VisualDSP++® 4.0
Kernel (VDK) User's Guide

Revision 1.0, January 2005

Part Number
82-000420-07

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

CONTENTS

PREFACE

USING VDK

VDK DATA TYPES

PROCESSOR-SPECIFIC NOTES

PREFACE

Thank you for purchasing Analog Devices (ADI) development software for digital signal processor applications.

Purpose of this Manual

The *VisualDSP++ 4.0 Kernel (VDK) User's Guide* contains information about VisualDSP++® Kernel, a Real Time Operating System integrated with the rest of the VisualDSP++ 4.0 development tools. The VDK incorporates scheduling and resource allocation techniques tailored specially for the memory and timing constraints of embedded programming and facilitates the development of structured applications using frameworks of template files.

The VDK is specially designed for effective operations on Analog Devices processor architectures.

The majority of the information in this manual is generic. However, information applicable to only a particular target processor, or to a particular processor family, is provided in Appendix A, "Processor-Specific Notes".

This manual is designed so that you can quickly learn about the kernel internal structure and operation.

Intended Audience

This manual is primarily intended for programmers who are familiar with Analog Devices processors. The manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual but should supplement it with other texts, such as Hardware Reference and Instruction Set Reference manuals, that describe your target architecture.

Manual Contents

The manual consists of:

- Chapter 1, "Introduction to VDK", concentrates on concepts, motivation, and general architectural principles of the VDK software.
- Chapter 2, "Configuration and Debugging of VDK Projects", describes the support for configuring and debugging a VDK-enabled project. For specific procedures on how to create, modify, and manage the kernel's components, refer to the VisualDSP++ online Help.
- Chapter 3, "Using VDK", describes the kernel's internal structure and components.
- Chapter 4, "VDK Data Types", describes built-in data types supported in the current release of the VDK.
- Chapter 5, "VDK API Reference", describes library functions and macros included in the current release of the VDK.

- Appendix A, "Processor-Specific Notes", provides processor-specific information for Blackfin, SHARC and TigerSHARC processor architectures.
- Appendix B, "Migrating Device Drivers", describes how to convert the device driver components created with VisualDSP++ 2.0 for use in projects built with VisualDSP++ 4.0.

What's New in This Manual

This first revision of the *VisualDSP++ 4.0 Kernel (VDK) User's Guide* documents VDK support for the new Blackfin processors (ADSP-BF539, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538 and ADSP-BF566) and SHARC processors (ADSP-21366, ADSP-21367, ADSP-21368 and ADSP-21369). See "Supported Processors" for a list of all of the processors supported in this release.

This manual also documents VDK functionality that is new in VisualDSP++ 4.0, including:

- Support for writing ISRs in C/C++
- Support for C++ exception handling
- Option to include modifiable startup code

The manual documents VisualDSP++ Kernel version 4.0.00.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at: `http://www.analog.com/processors/technicalSupport`
- E-mail tools questions to: `dsptools.support@analog.com`
- E-mail processor questions to: `dsp.support@analog.com`
- Phone questions to: **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor.
- Send questions by mail to:

```
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Supported Processors

The following is the list of Analog Devices, Inc. processors currently supported by VDK in VisualDSP++ 4.0.

TigerSHARC (ADSP-TSxxx) Processors

VDK currently supports the following TigerSHARC processors:

- ADSP-TS101, ADSP-TS201, ADSP-TS202, ADSP-TS203

SHARC (ADSP-21xxx) Processors

VDK currently supports the following SHARC processors:

- ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, ADSP-21160, ADSP-21161, ADSP-21261, ADSP-21262, ADSP-21266, ADSP-21267, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368, ADSP-21369

Blackfin (ADSP-BFxxx) Processors

VDK currently supports the following Blackfin processors:

- ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF535, ADSP-BF536, ADSP-BF537, ADSP-BF538, ADSP-BF539, ADSP-BF561, AD6532, ADSP-BF566

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

`MyAnalog.com` is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests. `MyAnalog.com` provides access to books, application notes, data sheets, code examples, and more.

Registration

Visit `www.myanalog.com` to sign up. Click **Register** to use `MyAnalog.com`. Registration takes about five minutes and serves as a means to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at `www.analog.com/processors`, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
  `dsp.support@analog.com`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  089/76 903-557 (Europe)
- Access the FTP Web site at
  `ftp ftp.analog.com` or `ftp 137.71.23.21`
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see these publications:

- *VisualDSP++ 4.0 Getting Started Guide*
- *VisualDSP++ 4.0 User's Guide*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for SHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for TigerSHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*
- *VisualDSP++ 4.0 Assembler and Preprocessor Manual*
- *VisualDSP++ 4.0 Linker and Utilities Manual*
- *VisualDSP++ 4.0 Loader Manual*
- *VisualDSP++ 4.0 Product Release Bulletin*
- *VisualDSP++ 4.0 Kernel (VDK) User's Guide*
- *VisualDSP++ 4.0 Quick Installation Reference Card*

For hardware information, refer to your processors's hardware reference, programming reference, or data sheet. All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

`http://www.analog.com/processors/resources/technicalLibrary`

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
`.CHM`	Help system files and manuals in Help format
`.HTM` or `.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the `.HTML` files requires a browser, such as Internet Explorer 4.0 (or higher).
`.PDF`	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the `.PDF` files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (`.CHM`) are located in the `Help` folder, and `.PDF` files are located in the `Docs` folder of your VisualDSP++ installation CD-ROM. The `Docs` folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.
- Double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.

Accessing Documentation From the Web

Download manuals at the following Web site:
`http://www.analog.com/processors/resources/technicalLibrary/manuals`

Select a processor family and book title. Download archive (`.ZIP`) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto `http://www.analog.com/salesdir/continent.asp`.

Hardware Tools Manuals

To purchase EZ-KIT Lite™ and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

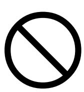
Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

Text conventions used in this manual are identified and described as follows.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

Additional conventions, which apply only to specific chapters, may appear throughout this document.

1 INTRODUCTION TO VDK

This chapter concentrates on concepts, motivation, and general architectural principles of the operating system kernel. It also provides information on how to partition a VDK application into independent, reusable functional units that are easy to maintain and debug.

The following sections provide information about the operating system kernel concepts.

- "Motivation" on page 1-2
- "Partitioning an Application" on page 1-4
- "Scheduling" on page 1-5
- "Protected Regions" on page 1-8
- "Thread and Hardware Interaction" on page 1-9

Motivation

All applications require control code as support for the algorithms that are often thought of as the “real” program. The algorithms require data to be moved to and/or from peripherals, and many algorithms consist of more than one functional block. For some systems, this control code may be as simple as a “superloop” blindly processing data that arrives at a constant rate. However, as processors become more powerful, considerably more sophisticated control may be needed to realize the processor’s potential, to allow the processor to absorb the required functionality of previously supported chips, and to allow a single processor to do the work of many. The following sections provide an overview of some of the benefits of using a kernel on a processor.

Rapid Application Development

The tight integration between the VisualDSP++ environment and the VDK allows rapid development of applications compared to creating all of the control code required by hand. The use of automatic code generation and file templates, as well as a standard programming interface to device drivers, allows you to concentrate on the algorithms and the desired control flow rather than on the implementation details. The VDK supports the use of C, C++, and assembly language. You are encouraged to develop code that is highly readable and maintainable, yet retaining the option of hand optimizing if necessary.

Debugged Control Structures

Debugging a traditional DSP application can be laborious because development tools (compiler, assembler, and linker among others) are not aware of the architecture of the target application and the flow of control that results. Debugging complex applications is much easier when instantaneous snapshots of the system state and statistical run-time data are clearly presented by the tools. To help offset the difficulties in debugging software, VisualDSP++ includes three versions of the VDK libraries containing full instrumentation (including error checking), only error checking, and neither instrumentation nor error checking.

In the instrumented mode, the kernel maintains statistical information and logging of all significant events into a history buffer. When the execution is paused, the debugger can traverse this buffer and present a graphical trace of the program's execution including context switches, pending and posting of signals, changes in a thread's status, and more.

Statistics are presented for each thread in a tabular view and show the total amount of time the thread has executed, the number of times it has been run, the signal it is currently blocked on, and other data. For more information, see "Debugging VDK Projects" on page 2-3 and the online Help.

Code Reuse

Many programmers begin a new project by writing the infrastructure portions that transfers data to, from, and between algorithms. This necessary control logic usually is created from scratch by each design team and infrequently reused on subsequent projects. The VDK provides much of this functionality in a standard, portable and reusable library. Furthermore, the kernel and its tight integration with the VisualDSP++ environment are designed to promote good coding practice and organization by partitioning large applications into maintainable and comprehensible blocks. By isolating the functionality of subsystems, the kernel helps to prevent the morass all too commonly found in systems programming.

The kernel is designed specifically to take advantage of commonality in user applications and to encourage code reuse. Each thread of execution is created from a user-defined template, either at boot time or dynamically by another thread. Multiple threads can be created from the same template, but the state associated with each created instance of the thread remains unique. Each thread template represents a complete encapsulation of an algorithm that is unaware of other threads in the system unless it has a direct dependency.

Hardware Abstraction

In addition to a structured model for algorithms, the VDK provides a hardware abstraction layer. Presented programming interfaces allow you to write most of the application in a platform independent, high-level language (C or C++). The VDK Application Programming Interface (API) is identical for all Analog Devices processors, allowing code to be easily ported to a different processor core.

When porting an application to a new platform, programmers must address the two areas necessarily specific to a particular processor—Interrupt Service Routines (ISR) and device drivers. The VDK architecture identifies a crisp boundary around these subsystems and supports the traditionally difficult development with a clear programming framework and code generation. Both interrupts and device drivers are declared with a graphical user interface in the VisualDSP++ Integrated Debugging and Development Environment (IDDE), which generates well-commented code that can be compiled without further effort.

Partitioning an Application

A VDK *thread* is an encapsulation of an algorithm and its associated data. When beginning a new project, use this notion of a thread to leverage the kernel architecture and to reduce the complexity of your system. Since many algorithms may be thought of as being composed of “subalgorithm” building blocks, an application can be partitioned into smaller functional units that can be individually coded and tested. These building blocks then become reusable components in more robust and scalable systems.

You define the behavior of VDK threads by creating *thread types*. Types are templates that define the behavior and data associated with all threads of that type. Like data types in C or C++, thread types are not used directly until an instance of the type is created. Many threads of the same thread type can be created, but for each thread type, only one copy of the code is linked into the executable code. Each thread has its own private set of variables defined for the thread type, its own stack, and its own C run-time context.

When partitioning an application into threads, identify portions of your design in which a similar algorithm is applied to multiple sets of data. These are, in general, good candidates for thread types. When data is present in the system in sequential blocks, only one instance of the thread type is required. If the same operation is performed on separate sets of data simultaneously, multiple threads of the same type can coexist and be scheduled for prioritized execution (based on when the results are needed).

Scheduling

The VDK is a *preemptive multitasking* kernel. Each thread begins execution at its entry point. Then, it either runs to completion or performs its primary function repeatedly in an infinite loop. It is the role of the scheduler to preempt execution of a thread and to resume its execution when appropriate. Each thread is given a *priority* to assist the scheduler in determining precedence of threads (see Figure 1-1).

The scheduler gives processor time to the thread with the highest priority that is in the *ready* state (see Figure 3-2 on page 3-15). A thread is in the ready state when it is not waiting for any system resources it has requested. A reference to each ready thread is stored in a structure that is internal to the kernel and known as the *ready queue*. For more information, see "Scheduling" on page 3-10.

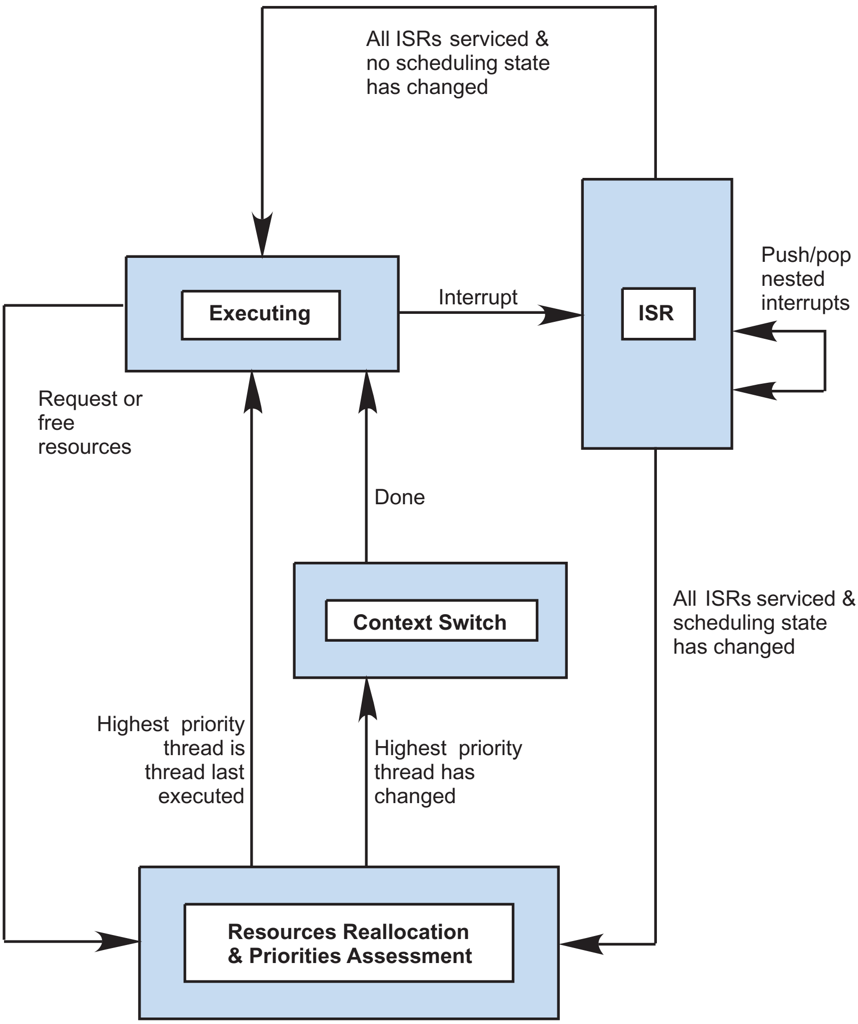


Figure 1-1. VDK State Diagram

Priorities

Each thread is assigned a dynamically modifiable priority based on the default for its thread type declared in the IDDE's **Project** window. An application is limited to thirty priority levels. However, the number of threads at each priority is limited, in practice, only by system memory. Priority level one is the highest priority, and priority thirty is the lowest. The system maintains an idle thread that is set to a priority lower than that of the lowest user thread.

Assigning priorities is one of the most difficult tasks of designing a real time preemptive system. Although there has been research in the area of rigorous algorithms for assigning priorities based on deadlines (for example, rate monotonic scheduling), most systems are designed by considering the interrupts and signals triggering the execution, while balancing the deadlines imposed by the system's input and output streams. For more information, see "Thread Parameters" on page 3-2.

Preemption

A running thread continues execution unless it requests a system resource using a kernel API. When a thread requests a signal (semaphore, event, device flag, or message) and the signal is available, the thread resumes execution. If the signal is not available, the thread is removed from the ready queue—the thread is *blocked* (see Figure 3-2 on page 3-15). The kernel does not perform a context switch as long as the running thread maintains the highest priority in the ready queue, even if the thread frees a resource and enables other threads to move to the ready queue at the same or lower priority. A thread can also be interrupted. When an interrupt occurs, the kernel yields to the hardware interrupt controller. When the ISR completes, the highest priority thread resumes execution.

For more information, see "Preemptive Scheduling" on page 3-12.

Protected Regions

Frequently, system resources must be accessed atomically. The kernel provides two levels of protection for code that needs to execute sequentially—*unscheduled regions* and *critical regions*.

Unscheduled and critical regions can be intertwined. You can enter critical regions from within unscheduled regions, or enter unscheduled regions from within critical regions. For example, if you are in an unscheduled region and call a function that pushes and pops a critical region, the system is still in an unscheduled region when the function returns.

Disabling Scheduling

The VDK scheduler can be disabled by entering an unscheduled region. The ability to disable scheduling is necessary when you need to free multiple system resources without being switched out, or access global variables that are modified by other threads without preventing interrupts from being serviced. While in an unscheduled region, interrupts are still enabled and ISRs execute. However, the kernel does not perform a thread context switch even if a higher priority thread becomes ready. Unscheduled regions are implemented using a stack style interface. This enables you to begin and end an unscheduled region within a function without concern for whether or not the calling code is already in an unscheduled region.

Disabling Interrupts

On occasions, disabling the scheduler does not provide enough protection to keep a block of thread code reentrant. A critical region disables both scheduling and interrupts. Critical regions are necessary when a thread is modifying global variables that may also be modified by an ISR. Similar to unscheduled regions, critical regions are implemented as a stack. Developers can enter and exit critical regions in a function without being concerned about the critical region state of the calling code. Care should be taken to keep critical regions as short as possible as they may increase interrupt latency.

Thread and Hardware Interaction

Threads should have minimal knowledge of hardware; rather, they should use *device drivers* for hardware control. A thread can control and interact with a device in a portable and hardware abstracted manner through a standard set of APIs.

The VDK Interrupt Service Routine framework encourages you to remove specific knowledge of hardware from the algorithms encapsulated in threads (see Figure 1-2). Interrupts relay information to threads through signals to device drivers or directly to threads. Using signals to connect hardware to the algorithms allows the kernel to schedule threads based on asynchronous events.

The VDK run-time environment can be thought of as a bridge between two domains, the *thread domain* and the *interrupt domain*. The interrupt domain services the hardware with minimal knowledge of the algorithms, and the thread domain is abstracted from the details of the hardware. Device drivers and signals bridge the two domains. For more information, see "Threads" on page 3-2.

Thread Domain With Software Scheduling

The thread domain runs under a C/C++ run-time model. The prioritized execution is maintained by a software scheduler with full context switching. Threads should have little or no direct knowledge of the hardware; rather, threads should request resources and then wait for them to become available. Threads are granted processor time based on their priority and

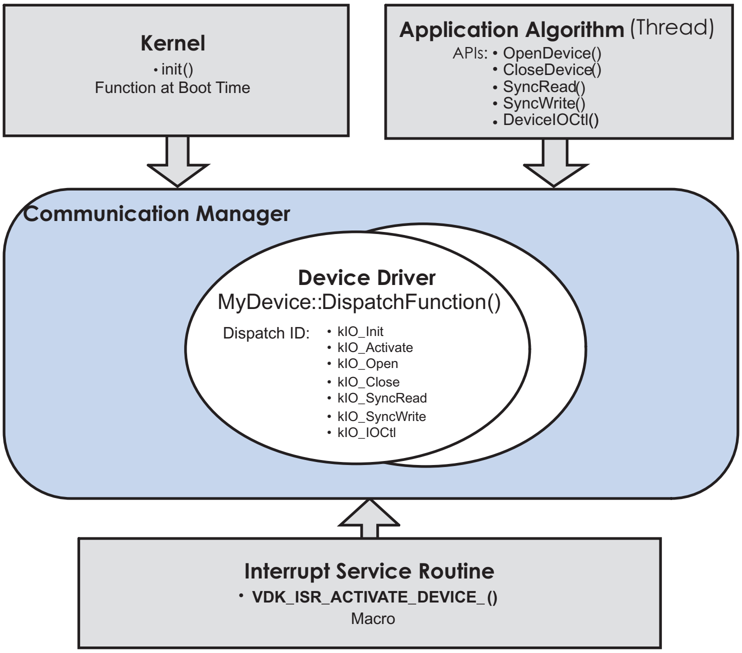


Figure 1-2. Device Drivers Entry Points

requested resources. Threads should minimize time spent in critical and unscheduled regions to avoid short-circuiting the scheduler and interrupt controller.

Interrupt Domain With Hardware Scheduling

The interrupt domain runs outside the C/C++ run-time model. The prioritized execution is maintained by the hardware interrupt controller. ISRs should be as small as possible. They should only do as much work as is necessary to acknowledge asynchronous external events and to allow peripherals to continue operations in parallel with the processor. ISRs should only signal that more processing can occur and leave the processing to threads. For more information, see "Interrupt Service Routines" on page 3-47.

Device Drivers

ISRs can communicate with threads directly using signals. Alternatively, an interrupt service routine and a thread can use a device driver to provide more complex device-specific functionality that is abstracted from the algorithm. A device driver is a single function with multiple entry conditions and domains of execution. For more information, see "Device Drivers" on page 3-53.

2 CONFIGURATION AND DEBUGGING OF VDK PROJECTS

This chapter contains information about the the configuration and debugging of VDK projects, including the VisualDSP++ Integrated Development and Debugging Environment (IDDE) support for VDK-enabled projects.

If you are new to VisualDSP++ application development software, we recommend that you start with the *VisualDSP++ 4.0 Getting Started Guide*.

The information included in this chapter is split into two areas:

- "Configuring VDK Projects"
- "Debugging VDK Projects"

Configuring VDK Projects

VisualDSP++ is extended to manage all of the VDK components. You start developing a VDK-based application by creating a set of source files. The IDDE automatically generates a source code framework for each user requested kernel object. Use the interface to supply the required information for these objects.

For specific procedures on how to set up VDK system parameters or how to create, modify, or delete a VDK component, refer to the VisualDSP++ online Help. Following the online procedures ensures your VDK projects build consistently and accurately with minimal project management. The process reduces development time and allows you to concentrate on algorithm development.

Linker Description File

When a new project makes use of the kernel, a reference to a VDK-specific default Linker Description File (`.LDF`) is added to the project. This file is copied to your project directory to allow modifications to be made to suit your individual hardware configurations.

Thread-Safe Libraries

Just as user threads must be reentrant, special "thread-safe" versions of the standard C and C++ libraries are included for use with the VDK. The default `.LDF` file included in VDK projects links with these libraries. If you modify your Linker Description File, ensure that the file links with the thread safe libraries. Your project's `.LDF` file resides in the **Linker Files** folder and is accessible via the **Project** tab of the **Project** window in VisualDSP++.

Header Files for the VDK API

When a VDK project is created in the development environment, one of the automatically generated files in the project directory is `VDK.h`. This header file contains enumerations for every user-defined object in the development environment and all VDK API declarations. Your source files must include `VDK.h` to access any kernel services.

Debugging VDK Projects

Debugging embedded software is a difficult task. To help offset the initial difficulties present in debugging VDK-enabled projects, the kernel offers special instrumented builds.

Instrumented Build Information

When building a VDK project, you have an option to include instrumentation in your executable by choosing **Full Instrumentation** as the instrumentation level in the **Kernel** tab of the **Project** window. An instrumented build differs from a release or non-instrumented build because the build includes extra code for thread statistic logging. In addition, an instrumented build creates a circular buffer of important system events. The extra logging introduces slight overhead in thread switches and certain API calls but helps you to trace system activities.

VDK State History Window

The VDK logs user-defined events and certain system state changes in a circular buffer. An event is logged in the history buffer with a call to "LogHistoryEvent()" on page 5-105. The call to `LogHistoryEvent()` logs four data values: the `ThreadID` of the calling thread, the tick when the call happened, the enumeration, and a value that is specific to the enumeration. Enumerations less than zero are reserved for use by the VDK. For more information about the history enumeration type, see "HistoryEnum" on page 4-19.

Using the history log, the IDDE displays a graph of running threads and system state changes in the **State History** window. Note that the data displayed in this window is only updated at halt. The **State History** window, the **Thread Status** and **Thread Event** legends are described in detail in the online Help.

Target Load Graph Window

Instrumented VDK builds allow you to analyze the average load of the processor over a period of time. The calculated load is displayed in the **Target Load** graph window. Although the calculation is not exact, the graph helps you to estimate the utilization level of the processor. Note that the information is updated at halt.

The **Target Load** graph shows the percent of time the target spent in the idle thread. A load of 0% means the VDK spent all of its time in the idle thread. A load of 100% means the target did not spend any time in the idle thread. Load data is processed using a moving window average. The load percentage is calculated for every clock tick, and all the ticks are averaged. The following formula is used to calculate the percentage of utilization for every clock tick.

```
Load = 1 - (# of times idle thread ran this tick) / (# of threads
run this tick)
```

For more information about the **Target Load** graph, refer to the online Help.

VDK Status Window

Besides history and processor load information, an instrumented build collects statistics for relevant VDK components, such as when a thread was created, last run, the number of times run, and so on. This data is displayed in the **Status** window and is updated at halt.

For more information about the VDK **Status** window, refer to the online Help.

General Debugging Tips

Even with the data collection features built into the VDK, debugging thread code is a difficult task. Due to the fact that multiple threads in a system are interacting asynchronously with device drivers, interrupts, and the idle thread, it can become difficult to track down the source of an error.

Unfortunately, one of the oldest and easiest debugging methods—inserting breakpoints—can have uncommon side effects in VDK projects. Since multiple threads (either multiple instantiations of the same thread type or different threads of different thread types) can execute the same function with completely different contexts, the utilization of non-thread-aware breakpoints is diminished. One possible workaround involves inserting some “thread-specific” breakpoints:

```
if (VDK_GetThreadID() == <thread_with_bug>)
   {
      <some statement>;    /* Insert breakpoint  */
   }
```

Kernel Panic

VDK calls an internal function named `KernelPanic()` under certain circumstances to indicate that an error has occurred that the system cannot recover from. By default, this function loops forever so that users can determine that a problem has occurred and to provide information to facilitate debugging.

The `KernelPanic()` function disables interrupts on entry to ensure that execution loops in the intended location. This function can be overridden by users in order to handle these types of errors differently, for example resetting the hardware.

The circumstances under which KernelPanic() is called include the following:

- Errors in the creation of a VDK boot item during startup
- Runtime errors that are not handled by a C++ Thread's error handler
- VDK internal errors

See "PanicCode" on page 4-33 for a complete list of the reasons for calling `KernelPanic()`.

To allow users to determine the cause of the "panic", the VDK sets up the following variables.

```
VDK::PanicCode  VDK::g_KernelPanicCode
VDK::SystemError VDK::g_KernelPanicError
int VDK::g_KernelPanicValue
int VDK::g_KernelPanicPC
```

where:

- `g_KernelPanicCode` indicates the reason why VDK needed to raise a Kernel Panic. For more information on the possible values of this variable, see "PanicCode" on page 4-33.
- `g_KernelPanicError` indicates in more detail the cause of the error. For example, if `g_KernelPanicCode` indicates a boot error, `g_KernelPanicError` specifies if the boot problem is in a semaphore, device flag, and so on. For more information, see "SystemError" on page 4-42.

- `g_KernelPanicValue` is a value whose meaning is determined by the error enumeration. For example, if the problem is creating the boot thread with ID 4, `g_KernelPanicValue` is 4.
- `g_KernelPanicPC` provides the address that produced the Kernel Panic.

3 USING VDK

This chapter describes how the VDK implements the general concepts described in Chapter 1, "Introduction to VDK". For information about the kernel library, see Chapter 5, "VDK API Reference".

The following sections provide information about the operating system kernel components and operations.

Threads

When designing an application, you partition it into threads, where each thread is responsible for a piece of the work. Each thread operates independently of the others. A thread performs its duty as if it has its own processor but can communicate with other threads.

Thread Types

You do not directly define threads; instead, you define thread types. A thread is an instance of a thread type and is similar to any other user defined type.

You can create multiple instantiations of the same thread type. Each instantiation of the thread type has its own stack, state, priority, and other local variables. You can distinguish between different instances of the same thread type. See "Thread Parameterization" on page 3-8 for further information. Each thread is individually identified by its "ThreadID" on page 4-48, a handle that can be used to reference that thread in kernel API calls. A thread can gain access to its ThreadID by calling "GetThreadID()" on page 5-88. A ThreadID is valid for the life of the thread—once a thread is destroyed, the ThreadID becomes invalid.

OldThreadIDs are eventually reused, but there is significant time between a thread's destruction and the ThreadID reuse—other threads have to recognize that the original thread is destroyed.

Thread Parameters

When a thread is created, the system allocates space in the heap to store a data structure that holds the thread-specific parameters. The data structure contains internal information required by the kernel and the thread type specifications provided by the user.

Stack Size

Each thread has its own stack. The full C/C++ run-time model, as specified in the appropriate *VisualDSP++ 4.0 C/C++ Compiler and Library Manual*, is maintained on a per thread basis. It is your responsibility to ensure that each thread has enough room on its stack for the return addresses and passed parameters of all function calls appropriate to the particular run-time model, user code structure, use of libraries, and so on. Stack overflows do not generate an exception, so an undersized stack has the potential to cause difficulties when reproducing bugs in your system.

Priority

Each thread type specifies a default priority. Threads may change their own (or another thread's) priority dynamically using the SetPriority() or ResetPriority() functions. Priorities are predefined by the kernel as an enumeration of type "Priority" on page 4-39 with a value of `kPriority1` being the highest priority (or the first to be scheduled) in the system. The priority enumeration such as `kPriority1 > kPriority2 >...` is set up. The number of priorities is limited to the processor's word size minus two.

Required Thread Functionality

Each thread type requires five particular functions to be declared and implemented. Default null implementations of all five functions are provided in the templates generated by the VisualDSP++ development environment. The thread's run function is the entry point for the thread. For many thread types, the thread's run and error functions are the only ones in the template you need to modify. The other functions allocate and free up system resources at appropriate times during the creation and destruction of a thread.

Run Function

The run function—called `Run()` in C++ and `RunFunction()` in C/assembly implemented threads—is the entry point for a fully constructed thread; `Run()` is roughly equivalent to `main()` in a C program. When a thread's run function returns, the thread is moved to the queue of threads waiting to free their resources. If the run function never returns, the thread remains running until destroyed.

Error Function

The thread's error function is called by the kernel when an error occurs in an API call made by the thread. The error function passes a description of the error in the form of an enumeration (see "SystemError" on page 4-42 for more details). It can also pass an additional piece of information whose exact definition depends on the error enumeration. A thread's default error-handling behavior makes the VDK go into "Kernel Panic" on page 2-5. See "Error Handling Facilities" on page 3-9 for more information about error handling in the VDK.

Create Function

The create function is similar to the C++ constructor. The function provides an abstraction used by the kernel API CreateThread() and CreateThreadEx() functions to enable dynamic thread creation. The create function is the first function called in the process of constructing a thread; it is also responsible for calling the thread's `init` function/constructor. Similar to the constructor, the create function executes in the context of the thread that is spawning a new thread by calling CreateThread() or CreateThreadEx(). The thread being constructed does not have a run-time context fully established until after these functions complete.

A create function calls the constructor for the thread and ensures that all of the allocations that the thread type required have taken place correctly. If any of the allocations failed, the create function deletes the partially created thread instantiation and returns a null pointer. If the thread has been constructed successfully, the create function returns the pointer to the thread. A create function should not call DispatchThreadError() because CreateThread() and CreateThreadEx() handle error reporting to the calling thread when the create function returns a null pointer.

The create function is exposed completely in C++ source templates. For C or assembly threads, the create function appears only in the thread's header file. If the thread allocates data in `InitFunction()`, you need to modify the create function in the thread's header to verify that the allocations are successful and delete the thread if not.

A thread of a certain thread type can be created at boot time by specifying a boot thread of the given thread type in the development environment. Additionally, if the number of threads in the system is known at build time, all the threads can be boot threads.

Init Function/Constructor

The `InitFunction()` (in C/assembly) and the constructor (in C++) provide a place for a thread to allocate system resources during the dynamic thread creation. A thread uses `malloc` (or `new`) when allocating the thread's local variables. A thread's `initFunction`/constructor cannot call any VDK APIs since the function is called during VDK initialization (for boot threads) or from within a different thread's context (for dynamically created threads).

Destructor

The destructor is called by the system when the thread is destroyed. A thread can do this explicitly with a call to DestroyThread(). The thread will also be destroyed if it runs to completion by reaching the end of its run function and falling out of scope. In all cases, you are responsible for freeing the memory and other system resources that the thread has claimed. Any memory allocated with `malloc` or `new` in the constructor should be released with a corresponding call to `free` or `delete` in the destructor.

A thread is not necessarily destroyed immediately when DestroyThread() is called. DestroyThread() takes a parameter that provides a choice of priority as to when the thread's destructor is called. If the second parameter, `inDestroyNow`, is `FALSE`, the thread is placed in a queue of threads to be cleaned up by the idle thread, and the destructor is called at a priority lower than that of any user threads. While this scheme has many advantages, it works, in essence, as the background garbage collector. This is not deterministic and presents no guarantees of when the freed resources are available to other threads.

If the `inDestroyNow` argument is passed to DestroyThread() with a value of `TRUE`, the destructor is called immediately. This assures the resources are freed when the function returns, but the destructor is effectively called at the priority of the currently running thread even if a lower priority thread is being destroyed.

Writing Threads in Different Languages

The code to implement different thread types may be written in C, C++, or assembly. The choice of language is transparent to the kernel. The development environment generates well commented skeleton code for all three choices.

One of the key properties of threads is that they are separate instances of the thread type templates—each with a unique local state. The mechanism for allocating, managing, and freeing thread local variables varies from language to language.

C++ Threads

C++ threads have the simplest template code of the three supported languages. User threads are derived classes of the abstract base class `VDK::Thread`. C++ threads have slightly different function names and include a `Create()` function as well a constructor.

Since user thread types are derived classes of the abstract base class `VDK::Thread`, member variables may be added to user thread classes in the header as with any other C++ class. The normal C++ rules for object scope apply so that threads may make use of `public`, `private`, and `static` members. All member variables are thread-specific (or instantiation-specific).

Additionally, calls to VDK APIs in C++ are different from C and assembly calls. All VDK APIs are in the `VDK` namespace. For example, a call to CreateThread() in C++ is `VDK::CreateThread()`. Do not expose the entire `VDK` namespace in your C++ threads with the `using` keyword.

C and Assembly Threads

Threads written in C rely on a C++ wrapper in their generated header file but are otherwise ordinary C functions. C thread function implementations are compiled without the C++ compiler extensions.

In C and assembly programming, the state local to the thread is accessed through a handle (a pointer to a pointer) that is passed as an argument to each of the four user thread functions. When more than a single word of state is needed, a block of memory is allocated with `malloc()` in the thread type's `InitFunction()`, and the handle is set to point to the new structure.

Each instance of the thread type allocates a unique block of memory, and when a thread of that type is executing, the handle references the correct memory reference. Note that, in addition to being available as an argument to all functions of the thread type, the handle can be obtained at any time for the currently running thread using the API GetThreadHandle(). The `InitFunction()` and `DestroyFunction()` implementations for a thread should not call GetThreadHandle() but should instead use the parameter passed to these functions, as they do not execute in the context of the thread being initialized or destroyed.

Thread Parameterization

In order to distinguish between different instances of the same thread type an 'initializer' value can be passed to a boot thread. The Initializer field in the kernel tab is used to do this for boot threads (see online **Help** for further details). In this case the initializer is passed into the thread constructor (or the thread's `InitFunction` in the case of a C thread) via the `user_data_ptr` field of the ThreadCreationBlock argument.

Because the `user_data_ptr` is a generic field (and a pointer), the initializer is passed by address. It can be extracted like this:

```
int initializer = *((int*)t.user_data_ptr);
```

in a C++ thread constructor, or:

```
int initializer;
initializer = *((int *)pTCB->user_data_ptr);
```

in a C thread `InitFunction`.

The usual thing to do with this value is to store it in a member variable of the thread itself. In a C thread the thread handle (which is also passed into the `InitFunction`) can be used for this purpose, but this is more easily achieved in C++.

Note also that for dynamically-created threads, the `user_data_ptr` can either be used in the same way (i.e. as a pointer to a unique integer) or as a completely general pointer to thread-specific data. This requires the use of CreateThreadEx() (which takes a ThreadCreationBlock as its argument) to create threads rather than CreateThread().

Global Variables

VDK applications can use global variables as normal variables. In C or C++, a variable defined in exactly one source file is declared as `extern` in other files in which that variable is used. In assembly, the `.GLOBAL` declaration exposes a variable outside a source file, and the `.EXTERN` declaration resolves a reference to a symbol at link time.

Plan carefully how you use global variables in a multithreaded system. Limit access to a single thread (a single instantiation of a thread type) whenever possible to avoid reentrancy problems. Critical and/or unscheduled regions should be used to protect operations on global entities that can potentially leave the system in an undefined state if not completed atomically.

Error Handling Facilities

VDK includes an error-handling mechanism that allows you to define behavior independently for each thread type. Each function call in Chapter 5, "VDK API Reference", lists the error codes that may result. For the complete list of all possible error codes, refer to "SystemError" on page 4-42.

The assumption underlying the error-handling mechanism in VDK is that all function calls normally succeed and, therefore, do not require an explicit error code to be returned and verified by the calling code. VDK's method differs from common C programming convention in which the return value of every function call must be checked to assure that the call has succeeded without an error. While that model is widely used in conventional systems programming, real-time embedded system function calls rarely, if ever, fail. When an error does occur, the system calls the user implemented `ErrorFunction()`.

You can call GetLastThreadError() to obtain an enumeration that describes the error condition. You can also call GetLastThreadErrorValue() to obtain an additional descriptive value whose definition depends on the specific error. For more information, see Table 5-21 on page 5-17. The thread's `ErrorFunction()` should check if the value returned by GetLastThreadError() is one that can be handled intelligently and can perform the appropriate operations. Any enumerated errors that the thread cannot handle must be passed to the default thread error function, which then raises "Kernel Panic" (on page 2-5). For instructions on how to pass an error to the error function, see comments included in the generated thread code.

Scheduling

The scheduler's role is to ensure that the highest priority ready thread is allowed to run at the earliest possible time. The scheduler is never invoked directly by a thread but is executed whenever a kernel API—called from either a thread or an Interrupt Service Routine (ISR) —changes the highest priority thread. The scheduler is not invoked during critical or unscheduled regions, but can be invoked immediately at the close of either type of protected region.

Ready Queue

The scheduler relies on an internal data structure known as the *ready queue*. The queue holds references to all threads that are not blocked or sleeping. All threads in the ready queue have every resource needed to run; they are only waiting for processor time. The exception is the currently running thread, which remains in the ready queue during execution.

The ready queue is called a queue because it is arranged as a prioritized First-In First-Out (FIFO) buffer. That is, when a thread is moved to the ready queue, it is added as the last entry at its priority. For example, there are four threads in the ready queue at the priorities `kPriority3`, `kPriority5`, and `kPriority7`, and an additional thread is made ready with a priority of `kPriority5` (see Figure 3-1).

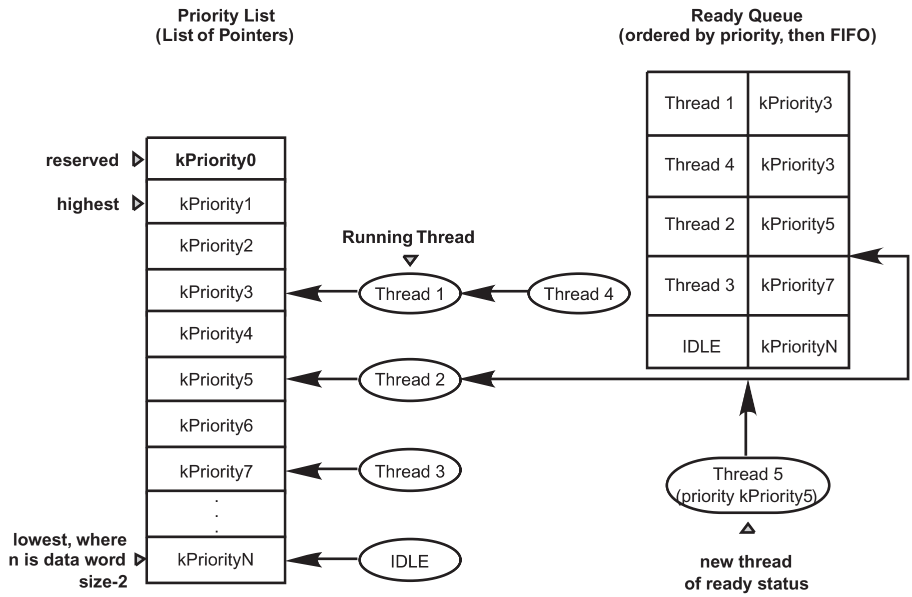


Figure 3-1. Ready Queue

The additional thread is inserted after the old thread with the priority of `kPriority5`, but before the thread with the priority of `kPriority7`. Threads are added to and removed from the ready queue in a fixed number of cycles regardless of the size of the queue.

Scheduling Methodologies

VDK always operates as a preemptive kernel. However, you can take advantage of a number of modes to expand the options for simpler or more complex scheduling in your applications.

Cooperative Scheduling

Multiple threads may be created at the same priority level. In the simplest scheduling scheme, all threads in the system are given the same priority, and each thread has access to the processor until it manually yields control. This arrangement is called *cooperative multithreading*.

When a thread is ready to defer to the next thread at the same priority level, the thread can do so by calling the Yield() function, placing the currently running thread at the end of the list. In addition, any system call that causes the currently running thread to block would have a similar result. For example, if a thread pends on a signal that is not currently available, the next thread in the queue at that priority starts running.

Round-Robin Scheduling

Round-robin scheduling, also called time slicing, allows multiple threads with the same priority to be given processor time automatically in fixed duration allotments. In VDK, priority levels may be designated as round-robin mode at build time and their period specified in system ticks. Threads at that priority are run for that duration, as measured by the number of VDK Ticks. If the thread is preempted by a higher priority thread for a significant amount of time, the time is not subtracted from the time slice. When a thread's round-robin period completes, it is moved to the end of the list of threads at its priority in the ready queue. Note that the round-robin period is subject to jitter when threads at that priority are preempted.

Preemptive Scheduling

Full *preemptive scheduling*, in which a thread gets processor time as soon as it is placed in the ready queue if it has a higher priority than the running thread, provides more power and flexibility than pure cooperative or round-robin scheduling.

VDK allows the use of all three paradigms without any modal configuration. For example, multiple non-time-critical threads can be set to a low priority in the round-robin mode, ensuring that each thread gets processor time without interfering with time critical threads. Furthermore, a thread can yield the processor at any time, allowing another thread to run. A thread does not need to wait for a timer event to swap the thread out when it has completed the assigned task.

Disabling Scheduling

Sometimes it is necessary to disable the scheduler when manipulating global entities. For example, when a thread tries to change the state of more than one signal at a time, the thread can enter an unscheduled region to ensure that all updates occur atomically. Unscheduled regions are sections of code that execute without being preempted by a higher priority thread. Note that interrupts are serviced in an unscheduled region, but the same thread runs on return to the thread domain. Unscheduled regions are entered through a call to PushUnscheduledRegion(). To exit an unscheduled region, a thread calls PopUnscheduledRegion().

Unscheduled regions (in the same way as critical regions, covered in "Enabling and Disabling Interrupts" on page 3-47) are implemented with a stack. Using nested critical and unscheduled regions allows you to write code that activates a region without being concerned about the region context when a function is called. For example:

```
void My_UnscheduledFunction()
{
  VDK_PushUnscheduledRegion();
  /* In at least one unscheduled region, but
     this function can be used from any number
     of unscheduled or critical regions */
  /* ... */
  VDK_PopUnscheduledRegion();
}
void MyOtherFunction()
```

```
{
  VDK_PushUnscheduledRegion();
  /* ... */
  /* This call adds and removes one unscheduled region */
  My_UnscheduledFunction();
  /* The unscheduled regions are restored here */
  /* ... */
  VDK_PopUnscheduledRegion();
}
```

An additional function for controlling unscheduled regions is PopNestedUnscheduledRegions(). This function completely pops the stack of all unscheduled regions. Although the VDK includes PopNestedUnscheduledRegions(), applications should use the function infrequently and balance regions correctly.

Entering the Scheduler From API Calls

Since the highest priority ready thread is the running thread, the scheduler is called only when a higher priority thread becomes ready. Because a thread interacts with the system through a series of API calls, the points at which the highest priority ready thread may change are well defined. Therefore, a thread invokes the scheduler only at these times, or whenever it leaves an unscheduled region.

Entering the Scheduler From Interrupts

ISRs communicate with the thread domain through a set of APIs that do not assume any context. Depending on the system state, an ISR API call may require the scheduler to be executed. VDK reserves the lowest priority software interrupt to handle the reschedule process.

If an ISR API call affects the system state, the API raises the lowest priority software interrupt. When the lowest priority software interrupt is scheduled to run by the hardware interrupt dispatcher, the interrupt reduces to subroutine and enters the scheduler. If the interrupted thread is not in an unscheduled region and a higher priority thread has become ready, the scheduler swaps out the interrupted thread and swaps in the new highest priority ready thread. The lowest priority software interrupt respects any unscheduled regions the running thread is in. However, interrupts can still service device drivers, post semaphores, and so on. On leaving the unscheduled region, the scheduler is run again, and the highest priority ready thread becomes the running thread (see Figure 3-2).

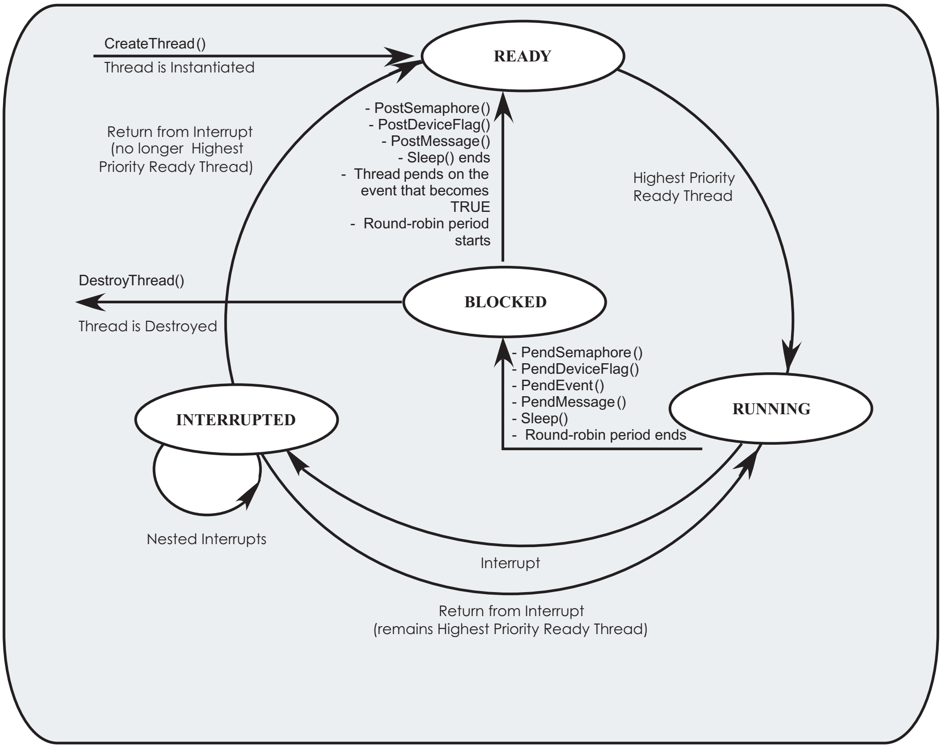


Figure 3-2. Thread State Diagram

Idle Thread

The idle thread is a predefined, automatically-created thread with the ThreadID set at zero and a priority lower than that of any user threads. Thus, when there are no user threads in the ready queue, the idle thread runs. The only substantial work performed by the idle thread is the freeing of resources of threads that have been destroyed. In other words, the idle thread handles destruction of threads that were passed to DestroyThread() with a value of FALSE for inDestroyNow. Depending on the platform, it may be possible to customize certain properties of the Idle thread, such as its stack size and the heap from which all its memory requirements (including the Idle thread stack) are allocated. (See online **Help** for further details.) There may be processor-specific requirements for particular Idle thread properties (See Appendix A, “Processor-Specific Notes”, for further details.)

The time spent in threads other than the idle thread is shown plotted as a percentage over time on the **Target Load** tab of the **State History** window in VisualDSP++. See “VDK State History Window” on page 2-3 and online Help for more information about the **State History** window.

Signals

Threads have four different methods for communication and synchronization:

- “Semaphores” on page 3-17
- “Messages” on page 3-23
- “Events and Event Bits” on page 3-39
- “Device Flags” on page 3-46

Each communication method has a different behavior and use. A thread pends on any of the four types of signals, and if a signal is unavailable, the thread blocks until the signal becomes available or (optionally) a timeout is reached.

Semaphores

Semaphores are protocol mechanisms offered by most operating systems. Semaphores are used to:

- Control access to a shared resource
- Signal a certain system occurrence
- Allow threads to synchronize
- Schedule periodic execution of threads

The maximum number of active semaphores and initial state of the semaphores enabled at boot time are set up when your project is built.

Behavior of Semaphores

A semaphore is a token that a thread acquires so that the thread can continue execution. If the thread pends on the semaphore and it is available (the count value associated with the semaphore is greater than zero), the semaphore is acquired, its count value is decremented by one and the thread continues normal execution. If the semaphore is not available (its count is zero), the thread trying to acquire (pend on) the semaphore blocks until the semaphore is available, or the specified timeout occurs. If the semaphore does not become available in the time specified, the thread continues execution in its error function.

Semaphores are global resources accessible to all threads in the system. Threads of different types and priorities can pend on a semaphore. When the semaphore is posted, the thread with the highest priority that has been waiting the longest is moved to the ready queue. If there are no threads pending on the semaphore, its count value is incremented by one. The count value is limited by the maximum value specified at the time of the semaphore creation. Additionally, unlike many operating systems, VDK semaphores are not owned. In other words, any thread is allowed to post a semaphore (make it available). If a thread has requested (pended on) and acquired a semaphore, and the thread is subsequently destroyed, the semaphore is not automatically posted by the kernel.

Besides operating as a flag between threads, a semaphore can be set up to be periodic. A periodic semaphore is posted by the kernel every *n* ticks, where *n* is the period of the semaphore. Periodic semaphores can be used to ensure that a thread is run at regular intervals.

Thread's Interaction With Semaphores

Threads interact with semaphores through the set of semaphore APIs. These functions allow a thread to create a semaphore, destroy a semaphore, pend on a semaphore, post a semaphore, get a semaphore's value, and add or remove a semaphore from the periodic queue.

Pending on a Semaphore

Figure 3-3 illustrates the process of pending on a semaphore.

Threads can pend on a semaphore with a call to PendSemaphore(). When a thread calls PendSemaphore(), it performs one of the following:

- Acquires the semaphore, decrements its count by one, and continues execution.
- Blocks until the semaphore is available or the specified timeout occurs.

If the semaphore becomes available before the timeout occurs or a timeout occurs and the `kNoTimeoutError` bit has been specified in the timeout parameter, the thread continues execution; otherwise, the thread's error function is called and the thread continues execution. You should not call

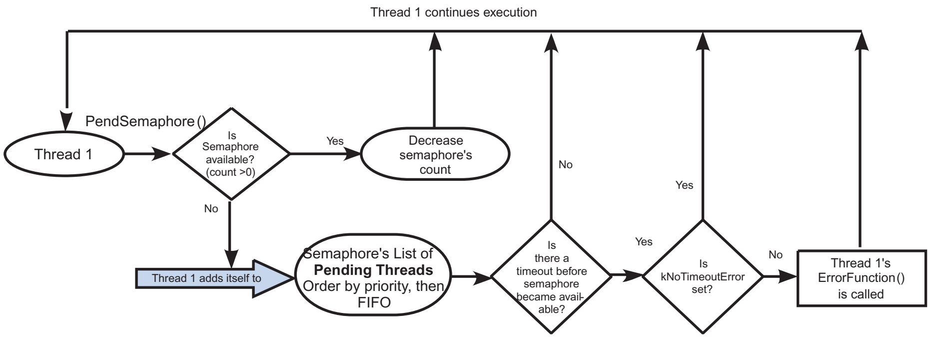


Figure 3-3. Pending on a Semaphore

PendSemaphore() within an unscheduled or critical region because if the semaphore is not available, then the thread will block. However, with the scheduler disabled, execution cannot be switched to another thread. Pending with a timeout of zero on a semaphore pends without timeout.

Posting a Semaphore

Semaphores can be posted from two different scheduling domains: the thread domain and the interrupt domain. If there are threads pending on the semaphore, posting it moves the highest priority thread from the semaphore's list of pending threads to the ready queue. All other threads are left blocked on the semaphore until their timeout occurs, or the semaphore becomes available for them. If there are no threads pending on the semaphore, posting it increments the count value by one. If the maximum count (which is specified when the semaphore is created) is reached, posting the semaphore has no effect.

Posting from the Thread Domain:
Figure 3-4 and Figure 3-5 illustrate the process of posting semaphores from the thread domain.

A thread can post a semaphore with a call to the PostSemaphore() API. If a thread calls PostSemaphore() from within a scheduled region (see Figure 3-4), and a higher priority thread is moved to the ready queue, the thread calling PostSemaphore() is context switched out.

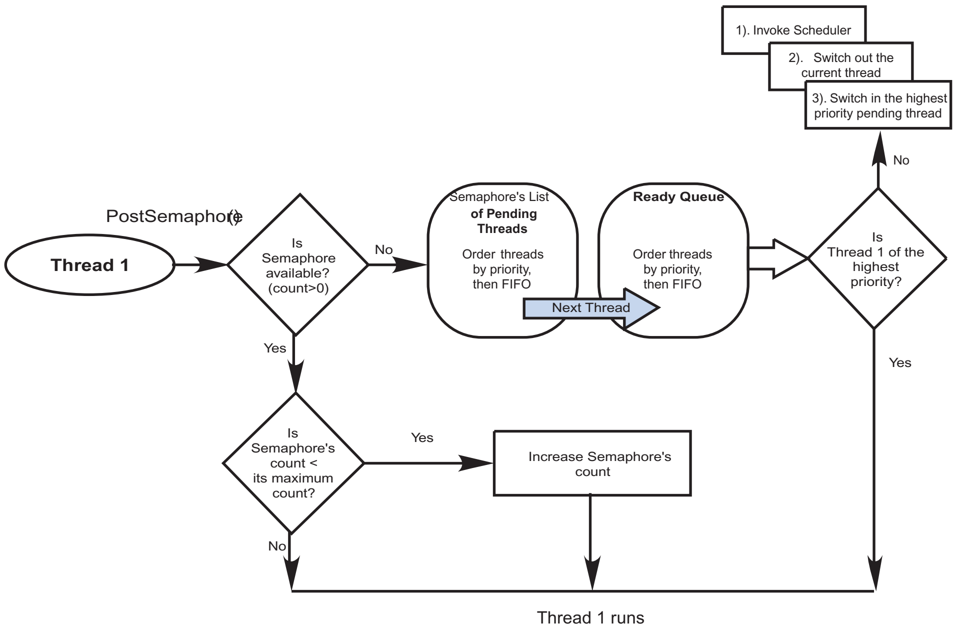


Figure 3-4. Thread Domain/Scheduled Region: Posting a Semaphore

If a thread calls PostSemaphore() from within an unscheduled region where the scheduler is disabled, the highest priority thread pending on the semaphore is moved to the ready queue, but no context switch occurs (see Figure 3-5).

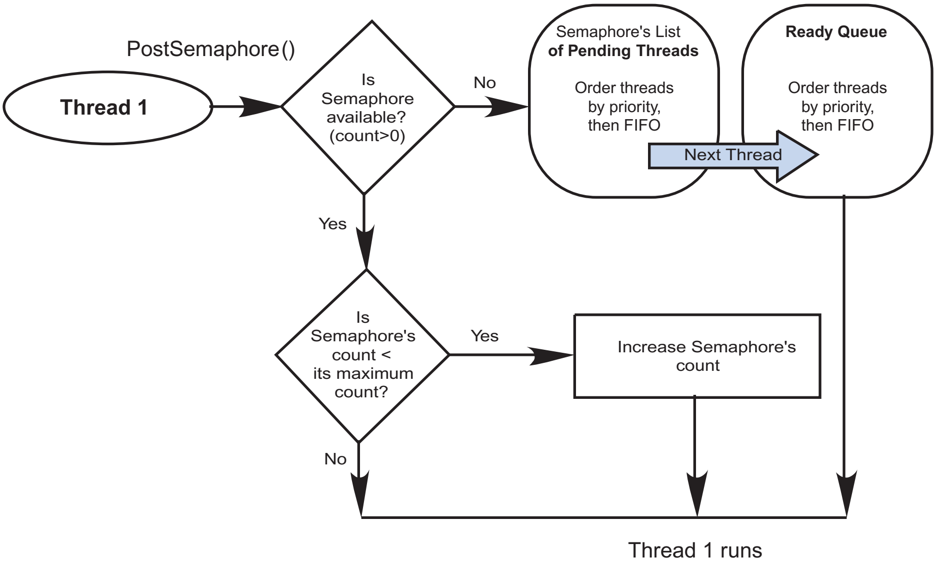


Figure 3-5. Thread Domain/Unscheduled Region: Posting a Semaphore

Posting from the Interrupt Domain:
Interrupt subroutines can also post semaphores. Figure 3-6 illustrates the process of posting a semaphore from the interrupt domain.

An ISR posts a semaphore by calling the VDK_ISR_POST_SEMAPHORE_() macro. The macro moves the highest priority thread to the ready queue and latches the low priority software interrupt if a call to the scheduler is required. When the ISR completes execution, and the low priority software interrupt is run, the scheduler is run. If the interrupted thread is in a scheduled region and a higher priority thread becomes ready, the interrupted thread is switched out and the new thread is switched in.

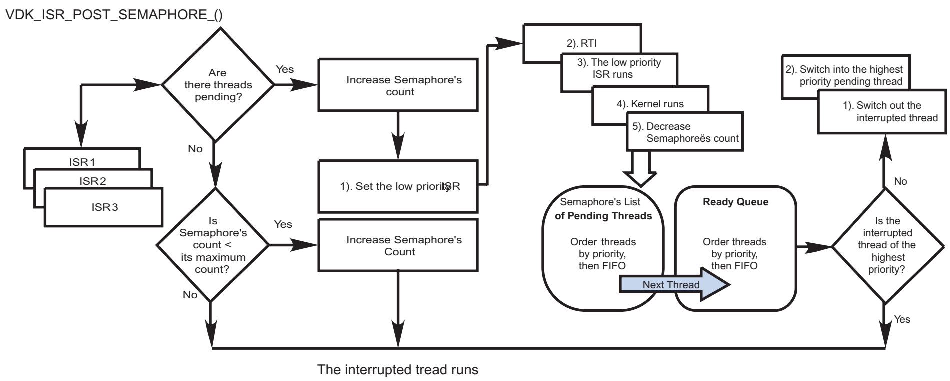


Figure 3-6. Interrupt Domain: Posting a Semaphore

Periodic Semaphores

Semaphores can also be used to schedule periodic threads. The semaphore is posted every *n* ticks (where *n* is the semaphore's period). A thread can then pend on the semaphore and be scheduled to run every time the semaphore is posted. A periodic semaphore does not guarantee that the thread pending on the semaphore is the highest priority scheduled to run, or that scheduling is enabled. All that is guaranteed is that the semaphore is posted, and the highest priority thread pending on that semaphore moves to the ready queue.

Periodic semaphores are posted by the kernel during the timer interrupt at system tick boundaries. Periodic semaphores can also be posted at any time with a call to PostSemaphore() or VDK_ISR_POST_SEMAPHORE_(). Calls to these functions do not affect the periodic posting of the semaphore.

Messages

Messages are an inter-thread communication mechanism offered by many operating systems. Messages can be used to:

- Communicate information between two threads
- Control access to a shared resource
- Signal a certain occurrence and communicate information about the occurrence
- Allow two threads to synchronize

The maximum number of messages supported in the system is set up when the project is built. When the maximum number of messages is non-zero, a system-owned memory pool is created to support messaging. The properties of this memory pool should not be altered. Further information on memory pools is given in "Memory Pools" on page 3-70.

Behavior of Messages

Messages allow two threads to communicate over logically separate channels. A message is sent on one of 15 possible channels, `kMsgChannel1` to `kMsgChannel15`. Messages are retrieved from these channels in priority order: `kMsgChannel1`, `kMsgChannel2`, ... `kMsgChannel15`. Each message can pass a reference to a data buffer, in the form of a message payload, from the sending thread to the receiving thread.

A thread creates a message (optionally associating a payload) and then posts (sends) the message to another thread. The posting thread continues normal execution unless the posting of the message activates a higher priority thread which is pending on (waiting to receive) the message.

A thread can pend on the receipt of a message on one or more of its channels. If a message is already queued for the thread, it receives the message and continues normal execution. If no suitable message is already queued, the thread blocks until a suitable message is posted to the thread, or until the specified timeout occurs. If a suitable message is not posted to the thread in the time specified, the thread continues execution in its error function.

Unlike semaphores, each message always has a defined owner at any given time, and only the owning thread can perform operations on the message. When a thread creates a message, it owns the message until it posts the message to another thread. The message ownership changes to the receiving thread following the post, when it is queued on one of the receiving message's channels. The receiving thread is now the owner of the message. The only operation that can be performed on the message at this time is pending on the message, making the message and its contents available to the receiving thread.

A message can only be destroyed by its owner; therefore, a thread that receives a message is responsible for either destroying or reusing a message. Ownership of the associated payload also belongs to the thread that owns the message. The owner of the message is responsible for the control of any memory allocation associated with the payload.

Each thread is responsible for destroying any messages it owns before it destroys itself. If there are any messages left queued on a thread's receiving channels when it is destroyed, then the system destroys the queued messages. As the system has no knowledge of the contents of the payload, the system does not free any resources used by the payload.

Thread's Interaction With Messages

Threads interact with messages through the set of message APIs. The functions allow a thread to create a message, pend on a message, post a message, get and set information associated with a message, and destroy a message.

Pending on a Message

Figure 3-7 illustrates the process of pending on a message.

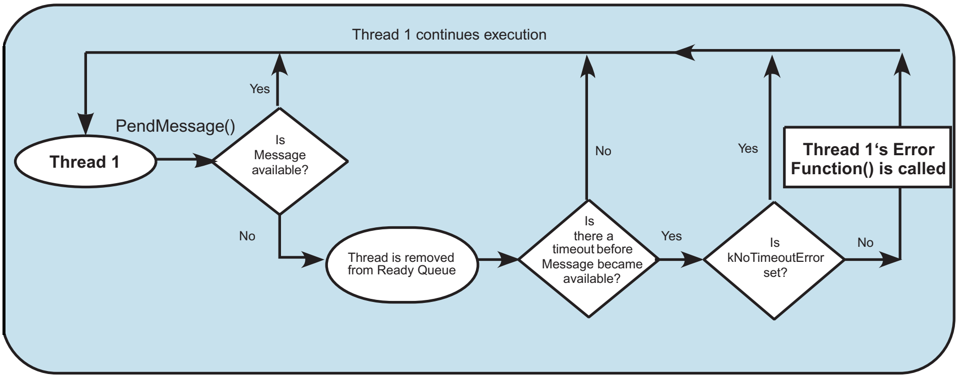


Figure 3-7. Pending on a Message

Threads can pend on a message with a call to PendMessage(), specifying one or more channels that a message is to be received on. When a thread calls PendMessage(), it does one of the following:

- Receives a message and continues execution
- Blocks until a message is available on the specified channel(s) or the specified timeout occurs

If messages are queued on the specified channels before the timeout occurs, or if a timeout occurs and the `kNoTimeoutError` bit is specified in the timeout parameter, the thread continues normal execution; otherwise, the thread continues execution in its error function.

Once a message has been received, you can obtain the identity of the sending thread and the channel the message was received on by calling GetMessageReceiveInfo(). You can also obtain information about the payload by calling GetMessagePayload(), which returns the type and length of the payload in addition to its location. Do not call PendMessage() within an unscheduled or critical region because if a message is not available, then the thread blocks, but with the scheduler disabled, execution cannot be switched to another thread. Pending with a timeout of zero on a message pends without timeout.

Posting a Message

Posting a message sends the specified message and its payload reference to the specified thread and transfers ownership of the message to the receiving thread. The details of the message payload can be specified by a call on the SetMessagePayload() function, which allows the thread to specify the payload type, length, and location before posting the message. A thread can send a message it currently owns with a call to PostMessage(), specifying the destination thread and the channel the message is to be sent on.

Figure 3-8 illustrates the process of posting a message from a scheduled region.

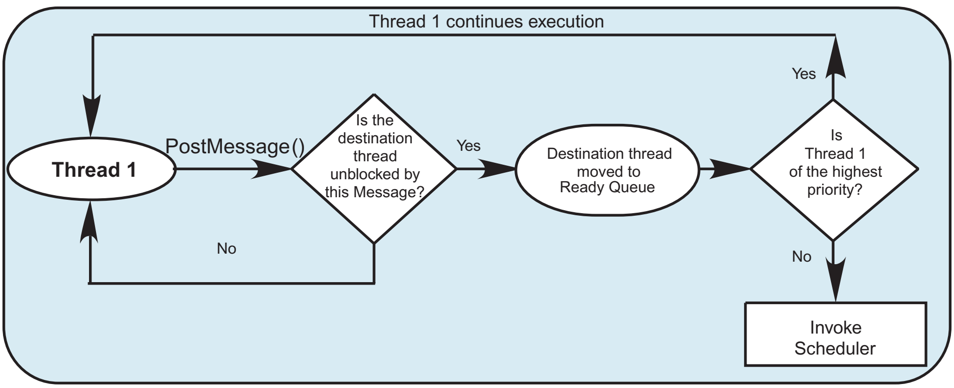


Figure 3-8. Posting a Message From a Scheduled Region

If a thread calls PostMessage() from within a scheduled region, and a higher priority thread is moved to the ready queue on receiving the message, then the thread calling PostMessage() is context switched out.

Figure 3-9 illustrates the process of posting a message from an unscheduled region.

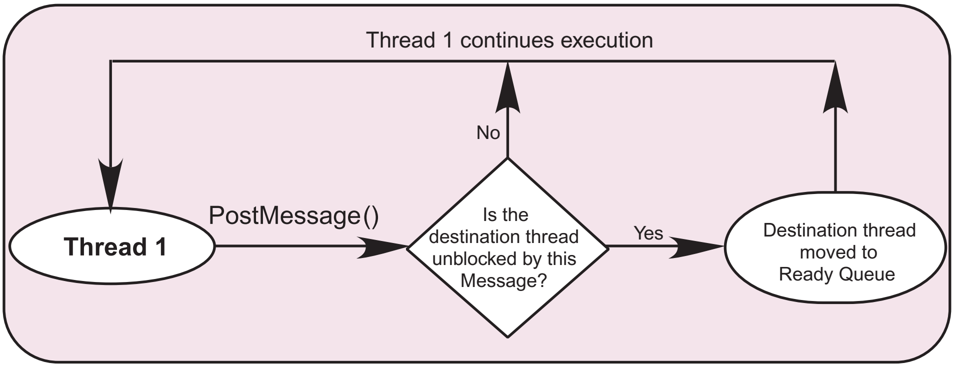


Figure 3-9. Posting a Message From an Unscheduled Region

If a thread calls PostMessage() from within an unscheduled region, even if a higher priority thread is moved to the ready queue to receive the message, the thread that calls PostMessage() continues to execute.

Multiprocessor Messaging

VDK messaging functionality was extended in VisualDSP++ 3.5 to allow messages to be passed between the processors in a multiprocessor configuration. The APIs and corresponding behaviors are, as much as possible, the same as for intra-processor messaging, but with extensions.

Each processor in a multiprocessor configuration is referred to as a *Node* and must have its own VisualDSP++ project. This means that each node runs its own instance of the VDK kernel and all VDK entities (such as semaphores, event bits, events, and so on, but excepting threads) are private to that node. Each node has a unique numeric node ID, which is set in the project's kernel tab.

Threads are uniquely identified across the multiprocessor system by embedding the node ID as a 5-bit field within the ThreadID. The size of this field limits the maximum number of nodes in the system to 32. Threads are permanently located on the node where they are created—there is no "migration" of threads between nodes.

In order for threads to be referenced on other nodes, each project in a multiprocessor system uses the kernel tab's **Import** list to import the project files for all the other nodes in the system. This makes the boot ThreadIDs for all the projects visible and usable across the system. Threads located on other nodes may then be used as destinations for the `VDK::PostMessage()` and `VDK::ForwardMessage()` functions, though not for any other thread-related API function.

Boot threads serve as "anchor points" for node-to-node communications, as their identities are known at build time. In order to communicate with dynamically-created threads on other nodes, it is necessary to pass the ThreadIDs as data between the nodes (that is, in a message payload). A reply to an incoming message can always be sent, regardless of the identity of the sending thread, as the sender's ID is carried in the message itself. Boot threads can therefore be used to provide information about dynamically-created threads, but such arrangements are application-specific and must form part of the system design.

Routing Threads (RThreads)

When a message is posted by a thread, the destination node ID (embedded in the destination ThreadID) is examined. If it matches the node ID of the node on which the thread is running, then the message is placed directly into the message queue of the destination thread, exactly as in single-processor messaging. If the node IDs do not match, then the message is passed to one of the Routing Threads (`RThreads`), which is responsible for the next stage in the process of moving the message to its destination.

Each `RThread` takes one of two roles, Incoming or Outgoing, which is fixed at the time of its creation.

Each `RThread` employs a device driver, which manages the physical details of moving messages between nodes. An Outgoing `RThread` has its device open for writing, while an Incoming `RThread` has its device open for reading.

Outgoing `RThreads` are referenced via a Routing Table, which is constructed by VisualDSP++ at build time. When a message must be sent to a different node, the destination node ID is used as an index into this table to select which outgoing `RThread` will handle transmission of the message.

Each node must contain at least one incoming and one outgoing `Rthread`, together with their corresponding device drivers. More `RThreads` may be included, depending on the number of physical connections to other nodes. However, the number of outgoing `RThreads` may be less than the number of nodes in the system, so that more than one entry in the routing table may map to the same `RThread`. This means that the topology of the multiprocessor system may require that a message make more than one "hop" to reach its final destination.

An outgoing `RThread`, when idle, waits for messages to be placed on any channel of its message queue, and then transmits that message (as a message packet) by making one or more `SyncWrite()` calls to its associated device driver. These `SyncWrite()` calls may block waiting I/O completion.

An incoming `RThread`, when idle, blocks in a `SyncRead()` call to its device driver awaiting reception of a message packet. Once the packet has been received, and expanded into a message object, the `RThread` forwards it to its destination. This may involve passing the message to an outgoing `RThread` if the current node is not the message's final destination.

The actual message objects, as referenced by particular `MessageIDs`, are each local to a particular node. When a message is transmitted between two nodes, only the message contents are passed over (as a message packet). The message object itself is destroyed on the sending side and recreated on the receiving side with the following consequences:

- The message usually has a different ID on the receiving side than it does on the sending.
- Message objects that are passed to an outgoing `RThread` are destroyed after transmission, and hence, returned to the pool of free messages.
- When a message packet is received by an incoming `RThread`, a message object must be created (from the pool of free messages).

Figure 3-10 shows the path taken by a message being sent between two threads on different nodes (A and B), where a direct connection exists between the two nodes.

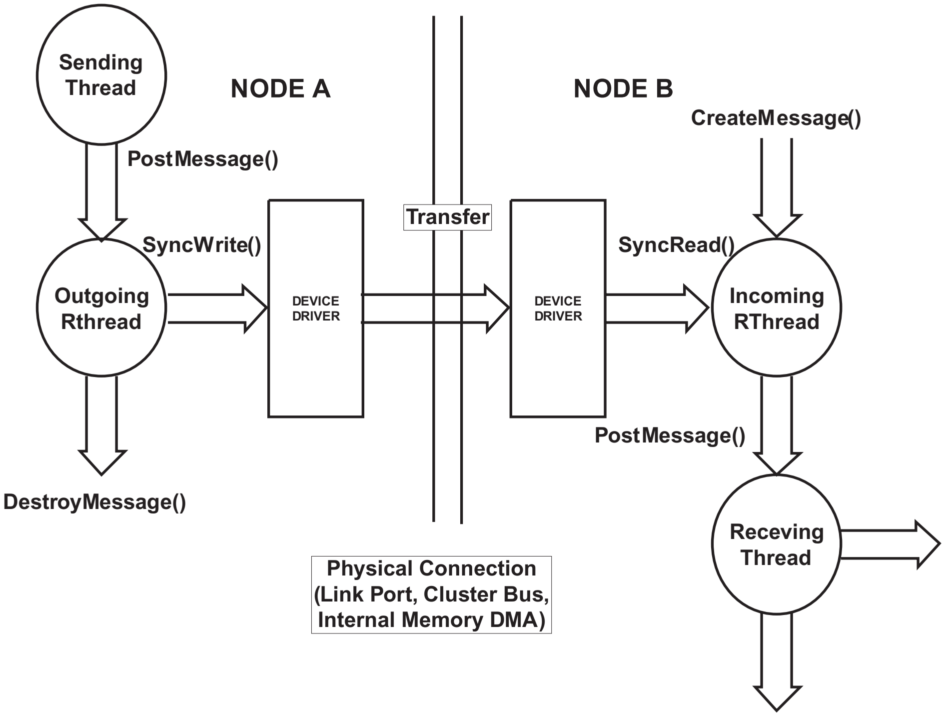


Figure 3-10. Sending Messages Between Adjacent Nodes

Figure 3-11 shows a scenario where node Y is an intermediate "hop" on the way to a third node, Z. The message is posted by Y's incoming `RThread`, directly into the message queue of the routing thread for the outgoing connection.

If the message allocation by the incoming `RThread` fails, then a system error is raised and execution stops on that node. This is necessary because the alternative of "dropping" the message is unacceptable (message delivery is defined as being reliable in the VDK).

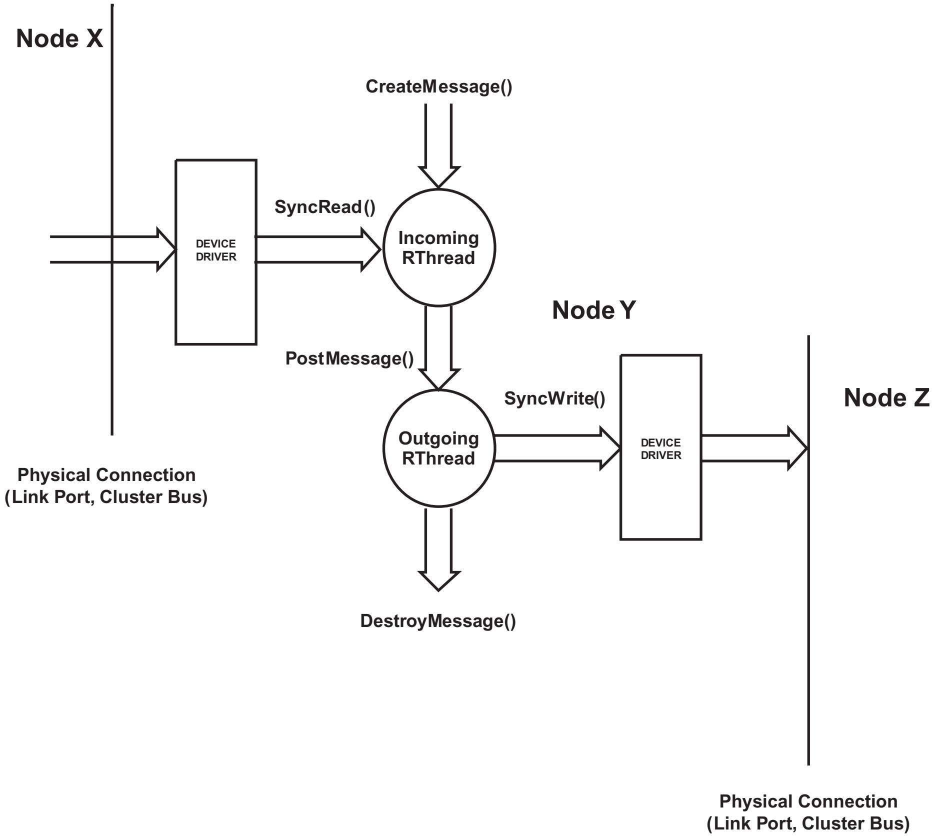


Figure 3-11. Sending Messages Between Non-Adjacent Nodes via an Intermediate Node

There are a number of ways of avoiding this problem:

1. Provide a careful design of the message flow in the application, and careful choice of priorities for the `RThreads`. The use of loopback (that is, returning messages to sender rather than destroying them) may assist with this.

2. Preallocate all messages during initialization and use loopback so that they never need to be explicitly destroyed. The maximum messages setting (in the kernel tab) for each node must be set equal to the total number of messages in the overall system. This ensures that there is no failure even in the worst case of all messages being sent to the same node at once.

3. A counting semaphore may be installed to regulate the message flow into the node, using the `VDK::InstallMessageControlSemaphore()` API function. The initial count of this semaphore should be set to less than or equal to the number of free messages which are reserved for use by the `RThreads`. This semaphore is pended on prior to each message allocation by an incoming `RThread`, and posted after each deallocation by an outgoing `RThread`. Provided that the semaphore's count is never less than the number of free messages on the node, then the allocations never fail. However, message flow into the node may stall if the semaphore count falls to zero.

Option 1 requires a thorough understanding of application behavior. Option 2 carries a memory space overhead, as more space may be reserved for messages than is actually needed at runtime, but is the simplest solution if this is no problem. Option 3 carries a performance overhead due to the semaphore pend and post operations. Additionally, if message flow stalls, which may occur with Option 3, it may have other consequences for the system.

Data Transfer (Payload Marshalling)

Very simple messages can be sent between nodes without interpretation, that is, if the message information is entirely conveyed by the two words of message data (internal payload). However, if the message actually has an in-memory (external) payload, then the address of this payload may not be meaningful once the message arrives on another node. In these cases the payload must be transferred along with the message. This is done via *payload marshalling*.

Any message type that has the MSB (sign bit) set (that is, a negative value) is considered to be a marshalled type, meaning that the system expects to allocate, deallocate, and transfer the payload automatically from node to node.

Since the organization of the payload for a particular message type is entirely the choice of the application designer (it might be a linked list or a tree, rather than a plain memory block), the allocation, deallocation, and transfer of the payload is the responsibility of a marshalling function. Pointers to these functions are held in a static marshalling table, which is indexed using the low-order bits of the message type.

The marshalling function implements (at least) the following operations:

- Allocate and receive
- Transmit and release

Note that it is not compulsory for the marshalling function to transfer the payload via the device driver. It may, for example, only be necessary for it to translate the payload address from a local value to a cluster bus address (on those processors that have cluster bus functionality), so as to permit in-place access to the payload from another processor. When the payloads for a particular message type are always stored in a memory (for example, SDRAM) which is visible to all nodes and mapped to the same address range on each, then no marshalling is needed. The message type can be given a non-marshalled value (that is, the sign bit is zero).

Since the most common form of marshalled payload is likely to be a plain memory block allocated either from a heap or from a VDK memory pool, the VDK provides built-in standard marshalling functions to handle these cases. (See Appendix A, "Processor-Specific Notes", for details of other standard marshalling functions that may be available only for certain processor families.) The more complex cases (linked data structures, shared memory, and so on) require user-written custom marshalling functions.

Marshalling functions are called from the routing threads and are passed these arguments:

- Marshalling code – indicates which operation is to be performed
- A pointer to the formatted message packet, which includes payload type, size, and address – for input and output
- Device descriptor – identifies the VDK device driver for the connection
- Heap index or PoolID – used by standard marshalling
- I/O timeout duration (usually set to zero, for indefinite wait)

For transmission, the marshalling function is also responsible for first transmitting the message packet. This allows the marshalling function to modify the payload attributes prior to transmission if required. For example, if the payload is to be accessed in-place across the cluster bus (on processors with cluster bus functionality), then node-specific addresses must be translated to the global address space and back.

The marshalling functions execute in the context of the Routing Threads. The `SyncRead()` or `SyncWrite()` calls made by the marshalling functions will (or may) cause the threads to block awaiting I/O completion, however the original sending thread(s) are not blocked. In this way, the Routing Threads act as a buffer between user threads and the interprocessor message transfer mechanism.

Note that it is not strictly necessary for the marshalling function to actually transfer the data. In certain circumstances, it may be sufficient for it merely to perform the allocations and deallocations. An example of where this may be useful is the message loopback. The message may be returned to the sender after changing its payload type to one whose marshalling function simply frees the payload when the message is transmitted and allocates a payload when the message is received. This avoids the overhead of transferring data that is no longer of interest but allows the payload to still be automatically managed by the system.

The only added complexity is the need for two marshalled types instead of one, and for the user threads to change the payload type between the two according to whether the message is "full" or "empty". The `Empty Pool` and `Empty Heap` standard marshalling functions are provided for this purpose.

When defining a marshalled payload type in the kernel tab, the user can select either standard or custom marshalling. For standard marshalling, the choice must be made between (at least) heap or pool marshalling, according to whether the payloads are allocated from a C/C++ heap (using the VisualDSP++ multiple heap API extensions) or a VDK memory pool. The heap or PoolID must also be specified. For custom marshalling, the name of the marshalling function must be supplied, and a source module containing a skeleton of the marshalling function is automatically created. It is then the user's task to add the code that allocates and deallocates, and reads and writes, the actual payload.

Device Drivers for Messaging

Device drivers employed in message transfer must provide certain properties assumed in the design of the routing threads:

- Synchronous operation – once a write call returns, the caller knows that the data has been sent.

- Flow control – no data is lost if a Write (by the sender) is initiated before the corresponding read (by the receiver).
- Reliable delivery – all data sent (written) will be received (Read) at the other side.

As mentioned above, the contents of messages are written to and read from the device driver as message packets. These packets are 16 bytes (128 bits) in size and are always read and written by a single operation of that size. Device drivers can therefore be optimized for these transfers, as they are the most frequent case, although other sizes must still be supported.

As well as the message packets, the device driver must also transfer the message payloads which are written and read by the marshalling functions. It is the responsibility of the application designer to ensure that the marshalling functions and the device drivers operate together correctly, that is any transfer size or alignment restrictions imposed by the drivers are met by the payloads. This is also true of marshalled payload types using standard marshalling – the sizes and alignments of the heap or memory pool blocks must be acceptable to the messaging device drivers.

Where a bidirectional hardware device (such as a link port on TigerSHARC or certain SHARC processors) is managed by a single device driver instance on each of the two nodes that it connects, then it is necessary for the device driver to permit itself to be opened by both an incoming and an outgoing routing thread. A generalized multiple-open capability is not required. The ability to be simultaneously open once for reading and once for writing (sometimes known as a “split open”) is sufficient. Alternatively, for some devices it may be preferable to create two device driver instances on each node, so that the hardware appears as two unidirectional connections.

Routing Topology

Application designers must choose the routing structure for a particular application. This choice is closely linked to the organization of the target hardware.

At one extreme, the ideal situation is to have a direct connection between each node. In such a configuration, no through-routing is required: that is, each message post requires only one "hop". This can be achieved for a small numbers of nodes (between two and five). However, the number of connections quickly becomes prohibitive as the number of nodes increases.

At the other extreme, the minimum number of connections required is one incoming and one outgoing per node. This is sufficient to allow the nodes to be connected in a simple "ring" configuration. However, a message post may require many "hops" if the sender and receiver are widely separated on the ring. If the connections are bidirectional (for example, link ports) and each node has two, then a bidirectional ring – with messages circulating in both directions – is possible.

Between these two extremes many configurations are possible, including grids, cubes and hypercubes (if sufficient links are available per node). Where a host system forms part of the design and is participating in messaging, then it must also be included in the routing topology.

The design of the routing network is best begun "on paper", as a Directed Graph of bubbles (nodes) and arrowed lines (connections). Designers should consider how to assign threads to nodes so that as much message communication as possible is over direct connections. Once the topology has been established within the constraints of the available hardware, then a system can be described in terms of node IDs, device drivers and routing threads. This information can then be entered into the kernel tab of the per-node projects for the application.

Example projects are supplied with VisualDSP++ for certain EZ-KIT Lite boards that have more than one processor core (for example, the ADSP-BF561 processor), or that have connections specifically designed for connecting processors (for example, link ports on TigerSHARC and certain SHARC processors). These examples have appropriate device drivers and Routing Threads already in place (for fully-connected topologies, since the numbers of nodes will be small) and may be used as a starting point for new applications.

Events and Event Bits

Events and *event bits* are signals used to regulate thread execution based on the state of the system. An event bit is used to signal that a certain system element is in a specified state. An event is a Boolean operation performed on the state of all event bits. When the Boolean combination of event bits is such that the event evaluates to `TRUE`, all threads that are pending on the event are moved to the ready queue and the event remains `TRUE`. Any thread that pends on an event that evaluates as `TRUE` does not block, but when event bits have changed causing the event to evaluate as `FALSE`, any thread that pends on that event blocks.

The number of events and event bits is limited to a processor's word size minus one. Therefore, on Blackfin, SHARC and TigerSHARC processors there can be 31 events and event bits.

Behavior of Events

Each event maintains the `VDK_EventData` data structure that encapsulates all the information used to calculate an event's value:

```
typedef struct
{
    bool          matchAll;
    VDK_Bitfield  values;
    VDK_Bitfield  mask;
} VDK_EventData;
```

When setting up an event, configure a flag describing how to treat a mask and target value:

- `matchAll: TRUE` when an event must have an exact match on all of the masked bits. `FALSE` if a match on any of the masked bits results in the event recalculating to `TRUE`.
- `values:` The target values for the event bits masked with the `mask` field of the `VDK_EventData` structure.
- `mask:` The event bits that the event calculation is based on.

Unlike semaphores, events are `TRUE` whenever their conditions are `TRUE`, and all threads pending on the event are moved to the ready queue. If a thread pends on an event that is already `TRUE`, the thread continues to run, and the scheduler is not called. Like a semaphore, a thread pending on an event that is not `TRUE` blocks until the event becomes true, or the thread's timeout is reached. Pending with a timeout of zero on an event pends without timeout.

Global State of Event Bits

The state of all the event bits is stored in a global variable. When a user sets or clears an event bit, the corresponding bit number in the global word is changed. If toggling the event bit affects any events, that event is recalculated. This happens either during the call to SetEventBit() or ClearEventBit() (if called within a scheduled region), or the next time the scheduler is enabled (with a call to PopUnscheduledRegion()).

Event Calculation

To understand how events use event bits, see the following examples.

Example 1: Calculation for an *All* Event

4	3	2	1	0		**event bit number**
0	1	0	1	0	<—	bit value
0	1	1	0	1	<—	mask
0	1	1	0	0	<—	target value

Event is `FALSE` because the global event bit 2 is not the target value.

Example 2: Calculation for an *All* Event

4	3	2	1	0		**event bit number**
0	1	1	1	0	<—	bit value
0	1	1	0	1	<—	mask
0	1	1	0	0	<—	target value

Event is `TRUE`.

Example 3: Calculation for an *Any* Event

4	3	2	1	0		**event bit number**
0	1	0	1	0	<—	bit value
0	1	1	0	1	<—	mask
0	1	1	0	0	<—	target value

Event is `TRUE` since bits 0 and 3 of the target and global match.

Example 4: Calculation for an *Any* Event

4	3	2	1	0		event bit number
0	1	0	1	1	<—	bit value
0	1	1	0	1	<—	mask
0	0	1	0	0	<—	target value

Event is `FALSE` since bits 0, 2, and 3 do not match.

Effect of Unscheduled Regions on Event Calculation

Each time an event bit is set or cleared, the scheduler is entered to recalculate all dependent event values. By entering an unscheduled region, you can toggle multiple event bits without triggering spurious event calculations that could result in erroneous system conditions. Consider the following code.

```
/* Code that accidentally triggers Event1 trying to set up
   Event2. Assume the prior event bit state = 0x00. */

VDK_EventData data1 = { true, 0x1, 0x3 };
VDK_EventData data2 = { true, 0x3, 0x3 };
VDK_LoadEvent(kEvent1, data1);
VDK_LoadEvent(kEvent2, data2);
VDK_SetEventBit(kEventBit1); /* will trigger Event1 by accident */
VDK_SetEventBit(kEventBit2); /* Event1 is FALSE, Event2 is TRUE */
```

Whenever you toggle multiple event bits, enter an unscheduled region to avoid the above loopholes. For example, to fix the accidental triggering of `Event1` in the above code, use the following code:

```
VDK_PushUnscheduledRegion();
VDK_SetEventBit(kEventBit1);  /* Event1 has not been triggered */
VDK_SetEventBit(kEventBit2);  /* Event1 is FALSE, Event2 is TRUE */
VDK_PopUnscheduledRegion();
```

Thread's Interaction With Events

Threads interact with events by pending on events, setting or clearing event bits, and by loading a new `VDK_EventData` into a given event.

Pending on an Event

Like semaphores, threads can pend on an event's condition becoming `TRUE` with a timeout. Figure 3-12 illustrates the process of pending on an event.

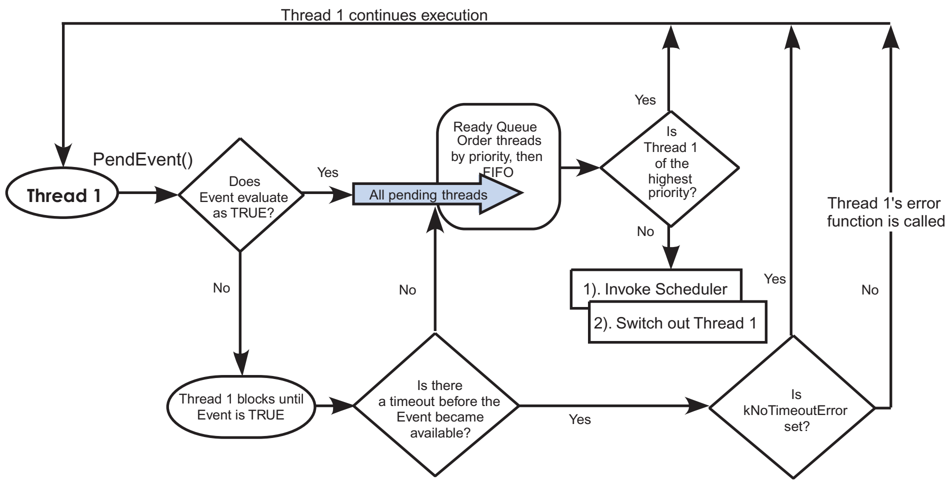


Figure 3-12. Pending on an Event

A thread calls PendEvent() and specifies the timeout. If the event becomes `TRUE` before the timeout is reached, the thread (and all other threads pending on the event) is moved to the ready queue. Calling PendEvent() with a timeout of zero means that the thread is willing to wait indefinitely.

Setting or Clearing of Event Bits

Changing the status of the event bits can be accomplished in both the interrupt domain and the thread domain. Each domain results in slightly different results.

From the Thread Domain:
Figure 3-13 illustrates the process of setting or clearing an event bit from the thread domain.

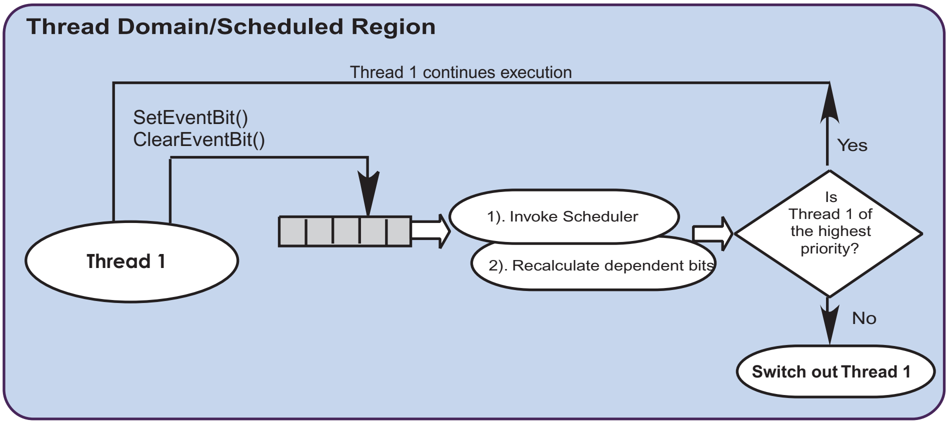


Figure 3-13. Thread Domain: Setting or Clearing an Event Bit

A thread can set an event bit by calling SetEventBit() and clear it by calling ClearEventBit(). Calling either from within a scheduled region recalculates all events that depend on the event bit and can result in a higher priority thread being context switched in.

From the Interrupt Domain:
Figure 3-14 illustrates the process of setting or clearing of an event bit from the interrupt domain.

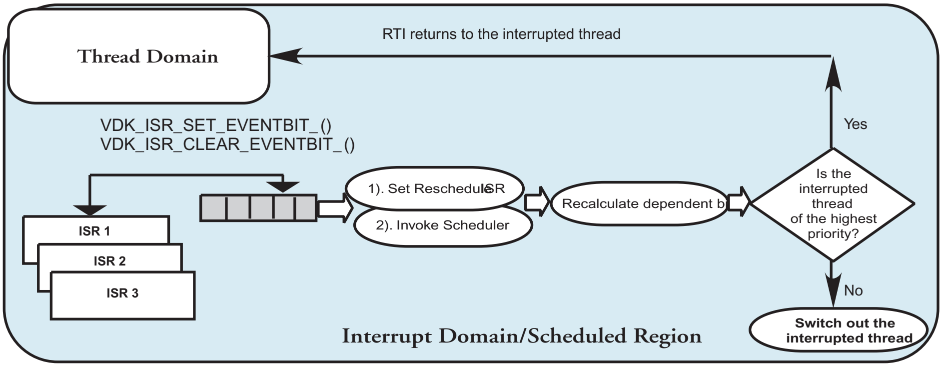


Figure 3-14. Interrupt Domain: Setting or Clearing an Event Bit

An ISR can call VDK_ISR_SET_EVENTBIT_() and VDK_ISR_CLEAR_EVENTBIT_() to change an event bit value and, possibly, free a new thread to run. Calling these macros *does not* result in a recalculation of the events; however, the low priority software interrupt is set and the scheduler entered. If the interrupted thread is in a scheduled region, an event recalculation takes place and can cause a higher priority thread to be context switched in. If an ISR sets or clears multiple event bits, the calls do not need to be protected with an unscheduled region (since there is no thread scheduling in the interrupt domain). For example,

```
/* The following two ISR calls do not need to be protected: */
VDK_ISR_SET_EVENTBIT_(kEventBit1);
VDK_ISR_SET_EVENTBIT_(kEventBit2);
```

Loading New Event Data into an Event

From the thread scheduling domain, a thread can get the VDK_EventData associated with an event with the GetEventData() API. Additionally, a thread can change the VDK_EventData with the LoadEvent() API. A call to LoadEvent() causes a recalculation of the event's value. If a higher priority thread becomes ready because of the call, it starts running if the scheduler is enabled.

Device Flags

Because of the special nature of device drivers, most require synchronization methods that are similar to those provided by events and semaphores, but with different operation. *Device flags* are created to satisfy the specific circumstances device drivers might require. Much of their behavior cannot be fully explained without an introduction to device drivers, which are covered extensively in "Device Drivers" on page 3-53.

Behavior of Device Flags

Like events and semaphores, a thread can pend on a device flag, but unlike semaphores and events, a device flag is always FALSE. A thread pending on a device flag immediately blocks. When a device flag is posted, all threads pending on it are moved to the ready queue.

Device flags are used to communicate to any number of threads that a device has entered a particular state. For example, assume that multiple threads are waiting for a new data buffer to become available from an A/D converter device. While neither a semaphore nor an event can correctly represent this state, a device flag's behavior can encapsulate this system state.

Thread's Interaction With Device Flags

A thread accesses a device flag through two APIs: PendDeviceFlag() and PostDeviceFlag(). Unlike most APIs that can cause a thread to block, PendDeviceFlag() *must* be called from within a critical region.

PendDeviceFlag() is set up this way because of the nature of device drivers. See "Device Drivers" on page 3-53 for a more information about device flags and device drivers.

Interrupt Service Routines

In VisualDSP++4.0 VDK ISRs can be written assembly, C or C++. All previous releases of VDK restricted users to using assembly for writing ISRs. Whilst the new support provides increased flexibility it should be noted that there are performance considerations and certain restrictions associated with writing your ISRs in C/C++, as described below.

The original VDK philosophy related to the writing of ISRs still holds true. ISRs should be short routines that perform essential tasks and then post semaphores, change event bit values, activate device drivers, and so on, in order to switch execution to the relevant thread or device driver. The bulk of any associated calculations should be performed at thread level. This approach reduces the number of context saves/restores required, decreases interrupt latency, and still keeps as much code as possible in a high-level language.

Enabling and Disabling Interrupts

Each processor architecture has a slightly different mechanism for masking and unmasking interrupts. Some architectures require that the state of the interrupt mask be saved to memory before servicing an interrupt or an exception, and the mask be manually restored before returning. Since the kernel installs interrupts (and exception handlers on some architectures), directly writing to the interrupt mask register may produce unintended results. Therefore, the VDK provides a simple and platform-independent API to simplify access to the interrupt mask.

A call to GetInterruptMask() returns the actual value of the interrupt mask, even if it has been saved temporarily by the kernel in private storage. Likewise, SetInterruptMaskBits() and ClearInterruptMaskBits() set and clear bits in the interrupt mask in a robust and safe manner. Interrupt levels with their corresponding bits set in the interrupt mask are enabled when interrupts are globally enabled. See the *Hardware Reference* manual for your target processor for more information about the interrupt mask.

VDK also presents a standard way of turning interrupts on and off globally. Like unscheduled regions (in which the scheduler is disabled), the VDK supports critical regions where interrupts are disabled. A call to PushCriticalRegion() disables interrupts, and a call to PopCriticalRegion() re-enables interrupts. These API calls implement a stack-style interface, as described in "Protected Regions" on page 1-8. Users are discouraged from turning interrupts off for long sections of code since this increases interrupt latency.

Interrupt Architecture

VDK ISRs can be written in assembly or C/C++. The following sections explain the advantages and disadvantages of each approach.

Assembly Interrupts

ISRs written in assembly are the most efficient way of servicing interrupts. The overhead of saving and restoring processor state is eliminated, along with need to set up a C run-time environment. ISRs written in assembly must save and restore only the registers that they use. The lightweight nature of assembly ISRs also encourages the use of interrupt nesting to further reduce latency. (VDK enables interrupt nesting by default on processors that support it.)

C/C++ Interrupts

ISRs written in C/C++ may simplify the coding of the routines, but there are inherent overheads with implementing ISRs in a high level language. A C/C++ run-time must be established on entry to the ISR, incurring a delay before any actual ISR code is executed. Also, the necessary processor state must be saved on entry to the ISR and restored on exit. If a C/C++ ISR calls any functions that are not present in the same module then the entire processor state will be saved and restored, unless the `regs_clobbered` pragma is used to specify the registers modified by the function. (Refer to your processor's *C/C++ Compiler and Library Manual* for details.) An additional point of note is that the majority of the run-time library is not interrupt-safe, and so can not be used in ISRs. (Refer to your processor's *C/C++ Compiler and Library Manual* for details.) Thus, there are certain limits imposed as to what can be done in a C/C++ ISR.

Vector Table

The method VDK uses to install interrupt handlers depends on the processor family used, and the underlying run-time library support. Refer to the generated skeleton ISR code for further details on any restrictions, requirements or options associated with adding your own code to ISRs.

By default VDK reserves (at least) two interrupts: the timer interrupt and the lowest priority software interrupt. For a discussion about the timer interrupt, see "Timer ISR" on page 3-52. For information about the lowest priority software interrupt, see "Reschedule ISR" on page 3-52. For information on any additional interrupts reserved by the VDK for particular processors, see Appendix A, "Processor-Specific Notes".

Global Data

Often ISRs need to communicate data back and forth to the thread domain besides semaphores, event bits, and device driver activations. ISRs can use global variables to get data to the thread domain, but you must remember to wrap any access to or from that global data in a critical region and to declare the variable as `volatile` (in C/C++). For example, consider the following:

```
/* MY_ISR.asm */
.EXTERN _my_global_integer;
<REG> = data;
DM(_my_global_integer) = <REG>;
/* finish up the ISR, enable interrupts, and RTI. */
```

And in the thread domain:

```
/* My_C_Thread.c */
volatile int my_global_integer;

/* Access the global ISR data */
VDK_PushCriticalRegion();
if (my_global_integer == 2)
   my_global_integer = 3;
VDK_PopCriticalRegion();
```

Communication With the Thread Domain

VDK supplies a set of assembly macros and APIs callable from C/C++ ISRs that can be used to communicate system state to the thread domain. (See "Assembly Macros and C/C++ ISR APIs" on page 5-161 for further details.)

The assembly macros are called from the interrupt domain, and so they make no assumptions about processor state, available registers, or parameters. In other words, the assembly macros can be called without consideration of saving state or having processor state trampled during a call.

Take for example, the following three equivalent VDK_ISR_POST_SEMAPHORE_() calls:

```
.VAR/DATA  semaphore_id;

/* Pass the value directly */
VDK_ISR_POST_SEMAPHORE_(kSemaphore1);

/* Pass the value in a register */
<REG> = kSemaphore1;
VDK_ISR_POST_SEMAPHORE_(<REG>);
/* <REG> was not trampled */

/* Post the semaphore one last time using a DM */
DM(semaphore_id) = <REG>;
VDK_ISR_POST_SEMAPHORE_(DM(semaphore_id));
```

Additionally, no condition codes are affected by the assembly macros, no assumptions are made about having space on any hardware stacks (for example, PC or status), and all VDK internal data structures are maintained.

The C/C++ ISR APIs provide equivalent functionality for use in ISRs written in C/C++.

The assembly macros and APIs callable from C/C++ ISRs raise the low priority software interrupt if thread domain scheduling is required after all other interrupts are serviced. For a discussion of the low priority software interrupt, see "Reschedule ISR" on page 3-52. Refer to Appendix A, "Processor-Specific Notes", for additional information about ISR APIs.

Within the interrupt domain, every effort should be made to enable interrupt nesting. Nesting may be disabled when an ISR begins. However, leaving it disabled is analogous to staying in an unscheduled region in the thread domain; other ISRs are prevented from executing, even if they have higher priority. Allowing nested interrupts potentially lowers interrupt latency for high-priority interrupts.

Timer ISR

By default VDK reserves a timer interrupt. The timer is used to calculate round-robin times, sleeping thread time to keep sleeping, and periodic semaphores. One VDK tick is defined as the time between timer interrupts. It is the finest resolution measure of time in the kernel. The timer interrupt can cause a low priority software interrupt (see “Reschedule ISR” on page 3-52). In VisualDSP++ 4.0, it is possible to change the interrupt used for the VDK timer interrupt from the default. (See online **Help** for further information.) Additionally, it is possible to specify "None" for the VDK timer interrupt if VDK timing services are not required.

Reschedule ISR

VDK designates the lowest priority interrupt that is not tied to a hardware device as the reschedule ISR. This ISR handles housekeeping when an interrupt causes a system state change that can result in a new high priority thread becoming ready. If a new thread is ready and the system is in a scheduled region, the software ISR saves off the context of the current thread and switches to the new thread. If an interrupt has activated a device driver, the low priority software interrupt calls the dispatch function for the device driver. For more information, see “Dispatch Function” on page 3-59.

On systems where the lowest priority non-hardware-tied interrupt is not the lowest priority interrupt, all lower priority interrupts must run with interrupts turned off for their entire duration. Failure to do so may result in undefined behavior.

I/O Interface

The I/O interface provides the mechanism for creating an interface between the external environment and VDK applications. In VisualDSP++ 4.0, only device driver objects can be used to construct the I/O interface.

I/O Templates

I/O templates are analogous to thread types. I/O templates are used to instantiate I/O objects. In VisualDSP++ 4.0, the only types of I/O templates available, and therefore the only classes of I/O objects, are for device drivers. In order to create an instance of a device driver, a boot I/O object must be added to the VDK project using the device driver template. You can distinguish between different instances of the same device driver. For more information, see “Init” on page 3-62.

Device Drivers

The role of a device driver is to abstract the details of the hardware implementation from the software designer. For example, a software engineer designing a Finite Impulse Response (FIR) filter does not need to understand the intricacies of the converters, and is able to concentrate on the FIR algorithm. The software can then be reused on different platforms, where the hardware interface differs.

The Communication Manager controls device drivers in the VDK. Using the Communication Manager APIs, you can maintain the abstraction layers between device drivers, interrupt service routines, and executing threads. This section details how the Communication Manager is organized.

In VisualDSP++ 4.0, device drivers are a part of the I/O interface. Device drivers are added to a VDK project as I/O objects. VisualDSP++ 2.0 device drivers are not compatible with

VisualDSP++ 4.0 device drivers. See Appendix B,"Migrating Device Drivers" for a description of how to convert existing VisualDSP++ 2.0 device drivers for use in VisualDSP++ 4.0 projects.

Execution

Device drivers and interrupt service routines are tied very closely together. Typically, DSP developers prefer to keep as much time critical code in assembly as possible. The Communication Manager is designed such that you can keep interrupt routines in assembly (the time critical pieces), and interface and resource management for the device in a high-level language without sacrificing speed. The Communication Manager attempts to keep the number of context switches to a minimum, to execute management code at reasonable times, and to preserve the order of priorities of running threads when a thread uses a device. However, you need to thoroughly understand the architecture of the Communication Manager to write your device driver.

There is only one interface to a device driver—through a dispatch function. The dispatch function is called when the device is initialized, when a thread uses a device (open/close, read/write, control), or when an interrupt service routine transfers data to or from the device. The dispatch function handles the request and returns. Device drivers should *not* block (pend) when servicing an initialize request or a request for more data by an interrupt service routine. However, a device driver can block when servicing a thread request and the relevant resource is not ready or available. Device driver initialization and ISR requests are handled within critical regions enforced by the kernel, so their execution does not have to be reentrant. A thread-level request must protect global variables within critical or unscheduled regions.

Parallel Scheduling Domains

This section focuses on a unique role of device drivers in the VDK architecture. Understanding device drivers requires some understanding of the time and method by which device driver code is invoked. VDK applications may be factored into two *domains*, referred to as the thread domain and the ISR domain (see Figure 3-15). This distinction is not an arbitrary or unnecessary abstraction. The hardware architecture of the processor as well as the software architecture of the kernel reinforces this notion. You should consider this distinction when you are designing your application and apportioning your code.

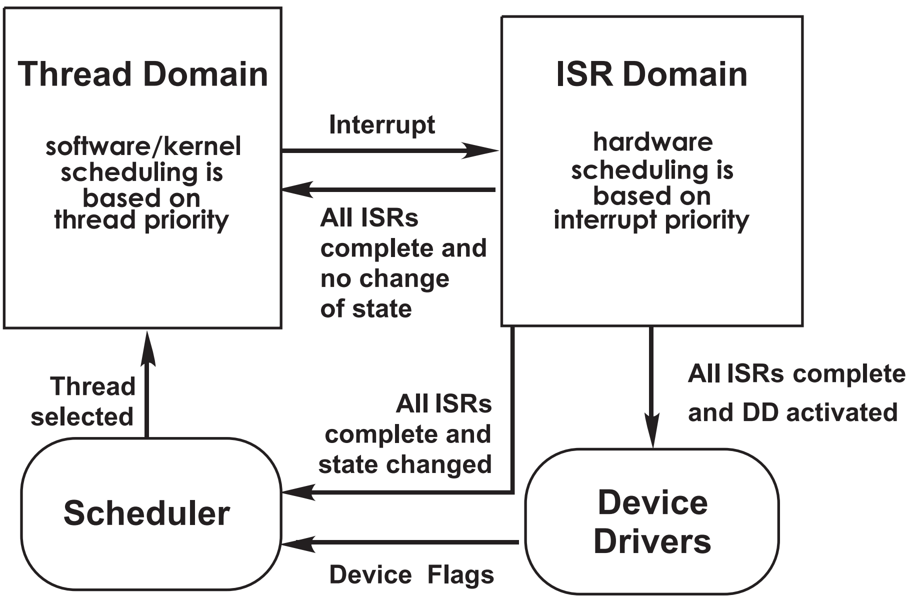


Figure 3-15. Parallel Scheduling Domains

Threads are scheduled based on their priority and the order in which they are placed in the ready queue. The scheduling portion of the kernel is responsible for selecting the thread to run. However, the scheduler does not have complete control over the processor. It may be preempted by a parallel and higher priority scheduler—the interrupt and exception hardware. While interrupts or exceptions are being serviced, thread priorities are temporarily moot. The position of threads in the ready queue becomes significant again only when the hardware relinquishes control back to the software-based scheduler.

Each of the domains has strengths and weaknesses that dictate the type of code suitable to be executed in that environment. The scheduler in the thread domain is invoked when threads are moved to or from the ready queue. Threads each have their own stack and may be written in a high-level language. Threads always execute in "supervisor" or "kernel mode" (if the processor makes this distinction). Threads implement algorithms and are allotted processor time based on the completion of higher priority activity.

In contrast, scheduling in the interrupt domain has the highest system wide priority. Any "ready" ISR takes precedence over any ready thread (outside critical regions), and this form of scheduling is implemented in hardware. ISRs are always written in assembly and must manually restore any registers they use. ISRs execute in "supervisor" or "kernel mode" (if the processor makes this distinction). ISRs respond to asynchronous peripherals at the lowest level only. The routine should perform only such activities that are so time critical that data would be lost if the code is not executed as soon as possible. All other activities should occur under the control of the kernel's scheduler based on priority.

Transferring from the thread domain to the interrupt domain is simple and automatic, but returning to the thread domain can be much more laborious. If the ready queue is not changed while in the interrupt domain, then the scheduler need not run when it regains control of the system. The interrupted thread resumes execution immediately. If the ready queue has changed, the scheduler must further determine whether the highest priority thread has changed. If it has changed, the scheduler must initiate a context switch.

Device drivers fill the gap between the two scheduling domains. They are neither thread code nor ISR code, and they are not directly scheduled by either the kernel or the interrupt controller. Device drivers are implemented as C++ objects and run on the stack of the currently running thread. However, they are not "owned" by any thread, and may be used by many threads concurrently.

Using Device Drivers

From the point of view of a thread, there are five functional interfaces to device drivers: OpenDevice(), CloseDevice(), SyncRead(), SyncWrite(), and DeviceIOCtl(). The names of the APIs are self-explanatory since threads mostly treat device drivers as black boxes. Figure 3-16 illustrates the device drivers' interface.

A thread uses a device by opening it, reading and/or writing to it, and closing it. The DeviceIOCtl() function is used for sending device-specific control information messages. Each API is a standard C/C++ function call that runs on the stack of the calling thread and returns when the function completes. However, when the device driver does not have a needed resource, one of these functions may cause the thread to be removed from the ready queue and block on a signal, similar to a semaphore or an event, called a "device flag."

Interrupt service routines have only one API call relating to device drivers: VDK_ISR_ACTIVATE_DEVICE_(). This macro is not a function call, and program flow does not transfer from the ISR to the device driver and back. Rather, the macro sets a flag indicating that the device driver's "activate" routine should execute after all interrupts have been serviced.

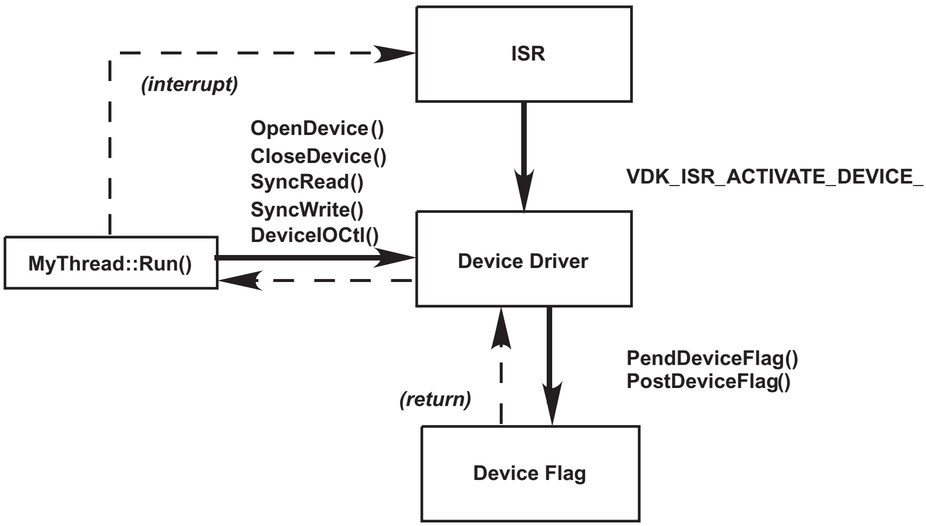


Figure 3-16. Device Driver APIs

The remaining two API functions, PendDeviceFlag() and PostDeviceFlag(), are typically called from within the device driver itself. For example, a call from a thread to SyncRead() might cause the device driver to call PendDeviceFlag() if there is no data currently available. This would cause the thread to block until the device flag is posted by another code fragment within the device driver that is providing the data.

As another example, when an interrupt occurs because an incoming data buffer is full, the ISR might move a pointer so that the device begins filling an empty buffer before calling VDK_ISR_ACTIVATE_DEVICE_(). The device driver's activate routine may respond by posting a device flag and moving a thread to the ready queue so that it can be scheduled to process the new data.

Dispatch Function

The dispatch function is the core of any device driver. This function takes two parameters and returns a `void*` (the return value depends on the input values). A dispatch function declaration for a device driver is as follows:

C Driver Code:

```
void* MyDevice_DispatchFunction(VDK_DispatchID     inCode,
                                VDK_DispatchUnion  inData);
```

C++ Driver Code:

```
void* MyDevice::DispatchFunction(VDK::DispatchID     inCode,
                                 VDK::DispatchUnion  &inData);
```

The first parameter is an enumeration that specifies why the dispatch function has been called:

```
enum VDK_DispatchID
{
   VDK_kIO_Init,
   VDK_kIO_Activate,
   VDK_kIO_Open,
   VDK_kIO_Close,
   VDK_kIO_SyncRead,
   VDK_kIO_SyncWrite,
   VDK_kIO_IOCtl
};
```

The second parameter is a union whose value depends on the enumeration value:

```
union VDK_DispatchUnion
{
  struct OpenClose_t
  {
     void         **dataH;
     char         *flags;  /* used for kIO_Open only */
  };
```

```
    struct ReadWrite_t
    {
       void         **dataH;
       VDK_Ticks    timeout;
       unsigned int dataSize;
       int          *data;
    };
    struct IOCtl_t
    {
       void         **dataH;
       void          *command;
       char         *parameters;
    };
    struct Init_t
    {
       void         *pInitInfo;
    };
  };
```

The values in the union are only valid when the enumeration specifies that the dispatch function has been called from the thread domain (`kIO_Open`, `kIO_Close`, `kIO_SyncRead`, `kIO_SyncWrite`, `kIO_IOCtl`).

A device driver's dispatch function can be structured as follows:

In C:

```
void* MyDevice_DispatchFunction(VDK_DispatchID    inCode,
                                VDK_DispatchUnion inData)
{
  switch(inCode)
  {
    case VDK_kIO_Init:
         /* Init the device */
    case VDK_kIO_Activate:
         /* Get more data ready for the ISR */
    case VDK_kIO_Open:
         /* A thread wants to open the device... */
         /* Allocate memory and prepare everything else */
    case VDK_kIO_Close:
```

```
        /* A thread is closing a connection to the device...*/
        /* Free all the memory, and do anything else */
    case VDK_kIO_SyncRead:
        /* A thread is reading from the device */
        /* Return an unsigned int of the num. of bytes read */
    case VDK_kIO_SyncWrite:
        /* A thread is writing to the device */
        /* Return an unsigned int of the number of bytes */
        /* written */
    case VDK_kIO_IOCtl:
        /* A thread is performing device-specific actions: */
    default:
        /* Invalid DispatchID code */
    return 0;
  }
}
```

In C++:

```
void* MyDevice::DispatchFunction(VDK::DispatchID    inCode,
                                 VDK::DispatchUnion &inData)
{
  switch(inCode)
  {
    case VDK::kIO_Init:
        /* Init the device */
    case VDK::kIO_Activate:
        /* Get more data ready for the ISR */
    case VDK::kIO_Open:
        /* A thread wants to open the device... */
        /* Allocate memory and prepare everything else */
    case VDK::kIO_Close:
        /* A thread is closing a connection to the device...*/
        /* Free all the memory, and do anything else */
    case VDK::kIO_SyncRead:
        /* A thread is reading from the device */
        /* Return an unsigned int of the num. of bytes read */
    case VDK::kIO_SyncWrite:
        /* A thread is writing to the device */
        /* Return an unsigned int of the number of bytes */
```

```
        /* written */
    case VDK::kIO_IOCtl:
        /* A thread is performing device-specific actions: */
    default:
        /* Invalid DispatchID code */
    return 0;
  }
}
```

Each of the different cases in the dispatch function are discussed below.

Init

The device dispatch function is called with the `VDK_kIO_Init` parameter for C-style device and `VDK::kIO_Init` for C++-style drivers at system boot time. All device-specific data structures and system resources should be set up at this time. The device driver `init` function is called within a critical region, and so *should not* call any APIs that throw an error or might block. As all device driver `init` functions are executed before any threads have been run, they can be used to run any required initialization code. This is applicable even if the device driver type is a stub whose only purpose is to contain the initialization code.

A union is passed to the device dispatch function whose value is defined with the `Init_t` of the `VDK_DispatchUnion`. The `Init_t` is defined as follows.

```
struct Init_t
{
  void *pInitInfo;
}
```

`Init_t.pInitInfo`: A pointer (type void*) to the value defined in the "Initializer" field of the kernel tab for Boot I/O Objects. (See online Help for further details.) Where more than one instance of a particular device driver is to be used this field can be used to distinguish between them by specifying a unique instance number for each device (i.e. each Boot I/O Object).

Open or Close

When a thread opens or closes a device with OpenDevice() or CloseDevice(), the device dispatch function is called with `VDK_kIO_Open` or `VDK_kIO_Close`. The dispatch function is called from the thread domain, so any stack-based variables are local to that thread. Access to shared data (data that may be accessed by threads and/or interrupts and/or device driver activate functions) should be appropriately protected by the use of unscheduled regions, critical regions, or other means.

When a thread calls the dispatch function attempting to open or close a device, the API passes a union to the device dispatch function whose value is defined with the `OpenClose_t` of the `VDK_DispatchUnion`. The `OpenClose_t` is defined as follows:

```
struct OpenClose_t
{
  void    **dataH;
  char    *flags;   /* used for kIO_Open only */
};
```

`OpenClose_t.dataH`: A pointer to a thread-specific location that a device driver can use to hold any thread-specific resources. For example, a thread can `malloc` space for a structure that describes the state of a thread associated with a device. The pointer to the structure can be stored in `*dataH`, which is then accessible to every other dispatch call involving this thread. A device driver can free the space when the thread calls CloseDevice().

`OpenClose_t.flags`: The second parameter passed to an OpenDevice() call is supplied to the dispatch function as the value of `OpenClose_t.flags`. This is used to pass any device-specific flags relevant to the opening of a device. Note that this part of the union is not used on a call to CloseDevice().

Read or Write

A thread that needs to read or write to a device it has opened calls SyncRead() or SyncWrite(). The dispatch function is called in the thread domain and on the thread's stack. These functions call the device dispatch function with the parameters passed to the API in the `VDK_DispatchUnion`, and the flags `VDK_kIO_SyncRead` or `VDK_kIO_SyncWrite`. The `ReadWrite_t` is defined as follows:

```
struct ReadWrite_t
{
  void          **dataH;
  VDK::Ticks    timeout;
  unsigned int  dataSize;
  int           *data;
};
```

`ReadWrite_t.dataH`: A thread-specific location, which is passed to the dispatch function on the opening of a device by an OpenDevice() call. This variable can be used to store a pointer to a thread-specific data structure detailing what state the thread is in while dealing with the device.

`ReadWrite_t.timeout`: The amount of time in Ticks that a thread is willing to wait for the completion of a SyncRead() or SyncWrite() call. If this timeout behavior is required, it must be implemented by using the value of `ReadWrite_t.timeout` as an argument to an appropriate PendDeviceFlag() call in the dispatch function.

`ReadWrite_t.dataSize:` The amount of data that the thread reads from or writes to the device.

`ReadWrite_t.data`: A pointer to the location that the thread writes the data to (on a read), or reads from (on a write).

Like calls to the device dispatch function for opening and closing, the calls to read and write are not protected with a critical or unscheduled region. If a device driver accesses global data structures during a read or write, the access should be protected with critical or unscheduled regions. See the discussion in "Device Drivers" on page 3-53 for more information about regions and pending.

IOCtl

VDK supplies an interface for threads to control a device's parameters with the DeviceIOCtl() API. When a thread calls DeviceIOCtl(), the function sets up some parameters and calls the specified device's dispatch function with the value `VDK_kIO_IOCtl` and the `VDK_DispatchUnion` set up as a `IOCtl_t`.

The `IOCtl_t` is defined as follows:

```
struct IOCtl_t
{
  void        **dataH;
  void         *command;
  char        *parameters;
};
```

`IOCtl_t.dataH`: A thread-specific location, which is passed to the dispatch function on the opening of a device by an OpenDevice() call. This variable can be used to store a pointer to a thread-specific data structure detailing what state the thread is in while dealing with the device.

`IOCtl_t.command`: A device-specific pointer (second parameter from the DeviceIOCtl() function).

`IOCtl_t.parameters`: A device-specific pointer (third parameter from the DeviceIOCtl() function).

Like read/write and open/close, a device dispatch function call for `IOCtl` is not protected by a critical or unscheduled region. If a device accesses global data structures, the device driver should protect them with a critical or an unscheduled region.

Activate

Often a device driver needs to respond to state changes caused by ISRs. The device dispatch function is called with a value `VDK_kIO_Activate` at some point after an ISR has called the macro VDK_ISR_ACTIVATE_DEVICE_().

When the ISR calls VDK_ISR_ACTIVATE_DEVICE_(), a flag is set indicating that a device has been activated, and the low priority software interrupt is triggered to run (see “Reschedule ISR” on page 3-52). When the scheduler is entered through the low priority software interrupt, the device's dispatch function is called with the `VDK_kIO_Activate` value.

The activate part of a device dispatch function should handle posting signals, so that threads waiting on certain device states can continue running. For example, assume that a D/A ISR runs out of data in its buffer. The ISR calls VDK_ISR_ACTIVATE_DEVICE_() with the `IOID` of the device driver. When the device dispatch function is called with the `VDK_kIO_Activate`, the device posts a device flag or semaphore that reschedules any threads that are pending.

The PostDeviceFlag(), PostSemaphore(), PushCriticalRegion(), and PopCriticalRegion() APIs are the only VDK APIs that are safe to call from the activate function.

Device Flags

Device flags are synchronization primitives, similar to semaphores, events, and messages, but device flags have a special association with device drivers. Like semaphores and events, a thread can pend on a device flag. This means that the thread waits until the flag is posted by a device driver. The post typically occurs from the activate function of a device driver's dispatch function.

Pending on a Device Flag

When a thread pends on a device flag (unlike with semaphores, events, and messages), the thread always blocks. The thread waits until the flag is posted by another call to the device's dispatch function. When the flag is posted, all threads that are pending on the device flag are moved to the ready queue. Since posting a device flag with the PostDeviceFlag() API moves an indeterminate number of threads to the ready queue, the call is not deterministic. For more information about posting device flags, see "Posting a Device Flag" on page 3-68.

The rules for pending on device flags are strict compared to other types of signals. The "stack" of critical regions must be exactly one level deep when a thread pends on a device flag. In other words, with interrupts enabled, call PushCriticalRegion() exactly once prior to calling PendDeviceFlag() from a thread. The reason for this condition becomes clear if you consider the reason for pending. A thread pends on a device flag when it is waiting for a condition to be set from an ISR. However, you must enter a critical region before examining any condition that may be modified from an ISR to ensure that the value you read is valid. Furthermore, PendDeviceFlag() pops the critical region stack once, effectively balancing the earlier call to PushCriticalRegion().

For example, a typical device driver uses device flags in the following manner.

```
VDK_PushCriticalRegion();
while(should_loop != 0)
{
    /* ... */
    /* access global data structures */
    /* and figure out if we should keep looping */
    /* ... */
    /* Wait for some device state */
    VDK_PendDeviceFlag();
    /* Must reenter the critical region */
    VDK_PushCriticalRegion();
```

```
}
VDK_PopCriticalRegion();
```

Figure 3-17 illustrates the process of pending on a device flag.

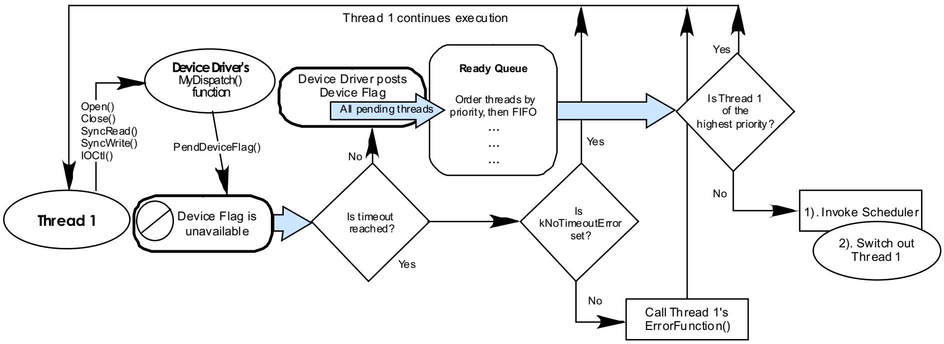


Figure 3-17. Pending on a Device Flag

Posting a Device Flag

Like semaphores and messages, a device flag can be posted. A device dispatch function posts a device flag with a call to PostDeviceFlag(). Unlike semaphores and messages, the call moves *all* threads pending on the device flag to the ready queue and continues execution. Once PostDeviceFlag() returns, subsequent calls to PendDeviceFlag() cause the thread to block (as before).

Note that the PostDeviceFlag() API does not throw any errors. The reason being is this API function is called typically from the dispatch function when the dispatch function is called with `VDK_kIO_Activate`. This happens because the device dispatch function operates on the kernel's stack when it is called with `VDK_kIO_Activate` rather than on the stack of a thread.

Figure 3-18 illustrates the process of posting a device flag.

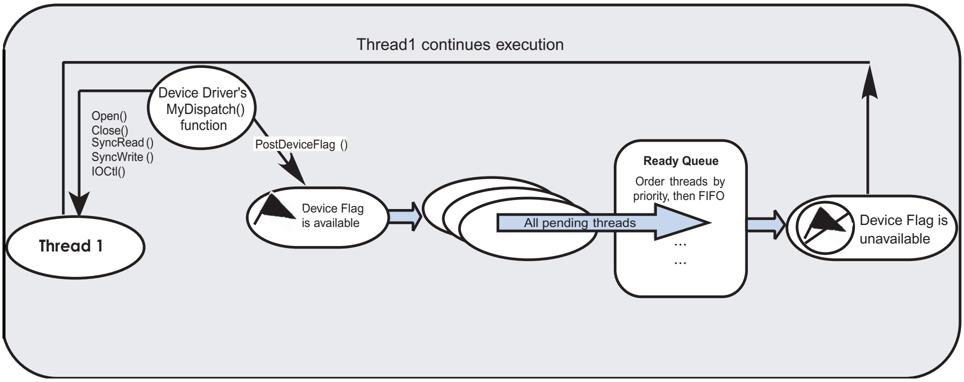


Figure 3-18. Posting a Device Flag

General Notes

Keep the following tips in mind while writing device drivers. Although many of these topics also apply to threads, they deserve special mention with respect to device drivers.

Variables

Device drivers and ISRs are closely linked. Since ISRs and the dispatch function access the same variables, declare the variables in the C/C++ device driver routine and access them as `extern` from within the assembly ISR. When declaring these variables in the C/C++ source file, you must declare them as `volatile` to ensure that the compiler optimizer is aware that their values may be changed externally to the C/C++ code at any time. Additionally, care must be taken in the ISR to refer to variables defined in C/C++ code correctly, by their decorated/mangled names.

Critical/Unscheduled Regions

Since many of the data structures and variables associated with a device driver are shared between multiple threads and ISRs, access to them must be protected within critical regions and unscheduled regions. Critical regions keep ISRs from modifying data structures unexpectedly, and unscheduled regions prevent other threads from modifying data structures.

When pending on device flags, care must be taken to remain in the correct regions. Device flags must be pended on from within a non-nested critical region, as discussed in "Pending on a Device Flag" on page 3-67.

Memory Pools

Common problems experienced with memory allocation using `malloc` are fragmentation of the heap as well as non-deterministic search times for finding a free area of the heap with the requested size. The memory pool manager uses the defined pools to provide an efficient, deterministic memory allocation scheme as an alternative to `malloc`. The use of memory pools for memory allocation can be advantageous when an application requires significant allocation and deallocation of objects of the same size.

A memory pool is an area of memory subdivided into equally-sized memory blocks. Each memory pool contains memory blocks of a single size, but multiple pools can be defined, each with a different block size. Furthermore, on architectures that support the definition of multiple heaps, the heap that is used by a pool can be specified. The maximum number of active memory pools in the system is set up when the project is built.

Memory Pool Functionality

Memory pools can be created either at boot time, or dynamically at runtime using the CreatePool() or CreatePoolEx() APIs. When creating a memory pool, the block size and number of blocks in the pool are specified. The memory pool manager allocates the memory required for the pool and splits it into blocks at creation time, if required. the VDK allows blocks to be created on demand at run time (during a call to MallocBlock()), rather than during the creation of a pool. On demand, creation of blocks reduces the overhead at the time the pool is created but increases the run-time overhead when obtaining a new block from the pool. Additionally, allocation and deallocation (by a call to FreeBlock() or LocateAndFreeBlock()) of a block from a pool is deterministic if the blocks are created when the pool is created.

In order to conform to memory alignment constraints, the block size specified for a pool is rounded up internally, so that its size is a multiple of the size of a pointer on the architecture in question—all block addresses returned by MallocBlock() are a multiple of `sizeof(void *)`.

The GetNumAllocatedBlocks() and GetNumFreeBlocks() APIs can be used to determine the number of used or available blocks respectively in a particular pool.

Multiple Heaps

By default, all VDK elements are allocated in the system heap(s).

Depending on the processor used, the VDK creates one or two system heaps. See Appendix A, “Processor-Specific Notes”, for information on specific processors.

In previous versions of the VDK, multiple heaps could be used in the definition of memory pools on processors where multiple heap support is provided. This mechanism has been extended and the VDK can now use multiple heaps defined at link time (dynamically created heaps are not allowed) to specify which area of memory is used to allocate the various VDK elements (semaphores, messages, thread stacks, and so on). The developer is responsible for setting up the heaps. For more information regarding how to specify multiple heaps, refer to the *C/C++ Compiler and Library Manual* for your target processor(s).

To specify a VDK heap, create a new heap in VisualDSP++ (which has a VDK `HeapID`). An ID is then associated with this name. This ID must be the same one used in setting up the heap under the C/C++ run-time (which is an integer or a string depending on the processor). For more information on how to set up VDK heaps, see the online documentation.

Thread Local Storage

Thread local storage allows the association of data with threads on a per thread basis. A typical usage of this functionality involves allocating the data required by individual threads for a thread-safe library function (for example, to store the thread-specific value of `errno` for each thread for the C runtime libraries).

There are eight thread local storage slots available for this purpose. Before a value is stored in the relevant slot in the thread's slot table, an entry must be allocated in the global slot table by using either AllocateThreadSlot() or AllocateThreadSlotEx(). If a slot is available in the global table, then the corresponding slot is also reserved in each thread slot table. These APIs return `FALSE` if there are no free slots available. An allocated entry in the global slot table can subsequently be freed by a call to FreeThreadSlot(). This mechanism for allocating slots provides one time initialization of slots for thread-specific data for library functions. Slots are allocated in every thread's slot table on the first calling of the library function by any thread.

Once a slot has been allocated in the global slot table, the corresponding value in the slot table of a particular thread can be set by a call to SetThreadSlotValue() from the thread in question. The value is of type `void *` and can be used to store an integer value or a pointer to allocated memory. The use of AllocateThreadSlotEx() to allocate a slot allows the specification of a cleanup function to be called on thread destruction to deal with any dynamically allocated memory that has been associated with a thread slot. Finally, GetThreadSlotValue() can be used to obtain the value stored in the slot table of a particular thread.

4 VDK DATA TYPES

VDK comes with a predefined set of data types. This chapter describes the current release of the kernel. Future releases may include further types.

This chapter contains:

- "Data Type Summary"
- "Data Type Descriptions"

Data Type Summary

VDK data types are summarized in Table 4-1. A description of each type begins on page 4-4.

Table 4-1. VDK Data Types

Data Type	Reference Page
`Bitfield`	on page 4-4
`DeviceDescriptor`	on page 4-5
`DeviceFlagID`	on page 4-6
`DeviceInfoBlock`	on page 4-7
`DispatchID`	on page 4-8
`DispatchUnion`	on page 4-9
`DSP_Family`	on page 4-11
`DSP_Product`	on page 4-12

Table 4-1. VDK Data Types (Cont'd)

Data Type	Reference Page
EventBitID	on page 4-15
EventID	on page 4-16
EventData	on page 4-17
HeapID	on page 4-18
HistoryEnum	on page 4-19
IMASKStruct	on page 4-21
IOID	on page 4-22
IOTemplateID	on page 4-23
MarshallingCode	on page 4-24
MarshallingEntry	on page 4-26
MessageDetails	on page 4-27
MessageID	on page 4-28
MsgChannel	on page 4-29
MsgWireFormat	on page 4-31
PanicCode	on page 4-33
PayloadDetails	on page 4-35
PFMarshaller	on page 4-36
PoolID	on page 4-38
Priority	on page 4-39
RoutingDirection	on page 4-40
SemaphoreID	on page 4-41
SystemError	on page 4-42
ThreadCreationBlock	on page 4-46
ThreadID	on page 4-48

Table 4-1. VDK Data Types (Cont'd)

Data Type	Reference Page
ThreadStatus	on page 4-49
ThreadType	on page 4-50
Ticks	on page 4-51
VersionStruct	on page 4-52

Data Type Descriptions

The following sections provide descriptions of VDK data types .

Bitfield

The `Bitfield` type is used to store a bit pattern. The size of a `Bitfield` item is the size of a data word word, which is 32 bits on SHARC, Tiger-SHARC and Blackfin processors.

In C:

```
typedef unsigned  int VDK_Bitfield;
```

In C++:

```
typedef unsigned int VDK::Bitfield;
```

DeviceDescriptor

The `DeviceDescriptor` type is used to store the unique identifier of an opened device. The value is obtained dynamically as the return value from OpenDevice().

In C:

```
typedef unsigned int VDK_DeviceDescriptor;
```

In C++:

```
typedef unsigned int VDK::DeviceDescriptor;
```

DeviceFlagID

The `DeviceFlagID` type is used to store the unique identifier of a device flag.

```
enum DeviceFlagID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

The enumeration in `VDK.h` will only contain the IDs of the device flags enabled at boot time. Any dynamically-created device flags will have an ID of the same type to allow the compiler to do type checking and prevent errors.

In C:

```
typedef enum DeviceFlagID VDK_DeviceFlagID;
```

In C/C++:

```
typedef enum DeviceFlagID VDK::DeviceFlagID;
```

DeviceInfoBlock

The `DeviceInfoBlock` structure holds information on the device driver that is being used by a routing thread and is passed as an argument to marshalling functions. All fields except the `DeviceDescriptor` are private to VDK and should not be used by user code.

In C:

```
typedef struct
{
     VDK_DeviceDescriptor dd;
     VDK_PFDispatchFunction pfDispatchFunction;
     struct VDK_IOAbstractBase *pDevObj;
     struct VDK_DeviceControlBlock *pDcb;
} VDK_DeviceInfoBlock;
```

In C++:

```
typedef struct
{
     VDK::DeviceDescriptor dd;
     VDK::PFDispatchFunction pfDispatchFunction;
     struct VDK::IOAbstractBase *pDevObj;
     struct VDK::DeviceControlBlock *pDcb;
} VDK::DeviceInfoBlock;
```

where `dd` is the descriptor for the device. Marshalling functions may use it as an argument for their SyncRead() and SyncWrite() calls.

DispatchID

The `DispatchID` type enumerates a device driver's dispatch commands.

In C:

```
enum VDK_DispatchID
{
  VDK_kIO_Init,
  VDK_kIO_Activate,
  VDK_kIO_Open,
  VDK_kIO_Close,
  VDK_kIO_SyncRead,
  VDK_kIO_SyncWrite,
  VDK_kIO_IOCtl
};
```

In C++:

```
enum VDK::DispatchID
{
  VDK::kIO_Init,
  VDK::kIO_Activate,
  VDK::kIO_Open,
  VDK::kIO_Close,
  VDK::kIO_SyncRead,
  VDK::kIO_SyncWrite,
  VDK::kIO_IOCtl
};
```

DispatchUnion

A variable of the DispatchUnion type is passed as a parameter to a device driver dispatch function. Calls to OpenDevice(), CloseDevice(), SyncRead(), SyncWrite(), and DeviceIOCtl() set up the relevant members of this union before calling the device driver's dispatch function.

In C:

```
union VDK_DispatchUnion
{
  struct
  {
     void   **dataH;
     char   *flags;   /* used for kIO_Open only */
  } OpenClose_t;
  struct
  {
    void          **dataH;
    VDK_Ticks     timeout;
    unsigned int  dataSize;
    char          *data;
  } ReadWrite_t;
  struct
  {
    void          **dataH;
    void          *command;
    char          *parameters;
  } IOCtl_t;
  struct
  {
    void          *pInitInfo;
  } Init_t ;
};
```

In C++:

```
union VDK::DispatchUnion
{
  struct
  {
    void    **dataH;
    char    *flags; /* used for kIO_Open only */
  } OpenClose_t;
  struct
  {
    void         **dataH;
    VDK::Ticks   timeout;
    unsigned int dataSize;
    char         *data;
  } ReadWrite_t;
  struct
  {
    void       **dataH;
    void       *command;
    char       *parameters;
  } IOCtl_t;
  struct
  {
    void       *pInitInfo;
  }  Init_t;
};
```

DSP_Family

The `DSP_Family` type enumerates the processor families supported by VDK. See also VersionStruct on page 4-52.

In C:

```
enum VDK_DSP_Family
{
  VDK_kUnknownFamily,
  VDK_kSHARC,
  VDK_kTSXXX,
  VDK_kBLACKFIN
};
```

In C++:

```
enum VDK::DSP_Family
{
  VDK::kUnknownFamily,
  VDK::kSHARC,
  VDK::kTSXXX,
  VDK::kBLACKFIN
};
```

DSP_Product

The `DSP_Product` type enumerates the devices supported by VDK. See also VersionStruct on page 4-52.

In C:

```
enum VDK_DSP_Product
{
  VDK_kUnknownProduct,
  VDK_k21060,
  VDK_k21061,
  VDK_k21062,
  VDK_k21065,
  VDK_k21160,
  VDK_k21161,
  VDK_k21262,
  VDK_k21266,
  VDK_k21261,
  VDK_k21267,
  VDK_k21363,
  VDK_k21364,
  VDK_k21365,
  VDK_k21366,
  VDK_k21367,
  VDK_k21368,
  VDK_k21369,
  VDK_kBF535,
  VDK_kBF532,
  VDK_kBF531,
  VDK_kBF533,
  VDK_kAD6532,
  VDK_kBF561,
  VDK_kBF539,
  VDK_kBF534,
  VDK_kBF536,
  VDK_kBF537,
  VDK_kBF538,
```

```
    VDK_kBF566,
    VDK_kTS101,
    VDK_kTS201,
    VDK_kTS202,
    VDK_kTS203
  };
```

In C++:

```
  enum VDK::DSP_Product
  {
    VDK::kUnknownProduct,
    VDK::k21060,
    VDK::k21061,
    VDK::k21062,
    VDK::k21065,
    VDK::k21160,
    VDK::k21161,
    VDK::k21262,
    VDK::k21266,
    VDK::k21261,
    VDK::k21267,
    VDK::k21363,
    VDK::k21364,
    VDK::k21365,
    VDK::k21366,
    VDK::k21367,
    VDK::k21368,
    VDK::k21369,
    VDK::kBF535,
    VDK::kBF532,
    VDK::kBF531,
    VDK::kBF533,
    VDK::kAD6532,
    VDK::kBF561,
    VDK::kBF539,
    VDK::kBF534,
    VDK::kBF536,
    VDK::kBF537,
    VDK::kBF538,
```

```
    VDK::kBF566,
    VDK::kTS101,
    VDK::kTS201,
    VDK::kTS202,
    VDK::kTS203
};
```

EventBitID

The `EventBitID` type is used to store the unique identifier of an event bit. The total number of event bits in a system is the size of a data word minus one one, which is 31 bits on SHARC, TigerSHARC and Blackfin processors.

```
enum EventBitID
{
    /* Defined by IDDE in the VDK.h file. */
};
```

In C:

```
typedef enum EventBitID VDK_EventBitID;
```

In C/C++:

```
typedef enum EventBitID VDK::EventBitID;
```

EventID

The `EventID` type is used to store the unique identifier of an event. The total number of events in a system is the size of a data word minus one, which is 31 bits on SHARC, TigerSHARC, and Blackfin processors.

```
enum EventID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

In C:

```
typedef enum EventID VDK_EventID;
```

In C/C++:

```
typedef enum EventID VDK::EventID;
```

EventData

The `EventData` type is used to store the data associated with an event. See also "Behavior of Events" on page 3-39.

In C:

```
typedef struct
{
  bool           matchAll;
  VDK_Bitfield   values;
  VDK_Bitfield   mask;
} VDK_EventData;
```

In C++:

```
typedef struct
{
  bool           matchAll;
  VDK::Bitfield  values;
  VDK::Bitfield  mask;
} VDK::EventData;
```

HeapID

The `HeapID` type is used to store the unique identifier of a VDK Heap. This data type is only available on processors for which multiple heap support is provided.

```
enum HeapID
{
  /* Defined by IDDE in the VDK.h file */
};
```

In C:

```
typedef enum HeapID VDK_HeapID;
```

In C/C++:

```
typedef enum HeapID VDK::HeapID;
```

HistoryEnum

The `HistoryEnum` type enumerates the events that can be logged with a call to the LogHistoryEvent() API or VDK_ISR_LOG_HISTORY_() macro.

In C:

```
enum VDK_HistoryEnum {
  VDK_kThreadCreated = INT_MIN,
  VDK_kThreadDestroyed
  VDK_kSemaphorePosted,
  VDK_kSemaphorePended,
  VDK_kEventBitSet,
  VDK_kEventBitCleared,
  VDK_kEventPended,
  VDK_kDeviceFlagPended,
  VDK_kDeviceFlagPosted,
  VDK_kDeviceActivated,
  VDK_kThreadTimedOut,
  VDK_kThreadStatusChange,
  VDK_kThreadSwitched,
  VDK_kMaxStackUsed,
  VDK_kPoolCreated,
  VDK_kPoolDestroyed,
  VDK_kDeviceFlagCreated,
  VDK_kDeviceFlagDestroyed,
  VDK_kMessagePosted,
  VDK_kMessagePended,
  VDK_kSemaphoreCreated,
  VDK_kSemaphoreDestroyed,
  VDK_kMessageCreated,
  VDK_kMessageDestroyed,
  VDK_kMessageTakenFromQueue,
  VDK_kThreadResourcesFreed,
  VDK_kUserEvent = 1
};
```

In C++:

```
enum VDK::HistoryEnum
{
  VDK::kThreadCreated = INT_MIN,
  VDK::kThreadDestroyed,
  VDK::kSemaphorePosted,
  VDK::kSemaphorePended,
  VDK::kEventBitSet,
  VDK::kEventBitCleared,
  VDK::kEventPended,
  VDK::kDeviceFlagPended,
  VDK::kDeviceFlagPosted,
  VDK::kDeviceActivated,
  VDK::kThreadTimedOut,
  VDK::kThreadStatusChange,
  VDK::kThreadSwitched,
  VDK::kMaxStackUsed,
  VDK::kPoolCreated,
  VDK::kPoolDestroyed,
  VDK::kDeviceFlagCreated,
  VDK::kDeviceFlagDestroyed,
  VDK::kMessagePosted,
  VDK::kMessagePended,
  VDK::kSemaphoreCreated,
  VDK::kSemaphoreDestroyed,
  VDK::kMessageCreated,
  VDK::kMessageDestroyed,
  VDK::kMessageTakenFromQueue,
  VDK::kThreadResourcesFreed,
  VDK::kUserEvent = 1
};
```

IMASKStruct

The `IMASKStruct` type is a platform-dependent type used by the ClearInterruptMaskBits(), GetInterruptMask(), and SetInterruptMaskBits() APIs to modify the interrupt mask.

For TigerSHARC processors, this type is defined as:

In C:

```
typedef unsigned long long VDK_IMASKStruct;
```

In C++:

```
typedef unsigned long long VDK::IMASKStruct;
```

On Blackfin and SHARC processors, the type is defined as:

In C:

```
typedef unsigned int VDK_IMASKStruct;
```

In C++:

```
typedef unsigned int VDK::IMASKStruct;
```

IOID

The `IOID` type is used to store the unique identifier of an I/O object.

```
enum IOID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

In C:

```
typedef enum IOID VDK_IOID;
```

In C/C++:

```
typedef enum IOID VDK::IOID;
```

IOTemplateID

The `IOTemplateID` type is used to store the unique identifier of an I/O object class.

```
enum IOTemplateID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

In C:

```
typedef enum IOTemplateID VDK_IOTemplateID;
```

In C/C++:

```
typedef enum IOTemplateID VDK::IOTemplateID;
```

MarshallingCode

The `MarshallingCode` type enumerates the possible reasons for calling a payload marshalling function.

In C:

```
enum VDK_MarshallingCode
{
       TRANSMIT_AND_RELEASE,
       ALLOCATE_AND_RECEIVE,
       ALLOCATE,
       RELEASE
};
```

In C++:

```
enum VDK::MarshallingCode
{
       TRANSMIT_AND_RELEASE,
       ALLOCATE_AND_RECEIVE,
       ALLOCATE,
       RELEASE
};
```

`TRANSMIT_AND_RELEASE` indicates that the marshalling function must perform the following steps in sequence:

1. Modify the message packet, if necessary (optional)
2. Transmit the message packet
3. Transmit the payload contents (optional in certain cases)
4. Deallocate the payload memory

`ALLOCATE_AND_RECEIVE` indicates that the marshalling function must perform the following steps in sequence:

1. Allocate memory for a payload of the type (and size) specified by the message packet
2. Receive the payload contents into the payload memory

`ALLOCATE` indicates that the marshalling function must allocate memory for a payload of the type (and size) specified by the message packet.

`RELEASE` indicates that the marshalling function must deallocate the payload memory.

MarshallingEntry

The `MarshallingEntry` structure forms the elements of the marshalling table array `g_vMarshallingTable` (defined by the IDDE in `Vdk.cpp`).

In C:

```
typedef struct
{
      VDK_PFMarshaller pfMarshaller;
      unsigned int area;
} VDK_MarshallingEntry;
```

In C++:

```
typedef struct
{
      VDK::PFMarshaller pfMarshaller;
      unsigned int area;
} VDK::MarshallingEntry;
```

`pfMarshaller` is a pointer to a system- or user-defined marshalling function.

`area` is used by standard marshalling to hold the Heap index (for heap marshalling) or PoolID (for pool marshalling) in order to parameterize the operation of the standard functions. It may also be used to parameterize the operation of custom marshalling functions.

MessageDetails

The `MessageDetails` structure combines the three attributes that describe the most recent posting of a message.

In C:

```
typedef struct
{
  VDK_MsgChannel channel;
  VDK_ThreadID  sender;
  VDK_ThreadID  target;
} VDK_MessageDetails;
```

In C++:

```
typedef struct
{
  VDK::MsgChannel channel;
  VDK::ThreadID  sender;
  VDK::ThreadID  target;
} VDK::MessageDetails;
```

MessageID

The `MessageID` type is used to store the unique identifier of a message.

```
enum MessageID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

The enumeration in `VDK.h` will be empty. All the messages are dynamically allocated and will have an ID of the `MessageID` type to allow the compiler to do type checking and prevent errors.

In C:

```
typedef enum MessageID VDK_MessageID;
```

In C/C++:

```
typedef enum MessageID VDK::MessageID;
```

MsgChannel

The `MsgChannel` type enumerates the channels a message can be posted or pended on.

In C:

```
enum VDK_MsgChannel
{
  VDK_kMsgWaitForAll = 1 << 15,
  VDK_kMsgChannel1  =  1 << 14,
  VDK_kMsgChannel2  =  1 << 13,
  VDK_kMsgChannel3  =  1 << 12,
  VDK_kMsgChannel4  =  1 << 11,
  VDK_kMsgChannel5  =  1 << 10,
  VDK_kMsgChannel6  =  1 << 9,
  VDK_kMsgChannel7  =  1 << 8,
  VDK_kMsgChannel8  =  1 << 7,
  VDK_kMsgChannel9  =  1 << 6,
  VDK_kMsgChannel10  = 1 << 5,
  VDK_kMsgChannel11  = 1 << 4,
  VDK_kMsgChannel12  = 1 << 3,
  VDK_kMsgChannel13  = 1 << 2,
  VDK_kMsgChannel14  = 1 << 1,
  VDK_kMsgChannel15  = 1 << 0
};
```

In C/C++:

```
enum VDK::MsgChannel
{
  VDK::kMsgWaitForAll = 1 << 15,
  VDK::kMsgChannel1  =  1 << 14,
  VDK::kMsgChannel2  =  1 << 13,
  VDK::kMsgChannel3  =  1 << 12,
  VDK::kMsgChannel4  =  1 << 11,
  VDK::kMsgChannel5  = 1 << 10,
```

```
    VDK::kMsgChannel6  = 1 << 9,
    VDK::kMsgChannel7  = 1 << 8,
    VDK::kMsgChannel8  = 1 << 7,
    VDK::kMsgChannel9  = 1 << 6,
    VDK::kMsgChannel10 = 1 << 5,
    VDK::kMsgChannel11 = 1 << 4,
    VDK::kMsgChannel12 = 1 << 3,
    VDK::kMsgChannel13 = 1 << 2,
    VDK::kMsgChannel14 = 1 << 1,
    VDK::kMsgChannel15 = 1 << 0
};
```

MsgWireFormat

The `MsgWireFormat` structure is used to transfer a message across a communication link. It is the structure written to and read from the device drivers that manage the links.

In C:

```
typedef struct
{
  unsigned int header;
  VDK_PayloadDetails payload;
} VDK_MsgWireFormat;
```

In C++:

```
typedef struct
{
  unsigned int header;
  VDK::PayloadDetails payload;
} VDK::MsgWireFormat;
```

`MsgWireFormat` is four words (16 bytes or 128 bits) in size, of which three words are made up of the payload description. The remaining (first) word is the message header, which contains the additional information about the message, packed using the following format:

Bit Position	31 to 28	27 to 23	22 to 14	13 to 9	8 to 0
Word 0	Channel	Destination Node	Destination Thread	Source Node	Source Thread
Word 1	Payload Type				
Word 2	Payload Address				
Word 3	Payload Length				

The header bit allocation allows for up to 32 nodes in the system and up to 512 threads per node.

Because there are only 15 message channel numbers (1 to 15) used by VDK, message packets having header bits 28 to 31 set to all zeros (that is, the non-existent channel 0) are special cases which may be used internally by VDK (or by the device drivers) as private control messages.

The message ID is *not* transferred in the packet, as the message will have a different ID on the destination processor.

PanicCode

The `PanicCode` type enumerates the possible causes of the VDK raising a Kernel Panic. When the VDK enters Kernel Panic, the cause is stored in the variable `VDK::g_KernelPanicCode` in C++ (C++ syntax must be used).

In C:

```
enum VDK_PanicCode
   {
        VDK_kNoPanic=0,
        VDK_kThreadError,
        VDK_kBootError
        VDK_kISRError,
        VDK_kDeprecatedAPI,
        VDK_kInternalError,
        VDK_kStackCheckFailure
   };
```

In C/C++:

```
enum VDK::PanicCode
   {
        VDK::kNoPanic=0,
        VDK::kThreadError,
        VDK::kBootError
        VDK::kISRError,
        VDK::kDeprecatedAPI,
        VDK::kInternalError,
        VDK::kStackCheckFailure
   };
```

The `g_KernelPanicCode` variable has a value of `kNoPanic` when `KernelPanic` has not been called.

The g_KernelPanicCode variable has a value of kThreadError when a thread's error function does not handle the error (default behavior) or when an attempt is made to dispatch an error when the running thread is the idle thread.

The g_KernelPanicCode variable has a value of kBootError when there has been a problem creating any of the VDK boot components (threads, semaphores, memory pools, and so on).

The g_KernelPanicCode variable has a value of kISRError when an ISR macro is invoked with an ID greater than the maximum allowed for the particular macro.

The g_KernelPanicCode variable has a value of kDeprecatedAPI when the API in question is no longer supported.

The g_KernelPanicCode variable has a value of InternalError when VDK detected internal problems which it cannot recover from.

The g_KernelPanicCode variable has a value of kStackCheckFailure when VDK has detected a thread which has overrun its stack. Users should not rely on this to verify that threads have not overrun their stacks as VDK can only detect a specific limited number of cases. This panic code is only used in fully instrumented builds.

PayloadDetails

The `PayloadDetails` structure combines the three attributes that describe a message payload.

In C:

```
typedef struct
{
      int type;
      unsigned int size;
      void *addr;
} VDK_PayloadDetails;
```

In C/C++:

```
typedef struct
{
      int type;
      unsigned int size;
      void *addr;
} VDK::PayloadDetails;
```

The `type` variable is an application-defined value that specifies the interpretation given to the contents of the payload. Negative values of payload type indicate a user-defined marshalled type, which can be managed automatically by the VDK for the purposes of inter-processor messaging.

The `size` variable is typically the size of the payload in the smallest addressable units of the processor (`sizeof(char)`).

The `addr` variable is typically a pointer to the beginning of the payload buffer.

However, depending on the application-defined interpretation of the payload's type, the payload `addr` and `size` attributes may contain any user-defined data that can be stored in two 32-bit fields.

PFMarshaller

The `PFMarshaller` type is a pointer-to-function type, used to hold the address of a system- or user-defined marshalling function.

In C:

```
typedef int (*VDK_PFMarshaller)(VDK_MarshallingCode code,
                          VDK_MsgWireFormat *inOutMsgPacket,
                          VDK_DeviceInfoBlock *pDev,
                          unsigned int area,
                          VDK_Ticks timeout);
```

In C/C++:

```
typedef int (*VDK::PFMarshaller)(VDK::MarshallingCode code,
                          VDK::MsgWireFormat *inOutMsgPacket,
                          VDK::DeviceInfoBlock *pDev,
                          unsigned int area,
                          VDK::Ticks timeout);
```

Parameters

`code` tells the marshalling function which operation(s) to perform (see "MarshallingCode" on page 4-24).

`inOutMsgPacket` is a pointer to the formatted message packet.

`pDev` is a pointer to a `DeviceInfoBlock` structure describing the VDK device driver for the connection.

`area` is the Heap index or PoolID used by standard marshalling.

`timeout` is the I/O timeout duration (usually set to 0, for indefinite wait).

The marshalling function may invoke the scheduler, depending on the implementation. The return value from a marshalling function will usually be the result of a SyncRead() or SyncWrite() call that has been performed internally, but this value is not presently used. Errors may be thrown by the marshalling function, or by functions called by it.

PoolID

The `PoolID` type is used to store the unique identifier of a memory pool.

```
enum PoolID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

The enumeration in `VDK.h` will contain only the IDs for the memory pools enabled at boot time. Any dynamically-created memory pools will have an ID of the same type to allow the compiler to perform type checking and prevent errors.

In C:

```
typedef enum PoolID VDK_PoolID;
```

In C/C++:

```
typedef enum PoolID VDK::PoolID;
```

Priority

The `Priority` type is used to denote the scheduling priority level of a thread:

- The highest priority is one (zero is reserved)
- The lowest priority is the size of a data word minus two.
  For SHARC, TigerSHARC, or Blackfin processors, this value is 30.

In C:

```
enum VDK_Priority
{
  VDK_kPriority1,
  VDK_kPriority2,
  VDK_kPriority3,
  …
  VDK_kPriority30
};
```

In C++:

```
enum VDK::Priority
{
  VDK::kPriority1,
  VDK::kPriority2,
  VDK::kPriority3,
  …
  VDK::kPriority30
};
```

RoutingDirection

The `RoutingDirection` type enumerates the two distinct operating modes of a routing thread. It is used to specify the operating mode of a routing thread at the time of its creation.

In C:

```
enum VDK_RoutingDirection
{
     kINCOMING,
     kOUTGOING
} ;
```

In C++:

```
enum VDK::RoutingDirection
{
     kINCOMING,
     kOUTGOING
 } ;
```

SemaphoreID

The `SemaphoreID` type is used to store the unique identifier of a semaphore.

```
enum SemaphoreID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

The enumeration in `VDK.h` will contain the IDs only for the semaphores enabled at boot time. Any dynamically-created semaphores will have an ID of the same type to allow the compiler to perform type checking and prevent errors.

In C:

```
typedef enum SemaphoreID VDK_SemaphoreID;
```

In C/C++:

```
typedef enum SemaphoreID VDK::SemaphoreID;
```

SystemError

The `SystemError` type enumerates system-defined errors thrown to the error handler.

In C:

```
enum VDK_SystemError
{
  VDK_kUnknownThreadType = INT_MIN,
  VDK_kUnknownThread,
  VDK_kInvalidThread,
  VDK_kThreadCreationFailure,
  VDK_kUnknownSemaphore,
  VDK_kUnknownEventBit,
  VDK_kUnknownEvent,
  VDK_kInvalidPriority,
  VDK_kInvalidDelay,
  VDK_kSemaphoreTimeout,
  VDK_kEventTimeout,
  VDK_kBlockInInvalidRegion,
  VDK_kDbgPossibleBlockInRegion,
  VDK_kInvalidPeriod,
  VDK_kAlreadyPeriodic,
  VDK_kNonperiodicSemaphore,
  VDK_kDbgPopUnderflow,
  VDK_kBadIOID,
  VDK_kBadDeviceDescriptor,
  VDK_kOpenFailure,
  VDK_kCloseFailure,
  VDK_kReadFailure,
  VDK_kWriteFailure,
  VDK_kIOCtlFailure,
  VDK_kInvalidDeviceFlag,
  VDK_kDeviceTimeout,
  VDK_kDeviceFlagCreationFailure,
  VDK_kMaxCountExceeded,
  VDK_kSemaphoreCreationFailure,
```

```
    VDK_kSemaphoreDestructionFailure,
    VDK_kPoolCreationFailure,
    VDK_kInvalidBlockPointer,
    VDK_kInvalidPoolParms,
    VDK_kInvalidPoolID,
    VDK_kErrorPoolNotEmpty,
    VDK_kErrorMallocBlock,
    VDK_kMessageCreationFailure,
    VDK_kInvalidMessageID,
    VDK_kInvalidMessageOwner,
    VDK_kInvalidMessageChannel,
    VDK_kInvalidMessageRecipient,
    VDK_kMessageTimeout,
    VDK_kMessageInQueue,
    VDK_kInvalidTimeout,
    VDK_kInvalidTargetDSP,
    VDK_kIOCreateFailure,
    VDK_kHeapInitialisationFailure,
    VDK_kInvalidHeapID,
    VDK_kNewFailure,
    VDK_kInvalidMarshalledType,
    VDK_kUncaughtException,
    VDK_kAbort,
    VDK_kInvalidMaskBit,
    VDK_kInvalidThreadStatus,
    VDK_kThreadStackOverflow,
    VDK_kMaxIDExceeded,
    VDK_kThreadDestroyedInInvalidRegion,
    VDK_kNoError = 0,
    VDK_kFirstUserError,
    VDK_kLastUserError = INT_MAX
  };
```

In C++:

```
  enum VDK::SystemError
  {
    VDK::kUnknownThreadType = INT_MIN,
    VDK::kUnknownThread,
```

```
VDK::kInvalidThread,
VDK::kThreadCreationFailure,
VDK::kUnknownSemaphore,
VDK::kUnknownEventBit,
VDK::kUnknownEvent,
VDK::kInvalidPriority,
VDK::kInvalidDelay,
VDK::kSemaphoreTimeout,
VDK::kEventTimeout,
VDK::kBlockInInvalidRegion,
VDK::kDbgPossibleBlockInRegion,
VDK::kInvalidPeriod,
VDK::kAlreadyPeriodic,
VDK::kNonperiodicSemaphore,
VDK::kDbgPopUnderflow,
VDK::kBadIOID,
VDK::kBadDeviceDescriptor,
VDK::kOpenFailure,
VDK::kCloseFailure,
VDK::kReadFailure,
VDK::kWriteFailure,
VDK::kIOCtlFailure,
VDK::kInvalidDeviceFlag,
VDK::kDeviceTimeout,
VDK::kDeviceFlagCreationFailure,
VDK::kMaxCountExceeded,
VDK::kSemaphoreCreationFailure,
VDK::kSemaphoreDestructionFailure,
VDK::kPoolCreationFailure,
VDK::kInvalidBlockPointer,
VDK::kInvalidPoolParms,
VDK::kInvalidPoolID,
VDK::kErrorPoolNotEmpty,
VDK::kErrorMallocBlock,
VDK::kMessageCreationFailure,
VDK::kInvalidMessageID,
VDK::kInvalidMessageOwner,
VDK::kInvalidMessageChannel,
VDK::kInvalidMessageRecipient,
VDK::kMessageTimeout,
VDK::kMessageInQueue,
VDK::kInvalidTimeout,
```

```
    VDK::kInvalidTargetDSP,
    VDK::kIOCreateFailure,
    VDK::kHeapInitialisationFailure,
    VDK::kInvalidHeapID,
    VDK::kNewFailure,
    VDK::kInvalidMarshalledType,
    VDK::kUncaughtException,
    VDK::kAbort,
    VDK::kInvalidMaskBit,
    VDK::kInvalidThreadStatus,
    VDK::kThreadStackOverflow,
    VDK::kMaxIDExceeded,
    VDK::kThreadDestroyedInInvalidRegion,
    VDK::kNoError = 0,
    VDK::kFirstUserError,
    VDK::kLastUserError = INT_MAX
};
```

ThreadCreationBlock

A variable of the type `ThreadCreationBlock` is passed to the CreateThreadEx() function.

In C:

```
typedef struct VDK_ThreadCreationBlock
{
  VDK_ThreadType  template_id;
  VDK_ThreadID    thread_id;
  unsigned int    thread_stack_size;
  VDK_Priority    thread_priority;
  void            *user_data_ptr;
  struct VDK_ThreadTemplate
                  *pTemplate;
} VDK_ThreadCreationBlock;
```

In C++:

```
typedef struct VDK::ThreadCreationBlock
{
  VDK::ThreadType  template_id;
  VDK::ThreadID    thread_id;
  unsigned int     thread_stack_size;
  VDK::Priority    thread_priority;
  void             *user_data_ptr;
  struct VDK::ThreadTemplate
                   *pTemplate;
} VDK::ThreadCreationBlock;
```

- `template_id` corresponds to a ThreadType defined in the `VDK.h` and `vdk.cpp` files. These files contain the default values for the stack size and initial priority, which may optionally be overridden by the following fields.
- `thread_id` is an output only field. On a successful return, it contains the same value as the function return.

- `thread_stack_size` overrides the default stack size implied by the ThreadType when it is nonzero.
- `thread_priority` overrides the default thread priority implied by the ThreadType when it is nonzero.
- `user_data_ptr` allows a generic argument to be passed (without interpretation) to the thread creation function and, hence, to the thread constructor. This allows individual thread instances to be parameterized at creation time, without the need to resort to global variables for argument passing.
- `pTemplate` is a pointer to the thread template that is used to generate the thread. This is only required if the `template_id` is set to `kDynamicThreadType`. The stack size and initial priority are (optionally) overridden by the values specified in the `thread_stack_size` and `thread_priority fields`.

ThreadID

The `ThreadID` type is used to store the unique identifier of a thread.

```
enum ThreadID
{
  /* Defined by IDDE in the VDK.h file. */
};
```

The enumeration in `VDK.h` will contain the IDs only for the threads enabled at boot time. Any dynamically-created threads will have an ID of the same type to allow the compiler to perform type checking and prevent errors.

In C:

```
typedef enum ThreadID VDK_ThreadID;
```

In C/C++:

```
typedef enum ThreadID VDK::ThreadID;
```

ThreadStatus

The `ThreadStatus` type is used to enumerate the state of a thread.

In C:

```
enum VDK_ThreadStatus
{
  VDK_kReady,
  VDK_kSemaphoreBlocked,
  VDK_kEventBlocked,
  VDK_kDeviceFlagBlocked,
  VDK_kSemaphoreBlockedWithTimeout,
  VDK_kEventBlockedWithTimeout,
  VDK_kDeviceFlagBlockedWithTimeout,
  VDK_kSleeping,
  VDK_MessageBlocked,
  VDK_kMessageBlockedWithTimeout,
  VDK_kUnknown
};
```

In C++:

```
enum VDK::ThreadStatus
{
  VDK::kReady,
  VDK::kSemaphoreBlocked,
  VDK::kEventBlocked,
  VDK::kDeviceFlagBlocked,
  VDK::kSemaphoreBlockedWithTimeout,
  VDK::kEventBlockedWithTimeout,
  VDK::kDeviceFlagBlockedWithTimeout,
  VDK::kSleeping,
  VDK::MessageBlocked,
  VDK::kMessageBlockedWithTimeout,
  VDK::kUnknown
};
```

ThreadType

`ThreadType` is used to store the unique identifier of a Thread class.

```
enum ThreadType
{
  /* Defined by IDDE in the VDK.h file. */
};
```

In C:

```
typedef enum ThreadType VDK_ThreadType;
```

In C/C++:

```
typedef enum ThreadType VDK::ThreadType;
```

Ticks

Time is measured in system `Ticks`. A tick is the amount of time between hardware interrupts generated by a hardware timer.

In C:

```
typedef unsigned int VDK_Ticks;
```

In C++:

```
typedef unsigned int VDK::Ticks;
```

VersionStruct

The `VersionStruct` constant is used to store four integers that describe the system parameters:

- VDK API version number
- Processor family supported
- base processor product supported
- build number

In C:

```
typedef struct
{
  int              mAPIVersion;
  VDK_DSP_Family   mFamily;
  VDK_DSP_Product  mProduct;
  long             mBuildNumber;
} VDK_VersionStruct;
```

In C++:

```
typedef struct
{
  int               mAPIVersion;
  VDK::DSP_Family   mFamily;
  VDK::DSP_Product  mProduct;
  long              mBuildNumber;
} VDK::VersionStruct;
```

The DSP_Family and DSP_Product types are described on page 4-11 and on page 4-12, respectively.

5 VDK API REFERENCE

The VDK Application Programming Interface (API) is a library of functions and macros that may be called from your application programs. Application programs depend on API functions to perform services that are basic to VDK. These services include interrupt handling, scheduler management, thread management, semaphore management, memory pool management, events and event bits, device drivers, and message passing. All of the VDK functions are written in the C++ programming language.

This chapter describes the current release of the API library. Future releases may include additional functions.

This chapter provides information on the following topics:

- "Calling Library Functions" on page 5-2
- "Linking Library Functions" on page 5-2
- "Working With VDK Library Header" on page 5-3
- "Passing Function Parameters" on page 5-3
- "Library Naming Conventions" on page 5-3
- "API Summary" on page 5-5
- "VDK Error Codes and Error Values" on page 5-10
- "API Functions" on page 5-18
- "Assembly Macros and C/C++ ISR APIs" on page 5-161

Calling Library Functions

To use an API function or a macro, call it by name and provide the appropriate arguments. The name and arguments for each library entity appear on its reference page. Note that the function names are C and C++ function names. If you call a C run-time library function from an assembly language program, prefix the function name with an underscore.

Similar to other functions, library functions should be declared. Declarations are supplied in the `VDK.h` header file. For more information about the kernel header file, see "Working With VDK Library Header" on page 5-3.

The reference pages appear in the "API Summary" on page 5-5.

Linking Library Functions

When your code calls an API function, the call creates a reference resolved by the linker when linking your program. One way to direct the linker to the library's location is to use the default VDK Linker Description File (`VDK-`<*your_target*>`.LDF`). The default VDK Linker Description File automatically directs the linker to the `*.DLB` file in the `lib` subdirectory of your VisualDSP++ installation.

If you do not use the default VDK `.LDF` file, add the library file to your project's `.LDF` file. Alternatively, use the compiler's `-l` (library directory) switch to specify the library to be added to the link line. Library functions are not linked into the `.DXE` file unless they are referenced.

Working With VDK Library Header

If one of your program source files needs to call a VDK API library function, include the `VDK.h` header file with the `#include` preprocessor command. The header file provides prototypes for all VDK public functions. The compiler uses prototypes to ensure each function is called with the correct arguments. The `VDK.h` file also provides declarations for user-accessible global variables, macros, type definitions, and enumerations.

Passing Function Parameters

All parameters passed through the VDK library functions listed in "API Summary" on page 5-5 are either passed by value or as constant objects. This means VDK does not modify any of the variables passed. Where arguments need to be modified, they are passed by the address (pointer).

Library Naming Conventions

Table 5-1 and Table 5-2 show coding style conventions that apply to the entities in the library reference section. By following the library and function naming conventions, you can review VDK sources or documentation and recognize whether the identifier is a function, macro, variable parameter, or a constant.

Table 5-1. Library Naming Conventions

Notation	Description
`VDK_Ticks`	VDK-defined types are written with the first letter uppercase.
`kPriority1`	Constants are prefixed with a "`k`".

Table 5-1. Library Naming Conventions (Cont'd)

Notation	Description
`inType`	Input parameters are prefixed with an "`in`".
`mDevice`	Data members are prefixed with an "`m`".

Table 5-2. Function and Macro Naming Conventions

Notation	Description
`VDK_`	C-callable function names are prefixed by "`VDK_`" to distinguish VDK library functions from user functions.
`VDK::`	C++-callable functions are located in the VDK namespace, thus function names are preceded by "`VDK::`".
`VDK_Yield(void)`	The remaining portion of the function name is written with the first letter of each sub-word in uppercase.
`VDK_ISR_SET_EVENTBIT_()`	Assembly macros are written in uppercase with words separated by underscores and a trailing underscore.

API Summary

Table 5-3 through Table 5-19 list the VDK library entities included in the current software release. These tables list the library entities grouped by a particular service. The reference pages, beginning on page 5-19, appear in alphabetic order.

Table 5-3. Interrupt Handling Functions

Function Name	Reference Page
PopCriticalRegion()	on page 5-123
PopNestedCriticalRegions()	on page 5-125
PushCriticalRegion()	on page 5-136

Table 5-4. Interrupt Mask Handling Functions

ClearInterruptMaskBits()	on page 5-25
ClearInterruptMaskBitsEx()	on page 5-26
GetInterruptMask()	on page 5-73
GetInterruptMaskEx()	on page 5-74
SetInterruptMaskBits()	on page 5-143
SetInterruptMaskBitsEx()	on page 5-144

Table 5-5. Scheduler Management Functions

PopNestedUnscheduledRegions()	on page 5-127
PopUnscheduledRegion()	on page 5-128
PushUnscheduledRegion()	on page 5-137

Table 5-6. Block Memory Management Functions

CreatePool()	on page 5-34
CreatePoolEx()	on page 5-36
DestroyPool()	on page 5-49
FreeBlock()	on page 5-61
GetNumAllocatedBlocks()	on page 5-83
GetNumFreeBlocks()	on page 5-84
LocateAndFreeBlock()	on page 5-104
MallocBlock()	on page 5-108

Table 5-7. Thread and System Information Functions

GetClockFrequency()	on page 5-68
GetHeapIndex()	on page 5-72
GetThreadHandle()	on page 5-87
GetThreadID()	on page 5-88
GetThreadStackUsage()	on page 5-90
GetThreadStack2Usage()	on page 5-92
GetThreadStatus()	on page 5-94
GetTickPeriod()	on page 5-95
GetUptime()	on page 5-96
GetVersion()	on page 5-97
InstrumentStack()	on page 5-100
LogHistoryEvent()	on page 5-105
SetClockFrequency()	on page 5-140
SetTickPeriod()	on page 5-152

Table 5-8. Thread Creation and Destruction Functions

Table 5-9. Thread Local Storage Functions

Table 5-10. Thread Error Management Functions

Table 5-11. Thread Priority Management Functions

Table 5-12. Thread Scheduling Control Functions

Table 5-13. Semaphore Management Functions

CreateSemaphore()	on page 5-38
DestroySemaphore()	on page 5-51
GetSemaphoreValue()	on page 5-86
MakePeriodic()	on page 5-106
PendSemaphore()	on page 5-121
PostSemaphore()	on page 5-134
RemovePeriodic()	on page 5-138

Table 5-14. Event and EventBit Functions

ClearEventBit()	on page 5-23
GetEventBitValue()	on page 5-69
GetEventData()	on page 5-70
GetEventValue()	on page 5-71
LoadEvent()	on page 5-102
PendEvent()	on page 5-116
SetEventBit()	on page 5-141

Table 5-15. Device Flags Functions

CreateDeviceFlag()	on page 5-31
DestroyDeviceFlag()	on page 5-44
PendDeviceFlag()	on page 5-114
PostDeviceFlag()	on page 5-130

Table 5-16. Device Driver Functions

CloseDevice()	on page 5-29
DeviceIOCtl()	on page 5-55
OpenDevice()	on page 5-112

Table 5-16. Device Driver Functions (Cont'd)

SyncRead()	on page 5-155
SyncWrite()	on page 5-157

Table 5-17. Message Functions

CreateMessage()	on page 5-32
DestroyMessage()	on page 5-45
DestroyMessageAndFreePayload()	on page 5-47
ForwardMessage()	on page 5-58
FreeMessagePayload ()	on page 5-64
GetMessageDetails ()	on page 5-77
GetMessagePayload()	on page 5-79
GetMessageReceiveInfo()	on page 5-81
InstallMessageControlSemaphore ()	on page 5-98
MessageAvailable()	on page 5-110
PendMessage()	on page 5-118
PostMessage()	on page 5-131
SetMessagePayload()	on page 5-146

Table 5-18. Assembly Macros

Macro Name	Reference Page
VDK_ISR_ACTIVATE_DEVICE_()	on page 5-163
VDK_ISR_CLEAR_EVENTBIT_()	on page 5-164
VDK_ISR_LOG_HISTORY_()	on page 5-165
VDK_ISR_POST_SEMAPHORE_()	on page 5-166
VDK_ISR_SET_EVENTBIT_()	on page 5-167

Table 5-19. C/C++ ISR API

Function Name	Reference Page
C_ISR_ActivateDevice()	on page 5-168
C_ISR_ClearEventBit()	on page 5-170
C_ISR_PostSemaphore()	on page 5-171
C_ISR_SetEventBit()	on page 5-173

VDK Error Codes and Error Values

The entry for each VDK API function lists the error codes that may result from calling that function. Additional information is provided, where appropriate, in the form of an integer value relating to the error code in question. Table 5-20 summarizes the possible error codes dispatched by all VDK API functions, along with the associated error value.

Table 5-20. VDK API Error Codes and Error Values

API	Error code	Error value
ClearEventBit	kUnknownEventBit	inEventBitID
ClearInterruptMaskEx	kInvalidMaskBit	bit that doesn't refer to an interrupt mask bit
CloseDevice	kBadDeviceDescriptor	inDD
CreateDeviceFlag	kDeviceFlagCreationFailure	-1
CreateMessage	kMaxCountExceeded	inPayloadType
CreatePool	kPoolCreationFailure	0
CreatePool	kInvalidPoolParms	0

Table 5-20. VDK API Error Codes and Error Values (Cont'd)

API	Error code	Error value
CreatePoolEx	kPoolCreationFailure	0
CreatePoolEx	kInvalidPoolParms	0
CreateSemaphore	kMaxCountExceeded	-1
CreateSemaphore	kSemaphoreCreationFailure	-1
CreateThread	kUnknownThreadType	inType
CreateThread	kThreadCreationFailure	inType
CreateThreadEx	kUnknownThreadType	inOutTCB->template_id
CreateThreadEx	kThreadCreationFailure	inOutTCB->template_id
DestroyDeviceFlag	kInvalidDeviceFlag	inDeviceFlagID
DestroyMessage	kInvalidMessageID	inMessageID
DestroyMessage	kMessageInQueue	inMessageID
DestroyMessage	kInvalidMessageOwner	inMessageID
DestroyMessageAndFree Payload	kInvalidMessageID	inMessageID
DestroyMessageAndFree Payload	kMessageInQueue	inMessageID
DestroyMessageAndFree Payload	kInvalidMessageOwner	inMessageID
DestroyPool	kErrorPoolNotEmpty	inPoolID
DestroyPool	kInvalidPoolID	inPoolID

Table 5-20. VDK API Error Codes and Error Values (Cont'd)

API	Error code	Error value
DestroySemaphore	kSemaphoreDestructionFailure	inSemaphoreID
DestroySemaphore	kUnknownSemaphore	inSemaphoreID
DestroyThread	kInvalidThread	inThreadID
DestroyThread	kUnknownThread	inThreadID
DeviceIOCtl	kBadDeviceDescriptor	inDD
ForwardMessage	kInvalidMessageID	inMessageID
ForwardMessage	kMessageInQueue	inMessageID
ForwardMessage	kInvalidMessageOwner	inMessageID
FreeBlock	kInvalidPoolID	inPoolID
FreeBlock	kInvalidBlockPointer	inBlockPtr
FreeMessagePayload	kInvalidMessageID	inMessageID
FreeMessagePayload	kMessageInQueue	inMessageID
FreeMessagePayload	kInvalidMessageOwner	inMessageID
GetEventBitValue	kUnknownEventBit	inEventBitID
GetEventData	kUnknownEvent	inEventID
GetEventValue	kUnknownEvent	inEventID
GetHeapIndex	kInvalidHeapID	inHeapID
GetMessageDetails	kInvalidMessageID	inMessageID
GetMessageDetails	kMessageInQueue	inMessageID

Table 5-20. VDK API Error Codes and Error Values (Cont'd)

API	Error code	Error value
GetMessageDetails	kInvalidMessageOwner	inMessageID
GetMessagePayload	kInvalidMessageID	inMessageID
GetMessagePayload	kMessageInQueue	inMessageID
GetMessagePayload	kInvalidMessageOwner	inMessageID
GetMessageReceiveInfo	kInvalidMessageID	inMessageID
GetMessageReceiveInfo	kMessageInQueue	inMessageID
GetMessageReceiveInfo	kInvalidMessageOwner	inMessageID
GetNumAllocatedBlocks	kInvalidPoolID	inPoolID
GetNumFreeBlocks	kInvalidPoolID	inPoolID
GetPriority	kUnknownThread	inThreadID
GetSemaphoreValue	kUnknownSemaphore	0
GetThreadStackUsage	kUnknownThread	inThreadID
GetThreadStack2Usage	kUnknownThread	inThreadID
LoadEvent	kUnknownEvent	inEventID
LocateAndFreeBlock	kInvalidBlockPointer	inBlkPtr
MakePeriodic	kInvalidPeriod	inPeriod
MakePeriodic	kUnknownSemaphore	inSemaphoreID
MakePeriodic	kInvalidDelay	inDelay
MakePeriodic	kAlreadyPeriodic	inSemaphoreID

Table 5-20. VDK API Error Codes and Error Values (Cont'd)

API	Error code	Error value
MallocBlock	kInvalidPoolID	inPoolID
MallocBlock	kErrorMallocBlock	NumFreeBlocks
MessageAvailable	kInvalidThread	0
MessageAvailable	kInvalidMessageChannel	inMsgChannelMask
OpenDevice	kBadIOID	inIDNum
OpenDevice	kOpenFailure	inIDNum
PendDeviceFlag	kDeviceTimeout	inTimeout
PendDeviceFlag	kBlockInInvalidRegion	Number of nested unscheduled regions
PendDeviceFlag	kInvalidDeviceFlag	inFlagID
PendDeviceFlag	kInvalidTimeout	inTimeout
PendEvent	kDbgPossibleBlockInRegion	Number of nested unscheduled regions
PendEvent	kBlockInInvalidRegion	Number of nested unscheduled regions
PendEvent	kEventTimeout	inTimeout
PendEvent	kUnknownEvent	inEventID
PendEvent	kInvalidTimeout	inTimeout
PendMessage	kInvalidThread	0
PendMessage	kInvalidTimeout	inTimeout

Table 5-20. VDK API Error Codes and Error Values (Cont'd)

API	Error code	Error value
PendMessage	kDbgPossibleBlockInRegion	Number of nested unscheduled regions
PendMessage	kInvalidMessageChannel	inMessageChannelMask
PendMessage	kBlockInInvalidRegion	Number of nested unscheduled regions
PendMessage	kMessageTimeout	inTimeout
PendMessage	kInvalidMessageID	0
PendSemaphore	kInvalidTimeout	inTimeout
PendSemaphore	kDbgPossibleBlockInRegion	Number of nested unscheduled regions
PendSemaphore	kUnknownSemaphore	inSemaphoreID
PendSemaphore	kSemaphoreTimeout	inTimeout
PendSemaphore	kBlockInInvalidRegion	Number of nested unscheduled regions
PopCriticalRegion	kDbgPopUnderflow	kFromPopCriticalRegion
PopNestedCritical Regions	kDbgPopUnderflow	kFromPopNestedCritical Regions
PopNestedUnscheduled Regions	kDbgPopUnderflow	kFromPopNestedUnscheduled Regions
PopUnscheduled Region	kDbgPopUnderflow	kFromPopUnscheduled Region
PostDeviceFlag	kInvalidDeviceFlag	inFlagID
PostMessage	kInvalidMessageChannel	inChannel

Table 5-20. VDK API Error Codes and Error Values (Cont'd)

API	Error code	Error value
PostMessage	kInvalidTargetDSP	dstNode (MP only)
PostMessage	kInvalidMessageOwner	ID Message Owner
PostMessage	kInvalidMessageRecipient	recipient thread ID
PostMessage	kMessageInQueue	ID Message Owner
PostMessage	kInvalidMessageID	inMessageID
PostMessage	kUnknownThread	inRecipient
PostSemaphore	kUnknownSemaphore	inSemaphoreID
RemovePeriodic	kUnknownSemaphore	inSemaphoreID
RemovePeriodic	kNonperiodicSemaphore	inSemaphoreID
ResetPriority	kInvalidThread	inThreadID
ResetPriority	kUnknownThread	inThreadID
SetEventBit	kUnknownEventBit	inEventBitID
SetInterruptMaskEx	kInvalidMaskBit	bit that doesn't refer to an interrupt mask bit
SetMessagePayload	kInvalidMessageID	inMessageID
SetMessagePayload	kMessageInQueue	inMessageID
SetMessagePayload	kInvalidMessageOwner	inMessageID
SetPriority	kInvalidPriority	inPriority
SetPriority	kUnknownThread	inThreadID

Table 5-20. VDK API Error Codes and Error Values (Cont'd)

API	Error code	Error value
SetPriority	kInvalidThread	inThreadID
Sleep	kBlockInInvalidRegion	Number of nested unscheduled regions
Sleep	kInvalidDelay	inSleepTicks
SyncRead	kBadDeviceDescriptor	inDD
SyncWrite	kBadDeviceDescriptor	inDD
Yield	kBlockInInvalidRegion	Number of nested unscheduled regions

When using the VDK multiprocessor messaging functionality, the Routing Thread Run functions dispatch errors when appropriate. Table 5-21 summarizes the possible error codes dispatched by Routing Threads, along with the associated error value.

Table 5-21. Routing Thread Run Function Error Codes and Error Values

Error code	Error value
kBadIOID	IOID
kInvalidMarshalledType	PayloadType
kInvalidMessageID	0
kMaxCountExceeded	PayloadType
kOpenFailure	IOID

API Functions

The following format applies to all of the entries in the library reference section.

- **C Prototype** – Provides the C prototype (as it is found in `VDK.h`) describing the interface to the function
- **C++ Prototype** – Provides the C++ prototype (as it is found in `VDK.h`) describing the interface to the function
- **Description** – Describes the function's operation
- **Parameters** – Describes the function's parameters
- **Scheduling** – Specifies whether the function invokes the scheduler
- **Determinism** – Specifies whether the function is deterministic
- **Return Value** – Describes the function's return value
- **Errors Thrown** – Specifies errors detected by VDK that can be dealt with by the thread's error-handling routines

AllocateThreadSlot()

C Prototype

```
bool VDK_AllocateThreadSlot( int *ioSlotNum );
```

C++ Prototype

```
bool VDK::AllocateThreadSlot( int *ioSlotNum );
```

Description

Assigns a new slot number if `*ioSlotNum = VDK::kTLSUnallocated` and enters the allocated `*ioSlotNum` into the global slot identifier table.

- Returns `FALSE` immediately if the value of `*ioSlotNum` is not equal to `VDK::kTLSUnallocated` (`INT_MIN`) to guard against multiple attempts to allocate the same key variable
- Returns `FALSE` if there are no free slots, in which case `*ioSlotNum` is still `VDK::kTLSUnallocated`
- Otherwise allocates the first available slot, places the slot number in `*ioSlotNum`, and returns `TRUE`
- Does not access (change) any thread state
- Guaranteed to return `TRUE` once only for a given key variable, so the return value may be used to control other one-time library initializations
- May be safely called during system initialization (that is, before any threads are running)
- Equivalent to calling `AllocateThreadSlotEx` on page 5-21 with a `NULL` cleanup function

Parameters

`ioSlotNum` is a pointer to a slot identifier.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

`TRUE` upon success and `FALSE` upon failure

Errors Thrown

None

AllocateThreadSlotEx()

C Prototype

```
bool VDK_AllocateThreadSlotEx(int *ioSlotNum,
                              void(*cleanupFn)(void*));
```

C++ Prototype

```
bool VDK::AllocateThreadSlotEx(int *ioSlotNum,
                               void(*cleanupFn)(void*));
```

Description

Assigns a new slot number if `*ioSlotNum = VDK::kTLSUnallocated` and enters the allocated `*ioSlotNum` into the global slot identifier table.

- Returns `FALSE` immediately if the value of `*ioSlotNum` is not equal to `VDK::kTLSUnallocated` (`INT_MIN`) to guard against multiple attempts to allocate the same key variable
- Returns `FALSE` if there are no free slots, in which case `*ioSlotNum` is still `VDK::kTLSUnallocated`
- Otherwise allocates the first available slot, places the slot number in `*ioSlotNum`, stores the `cleanupFn` pointer internally, and returns `TRUE`
- Does not access (change) any thread state
- Guaranteed to return `TRUE` once only for a given key variable, so the return value may be used to control other one time library initialization
- May be safely called during system initialization, that is, before any threads are running

Parameters

`ioSlotNum` is a pointer to a slot identifier.

`cleanupFn` is a pointer to a function to handle cleanup of thread-specific data in the event of thread destruction and:

- May be `NULL`, in which case it does nothing
- Is called from within `DestroyThread()` (see on page 5-53)
- Executes in the context of the calling thread, not the thread that is being destroyed
- Is only called when the slot value is not `NULL`
- The `free()` function may be used as the cleanup function where the slot is used to hold "malloced" data.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

`TRUE` upon success and `FALSE` upon failure

Errors Thrown

None

ClearEventBit()

C Prototype

```
void VDK_ClearEventBit( VDK_EventBitID inEventBitID );
```

C++ Prototype

```
void VDK::ClearEventBit( VDK::EventBitID inEventBitID );
```

Description

The `ClearEventBit` clears the value of the event bit, that is, sets it to `FALSE`, `NULL`, or `0`. Once the event bit is cleared, the value of each dependent event is recalculated. If several event bits are to be cleared (or set) as a single operation, then the `SetEventBit()` (on page 5-141) and/or `ClearEventBit()` (on page 5-23) calls are made from within an unscheduled region. Event recalculation does not occur until the unscheduled region is popped.

Parameters

`inEventBitID` is the system event bit to clear.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownEventBit` indicates `inEventBitID` is not a valid identifier.

Non error-checking libraries: None

ClearInterruptMaskBits()

C Prototype

```
void VDK_ClearInterruptMaskBits(VDK_IMASKStruct inMask);
```

C++ Prototype

```
void VDK::ClearInterruptMaskBits(VDK::IMASKStruct inMask);
```

Description

Clears bits in the interrupt mask. In other words, the new mask is computed as the bitwise `AND` of the old mask and the one's complement of the `inMask` parameter.

Parameters

`inMask` specifies which bits are cleared in the interrupt mask.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

ClearInterruptMaskBitsEx()

C Prototype

```
void VDK_ClearInterruptMaskBitsEx(VDK_IMASKStruct inMask,
                                  VDK_IMASKStruct inLMask);
```

C++ Prototype

```
void VDK::ClearInterruptMaskBitsEx(VDK::IMASKStruct inMask,
                                   VDK::IMASKStruct inLMask);
```

Description

Clears bits in the `IMASK` and `LMASK` interrupt masks (where `LMASK` is the mask component of the `IRPTL` register). Any bits set in the parameters are cleared in the relevant interrupt mask. In other words, the new masks are computed as the bitwise `AND` of the old mask and the one's complement of the specified `IMASKStruct`. The bits specified for `inLMask` are shifted by the relevant amount for the processor in question, thereby allowing the use of the bit definitions for the `LIRPTL` register.

This API is applicable only to certain processors, currently those in the ADSP-2116x, ADSP-2126x, and ADSP-2136x processor families.

Parameters

`inMask` specifies which bits should be cleared in the `IMASK` interrupt mask.

`inLMask` specifies which bits should be cleared in the `LMASK` interrupt mask.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

`kInvalidMaskBit` if any of the bits in `inLmask` do not refer to an interrupt mask.

Non error-checking libraries: None

ClearThreadError()

C Prototype

```
void VDK_ClearThreadError(void);
```

C++ Prototype

```
void VDK::ClearThreadError(void);
```

Description

Sets the running thread's error status to `kNoError` and the error value to zero

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

CloseDevice()

C Prototype

```
void VDK_CloseDevice(VDK_DeviceDescriptor inDD);
```

C++ Prototype

```
void VDK::CloseDevice(VDK::DeviceDescriptor inDD);
```

Description

Closes the specified device. The function calls the dispatch function of the device opened with `inDD`.

Parameters

`inDD` is the `DeviceDescriptor` (see on page 4-5) returned from `OpenDevice()` (see on page 5-112)

Scheduling

Does not invoke the scheduler, but the user-written device driver can call the schedulerthat

Determinism

Constant time. Note that this function calls user-written device driver code that may not be deterministic.

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

`kBadDeviceDescriptor` indicates that `inDD` is not a valid `DeviceDescriptor` (see on page 4-5).

Non error-checking libraries: None

Note that other errors may be thrown by user-written device driver code executed by this API.

CreateDeviceFlag()

C Prototype

```
VDK_DeviceFlagID VDK_CreateDeviceFlag(void);
```

C++ Prototype

```
VDK::DeviceFlagID VDK::CreateDeviceFlag(void);
```

Description

Creates a new device flag and returns its identifier.

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

New device flag identifier upon success and `UINT_MAX` upon failure

Errors Thrown

Full instrumentation and error-checking libraries:

`kDeviceFlagCreationFailure` indicates that the kernel is not able to allocate and/or initialize memory for the device flag.

Non error-checking libraries: None

CreateMessage()

C Prototype

```
VDK_MessageID VDK_CreateMessage(int          inPayloadType,
                                unsigned int inPayloadSize,
                                void         *inPayloadAddr);
```

C++ Prototype

```
VDK::MessageID VDK::CreateMessage(int          inPayloadType,
                                  unsigned int inPayloadSize,
                                  void         *inPayloadAddr);
```

Description

Creates and initializes a new message object. The return value is the identifier of the new message. The values passed to `CreateMessage()` (on page 5-32) may be read by calling `GetMessagePayload()` (on page 5-79) and may be reset by calling `SetMessagePayload()` (on page 5-146). The calling thread becomes the owner of the new message.

Parameters

The `inPayloadType` is a user-defined value that may be used to convey additional information about the message and/or the payload to the receiving thread. This value is not used or modified by the kernel, except that negative values of payload type are reserved for use by VDK. Positive payload types are reserved for use by the application code. It is recommended that the payload address and size are always interpreted in the same way for each distinct message type.

The `inPayloadSize` is the length of the payload buffer in the smallest addressable unit on the processor architecture (`sizeof(char)`). When `inPayloadSize` has a value of zero, the kernel assumes `inPayloadAddr` is not a pointer and may contain any user value of the same size.

The `inPayloadAddr` is a pointer to the start of the data being passed in the message.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

New message identifier upon success and `UINT_MAX` upon failure

Errors Thrown

Full instrumentation and error-checking libraries:

- `kMaxCountExceeded` indicates that the number of simultaneous messages in the system exceeds the value specified in the GUI.
- `kErrorMallocBlock` indicates that there are no free blocks in the system memory pool used to allocate messages.

Non error-checking libraries: None

CreatePool()

C Prototype

```
VDK_PoolID VDK_CreatePool(unsigned int  inBlockSz,
                          unsigned int  inBlockCount,
                          bool          inCreateNow);
```

C++ Prototype

```
VDK::PoolID VDK::CreatePool(unsigned int  inBlockSz,
                            unsigned int  inBlockCount,
                            bool          inCreateNow);
```

Description

Creates a new memory pool in the system heap and returns the pool identifier

Parameters

`inBlockSz` specifies the block size in the lowest addressable unit.

`inBlockCount` specifies the total number of blocks in the pool.

`inCreateNow` indicates whether the block construction is done at runtime on an on-demand basis (`FALSE`) or as a part of the creation process (`TRUE`).

Scheduling

Does not invoke the scheduler

Determinism

Constant time if `inCreateNow` is `FALSE`.
Not deterministic if `inCreateNow` value is `TRUE`.

Return Value

New pool identifier upon success and `UINT_MAX` upon failure

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidPoolParms` indicates that either `inBlockSz` or `inBlockCount` is zero.
- `kPoolCreationFailure` indicates that the kernel is not able to allocate and/or initialize memory for the pool.

Non error-checking libraries: None

CreatePoolEx()

C Prototype

```
VDK_PoolID VDK_CreatePoolEx(unsigned int  inBlockSz,
                            unsigned int  inBlockCount,
                            bool          inCreateNow,
                            int           inWhichHeap);
```

C++ Prototype

```
VDK::PoolID VDK::CreatePoolEx(unsigned int  inBlockSz,
                              unsigned int  inBlockCount,
                              bool          inCreateNow,
                              int           inWhichHeap);
```

Description

Creates a new memory pool in the specified heap and returns the pool identifier. When architectures do not support multiple heaps, `inWhichHeap` must be initialized to zero. Refer to Appendix A, "Processor-Specific Notes" for architecture-specific information.

Parameters

`inBlockSz` specifies the block size in the lowest addressable units.

`inBlockCount` specifies the total number of blocks in the pool.

`inCreateNow` indicates whether block construction is done at runtime on an done on-demand basis (`FALSE`) or as a part of the creation process (`TRUE`).

`inWhichHeap` specifies the heap in which the pool is to be created. This parameter is ignored on single heap architectures. Setting the value of `inWhichHeap` to zero specifies the default heap to be used.

Scheduling

Does not invoke the scheduler

Determinism

Constant time if `inCreateNow` is `FALSE`.
Not deterministic if `inCreateNow` value is `TRUE`.

Return Value

New pool identifier upon success and `UINT_MAX` upon failure

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidPoolParms` indicates that either `inBlockSz` or `inBlockCount` is zero.
- `kPoolCreationFailure` indicates that the kernel is not able to allocate and/or initialize memory for the pool.

Non error-checking libraries: None

CreateSemaphore()

C Prototype

```
VDK_SemaphoreID VDK_CreateSemaphore(
                                   unsigned int inInitialValue,
                                   unsigned int inMaxCount,
                                   VDK_Ticks    inInitialDelay,
                                   VDK_Ticks    inPeriod);
```

C++ Prototype

```
VDK::SemaphoreID VDK::CreateSemaphore(
                                   unsigned int inInitialValue,
                                   unsigned int inMaxCount,
                                   VDK::Ticks   inInitialDelay,
                                   VDK::Ticks   inPeriod);
```

Description

Creates and initializes a dynamic semaphore. If the value of `inPeriod` is non-zero, a periodic semaphore is created.

Parameters

The `inInitialValue` is the value the semaphore has once it is created. A value of `zero` indicates that the semaphore is unavailable. This value should be between zero and `inMaxCount`.

The `inMaxCount` is the maximum number the semaphore's count can reach when posting it. An `inMaxCount` of one creates a binary semaphore, which is equivalent to the semaphores in the ????VisualDSP++ 2.0 release of VDK.

The `inInitialDelay` is the number of ticks before the first posting of a periodic semaphore. `InInitialDelay` must be equal to or greater than one.

The `inPeriod` specifies the period property of the semaphore and the number of ticks to sleep at each cycle after the semaphore is first posted. If `inPeriod` is zero, the created semaphore is not periodic.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

New semaphore identifier upon success and `UINT_MAX` upon failure

Errors Thrown

Full instrumentation and error-checking libraries:

- `kMaxCountExceeded` indicates that `inInitialValue` is greater than `inMaxCount`.
- `kSemaphoreCreationFailure` indicates that the kernel is not able to allocate and/or initialize memory for the semaphore.

Non error-checking libraries: None

CreateThread()

C Prototype

```
VDK_ThreadID VDK_CreateThread(VDK_ThreadType inType);
```

C++ Prototype

```
VDK::ThreadID VDK::CreateThread(VDK::ThreadType inType);
```

Description

Creates a thread of the specified type and returns the new thread.

Parameters

The `inType` corresponds to a thread type defined in the `VDK.h` and `vdk.cpp` files. These files contain the default values for the stack size, initial priority, and other properties.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

New thread identifier upon success and `UINT_MAX` upon failure

Errors Thrown

Full instrumentation and error-checking libraries:

- `kUnknownThreadType` indicates that `inType` is not an element of the `ThreadType` type (see on page 4-50), as defined in `VDK.h`.
- `kThreadCreationFailure` indicates that the kernel is not able to allocate and/or initialize memory for the thread.

Non error-checking libraries: None

CreateThreadEx()

C Prototype

```
VDK_ThreadID VDK_CreateThreadEx(VDK_ThreadCreationBlock *inOutTCB);
```

C++ Prototype

```
VDK::ThreadID VDK::CreateThreadEx(VDK::ThreadCreationBlock *inOutTCB);
```

Description

Creates a thread with the specified characteristics and returns the new thread

Parameters

`inOutTCB` is a pointer to a structure of the `ThreadCreationBlock` data type.(see on page 4-46).

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

New thread identifier upon success and `UINT_MAX` upon failure

Errors Thrown

Full instrumentation and error-checking libraries:

- `kUnknownThreadType` indicates that `inType` is not an element of the `ThreadType` type (see on page 4-50), as defined in `VDK.h`.
- `kThreadCreationFailure` indicates that the kernel is not able to allocate and/or initialize memory for the thread.

Non error-checking libraries: None

DestroyDeviceFlag()

C Prototype

```
void VDK_DestroyDeviceFlag(VDK_DeviceFlagID inDeviceFlagID);
```

C++ Prototype

```
void VDK::DestroyDeviceFlag(VDK::DeviceFlagID inDeviceFlagID);
```

Description

Deletes the device flag from the system and releases the associated memory

Parameters

`inDeviceFlagID` specifies the device flag to be destroyed.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kInvalidDeviceFlag` indicates that `inDeviceFlagID` is not a valid identifier.

Non error-checking libraries: None

DestroyMessage()

C Prototype

```
void VDK_DestroyMessage(VDK_MessageID inMessageID);
```

C++ Prototype

```
void VDK::DestroyMessage(VDK::MessageID inMessageID);
```

Description

Destroys a message object. Only the thread that is the owner of a message can destroy it. The message payload memory is assumed to be freed by the user thread. `DestroyMessage` does not free the payload and results in a memory leak if the memory is not freed.

Parameters

`inMessageID` is the identifier of the message to be destroyed.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageID` indicates the `inMessageID` is not a valid message identifier.
- `kInvalidMessageOwner` indicates the thread attempting to destroy the message is not the current owner.
- `kMessageInQueue` indicates the message is posted to a thread (see `ThreadID` on page 4-48) is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

DestroyMessageAndFreePayload()

C Prototype

```
void VDK_DestroyMessageAndFreePayload(
                                      VDK_MessageID inMessageID);
```

C++ Prototype

```
void VDK::DestroyMessageAndFreePayload(
                                      VDK::MessageID inMessageID);
```

Description

Destroys a message object. Only the thread that is the owner of a message can destroy it. If the payload is of a marshalled type (that is, the sign bit of the payload type code is set), then the payload is freed by calling the type marshalling function with the `RELEASE` code.

Parameters

`inMessageID` is the identifier of the message to be destroyed.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error0checking libraries:

- `kInvalidMessageID` indicates the `inMessageID` is not a valid message identifier.
- `kInvalidMessageOwner` indicates the thread attempting to destroy the message is not the current owner.
- `kMessageInQueue` indicates the message is posted to a thread (`ThreadID` on page 4-48) is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

ⓘ Other errors may be thrown by the user-supplied marshalling function, or by functions called by it.

DestroyPool()

C Prototype

```
void VDK_DestroyPool(VDK_PoolID inPoolID);
```

C++ Prototype

```
void VDK::DestroyPool(VDK::PoolID inPoolID);
```

Description

Deletes the pool and cleans up the memory associated with it. If there are any allocated blocks (which are not yet freed), an error is thrown in the fully instrumented and error-checking builds, and the pool is not destroyed. In the non error-checking build, the pool is destroyed.

Parameters

`inPoolID` specifies the pool to delete.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidPoolID` indicates `inPoolID` is not valid.
- `kErrorPoolNotEmpty` indicates the pool is not empty (there are some blocks that are not freed) and cannot be destroyed.

Non error-checking libraries: None

DestroySemaphore()

C Prototype

```
void VDK_DestroySemaphore(VDK_SemaphoreID inSemaphoreID);
```

C++ Prototype

```
void VDK::DestroySemaphore(VDK::SemaphoreID inSemaphoreID);
```

Description

Destroys the semaphore associated with `inSemaphoreID`. The destruction does not take place if there is any thread pending on the semaphore, resulting in an error thrown in full instrumentation and error-checking builds.

Parameters

`inSemaphoreID` is the semaphore to destroy.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kSemaphoreDestructionFailure` indicates the semaphore cannot be destroyed because there are threads pending on it.
- `kUnknownSemaphore` indicates `inSemaphoreID` is not a valid identifier.

Non error-checking libraries: None

DestroyThread()

C Prototype

```
void VDK_DestroyThread(VDK_ThreadID  inThreadID,
                       bool          inDestroyNow);
```

C++ Prototype

```
void VDK::DestroyThread(VDK::ThreadID  inThreadID,
                        bool           inDestroyNow);
```

Description

Initiates the process of removing the specified thread from the system. Although the scheduler never runs the thread again once this function completes, the kernel may optionally defer deallocation of the memory resources associated with the thread to the `Idle Thread`. Any references to the destroyed thread are invalid and may throw an error. For more information about the low-priority thread, see "Idle Thread" on page 3-16.

Parameters

`inThreadID` specifies the thread to remove from the system.

`inDestroyNow` indicates whether the thread's memory is recovered now (`TRUE`) or recovered in the low-priority `IDLE` thread (`FALSE`).

Scheduling

Invokes the scheduler and results in a context switch only if a thread passes itself to `DestroyThread`

Determinism

Constant time if `inDestroyNow` is `FALSE`; otherwise, not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kUnknownThread` indicates `inThreadID` is not a valid identifier.
- `kInvalidThread` indicates there is a intention to destroy the `Idle Thread`.

Non error-checking libraries: None

DeviceIOCtl()

C Prototype

```
int VDK_DeviceIOCtl(VDK_DeviceDescriptor inDD,
                    void                 *inCommand,
                    char                 *inParameters);
```

C++ Prototype

```
int VDK::DeviceIOCtl(VDK::DeviceDescriptor inDD,
                     void                  *inCommand,
                     char                  *inParameters);
```

Description

Controls the specified device. The `inCommand` and `inParameters` are passed unchanged to the device driver.

Parameters

`inDD` is the `DeviceDescriptor` (see on page 4-5) returned from `OpenDevice()` (see on page 5-112).

`inCommand` are the device driver's specific commands.

`inParameters` are the device driver's specific parameters for the above commands.

Scheduling

Does not invoke the scheduler, but the user-written device driver can call the scheduler

Determinism

Constant time.
Note this function calls user-written device driver code that may not be deterministic.

Return Value

Return value of the dispatch function if the device exists and `UINT_MAX` if it does not

Errors Thrown

Full instrumentation and error-checking libraries:
`kBadDeviceDescriptor` indicates that `inDD` is not a valid identifier.

Non error-checking libraries: None

Other errors may be thrown by user-written device driver code executed by this API.

DispatchThreadError()

C Prototype

```
int VDK_DispatchThreadError(VDK_SystemError  inErr,
                            const int        inVal);
```

C++ Prototype

```
int VDK::DispatchThreadError(VDK::SystemError  inErr,
                             const int         inVal);
```

Description

Sets the error and error's value in the currently running thread and calls the thread's error function.

Parameters

`inErr` is the error enumeration. See "SystemError" on page 4-42 for more information about errors.

`inVal` is the value whose meaning is determined by the error enumeration.

Scheduling

Does not invoke the scheduler, but the thread exception handler may do so

Determinism

Not deterministic

Return Value

The current thread's error handler

Errors Thrown

None

ForwardMessage()

C Prototype

```
void VDK_d(VDK_ThreadID inRecipient,
                        VDK_MessageID inMessageID,
                        VDK_MsgChannel inChannel,
                        VDK_ThreadID inPsuedoSender);
```

C++ Prototype

```
void VDK::ForwardMessage(VDK::ThreadID inRecipient,
                         VDK::MessageID inMessageID,
                         VDK::MsgChannel inChannel,
                         VDK::ThreadID inPsuedoSender);
```

Description

Identical to `PostMessage()` (see on page 5-131), except that the `Sender` attribute of the message is set to the value of the `inPsuedoSender` argument, instead of the ID of the current thread.

This function is useful where *message loopback* is being employed between two threads (that is, the received message is returned to the sender rather than being destroyed), and a third thread needs to be inserted transparently into the loop.

By querying the message's `sender` attribute (using `GetMessageReceiveInfo()` as shown on page 5-81), and then passing it as the `inPsuedoSender` argument to `ForwardMessage()` (see on page 5-58), this third thread can ensure that the message is returned to the original sender, rather than to itself.

Parameters

`inRecipient` is the `ThreadID` (see on page 4-48) of the thread receivimg the message.

`inMessageID` is the `MessageID` (see on page 4-28) of the sent message. A message must be created before it is posted. This parameter is a return value of the call to `CreateMessage()` (see on page 5-32).

`inChannel` is the FIFO within the recipient's message queue on which the message is appended. Its value is `kMsgChannel1` through `kMessageChannel15`.

`inPsuedoSender` is the `ThreadID` (see on page 4-48), which is stored in the `Sender` attribute of the message.

Scheduling

Non-blocking, but invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageChannel` indicates the `inChannel` is not a valid channel value.
- `kUnknownThread` indicates `inRecipient` is not a valid thread identifier.
- `kInvalidMessageID` indicates `inMessageID` is not a valid message identifier.

- `kInvalidMessageRecipient` indicates `inRecipient` does not have a message queue as it has not been enabled for messaging.
- `kInvalidMessageOwner` indicates the thread attempting to post the message is not the current owner. The error value is the `ThreadID` of the owner.
- `kMessageInQueue` indicates the message is posted to a thread (the `ThreadID` is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

FreeBlock()

C Prototype

```
void VDK_FreeBlock(VDK_PoolID inPoolID, void *inBlockPtr);
```

C Prototype

```
void VDK::FreeBlock(VDK::PoolID inPoolID, void *inBlockPtr);
```

Description

Frees the specified block and returns it to the free block list.

Parameters

`inPoolID` specifies the pool from which the block is to be freed.

`inBlockPtr` specifies the block to free.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidBlockPointer` indicates `inBlockPtr` is not a valid pointer from `inPoolID`.
- `kInvalidPoolID` indicates `inPoolID` is not a valid identifier.

Non error-checking libraries: None

FreeDestroyedThreads()

C Prototype

```
void VDK_FreeDestroyedThreads(void);
```

C++ Prototype

```
void VDK::FreeDestroyedThreads(void);
```

Description

Frees the memory held by the destroyed threads whose resources have not been released by the `IDLE` thread. For more information, see “Idle Thread” on page 3-16.

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

None

FreeMessagePayload ()

C Prototype

```
void VDK_FreeMessagePayload(VDK_MessageID inMessageID);
```

C++ Prototype

```
void VDK::FreeMessagePayload(VDK::MessageID inMessageID);
```

Description

If the payload of the specified message object is of a marshalled type (that is, the sign bit of the payload type code is set), then the payload is freed without destroying the message object itself. Only the thread that is the owner of a message can free its payload. The payload is freed by calling the type marshalling function with the `RELEASE` code.

The payload `Type`, `Size` and `Addr` attributes of the message object are all set to zero.

Parameters

`inMessageID` is the identifier of the message to be destroyed.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageID` indicates `inMessageID` is not a valid message identifier.
- `kInvalidMessageOwner` indicates the thread attempting to destroy the message is not the current owner.
- `kMessageInQueue` indicates that the message has been posted to a thread (the `ThreadID` is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

Note that other errors may be thrown by the user-supplied marshalling function, or by functions called by it.

FreeThreadSlot()

C Prototype

```
bool VDK_FreeThreadSlot(int inSlotNum);
```

C++ Prototype

```
bool VDK::FreeThreadSlot(int inSlotNum);
```

Description

Releases and clears the slot table entry in the currently running thread's slot table associated with `inSlotNum` and:

- Returns `FALSE` if (and only if) the key does not identify a currently allocated slot.
- Releases the slot identified by `inSlotNum`, which was previously created with the `AllocateThreadSlot()` function (see on page 5-19).
- The application must ensure that no thread local data is associated with the key at the time it is freed. Any specified cleanup functions (see `AllocateThreadSlotEx()` on page 5-21) are only called on thread destruction.

Parameters

`inSlotNum` is the static library's preallocated slot number.

Scheduling

Does not invoke the scheduler

Determinism

Constant Time

Return Value

`TRUE` upon success and `FALSE` upon failure

Errors Thrown

None

GetClockFrequency()

C Prototype

```
unsigned int VDK_GetClockFrequency (void);
```

C++ Prototype

```
unsigned int VDK::GetClockFrequency (void);
```

Description

Returns the value of the clock frequency for the application. The value of clock frequency is specified as part of the configuration of a VDK project and can be changed at runtime by `SetClockFrequency()` (see on page 5-140). It is the responsibility of the application designer to ensure that the clock frequency matches that of the hardware used.

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Value of the clock frequency

Errors Thrown

None

GetEventBitValue()

C Prototype

```
bool VDK_GetEventBitValue(VDK_EventBitID  inEventBitID);
```

C++ Prototype

```
bool VDK::GetEventBitValue(VDK::EventBitID  inEventBitID);
```

Description

Returns the value of the event bit.

Parameters

`inEventBitID` specifies the system event bit to query.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

The value of either the specified event bit (if such a bit exists) or the current thread's error handler (if the specified event bit does not exist)

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownEventBit` indicates that `inEventBitID` is not a valid identifier.

Non error-checking libraries: None

GetEventData()

C Prototype

```
VDK_EventData VDK_GetEventData(VDK_EventID inEventID);
```

C++ Prototype

```
VDK::EventData VDK::GetEventData(VDK::EventID inEventID);
```

Description

Returns the `EventData` associated with the queried event. Threads can use this function to get an event's current values. For more information, see "EventData" on page 4-17.

Parameters

`inEventID` is the event to query.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Data associated with the specified event bit if it exists, and a structure filled with zeros if it does not

Errors Thrown

Full instrumentation and error-checking libraries
`kUnknownEvent` indicates `inEventID` is not a valid event identifier.

Non error-checking libraries: None

GetEventValue()

C Prototype

```
bool VDK_GetEventValue(VDK_EventID inEventID);
```

C++ Prototype

```
bool VDK::GetEventValue(VDK::EventID inEventID);
```

Description

Returns the value of the specified event.

Parameters

`inEventID` specifies the event to query.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Value of the specified event if it exists, and the return value of the current thread's error handler if it does not

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownEvent` indicates `inEventID` is not a valid identifier.

Non error-checking libraries: None

GetHeapIndex()

C Prototype

```
unsigned int VDK_GetHeapIndex(VDK_HeapID inHeapID);
```

C++ Prototype

```
unsigned int VDK::GetHeapIndex(VDK::HeapID inHeapID);
```

Description

Translates a `HeapID` (as configured in the **Kernel** pane of the IDDE **Project** window) to a heap index, which can be passed to `heap_malloc()`, `heap_calloc()`, `heap_realloc()`, and `heap_free()`

Parameters

`inHeapID` is the `HeapID` (see on page 4-18) for the returned corresponding heap index.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Heap index

Errors Thrown

Full instrumentation and error-checking libraries:
`kInvalidHeapID` indicates that `inHeapID` is not a valid `HeapID`.

Non error-checking libraries: None

GetInterruptMask()

C Prototype

```
VDK_IMASKStruct VDK_GetInterruptMask(void);
```

C++ Prototype

```
VDK::IMASKStruct VDK::GetInterruptMask(void);
```

Description

Returns the current value of the interrupt mask

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Current value of the interrupt mask

Errors Thrown

None

GetInterruptMaskEx()

C Prototype

```
void VDK_GetInterruptMaskEx(VDK_IMASKStruct *outMask,
                            VDK_IMASKStruct *outLMask);
```

C++ Prototype

```
Void VDK::GetInterruptMaskEx(VDK::IMASKStruct *outMask,
                             VDK::IMASKStruct *outLMask);
```

Description

Returns the current value of the `IMASK` and `LMASK` interrupt masks in `*outMask` and `*outLMask` (where `LMASK` is the mask component of the `IRPTL` register). This API is applicable only to certain processors, currently those in the ADSP-2116x, ADSP-2126x, and ADSP-2136x processor families.

Parameters

`*outMask` is the current value of `IMASK`.

`*outLMask` is the current value of `LMASK`.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

GetLastThreadError()

C Prototype

```
VDK_SystemError VDK_GetLastThreadError(void);
```

C++ Prototype

```
VDK::SystemError VDK::GetLastThreadError(void);
```

Description

Returns the running thread's most recent error. See "SystemError" on page 4-42 for more information about errors.

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

The running thread's most recent error

Errors Thrown

None

GetLastThreadErrorValue()

C Prototype

```
int VDK_GetLastThreadErrorValue(void);
```

C++ Prototype

```
int VDK::GetLastThreadErrorValue(void);
```

Description

Returns the value parameter of the call with the most recent error

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

An additional descriptive value whose definition depends on the last error that has been dispatched. (See Table 5-20 on page 5-10 for further details.)

Errors Thrown

None

GetMessageDetails ()

C Prototype

```
void VDK_GetMessageDetails(VDK_MessageID inMessageID,
                  VDK_MessageDetails *pOutMessageDetails,
                  VDK_PayloadDetails *pOutPayloadDetails);
```

C++ Prototype

```
void VDK::GetMessageDetails(VDK::MessageID inMessageID,
                  VDK::MessageDetails *pOutMessageDetails,
                  VDK::PayloadDetails *pOutPayloadDetails);
```

Description

Returns the full set of attributes associated with a message object. The results are divided into details about the message itself (channel, sender and target), and about the payload (type, size and address).

The meaning of the message attributes corresponds to the arguments from the most recent posting of the message. The meaning of the payload values is application-specific and corresponds to the arguments passed to `CreateMessage()` (see on page 5-32).

Only the thread that is the owner of a message may examine the attributes of its payload. If other threads call this API, an error is thrown, and the contents of `*pOutMessageDetails` and `*pOutPayloadDetails` remain unchanged.

Parameters

The `inMessageID` specifies the message to query.

The `pOutMessageDetails` is a pointer to a structure of type `MessageDetails` (see on page 4-27), which contains channel, sender and target fields. Channel is of type `MsgChannel` (see on page 4-29), and sender and target are of type `ThreadID` (see on page 4-48). The `pOutMessageDetails` may be NULL, in which case no message details are returned.

The `pOutPayloadDetails` is a pointer of type `PayloadDetails` (see on page 4-35), which contains type, size and addr fields to describe the message payload. This information is the same as the one retrieved by `GetMessagePayload()` (see on page 5-79). The `pOutPayloadDetails` may be NULL, in which case no payload details are returned.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageID` indicates `inMessageID` is not a valid message identifier.
- `kInvalidMessageOwner` indicates the thread attempting to destroy the message is not the current owner.
- `kMessageInQueue` indicates the message is posted to a thread (the `ThreadID` is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

GetMessagePayload()

C Prototype

```
void VDK_GetMessagePayload(VDK_MessageID  inMessageID,
                           int            *outPayloadType,
                           unsigned int   *outPayloadSize,
                           void           **outPayloadAddr);
```

C++ Prototype

```
void VDK::GetMessagePayload(VDK::MessageID  inMessageID,
                            int             *outPayloadType,
                            unsigned int    *outPayloadSize,
                            void            **outPayloadAddr);
```

Description

Returns the attributes associated with a message payload—type, size and address.

The meaning of these values is application-specific and corresponds to the arguments passed to `CreateMessage()` (see on page 5-32). Only the thread that is the owner of a message may examine the attributes of its payload. If other threads call this API, an error is thrown, and the contents of `outPayloadType`, `outPayloadSize`, and `outPayloadAddress` remain unchanged.

Parameters

The `inMessageID` specifies the message to query.

The `*outPayloadType` is an application-specific value that may be used to describe the contents of the payload. Negative values of payload type are reserved for use by VDK.

The `*outPayloadSize` is typically the size of the payload in the smallest addressable units of the processor (`sizeof(char)`).

The `*outPayloadAddr` is typically a pointer to the beginning of the payload buffer. However, if the payload size has a value of zero, then the payload address may contain any user-defined data.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageOwner` indicates the argument `inMessageID` is not the current owner of the message.
- `kInvalidMessageID` indicates the argument `inMessageID` is not a valid message identifier.
- `kMessageInQueue` indicates that the message is posted to a thread (the `ThreadID` is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

GetMessageReceiveInfo()

C Prototype

```
void VDK_GetMessageReceiveInfo(VDK_MessageID   inMessageID,
                               VDK_MsgChannel  *outChannel,
                               VDK_ThreadID    *outSender);
```

C++ Prototype

```
void VDK::GetMessageReceiveInfo(VDK::MessageID   inMessageID,
                                VDK::MsgChannel  *outChannel,
                                VDK::ThreadID    *outSender);
```

Description

Returns the parameters associated with how a message is received.

Only the thread that is the owner of a message should call this API. If a different thread calls the API, there is an error thrown and the `outChannel` and `outSender` variables do not contain the right information.

Parameters

`inMessageID` specifies the message to query.

`*outChannel` identifies the channel of the recipient thread's message queue the message is posted on.

`*outSender` identifies the `ThreadID` of the thread that posted the message.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageOwner` indicates the argument `inMessageID` is not the current owner of the message.
- `kInvalidMessageID` indicates the argument `inMessageID` is not a valid message identifier.
- `kMessageInQueue` indicates that the message is posted to a thread (the `ThreadID` is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

GetNumAllocatedBlocks()

C Prototype

```
unsigned int VDK_GetNumAllocatedBlocks(VDK_PoolID inPoolID);
```

C++ Prototype

```
unsigned int VDK::GetNumAllocatedBlocks(VDK::PoolID inPoolID);
```

Description

Gets the number of allocated blocks in the pool

Parameters

`inPoolID` specifies the pool.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Number of allocated blocks for the specified pool upon success and `UINT_MAX` otherwise

Errors Thrown

Full instrumentation and error-checking libraries:
`kInvalidPoolID` indicates that `inPoolID` is not a valid identifier.

Non error-checking libraries: None

GetNumFreeBlocks()

C Prototype

```
unsigned int VDK_GetNumFreeBlocks(VDK_PoolID inPoolID);
```

C ++ Prototype

```
unsigned int VDK::GetNumFreeBlocks(VDK::PoolID inPoolID);
```

Description

Gets the number of free blocks in the pool

Parameters

`inPoolID` specifies the pool to query.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Number of free blocks for the specified pool upon success and `UINT_MAX` otherwise

Errors Thrown

Full instrumentation and error-checking libraries:
`kInvalidPoolID` indicates that `inPoolID` is not a valid identifier.

Non error-checking libraries: None

GetPriority()

C Prototype

```
VDK_Priority VDK_GetPriority(VDK_ThreadID inThreadID);
```

C++ Prototype

```
VDK::Priority VDK::GetPriority(VDK::ThreadID inThreadID);
```

Description

Returns the priority of the specified thread

Parameters

`inThreadID` is the thread whose priority is being queried.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Priority of the specified thread if it exists and `UINT_MAX` if it does not. See "Priority" on page 4-39 for more information.

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownThread` indicates that `inThreadID` is not a valid identifier.

Non error-checking libraries: None

GetSemaphoreValue()

C Prototype

```
unsigned int VDK_GetSemaphoreValue(VDK_SemaphoreID inSemaphoreID);
```

C++ Prototype

```
unsigned int VDK::GetSemaphoreValue(VDK::SemaphoreID inSemaphoreID);
```

Description

Returns the value of the specified semaphore

Parameters

`inSemaphoreID` is the semaphore to query.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Value of the specified semaphore if it exists and `UINT_MAX` if it does not

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownSemaphore` indicates that `inSemaphoreID` is not a valid identifier.

Non error-checking libraries: None

GetThreadHandle()

C Prototype

```
void** VDK_GetThreadHandle(void);
```

C++ Prototype

```
void** VDK::GetThreadHandle(void);
```

Description

Returns a pointer to a thread's user-defined, allocated data pointer. This pointer can be used in C and assembly threads for holding thread local state (for example, member variables).

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Pointer to a thread's user-defined, allocated data pointer

Errors Thrown

None

GetThreadID()

C Prototype

```
VDK_ThreadID VDK_GetThreadID(void);
```

C++ Prototype

```
VDK::ThreadID VDK::GetThreadID(void);
```

Description

Returns the identifier of the currently running thread

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Identifier of the currently running thread or VDK_KERNEL_LEVEL_ if the code is running at kernel level

Errors Thrown

None

GetThreadSlotValue()

C Prototype

```
void* VDK_GetThreadSlotValue(int inSlotNum);
```

C++ Prototype

```
void* VDK::GetThreadSlotValue(int inSlotNum);
```

Description

Returns the value in the currently running thread's slot table associated with `inSlotNum`. Returns `NULL` if the key does not identify a currently allocated slot, otherwise returns the current value held in the slot, which may also be `NULL`.

Parameters

`inSlotNum` is the static library's preallocated slot number.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Slot value for the slot number specified

Errors Thrown

None

GetThreadStackUsage()

C Prototype

```
unsigned int VDK_GetThreadStackUsage(VDK_ThreadID inThreadID);
```

C++ Prototype

```
unsigned int VDK::GetThreadStackUsage(VDK::ThreadID inThreadID);
```

Description

Gets the maximum used stack for the specified thread at the time of the call. For applications built with "Full Instrumentation", the maximum stack usage returned is either the amount used since the thread creation, or the last call to the `InstrumentStack()` API (see on page 5-100). For applications *not* built with "Full Instrumentation", the thread stacks are not instrumented by default. Therefore, this function does not return a meaningful value in these cases unless the `InstrumentStack()` API has previously been called to instrument the stack.

Parameters

`inThreadID` specifies the thread's stack usage to query.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

Maximum stack used since the thread was created (if the application was built with "Full Instrumentation") or since the last call to `InstrumentStack()`. Note that the stack is expressed in words.

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownThread` indicates that `inThreadID` is not a valid identifier.

Non error-checking libraries: None

GetThreadStack2Usage()

C Prototype

```
unsigned int VDK_GetThreadStack2Usage(VDK_ThreadID inThreadID);
```

C++ Prototype

```
unsigned int VDK::GetThreadStack2Usage(VDK::ThreadID
inThreadID);
```

Description

Gets the maximum used stack2 for the specified thread at the time of the call. For applications built with “Full Instrumentation”, the maximum stack usage returned is either the amount used since the thread creation, or the last call to the `InstrumentStack()` API (see on page 5-100). For applications *not* built with “Full Instrumentation”, the thread stacks are not instrumented by default. Therefore, this function does not return a meaningful value in these cases unless the `InstrumentStack()` API has previously been called to instrument the stack.

This API is applicable only to certain processors which use two stacks, currently those in the ADSP-TSxxx processor families

Parameters

`inThreadID` specifies the thread’s stack usage to query.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

Maximum stack2 used since the thread was created (if the application was built with "Full Instrumentation") or since the last call to `InstrumentStack()`. Note that the stack is expressed in words.

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownThread` indicates that `inThreadID` is not a valid identifier.

Non error-checking libraries: None

GetThreadStatus()

C Prototype

```
VDK_ThreadStatus VDK_GetThreadStatus(const VDK_ThreadID inThreadID);
```

C++ Prototype

```
VDK::ThreadStatus VDK::GetThreadStatus(const VDK::ThreadID inThreadID);
```

Description

Reports the enumerated status of the specified thread.

Parameters

`inThreadID` is the thread whose status is being queried.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Status of the specified thread if it exists and `VDK::kUnknown` if it does not. For more information, see "ThreadStatus" on page 4-49.

Errors Thrown

None

GetTickPeriod()

C Prototype

```
float VDK_GetTickPeriod (void);
```

C++ Prototype

```
float VDK::GetTickPeriod (void);
```

Description

Returns the value (in ms) of the tick period for the application

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Value (in ms) of the tick period

Errors Thrown

None

GetUptime()

C Prototype

```
VDK_Ticks VDK_GetUptime(void);
```

C++ Prototype

```
VDK::Ticks VDK::GetUptime(void);
```

Description

Returns the time in ticks since the last system reset

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Time in ticks since the last system reset

Errors Thrown

None

GetVersion()

C Prototype

```
VDK_VersionStruct VDK_GetVersion(void);
```

C++ Prototype

```
VDK::VersionStruct VDK::GetVersion(void);
```

Description

Returns the current version of the VDK, `VersionStruct`, described on page 4-52

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

Current version of VDK

Errors Thrown

None

InstallMessageControlSemaphore ()

C Prototype

```
void VDK_InstallMessageControlSemaphore(
                                VDK_SemaphoreID inSemaphore);
```

C++ Prototype

```
void VDK::InstallMessageControlSemaphore(
                                VDK::SemaphoreID inSemaphore);
```

Description

Sets up a counting semaphore to regulate the allocation and deallocation of message objects by the routing threads. The initial value of the semaphore should be set to the number of free messages reserved for use by the incoming routing threads. The semaphore is pended (by the incoming routing threads) prior to each message allocation, and posted (by the outgoing routing threads) after each message deallocation. Provided that the value of the semaphore is always less than or equal to the number of free messages, the allocation by the routing threads never fails (although the routing threads may block, pended on the semaphore) waiting for a free message to become available.

Parameters

`inSemaphore` is the identifier of the semaphore to be installed.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

InstrumentStack()

C Prototype

```
void VDK_InstrumentStack(void);
```

C++ Prototype

```
void VDK::InstrumentStack(void);
```

Description

Instruments the stack of the calling thread to allow the determination of maximum thread stack usage. The `GetThreadStackUsage()` API (see on page 5-90) is used to obtain the maximum stack usage for instrumented thread stacks.

If the fully instrumented libraries are used, the thread's stack is instrumented on creation. In this case, the `InstrumentStack()` API is used to reset the instrumentation of the stack to cover the currently unused section of the stack (for example, to determine the maximum stack used while executing a section of code). If the libraries without full instrumentation are used, the thread's stack is not instrumented by default and so `InstrumentStack()` has to be used to obtain meaningful results from `GetThreadStackUsage()`.

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

None

LoadEvent()

C Prototype

```
void VDK_LoadEvent(VDK_EventID          inEventID,
                   const VDK_EventData   inEventData);
```

C++ Prototype

```
void VDK::LoadEvent(VDK::EventID          inEventID,
                    const VDK::EventData  inEventData);
```

Description

Loads the `EventData` associated with the event. For more information, see "EventData" on page 4-17.

Parameters

`inEventID` is the event to be reinitialized.

`inEventData` contains the new values for the event.

Scheduling

Causes the value of the event to be recalculated, invokes the scheduler, and may result in a context switch

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownEvent` indicates that `inEventID` is not a valid identifier.

Non error-checking libraries: None

LocateAndFreeBlock()

C Prototype

```
void VDK_LocateAndFreeBlock(void *inBlkPtr);
```

C++ Prototype

```
void VDK::LocateAndFreeBlock(void *inBlkPtr);
```

Description

Determines the pool in which the to-be-freed block tesides, then frees the block, and returns it to the free block list

Parameters

`inBlockPtr` specifies the block to be freed.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

`kInvalidBlockPointer` indicates that `inBlkPtr` does not belong to any of the active memory pools and cannot be freed.

Non error-checking libraries: None

LogHistoryEvent()

C Prototype

```
void VDK_LogHistoryEvent(VDK_HistoryEnum inEnum, int inValue);
```

C++ Prototype

```
void VDK::LogHistoryEvent(VDK::HistoryEnum inEnum, int inValue);
```

Description

Adds a record to the history buffer. This function does not perform any action if the project is not linked with the fully instrumented libraries.

Parameters

`inEnum` is the enumeration value for this type of event. For more information, see "HistoryEnum" on page 4-19.

`inValue` is the value defined by enumeration.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

MakePeriodic()

C Prototype

```
void VDK_MakePeriodic(VDK_SemaphoreID  inSemaphoreID,
                      VDK_Ticks        inDelay,
                      VDK_Ticks        inPeriod);
```

C++ Prototype

```
void VDK::MakePeriodic(VDK::SemaphoreID  inSemaphoreID,
                       VDK::Ticks        inDelay,
                       VDK::Ticks        inPeriod);
```

Description

Directs the scheduler to post the specified semaphore after `inDelay` number of ticks. After every `inPeriod` ticks, the semaphore is posted and the scheduler is invoked. This allows the running thread to acquire the signal and, if the thread is at the highest priority level, to continue execution.

To be periodic, the running thread must repeat in sequence – perform task and then pend on the semaphore. Note that this differs from "sleeping" at the completion of activity.

Parameters

`inSemaphoreID` is the semaphore to make periodic.

`inDelay` is the number of ticks before the first posting of the semaphore. `inDelay` must be equal to or greater than one and less than `INT_MAX`.

`inPeriod` is the number of ticks to sleep at each cycle after the first cycle. `inPeriod` must be equal to or greater than one and less than `INT_MAX`.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kUnknownSemaphore` indicates `inSemaphoreID` is not a valid identifier.
- `kInvalidPeriod` indicates `inPeriod` is zero or greater than `INT_MAX`.
- `kInvalidDelay` indicates `inDelay` is zero or greater than `INT_MAX`. Zero is not an accepted delay because the first posting does not occur until the next tick.
- `kAlreadyPeriodic` indicates the semaphore is already periodic and cannot be made periodic again. If the intention is to change the period, the semaphore has to be made non-periodic first.

Non error-checking libraries: None

MallocBlock()

C Prototype

```
void* VDK_MallocBlock(VDK_PoolID inPoolID);
```

C++ Prototype

```
void* VDK::MallocBlock(VDK::PoolID inPoolID);
```

Description

Returns pointer to the next available block from the specified pool

Parameters:

`inPoolID` specifies the pool from which the block is to be allocated.

Scheduling:

Does not invoke the scheduler

Determinism

Constant time if `inCreateNow` was specified to be `TRUE` when the specified pool was created

Not deterministic if `inCreateNow` was specified to be `FALSE`

Return Value

Void pointer to a free memory block upon success; `NULL` if the call fails to allocate a block

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidPoolID` indicates `inPoolID` is not a valid identifier.
- `kErrorMallocBlock` indicates there are no free blocks in the pool, so a new block cannot be allocated.

Non error-checking libraries: None

MessageAvailable()

C Prototype

```
bool VDK_MessageAvailable(unsigned int inMessageChannelMask);
```

C++ Prototype

```
bool VDK::MessageAvailable(unsigned int inMessageChannelMask);
```

Description

Enables a thread to use a polling model (rather than a blocking model) to wait for messages in its message queue. This function returns `TRUE` if a subsequent call to `PendMessage()` (see on page 5-118) with the same channel mask does not block.

Parameters

`inMessageChannelMask` specifies the receive channels. A set bit corresponds to a receive channel, and a clear bit corresponds to a channel that is ignored.

If the `VDK::kMsgWaitForAll` flag is set in the channel mask, then the query operates with `AND` logic, rather than the default `OR` logic. By default, only one message—on any of the receive channels designated in the channel mask—is required for a true result. The `VDK::kMsgWaitForAll` flag requires at least one message be queued on each of the specified receive channels channel in order for the function to return `TRUE`.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

`TRUE` if there is a message available and `FALSE` if there is not.

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageChannel` indicates `inMessageChannelMask` is not a valid mask.
- `kInvalidThread` indicates the current thread does not have a message queue because it has not been enabled for messaging.

Non error-checking libraries: None

OpenDevice()

C Prototype

```
VDK_DeviceDescriptor VDK_OpenDevice(VDK_IOID    inIDNum,
                                    char        *inFlags);
```

C++ Prototype

```
VDK::DeviceDescriptor VDK::OpenDevice(VDK::IOID  inIDNum,
                                      char       *inFlags);
```

Description

Opens the specified device

Parameters

`inIDNum` is the boot I/O identifier.

`inFlags` is uninterpreted data passed through to the device being opened.

Scheduling

Does not call the scheduler, but the user-written device driver can call the scheduler

Determinism

Constant time.
Note that this function calls user-written device driver code that may not be deterministic.

Return Value

New device descriptor on success and UINT_MAX on failure

Errors Thrown

Full instrumentation and error-checking libraries:

- `kBadIOID` indicates `inIDNum` is not a valid `IOID` (see on page 4-22).
- `kOpenFailure` indicates no more devices can be open simultaneously.

Non error-checking libraries: None

ⓘ Other errors may be thrown by user-written device driver code executed by this API.

PendDeviceFlag()

C Prototype

```
bool VDK_PendDeviceFlag(VDK_DeviceFlagID  inFlagID,
                        VDK_Ticks         inTimeout);
```

C++ Prototype

```
bool VDK::PendDeviceFlag(VDK::DeviceFlagID  inFlagID,
                         VDK::Ticks         inTimeout);
```

Description

Allows a thread to block on a specified device flag.

The thread is blocked and swapped out. Once the device flag is made available via `PostDeviceFlag()` (see on page 5-130), *all* threads waiting for this flag are made ready-to-run. If the thread does not resume execution within `inTimeout` ticks, the thread's reentry point is changed to its error function, and the thread is made available for scheduling. This behavior can be changed by ORing the timeout with the constant `VDK_kNoTimeoutError` in C or `VDK::kNoTimeoutError` in C++. In this case, no errors are dispatched on timeout and the API simply returns after making the thread available for scheduling. If the value of `inTimeout` is passed as zero, then the thread may pend indefinitely.

Note that `PendDeviceFlag()` (see on page 5-114) must be called from within a non-nested critical region (a critical region with a stack depth of one), but from outside of any unscheduled regions (as explained in "Pending on a Device Flag" on page 3-67. The `PendDeviceFlag()` pops one level of the critical region stack.

Parameters

`inFlagID` is the device flag on which the thread pends.

`inTimeout` is a value less than `INT_MAX` that specifies the maximum duration in ticks for which the thread pends on the device flag.

Scheduling

Invokes the scheduler

Determinism

Not deterministic

Return Value

`TRUE` if the device flag has been successfully pended on, `FALSE` if the `PendDeviceFlag()` call has timed out but `kNoTimeoutError` was specified.

Errors Thrown

Full instrumentation and error-checking libraries:

- `kBlockInInvalidRegion` indicates `PendDeviceFlag()` is called in an unscheduled region or nested critical region (must be called in a non-nested critical region).
- `kInvalidDeviceFlag` indicates `inFlagID` is not a valid identifier.
- `kDeviceTimeout` indicates the timeout value has expired before the device flag was posted. This error is not dispatched if the timeout was ORed with the constant `kNoTimeoutError`.
- `kInvalidTimeout` indicates `inTimeout` is either `INT_MAX` or `UINT_MAX`.

Non error-checking libraries:
`kDeviceTimeout` as above

PendEvent()

C Prototype

```
bool VDK_PendEvent(VDK_EventID inEventID, VDK_Ticks inTimeout);
```

C++ Prototype

```
bool VDK::PendEvent(VDK::EventID inEventID, VDK::Ticks inTimeout);
```

Description

Provides the mechanism by which threads pend on events.

If the named event calculates as being available, execution returns to the running thread. If the event is *not* available, the thread pauses execution until the event is available. When the event becomes available, *all* threads pending on the event are moved to the ready queue.

If the thread does not resume execution within `inTimeout` ticks, the thread's reentry point is changed to its error function, and the thread is made available for scheduling. This behavior can be changed by ORing the timeout with the constant `VDK_kNoTimeoutError` in C or `VDK::kNoTimeoutError` in C++. In this case, no errors are dispatched on timeout and the API simply returns after making the thread available for scheduling. If the value of `inTimeout` is passed as zero, then the thread may pend indefinitely.

Parameters

`inEventID` is the event on which the thread pends.

`inTimeout` is a value less than `INT_MAX` that specifies the maximum duration in ticks for which the thread pends on the event.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Constant time if event is available

Return Value

`TRUE` if the event has been successfully pended on, `FALSE` if the `PendEvent()` call has timed out but `kNoTimeoutError` was specified

Errors Thrown

Full instrumentation and error-checking libraries:

- `kEventTimeout` indicates the timeout value has expired before the event was available. This error is not dispatched if the timeout was `OR`ed with the constant `kNoTimeoutError`.
- `kUnknownEvent` indicates `inEventID` is not a valid identifier.
- `kBlockInInvalidRegion` indicates `PendEvent` (see on page 5-116) is trying to block in an unscheduled region, causing a scheduling conflict.
- `kDbgPossibleBlockInRegion` indicates `PendEvent` is being called in an unscheduled region, causing a potential scheduling conflict.
- `kInvalidTimeout` indicates `inTimeout` is either `INT_MAX` or `UINT_MAX`.

Non error-checking libraries:

`kEventTimeout` as above

PendMessage()

C Prototype

```
VDK_MessageID VDK_PendMessage(unsigned int inMessageChannelMask,
                              VDK_Ticks    inTimeout);
```

C++ Prototype

```
VDK::MessageID VDK::PendMessage(unsigned int inMessageChannelMask,
                                VDK::Ticks   inTimeout);
```

Description

Retrieves a message from a thread's message queue. The `PendMessage` is a blocking call—when the specified conditions for a valid message in the queue are not met, the thread suspends execution. The channel mask allows you to specify which channels (`kMsgChannel1` through `kMsgChannel15`) to examine for incoming messages.

In addition, the flag `VDK::kMsgWaitForAll` may be included in (`OR`ed into) the channel mask to specify that at least one message must be present on each of the channels specified in the mask. Messages are retrieved from the lowest numbered channels first (`kMsgChannel1`, then `kMsgChannel2`, ...). Once a `MessageID` is returned by `PendMessage`, the message is no longer in the queue and is owned by the calling thread. If the thread does not resume execution within `inTimeout` ticks, the thread's reentry point is changed to its error function, and the thread is made available for scheduling. This behavior can be changed by ORing the timeout with the constant `VDK_kNoTimeoutError` in C or `VDK::kNoTimeoutError` in C++. In this case, no errors are dispatched on timeout and the API simply returns after making the thread available for scheduling. If the value for `inTimeout` is passed as zero, then the thread may pend indefinitely.

Parameters

`inMessageChannelMask` specifies the receive channels. A set bit corresponds to a receive channel. A clear bit corresponds to a channel that is ignored. The parameter may not be zero.

If the `VDK::kMsgWaitForAll` flag is set in the channel mask, then the pend operates with `AND` logic, rather than the default `OR` logic. By default, only one message —on any of the receive channels designated in the channel mask—is required to unblock the pending thread. The `VDK::kMsgWaitForAll` flag requires at least one message to be queued on each of the specified receive channels channel in order to unblock.

`inTimeout` is a value less than `INT_MAX` that specifies the maximum duration in ticks for which the thread pends on the receipt of the required message(s).

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Constant time if there is no need to block

Return Value

Identifier of the message the thread pended on upon success; `UINT_MAX` otherwise

Errors Thrown

Full instrumentation and error-checking libraries:

- `kDbgPossibleBlockInRegion` indicates `PendMessage` (on page 5-118) is being called in an unscheduled region, causing a potential scheduling conflict.

- `kInvalidMessageChannel` indicates `inMessageChannelMask` does not specify a correct group of channels to mask.
- `kMessageTimeout` indicates the timeout value has expired before the thread removed the message from its message queue. This error is not dispatched if the timeout was `OR`ed with the constant `kNoTimeoutError`.
- `kBlockInInvalidRegion` indicates `PendMessage` is trying to block in an unscheduled region, causing a scheduling conflict.
- `kInvalidTimeout` indicates `inTimeout` is either `INT_MAX` or `UINT_MAX`.
- `kInvalidThread` indicates the current thread does not have a message queue as it has not been enabled for messaging.

Non error-checking libraries:
`kMessageTimeout` as above

PendSemaphore()

C Prototype

```
bool VDK_PendSemaphore(VDK_SemaphoreID    inSemaphoreID,
                       VDK_Ticks          inTimeout);
```

C++ Prototype

```
bool VDK::PendSemaphore(VDK::SemaphoreID   inSemaphoreID,
                        VDK::Ticks         inTimeout);
```

Description

Provides the mechanism that allows threads to pend on semaphores.

If the named semaphore is available (its count is greater than zero), the semaphore's count is decremented by one, and processor control returns to the running thread. If the semaphore is *not* available (its count is zero), the thread pauses execution until the semaphore is posted. If the thread does not resume execution within `inTimeout` ticks, the thread's reentry point is changed to its error function, and the thread is made available for scheduling. This behavior can be changed by ORing the timeout with the constant `VDK_kNoTimeoutError` in C or `VDK::kNoTimeoutError` in C++. In this case, no errors are dispatched on timeout and the API simply returns after making the thread available for scheduling. If the value of `inTimeout` is passed as zero, then the thread may pend indefinitely.

Parameters

`inSemaphoreID` is the semaphore on which the thread pends.

`inTimeout` is a value less than `INT_MAX` that specifies the maximum duration in ticks for which the thread pends on the semaphore.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Constant time if semaphore is available

Return Value

`TRUE` if the semaphore has been successfully pended on, `FALSE` if the `PendSemaphore()` call has timed out but `kNoTimeoutError` was specified

Errors Thrown

Full instrumentation and error-checking libraries:

- `kSemaphoreTimeout` indicates that the timeout value has expired before the semaphore became available. This error will not be dispatched if the timeout was ORed with the constant `kNoTimeoutError`.
- `kUnknownSemaphore` indicates `inSemaphoreID` is not a valid identifier.
- `kBlockInInvalidRegion` indicates `PendSemaphore()` (see on page 5-121) is called in an unscheduled region, causing a scheduling conflict.
- `kDbgPossibleBlockInRegion` indicates `PendSemaphore()` may be called in an unscheduled region, causing a potential scheduling conflict.
- `kInvalidTimeout` indicates `inTimeout` is either `INT_MAX` or `UINT_MAX`.

Non error-checking libraries:
`kSemaphoreTimeout` as above

PopCriticalRegion()

C Prototype

```
void VDK_PopCriticalRegion(void);
```

C++ Prototype

```
void VDK::PopCriticalRegion(void);
```

Description

Decrements the count of nested critical regions. Use it as a close bracket call to `PopCriticalRegion` (see on page 5-136). A count is maintained to ensure that each entered critical region calls `PopCriticalRegion` before interrupts are re-enabled. The kernel ignores additional calls to `PopCriticalRegion` while interrupts are enabled.

Each critical region is also (implicitly) an unscheduled region.

Parameters

None

Scheduling

Invokes the scheduler and may result in a context switch if interrupts are re-enabled by this call

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kDbgPopUnderflow` indicates that there were no critical regions to pop.

Non error-checking libraries: None

PopNestedCriticalRegions()

C Prototype

```
void VDK_PopNestedCriticalRegions(void);
```

C++ Prototype

```
void VDK::PopNestedCriticalRegions(void);
```

Description

Resets the count of nested critical regions to zero, thereby, re-enabling interrupts. The kernel ignores additional calls to `PopNestedCriticalRegions` (see on page 5-125) while interrupts are enabled.

This function does not change the interrupt mask.

Parameters

None

Scheduling

Invokes the scheduler and may result in a context switch (unless interrupts are already enabled)

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kDbgPopUnderflow` indicates there are no critical regions to pop.

Non error-checking libraries: None

PopNestedUnscheduledRegions()

C Prototype

```
void VDK_PopNestedUnscheduledRegions(void);
```

C++ Prototype

```
void VDK::PopNestedUnscheduledRegions(void);
```

Description

Resets the count of nested unscheduled regions to zero, thereby, re-enabling scheduling. The kernel ignores additional calls to `PopNestedUnscheduledRegions` (see on page 5-127) while scheduling is enabled.

Parameters

None

Scheduling

Invokes the scheduler and may result in a context switch (unless scheduling is already enabled)

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kDbgPopUnderflow` indicates there were no critical regions to pop.

Non error-checking libraries: None

PopUnscheduledRegion()

C Prototype

```
void VDK_PopUnscheduledRegion(void);
```

C++ Prototype

```
void VDK::PopUnscheduledRegion(void);
```

Description

Decrements the count of nested unscheduled regions. Use it as a close bracket call to `PopUnscheduledRegion` (see on page 5-137). A nesting count is maintained to ensure that each entered unscheduled region calls `PopUnscheduledRegion` before scheduling is resumed.

The kernel ignores additional calls to `PopUnscheduledRegion` while scheduling is enabled.

Parameters

None

Scheduling

Invokes the scheduler and may result in a context switch if scheduling is reenabled by this call

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kDbgPopUnderflow` indicates there are no critical regions to pop.

Non error-checking libraries: None

PostDeviceFlag()

C Prototype

```
void VDK_PostDeviceFlag(VDK_DeviceFlagID inFlagID);
```

C++ Prototype

```
void VDK::PostDeviceFlag(VDK::DeviceFlagID inFlagID);
```

Description

Posts the specified device flag. Once the device flag is made available, *all* threads waiting for the flag are in the ready-to-run state.

Parameters

`inFlagID` is the `DeviceDescriptor` (see on page 4-5) returned from `OpenDevice()` (see on page 5-112)

Scheduling

Invokes the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

`kInvalidDeviceFlag` indicates `inFlagID` is not a valid identifier.

Non error-checking libraries: None

PostMessage()

C Prototype

```
void VDK_PostMessage(VDK_ThreadID        inRecipient,
                     VDK_MessageID       inMessageID,
                     VDK_MsgChannel      inChannel);
```

C++ Prototype

```
void VDK::PostMessage(VDK::ThreadID      inRecipient,
                      VDK::MessageID     inMessageID,
                      VDK::MsgChannel    inChannel);
```

Description

Appends the message `inMessageID` to the message queue of the thread with identifier `inRecipient` on the channel `inChannel`. The `PostMessage` (see on page 5-131) is a non-blocking function—returns execution to the calling thread without waiting for the recipient to run or to acknowledge the new message in its queue. The message is considered delivered when `PostMessage` returns. Only the thread that is the owner of a message may post it.

At delivery time, ownership of the message and the associated payload is transferred from the sending thread to the recipient thread. Once delivered, all memory references to the payload, which may be held by the sending thread, are invalid. Memory read and write privileges and the responsibility for freeing the payload memory are passed to the recipient thread along with ownership.

Parameters

`inRecipient` is the `ThreadID` of the thread receiving the message.

`inMessageID` is the `MessageID` (see on page 4-28) of the message being sent. A message must be created before it is posted. This parameter is a return value of the call to `CreateMessage()` (see on page 5-32).

`inChannel` is the FIFO within the recipient's message queue on which the message is appended. Its value is `kMsgChannel1` through `kMessageChannel15`.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageChannel` indicates the `inChannel` is not a channel value.
- `kUnknownThread` indicates `inRecipient` is not a valid thread identifier.
- `kInvalidMessageID` indicates `inMessageID` is not a valid message identifier.

- `kInvalidMessageRecipient` indicates `inRecipient` does not have a message queue because it has not been enabled for messaging.
- `kInvalidMessageOwner` indicates the thread attempting to post the message is not the current owner. The error value is the `ThreadID` (see on page 4-48) of the owner.
- `kMessageInQueue` indicates that the message is posted to a thread (the `ThreadID` is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118)

Non error-checking libraries: None

PostSemaphore()

C Prototype

```
void VDK_PostSemaphore(VDK_SemaphoreID inSemaphoreID);
```

C++ Prototype

```
void VDK::PostSemaphore(VDK::SemaphoreID inSemaphoreID);
```

Description

Provides the mechanism by which threads post semaphores. Every time a semaphore is posted, its count increases by one until it reaches the maximum value, as specified on creation. Any further posts have no effect. Note that Interrupt Service Routines (ISRs) must use a different interface, as described in "VDK_ISR_POST_SEMAPHORE_()" on page 5-166.

Parameters

`inSemaphoreID` is the semaphore to post.

Scheduling

May invoke the scheduler and may result in a context switch

Determinism

- No thread pending: Constant time
- Low priority thread pending: Constant time
- High priority thread pending: Constant time plus a context switch

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownSemaphore` indicates that `inSemaphoreID` is not a valid identifier.

Non error-checking libraries: None

PushCriticalRegion()

C Prototype

```
void VDK_PushCriticalRegion(void);
```

C++ Prototype

```
void VDK::PushCriticalRegion(void);
```

Description

Disables interrupts to enable atomic execution of a critical region of code. Note that critical regions may be nested. A count is maintained to ensure a coequal number of calls to `PopCriticalRegion()` (see on page 5-123) are made before restoring interrupts. Each critical region is also (implicitly) an unscheduled region.

Parameters

None

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

PushUnscheduledRegion()

C Prototype

```
void VDK_PushUnscheduledRegion(void);
```

C++ Prototype

```
void VDK::PushUnscheduledRegion(void);
```

Description

Disables the scheduler. While in an unscheduled region, the current thread does not become de-scheduled, even if a higher-priority thread becomes ready to run. Note that unscheduled regions may be nested. A count is maintained to ensure a coequal number of calls to `PopUnscheduledRegion()` (see on page 5-128) are made before scheduling is re-enabled.

Scheduling

Suspends scheduling until a matching `PopUnscheduledRegion()` call

Parameters

None

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

RemovePeriodic()

C Prototype

```
void VDK_RemovePeriodic(VDK_SemaphoreID inSemaphoreID);
```

C++ Prototype

```
void VDK::RemovePeriodic(VDK::SemaphoreID inSemaphoreID);
```

Description

Stops periodic posting of the specified semaphore. Trying to stop a non-periodic semaphore has no effect and raises a run-time error.

Parameters

`inSemaphoreID` is the semaphore for which periodic posting is halted.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kUnknownSemaphore` indicates `inSemaphoreID` is not a valid identifier.
- `kNonperiodicSemaphore` indicates the semaphore is not periodic.

Non error-checking libraries: None

ResetPriority()

C Prototype

```
void VDK_ResetPriority(const VDK_ThreadID inThreadID);
```

C++ Prototype

```
void VDK::ResetPriority(const VDK::ThreadID inThreadID);
```

Description

Restores the priority of the named thread to the default value specified in the thread's template

Parameters

`inThreadID` is the thread whose priority is reset.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kUnknownThread` indicates `inThreadID` is not a valid identifier.
- `kInvalidThread` indicates `inThreadID` specified the `IdleThread`.

Non error-checking libraries: None

SetClockFrequency()

C Prototype

```
void VDK_SetClockFrequency (unsigned int inFrequency);
```

C++ Prototype

```
void VDK::SetClockFrequency (unsigned int inFrequency);
```

Description

Sets the clock frequency to `inFrequency`. The clock is stopped, the clock parameters are recalculated using the new value for clock frequency, and then the clock is restarted. It is the responsibility of the application designer to ensure that the clock frequency matches that of the hardware being used.

Parameters

`inFrequency` is the new value for the clock frequency.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

None

SetEventBit()

C Prototype

```
void VDK_SetEventBit(VDK_EventBitID inEventBitID);
```

C++ Prototype

```
void VDK::SetEventBit(VDK::EventBitID inEventBitID);
```

Description

Sets the value of the event bit. Once the event bit is set (meaning its value equals to `TRUE`, `1`, occurred, and so on), the value of all dependent events is recalculated.

If several event bits are set (or cleared) as a single operation, then the `SetEventBit()` (see on page 5-141) and/or `ClearEventBit()` (see on page 5-23) calls should be made from within an unscheduled region. Event recalculation does not occur until the unscheduled region is popped.

Parameters

`inEventBitID` is the system event bit to set.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:
`kUnknownEventBit` indicates `inEventBitID` is not a valid identifier.

Non error-checking libraries: None

SetInterruptMaskBits()

C Prototype

```
void VDK_SetInterruptMaskBits(VDK_IMASKStruct inMask);
```

C++ Prototype

```
void VDK::SetInterruptMaskBits(VDK::IMASKStruct inMask);
```

Description

Sets bits in the interrupt mask. Any bits set in the parameter are set in the interrupt mask. In other words, the new mask is computed as the bitwise `OR` of the old mask and the parameter `inMask`.

Parameters

`inMask` specifies which bits should be set in the interrupt mask.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

SetInterruptMaskBitsEx()

C Prototype

```
void VDK_SetInterruptMaskBitsEx(VDK_IMASKStruct inMask,
                                VDK_IMASKStruct inLMask);
```

C++ Prototype

```
void VDK::SetInterruptMaskBitsEx(VDK::IMASKStruct inMask,
                                 VDK::IMASKStruct inLMask);
```

Description

Sets bits in the `IMASK` and `LMASK` interrupt masks (where `LMASK` is the mask component of the `IRPTL` register). Any bits set in the parameters are set in the relevant interrupt mask. In other words, the new masks are computed as the bitwise `OR` of the old mask and the specified `IMASKStruct`. The bits specified for `inLMask` are shifted by the relevant amount for the processor in question, thereby allowing the use of the bit definitions for the `LIRPTL` register.

This API is applicable only to certain processors, currently those in the ADSP-2116x, ADSP-2126x, and ADSP-2136x processor families.

Parameters

`inMask` specifies which bits should be set in the `IMASK` interrupt mask.

`inLMask` specifies which bits should be set in the `LMASK` interrupt mask.

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

`kInvalidMaskBit` if any of the bits in `inLmask` do not refer to an interrupt mask.

Non error-checking libraries: None

SetMessagePayload()

C Prototype

```
void VDK_SetMessagePayload(VDK_MessageID   inMessageID,
                           int             inPayloadType,
                           unsigned int    inPayloadSize,
                           void           *inPayloadAddr);
```

C++ Prototype

```
void VDK::SetMessagePayload(VDK::MessageID  inMessageID,
                            int             inPayloadType,
                            unsigned int    inPayloadSize,
                            void           *inPayloadAddr);
```

Description

Sets the values in a message header that describes the payload. The meaning of these values is application-specific and corresponds to the arguments passed to `CreateMessage()` (see on page 5-32). This function overwrites the existing values in the message. Only the thread that is the owner of a message may set the attributes of its payload.

Parameters

`inMessageID` specifies the message to be modified.

`inPayloadType` is an application-specific value that may be used to describe the contents of the payload. Negative values of payload type are reserved for use by VDK.

`inPayloadSize` indicates the size of the payload in the smallest addressable units of the processor `(sizeof(char))`.

`inPayloadAddr` is (typically) a pointer to the beginning of the payload buffer.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kInvalidMessageOwner` indicates the running thread is not the current owner of the message.
- `kInvalidMessageID` indicates the argument `inMessageID` is not a valid message identifier.
- `kMessageInQueue` indicates the message is posted to a thread (the `ThreadID` is not known at this point), and it needs to be removed from the message queue by a call to `PendMessage()` (see on page 5-118).

Non error-checking libraries: None

SetPriority()

C Prototype

```
void VDK_SetPriority(const VDK_ThreadID    inThreadID,
                     const VDK_Priority    inPriority);
```

C++ Prototype

```
void VDK::SetPriority(const VDK::ThreadID   inThreadID,
                      const VDK::Priority   inPriority);
```

Description

Dynamically sets the priority of the named thread while overriding the default value. All threads are given an initial priority level at creation time. The thread's template specifies the priority initial value.

Parameters

`inPriority` is the new priority level. The `Priority` data type is described on page 4-39.

`inThreadID` is the thread to modify.

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kUnknownThread` indicates `inThreadID` is not a valid identifier.
- `kInvalidPriority` indicates `inPriority` is not a priority of the `Priority` data type.
- `kInvalidThread` indicates `inThreadID` specified the `IdleThread`.

Non error-checking libraries: None

SetThreadError()

C Prototype

```
void VDK_SetThreadError(VDK_SystemError inErr, int inVal);
```

C++ Prototype

```
void VDK::SetThreadError(VDK::SystemError inErr, int inVal);
```

Description

Sets the running thread's error value.

Parameters

`inErr` is the error enumeration. See "SystemError" on page 4-42 for more information about errors.

`inVal` is the value whose meaning is determined by the error enumeration.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

No return value

Errors Thrown

None

SetThreadSlotValue()

C Prototype

```
bool VDK_SetThreadSlotValue(int inSlotNum, void *inValue);
```

C++ Prototype

```
bool VDK::SetThreadSlotValue(int inSlotNum, void *inValue);
```

Description

Sets the value in the currently running thread's slot table associated with `inSlotNum`. Returns `FALSE` if (and only if) `inSlotNum` does not identify a currently allocated slot. Otherwise, stores `inValue` in the thread slot identified by `inSlotNum` and returns `TRUE`.

Parameters

`inSlotNum` is the static library's preallocated slot number.

`inValue` is the value to store in thread's slot table (at `inSlotNum` entry).

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Return Value

`FALSE` if `inSlotNum` does not identify a currently allocated slot and `TRUE` otherwise

Errors Thrown

None

SetTickPeriod()

C Prototype

```
void VDK_SetTickPeriod (float inPeriod);
```

C++ Prototype

```
void VDK::SetTickPeriod (float inPeriod);
```

Description

Sets the tick period to `inPeriod`. The clock is stopped, the clock parameters are recalculated using the new value for tick period, and then the clock is restarted.

Parameters

`inPeriod` is the new value for the tick period (in ms).

Scheduling

Does not invoke the scheduler

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

None

Sleep()

C Prototype

```
void VDK_Sleep(VDK_Ticks inSleepTicks);
```

C++ Prototype

```
void VDK::Sleep(VDK::Ticks inSleepTicks);
```

Description

Causes a thread to pause execution for at least the given number of clock ticks. Once delay ticks have elapsed, the calling thread is in the ready-to-run state. The thread resumes execution only if it is the highest priority thread with ready status. The minimum delay is one.

Parameters

`inSleepTicks` is a value less than `INT_MAX` that specifies the duration (in ticks) for the thread “sleep”.

Scheduling

Invokes the scheduler and results in a context switch.

Determinism

Not deterministic

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

- `kBlockInInvalidRegion` indicates `Sleep()` (see on page 5-153) is being called in an unscheduled region, causing a scheduling conflict.
- `kInvalidDelay` indicates `inSleepTicks` is not within the valid range of 1 to (`INT_MAX -1`).

Non error-checking libraries: None

SyncRead()

C Prototype

```
unsigned int VDK_SyncRead(VDK_DeviceDescriptor  inDD,
                          char                  *outBuffer,
                          const unsigned int    inSize,
                          VDK_Ticks             inTimeout);
```

C++ Prototype

```
unsigned int VDK::SyncRead(VDK::DeviceDescriptor  inDD,
                           char                   *outBuffer,
                           const unsigned int     inSize,
                           VDK::Ticks             inTimeout);
```

Description

Invokes the read functionality of the driver

Parameters

`inDD` is the `DeviceDescriptor` (see on page 4-5) returned from `OpenDevice()` (see on page 5-112).

`outBuffer` is the address of the data buffer filled by the device.

`inSize` is the number of words tread from the device.

`inTimeout` is the number of ticks before timeout occurs.

Scheduling

Does not call the scheduler, but the user-written device driver can call the scheduler

Determinism

Constant time.
Note that this function calls user-written device driver code that may not be deterministic.

Return Value

Return value of the dispatch function if the device exists and `UINT_MAX` if it does not

Errors Thrown

Full instrumentation and error-checking libraries:

`kBadDeviceDescriptor` indicates that `inDD` is not a valid `DeviceDescriptor`.

Non error-checking libraries: None

Other errors may be thrown by user-written device driver code executed by this API.

SyncWrite()

C Prototype

```
unsigned int VDK_SyncWrite(VDK_DeviceDescriptor  inDD,
                           char                  *outBuffer,
                           const unsigned int    inSize,
                           VDK_Ticks             inTimeout);
```

C++ Prototype

```
unsigned int VDK::SyncWrite(VDK::DeviceDescriptor  inDD,
                            char                   *outBuffer,
                            const unsigned int     inSize,
                            VDK::Ticks             inTimeout);
```

Description

Invokes the write functionality of the driver.

Return Value

Return value of the dispatch function if the device exists and `UINT_MAX` if it does not

Parameters

`inDD` is the `DeviceDescriptor` (see on page 4-5) returned from `OpenDevice()` (see on page 5-112).

`outBuffer` is the address of the data buffer read by the device.

`inSize` is the number of words written to the device.

`inTimeout` is the number of ticks before timeout occurs.

Scheduling

Does not call the scheduler, but the use- written device driver can call the scheduler

Determinism

Constant time. Note that this function calls user-written device driver code that may not be deterministic.

Errors Thrown

Full instrumentation and error-checking libraries:

`kBadDeviceDescriptor` indicates that `inDD` is not a valid `DeviceDescriptor`.

Non error-checking libraries: None

Other errors may be thrown by user-written device driver code executed by this API.

Yield()

C Prototype

```
void VDK_Yield(void);
```

C++ Prototype

```
void VDK::Yield(void);
```

Description

Yields control of the processor and moves the thread to the end of the wait queue of threads at its priority level. When `Yield()` (see on page 5-159) is called from a thread at a priority level using round-robin multithreading, the call also yields the remainder of the thread's time slice.

Parameters

None

Scheduling

Invokes the scheduler and may result in a context switch

Determinism

Constant time and conditional context switch

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

`kBlockInInvalidRegion` indicates that `Yield()` is called in an unscheduled region, causing a scheduling conflict.

Non error-checking libraries: None

Assembly Macros and C/C++ ISR APIs

This section describes the assembly-language macros and C/C++ ISR APIs that allow Interrupt Service Routines (ISRs) to communicate with VDK. These are known collectively as the ISR API and are the only part of VDK that can be safely called from interrupt level.

In VDK applications, ISRs should execute quickly and should leave as much of the processing as possible to be performed either by a thread or by a device driver activation. The principle purpose of the ISR API is therefore to provide the means to initiate such (thread or driver) activity.

In all previous VisualDSP++ releases VDK only provided direct support for the writing of ISRs in assembly language. However, the VisualDSP++ 4.0 release introduces VDK support for the writing of ISRs in C/C++.

Writing ISRs in assembly eliminates the overhead of saving and restoring the processor state, and of setting up a C run-time environment for each ISR entry. Assembly ISRs are responsible for saving and restoring any registers that they use. No assumptions can be made about the processor state at the time of entry to an ISR. Each ISR assembly macro saves and restores all of the registers that it uses, and the macros are safe to use with nested interrupts enabled.

The ISR assembly macros are:

- "VDK_ISR_ACTIVATE_DEVICE_()" on page 5-163
- "VDK_ISR_CLEAR_EVENTBIT_()" on page 5-164
- "VDK_ISR_LOG_HISTORY_()" on page 5-165
- "VDK_ISR_POST_SEMAPHORE_()" on page 5-166
- "VDK_ISR_SET_EVENTBIT_()" on page 5-167

Writing ISRs in C/C++ may simplify the coding of these routines. However, it must be remembered that there is a performance cost associated with executing ISRs that have been written in C/C++. Additionally, it should be noted that the majority of the run-time library is not interrupt-safe and so can not be used in ISRs (See the *VisualDSP++ 4.0 C/C++ Compiler and Libraries Manual* for details).

The VDK APIs callable from C/C++ ISRs are:

- "C_ISR_ActivateDevice()" on page 5-168
- "C_ISR_ClearEventBit()" on page 5-170
- "C_ISR_PostSemaphore()" on page 5-171
- "C_ISR_SetEventBit()" on page 5-173

VDK_ISR_ACTIVATE_DEVICE_()

Prototype

```
VDK_ISR_ACTIVATE_DEVICE_(VDK_IOID inID);
```

Description

Executes the named device driver prior to execution returning to the thread domain

Parameters

`inID` is the device driver to run.

Scheduling

Invokes the scheduler prior to returning to the thread domain

Determinism

Constant time

Errors Thrown

Full instrumentation and error-checking libraries:

Goes to `KernelPanic` with a `g_KernelPanicCode` of `kISRError` and a `g_KernelPanicError` of `kBadIOID` if `inID` is greater than the maximum number of I/O objects allowed in the system.

VDK_ISR_CLEAR_EVENTBIT_()

Prototype

```
VDK_ISR_CLEAR_EVENTBIT_(VDK_EventBitID inEventBit);
```

Description

Clears the value of `inEventBit` by setting it to `0` (`FALSE`). *All* event bit clears that occur in the interrupt domain are processed immediately prior to returning to the thread domain.

Parameters

`inEventBit` specifies the event bit to clear.

Scheduling

If `inEventBit` is currently set (`1`), the macro invokes the scheduler prior to returning to the thread domain. This allows the value of all dependent events to be recalculated and may cause a thread context switch.

Determinism

Constant time

Errors Thrown

None

VDK_ISR_LOG_HISTORY_()

Prototype

```
VDK_ISR_LOG_HISTORY_(VDK_HistoryEnum  inEnum,
                     int              inVal,
                     VDK_ThreadID     inThreadID);
```

Description

Adds a record to the history buffer. It is `NULL` if the `VDK_INSTRUMENTATION_LEVEL_` macro is set to `0` or `1`. (The value of `2` indicates that fully instrumented libraries are in use.) See online Help for more information.

Parameters

`inEnum` is the enumeration value for this type of event. For more information, see "HistoryEnum" on page 4-19.

`inVal` is the information defined by the enumeration.

`inThreadID` is the `ThreadID` stored with History Event.

Scheduling

Does not invoke the scheduler

Determinism

Constant time

Errors Thrown

None

VDK_ISR_POST_SEMAPHORE_()

Prototype

```
VDK_ISR_POST_SEMAPHORE_(VDK_SemaphoreID inSemaphoreID);
```

Description

Posts the named semaphore. Every time a semaphore is posted, its count increases until it reaches its maximum value, as specified on creation. Any further posts have no effect. All semaphore posts that occur in the interrupt domain are processed immediately prior to returning to the thread domain.

Parameters

`inSemaphoreID` is the semaphore to post.

Scheduling

If a thread is pending on `inSemaphoreID`, the macro invokes the scheduler prior to returning to the thread domain.

Determinism

Constant time

Errors Thrown

Full instrumentation and error-checking libraries:

Goes to `KernelPanic` with a `g_KernelPanicCode` of `kISRError` and a `g_KernelPanicError` of `kUnknownSemaphore` if `inSemaphoreID` is greater than the maximum number of semaphores allowed in the system.

VDK_ISR_SET_EVENTBIT_()

Prototype

```
VDK_ISR_SET_EVENTBIT_(VDK_EventBitID inEventBit);
```

Description

Sets the value of `inEventBit` by setting it to `1` (`TRUE`). *All* event bit sets that occur in the interrupt domain are processed immediately prior to returning to the thread domain.

Parameters

`inEventBit` specifies the event bit to set.

Scheduling

If `inEventBit` is currently clear (zero), the macro invokes the scheduler prior to returning to the thread domain. This allows the value of all dependent events to be recalculated and may cause a thread context switch.

Determinism

Constant time

Errors Thrown

None

C_ISR_ActivateDevice()

C Prototype

```
void VDK_C_ISR_ActivateDevice(VDK_IOID inID);
```

C++ Prototype

```
void VDK::C_ISR_ActivateDevice(VDK::IOID inID);
```

Description

Executes the named device driver prior to execution returning to the thread domain.

Parameters

`inID` is the device driver to run.

Scheduling

Invokes the scheduler prior to returning to the thread domain.

Determinism

Constant time.

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

Goes to `KernelPanic` with a `g_KernelPanicCode` of `kISRError` and a `g_KernelPanicError` of `kBadIOID` if `inID` is greater than the maximum number of I/O objects allowed in the system.

non error-checking libraries: None

C_ISR_ClearEventBit()

C Prototype

```
void VDK_C_ISR_ClearEventBit(VDK_EventBitID inEventBitID);
```

C++ Prototype

```
void VDK::C_ISR_ClearEventBit(VDK::EventBitID inEventBitID);
```

Description

Clears the value of `inEventBitID` by setting it to `0` (`FALSE`). *All* event bit clears that occur in the interrupt domain are processed immediately prior to returning to the thread domain.

Parameters

`inEventBitID` specifies the event bit to clear.

Scheduling

If `inEventBitID` is currently set (`1`), the macro invokes the scheduler prior to returning to the thread domain. This allows the value of all dependent events to be recalculated and may cause a thread context switch.

Determinism

Constant time.

Return Value

No return value

Errors Thrown

None

C_ISR_PostSemaphore()

C Prototype

```
void VDK_C_ISR_PostSemaphore(VDK_SemaphoreID inSemaphoreID);
```

C++ Prototype

```
void VDK::C_ISR_PostSemaphore(VDK::SemaphoreID inSemaphoreID);
```

Description

Posts the named semaphore. Every time a semaphore is posted, its count increases until it reaches its maximum value, as specified on creation. Any further posts have no effect. All semaphore posts that occur in the interrupt domain are processed immediately prior to returning to the thread domain.

Parameters

`inSemaphoreID` is the semaphore to post.

Scheduling

If a thread is pending on `inSemaphoreID`, the scheduler is invoked prior to returning to the thread domain.

Determinism

Constant time

Return Value

No return value

Errors Thrown

Full instrumentation and error-checking libraries:

Goes to `KernelPanic` with a `g_KernelPanicCode` of `kISRError` and a `g_KernelPanicError` of `kUnknownSemaphore` if `inSemaphoreID` is greater than the maximum number of semaphores allowed in the system.

non error-checking libraries: None

C_ISR_SetEventBit()

C Prototype

```
void VDK_C_ISR_SetEventBit(VDK_EventBitID inEventBitID);
```

C++ Prototype

```
void VDK::C_ISR_SetEventBit(VDK::EventBitID inEventBitID);
```

Description

Sets the value of `inEventBitID` by setting it to `1` (`TRUE`). *All* event bit sets that occur in the interrupt domain are processed immediately prior to returning to the thread domain.

Parameters

`inEventBitID` specifies the event bit to set.

Scheduling

If `inEventBitID` is currently clear (zero), the macro invokes the scheduler prior to returning to the thread domain. This allows the value of all dependent events to be recalculated and may cause a thread context switch.

Determinism

Constant time.

Return Value

No return value

Errors Thrown

None

A PROCESSOR-SPECIFIC NOTES

This appendix provides processor-specific information for Blackfin, SHARC and TigerSHARC processors.

This section describes:

VDK for Blackfin Processors

This section provides information relevant to the use of VDK on ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF535, ADSP-BF536, ADSP-BF537, ADSP-BF538, ADSP-BF539, ADSP-BF561, ADSP-BF566 and AD6532.

User and Supervisor Modes

The Blackfin processor's architecture makes a distinction between execution in user and supervisor modes. The initial release of VDK (in VisualDSP++ 2.0) supported both modes and switched back and forth as necessary. However, all subsequent releases run entirely in supervisor mode, including all user thread code.

Since supervisor mode provides a superset of the capabilities of user mode, applications using VDK no longer need to be aware of the processor mode and do not need to raise exceptions in order to access protected resources.

Thread, Kernel, and Interrupt Execution Levels

VDK reserves execution level 15 as the run-time execution level for most user and VDK API code, and reserves execution level 14 for internal VDK operations. In the following text, these are referred to as "Thread Level" and "Kernel Level", respectively. Execution levels 13–6 are collectively referred to as "Interrupt Level". All of these levels (15–6) execute in supervisor mode.

All thread functions execute at Thread Level (execution level 15), including `Run()` and `ErrorHandler()`. Conversely, all Interrupt Service Routines (ISRs) execute at higher priority (lower numbered) execution levels, according to the interrupt source that invoked them. The implementation function for device drivers (their single entry point) may be called by the kernel at either Thread Level (execution level 15) or at Kernel Level (execution level 14), depending on the purpose of the call.

Device driver "activate" (`kIO_Activate`) functionality is the only user code that executes at Kernel Level. All other device driver code executes at Thread Level. Entry to Kernel Level is, in this case, initiated by an ISR calling `VDK_ISR_ACTIVATE_DEVICE_()` or `C_ISR_ActivateDevice()` and is, therefore, asynchronous with respect to Thread Level, except within a critical region. Thus, care must be taken to synchronize access to shared data between Thread level and Kernel Level, as well as between Thread Level and Interrupt Level (and also between Kernel Level and Interrupt Level). Critical regions may be used for both of these purposes.

As VDK now runs entirely in supervisor mode, system memory-mapped registers (MMRs) may be accessed directly and at any time. It is, however, the user's responsibility to ensure the operations are performed correctly and at appropriate times.

Critical and Unscheduled Regions

Because VDK now executes entirely in supervisor mode, the execution-time cost of entering or leaving a critical region is reduced, compared to that in the VisualDSP++ 2.0 release.. However, because VDK now disables interrupts for much shorter periods of time, it is more likely that the worst-case interrupts-off time will be set by critical regions in user code. Therefore, care must be taken that such usage does not impact the interrupt latency of the system to an unacceptable degree.

Exceptions

VDK reserves service exception `ID 0 (EXCPT 0)` for internal use. Additionally, the IDDE automatically generates a source file for all VDK projects for Blackfin processors. The source file defines an entry point for any service or error exceptions you wish to trap. When an exception occurs, VDK intercepts the exception; if it is not the VDK exception, the user-defined exception handler executes.

Do not manipulate the `IMASK` system register from within your exception handler. If you need to mask or unmask an interrupt in response to an exception, raise an interrupt and change the value of `IMASK` in the ISR after the exception handler.

ISR API Assembly Macros

The Blackfin processor's assembly syntax requires the use of separate API macros, depending on whether the arguments are constants (immediate values, enumerations) or data registers (`R0` through `R7`). The arguments to the default assembly macros, described in Chapter 5, "VDK API Reference", must be constants. When passing data registers as arguments, append "`REG_`" to the macro name as follows:

```
VDK_ISR_POST_SEMAPHORE_REG_(semaphore_num_);
VDK_ISR_ACTIVATE_DEVICE_REG_(dev_num_);
```

```
VDK_ISR_SET_EVENTBIT_REG_(eventbit_num_);
VDK_ISR_CLEAR_EVENTBIT_REG_(eventbit_num_);
VDK_ISR_LOG_HISTORY_REG_(enum_, value_, threadID_);
```

The assembly macros, as defined in "Assembly Macros and C/C++ ISR APIs" (without "REG_"), only accept constants as arguments. Passing a register name results in an assembler error.

Interrupts

The following hardware events (interrupts) are reserved for use by VDK on Blackfin processors.

- EVT_EVX – the software exception handler. User code handles software exceptions by modifying the source file created by VisualDSP++ named ExceptionHandler-<processor_name>.asm, for example ExceptionHandler-BF533.asm.
- EVT_IVTMR – the interrupt associated with the timer integral to the processor core. This timer generates the interrupts for system ticks and provides all VDK timing services. Disabling this timer stops sleeping, round-robin scheduling, pending with timeout, and periodic semaphores.
- EVT_IVG14 – general interrupt #14. This interrupt is reserved for use by VDK and may not be used in any other manner.
- EVT_IVG15 – general interrupt #15. This interrupt is reserved to provide a supervisor mode runtime for user and VDK code and may not be used in any other manner.

The ADSP-BF53x processor designates hardware events seven (EVT_IVG7) through thirteen (EVT_IVG13) as "general interrupts" and maps each to more than one physical peripheral. The IDDE generates source code templates with a single entry point per interrupt level, rather than for each peripheral. Therefore, you are responsible for dispatching interrupts when more than one peripheral is used at the same level. The technique used depends on the application, but, typically, either chaining or a jump table is used. When chaining, a handler will check to see whether the interrupt was caused by the specific peripheral it knows how to handle. If not, execution is passed to the next handler in the chain. A jump table uses an identifier or a constant as an index to a table of function pointers when searching for the appropriate interrupt handler.

Timer

VDK `Ticks` are derived from the timer implemented in the inner processor core of Blackfin processors and are synchronized to the main core clock `CCLK`. However, this timer is disabled when a Blackfin processor enters low-power mode. Thus, all VDK timing services (such as sleeping, timeouts, and periodic semaphores) do not operate while the core is in `IDLE` or low power mode.

ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF534, ADSP-BF536, ADSP-BF537, ADSP-BF538 and ADSP-BF539 Processor Memory

The default VDK linker description files for these processors (`VDK-BF531.LDF`, `VDK-BF532.LDF`, etc.) place all code and data, and the default heap, into L1 SRAM. These default assignments may be changed by customizing the `.LDF` file used by a project. Refer to the *VisualDSP++ 4.0 Linker and Utilities Manual* for details on how to do this. An alternative mapping is included in the default `.LDF` files, which uses part or all of the L1 SRAM as cache (assumes that external memory is present). However, caching of code and data is not enabled by default. For details on how to enable and configure caching, see the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*.

ADSP-BF535 and AD6532 Processor Memory

The default VDK linker description files for these processors (`VDK-BF535.LDF` and `VDK-AD6532.LDF`) place all code and data, and the default heap, into L2 memory. The L1 memory regions are not used by default. These default assignments may be changed by customizing the `.LDF` file used by a project. Refer to the *VisualDSP++ 4.0 Linker and Utilities Manual* for details on how to do this. An alternative mapping is included in the default `.LDF` files, which uses the L1 SRAM as cache. However, caching of code and data is not enabled by default. For details on how to enable and configure caching, see the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*.

ADSP-BF561and ADSP-BF566 Processor Memory

The default VDK linker description files for the ADSP-BF561 and ADSP-BF566 processors (`VDK-BF561.LDF` and `VDK-BF566.LDF`) place code and data, and the default heap, into both L1 SRAM and L2 memory. These default assignments may be changed by customizing the `.LDF` file used by a project. Refer to the *VisualDSP++ 4.0 Linker and Utilities Manual* for details on how to do this. An alternative mapping is included in the default `.LDF` files which uses part of the L1 SRAM as cache. However, caching of code and data is not enabled by default. For details on how to enable and configure caching, see the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*.

Interrupt Nesting

VDK fully supports nested interrupts:

- The skeleton Interrupt Service Routines, which are generated by the VisualDSP++ 4.0 IDDE for user-defined interrupt handlers, enable nesting by pushing the contents of the `RETI` register onto the stack on entry and popping it immediately prior to exit.
- The ISR API macros are fully reentrant.

Note that the skeleton ISRs in the VisualDSP++ 2.0 release did not enable nesting. When converting existing VisualDSP++ 2.0 projects to VisualDSP++ 4.0, manually add the `[--SP] = RETI;` and `RETI = [SP++];` instructions to the existing ISRs to obtain the benefits of interrupt nesting. Conversely, if nesting is not required in VisualDSP++ 4.0 projects, it is acceptable to manually delete these instructions from the ISRs.

Thread Stack Usage by Interrupts

Because all thread code executes in supervisor mode, there is no automatic switching between user and system stack pointers. Therefore, all ISRs execute using the stack of the current thread, which is the thread that is executing at the time the interrupt is serviced. This means each thread stack must—in addition to the thread's own requirements—reserve sufficient space for the requirements of ISRs. This is also applicable to the `Idle` thread's stack and, in VisualDSP++ 4.0, the size of the `Idle` thread's stack can be configured within the IDDE (see online Help for further information). When interrupt nesting is enabled (as it is by default), the worst-case space requirement is the total of the requirements of the individual ISRs. When nesting is disabled, the requirement is only the largest of the individual ISR requirements, which is one possible reason for disabling interrupt nesting.

Interrupt Latency

Every effort has been made to minimize the duration of the intervals in which interrupts are disabled by VDK. Interrupts are disabled only where necessary for synchronization with Interrupt Level and for the shortest feasible number of instructions. The instruction sequences executed during these interrupts-off periods are deterministic.

Within VDK itself, synchronization between Thread Level and Kernel Level is achieved by selectively masking the Kernel Level interrupt, while leaving the higher priority interrupts unmasked.

Multiprocessor Messaging

The dual-core ADSP-BF561 processor is the only processor in the Blackfin family for which out-of-the-box multiprocessor messaging support is provided. A device driver that uses the two Internal Memory DMA (`IMDMA0` and `IMDMA1`) channels for communication between the two cores is included under the examples directories in the VisualDSP++ 4.0 installation.

Because the `IMDMA` channels only support L1 and L2 memory, care needs to be exercised if external memory (SDRAM) is also in use. Memory payloads placed in external memory cannot be written or read by the `IMDMA` device driver and, therefore, cannot automatically be tranferred by the marshalling functions. However, since external memory is visible to both cores at the same addresses, it is not normally necessary to copy the payload contents between the cores. Provided that the application is carefully designed, it should be possible to pass the payload address and size as an unmarshalled payload type and to access the payload contents in place from either core. This is also the more efficient solution.

Note that there is no cache-coherency between the two cores of the ADSP-BF561 processor. Therefore, if caching is enabled, then any memory regions that are accessed from both cores (this applies both to L2 memory and to external SDRAM) must be defined as uncached in the CPLB tables. See the “Caching and Memory Protection” section in the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors* for more information about CPLBs.

Additionally, the thread stacks for Routing Threads must not be placed in external memory. This is because the buffer structures used for the transmission and reception of message packets are stored on the stack—these must be located in either L1 or L2 memory.

VDK for TigerSHARC Processors

This section provides information relevant to the use of VDK on the ADSP-TS101, ADSP-TS201, ADSP-TS202 and ADSP-TS203.

Thread, Kernel, and Interrupt Execution Levels

VDK runs most user and VDK API code at the normal (non-interrupt) execution level but also reserves one of the low-priority interrupt levels for internal VDK operations. In the following text, these are referred to as “Thread Level” and “Kernel Level,” respectively. The remaining interrupt levels are collectively referred to as “Interrupt Level.”

All thread functions execute at Thread Level, including `Run()` and `ErrorHandler()`. Conversely, all Interrupt Service Routines (ISRs) execute at higher priority execution levels, according to the interrupt source that invoked them. The implementation function for device drivers (their single entry point) may be called by the kernel at either Thread Level or at Kernel Level, depending on the purpose of the call.

Device driver “activate” (`kIO_Activate`) functionality is the only user code which executes at Kernel Level (all other device driver code executes at Thread Level). Entry to Kernel Level is, in this case, initiated by an ISR calling `VDK_ISR_ACTIVATE_DEVICE_()` or `C_ISR_ActivateDevice()` and is, therefore, asynchronous with respect to Thread Level (except within a critical region). For this reason, care must be taken to synchronize access to shared data between Thread Level and Kernel Level, as well as between Thread Level and Interrupt Level (and also between Kernel Level and Interrupt Level). Critical regions may be used for both of these purposes.

Critical and Unscheduled Regions

VDK in VisualDSP++ 4.0 disables interrupts internally for much shorter periods of time than in the VisualDSP++ 2.0 release. It is, therefore, more likely that the worst-case interrupts-off time will be set by the use of critical regions in user code. Care must be taken that such usage does not impact the interrupt latency of the system to an unacceptable degree.

Interrupts

The following interrupts are reserved for use by VDK on TigerSHARC processors.

- `Timer1HP` – the timer interrupt. The `Timer1` generates the interrupts for system ticks and provides all VDK timing services. Disabling this timer stops sleeping, round-robin scheduling, pending with timeout and periodic semaphores. This interrupt is reserved for use by the scheduler and may not be used in any other manner.

- `Timer0LP` – the kernel interrupt. This interrupt is reserved for use by VDK and may not be used in any other manner.

The corresponding priority levels for each timer, `Timer1LP` and `Timer0HP`, are also reserved and may not be used in any other manner.

On TS20x 1.x silicon VDK uses the specific Kernel interrupt instead of Timer0LP, thereby freeing Timer 0 for users.

Timer

On TS101 processors and TS20x processors with 0.x silicon, both `Timer0` and `Timer1` are reserved by VDK for its internal functions and may not be used in any other manner.

On TS20x processors with 1.x silicon, Timer 1 is reserved by VDK for its internal functions and may not be used in any other manner.

Memory

By default, the VDK `.LDF` files place all user and VDK code into a single section named `program`, and global and static data into specific named sections. Other named sections are available for explicit access to specific memory blocks.

VDK defines a primary heap (`system_heap0`) and a secondary heap (`system_heap1`) in two different memory blocks. VDK thread stacks are allocated from heap space, and the use of two heaps allows VDK to allocate the thread `J` and `K` stacks in different memory blocks.

"System" `J` and `K` stacks are also defined in two different memory blocks. This stack pair is used by VDK itself for Kernel-level processing. This means that when device-driver "activate" code is invoked, it is these system stacks which are in use. Therefore, you need to ensure that the size of these stacks is sufficient for the requirements of the user supplied activate code.

These default arrangements can be customized by editing the project's LDF file. Refer to the *VisualDSP++ 4.0 Linker and Utilities Manual* for information on segmenting your code. Refer to the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for TigerSHARC Processors* for information about defining additional heaps.

Interrupt Nesting

VDK fully supports nested interrupts:

- The skeleton Interrupt Service Routines, which are generated by the VisualDSP++ 4.0 IDDE for user defined interrupt handlers, enable nesting by storing the `RETIB` register to memory after entry.
- The ISR API macros are re-entrant between different interrupt levels.

Note that the skeleton ISRs in the VisualDSP++ 2.0 release did not enable nesting. When you convert existing VisualDSP++ 2.0 projects, it is, therefore, necessary to add the instructions to store and reload `RETIB` to the existing ISRs in order to obtain the benefits of interrupt nesting. Conversely, if nesting is not required in VisualDSP++ 4.0 projects, it is acceptable to manually delete these instructions from the ISRs.

Interrupt Latency

Every effort has been made to minimize the duration of the intervals where interrupts are disabled by VDK. Interrupts are disabled only when necessary for synchronization with Interrupt Level, and then for the shortest feasible number of instructions. The instruction sequences executed during these interrupts-off periods are deterministic.

Within VDK itself, synchronization between Thread Level and Kernel Level is achieved by selectively masking the Kernel Level interrupt, while leaving the higher priority interrupts unmasked.

Multiprocessor Messaging

Out-of-the-box multiprocessor messaging support is provided for the ADSP-TS101, ADSP-TS201, ADSP-TS202 and ADSP-TS203 processors. Device drivers that use the link port hardware for communication between

Processors are included under the examples directories in the VisualDSP++ 4.0 installation.

In the example projects, device driver instances (boot I/O objects) are provided for all four link ports. The selection of which devices to use for message routing can be configured by the user according to the routing topology that best fits their hardware, but by default the projects are configured for the ADSP-TS101 and ADSP-TS201 EZ-KIT Lite boards, as follows.

- ADSP-TS101 EZ-KIT Lite board

  Link port 2 carries messages out of CoreA (outgoing) and into CoreB (incoming). Link port 3 carries messages out of CoreB (outgoing) and into CoreA (incoming).

- ADSP-TS201 EZ-KIT Lite board

  Link port 2 carries messages both out of and into both cores (outgoing and incoming).

These arrangements reflect the hardwired port interconnections on the EZ-KIT boards and hence do not require the use of external link port patch cables.

Note that it **is** possible to use ADSP-TS101 link ports bi-directionally, so as to carry both incoming and outgoing messages over the same port (as on ADSP-TS201 processors). Where two ports are available, however, the unidirectional mode may be slightly more efficient as the link hardware never needs to switch direction. This is not an issue on the ADSP-TS201 processor as the transmit and receive sides of each link port are independent.

Support for the cluster bus address space is provided via a standard marshalling type (`ClusterBusMarshaller`), which automatically translates local payload addresses into multiprocessor space addresses prior to transmission in a message, and translates incoming payload addresses to local addresses whenever the portion of the multiprocessor space corresponds to the local processor. In an appropriately designed application, this feature can allow payloads to be passed by reference, rather than by copying, even between processors. However, the overall efficiency of such an approach (accessing payload contents in-place across the cluster bus rather than copying to local memory) is an issue for the application designer to consider. It will also be necessary to ensure that payload memory blocks are freed only by the processor that originally allocated them.

VDK for SHARC Processors

This section provides information relevant to the use of VDK on ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, ADSP-21160, ADSP-21161, ADSP-21261, ADSP-21262, ADSP-21266, ADSP-21267, ADSP-21363, ADSP-21364, and ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368 and ADSP-21369.

Thread, Kernel and Interrupt Execution Levels

VDK runs most user and VDK API code at the normal (non-interrupt) execution level but also reserves one of the low-priority interrupt levels for internal VDK operations. In the following text, these are referred to as “Thread Level” and “Kernel Level”, respectively. The remaining interrupt levels are collectively referred to as “Interrupt Level.”

All thread functions execute at Thread Level, including `Run()` and `ErrorHandler()`. Conversely, all Interrupt Service Routines execute at higher-priority execution levels according to the interrupt source that invoked them. The implementation function for device drivers (their single entry point) may be called by the kernel at either Thread Level or at Kernel Level, depending on the purpose of the call.

Device driver “activate” (kIO_Activate) functionality is the only user code which executes at Kernel Level (all other device driver code executes at Thread Level). Entry to Kernel Level is, in this case, initiated by an ISR calling VDK_ISR_ACTIVATE_DEVICE_() or C_ISR_ActivateDevice() and is, therefore, asynchronous with respect to Thread Level (except within a critical region). For this reason, care must be taken to synchronize access to shared data between Thread level and Kernel Level, as well as between Thread Level and Interrupt Level (and also between Kernel Level and Interrupt Level). Critical regions may be used for both of these purposes.

Critical and Unscheduled Regions

VDK in VisualDSP++ 4.0 disables interrupts internally for much shorter periods of time than in the VisualDSP++ 2.0 release. It is, therefore, more likely that the worst-case interrupts-off time will now be set by the use of critical regions in user code. Care must be taken that such usage does not impact the interrupt latency of the system to an unacceptable degree.

Interrupts on ADSP-2106x Processors

The following interrupts are reserved for use by VDK on the ADSP-21060, ADSP-21061, ADSP-21062 and ADSP-21065L processors.

- TMZHI – the timer interrupt. The core timer (Timer0 on ADSP-21065L processors) generates the interrupts for system ticks and provides all VDK timing services. Disabling this timer stops sleeping, round-robin scheduling, pending with timeout, and periodic semaphores. This interrupt is reserved for use by the scheduler and may not be used in any other manner.
- SFT3I – the kernel interrupt. This interrupt is reserved for use by VDK and may not be used in any other manner.

Note that the low priority interrupt for the timer (TMZLI) is available for use, either as a software interrupt, or to support a second timer where one is present in the hardware (as with ADSP-21065L processors). By default, VDK assigns `Timer1` to the low-priority timer interrupt (`IRPTL/IMASK` bit 23), so using it only requires an interrupt handler to be defined for `TMZLI` and for `Timer1` to be initialized.

Interrupts on ADSP-2116x, ADSP-2126x and ADSP-2136x Processors

The following interrupts are reserved for use by VDK on the ADSP-21160, ADSP-21161, ADSP-21261, ADSP-21262, ADSP-21266, ADSP-21267, ADSP-21363, ADSP-21364 and ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368 and ADSP-21369 processors.

- `TMZHI` – the timer interrupt. The core timer generates the interrupts for system ticks and provides all VDK timing services. Disabling this timer stops sleeping, round-robin scheduling, pending with timeout, and periodic semaphores. This interrupt is reserved for use by the scheduler and may not be used in any other manner.
- `SFT3I` – the kernel interrupt. This interrupt is reserved for use by VDK and may not be used in any other manner.

The `ClearInterruptMaskBitsEx()` (on page 5-26) and `SetInterruptMaskBitsEx()` (on page 5-144) APIs must be used to manipulate the mask component of the `LIRPTL` register.

Timer

Where only one timer is provided by the hardware, it is reserved by VDK for its internal timekeeping functions and may not be used in any other manner.

Where more than one timer is provided, one timer is reserved by VDK for its internal timekeeping functions and may not be used in any other manner. Any other timers may be used by user code.

Memory

By default, the VDK .LDF files place all user and VDK code into a single section called program, which is mapped into PM memory. Most global and static data is placed in the data1 section, which is mapped into DM memory. A separate memory region in DM memory is used for the system heap.

A system stack also exists in DM memory. This stack is used by VDK itself for Kernel Level processing. This means that whenever device driver "activate" code is invoked, it is this system stack which is in use. Therefore, you need to ensure that the size of this stack is sufficient for the requirements of user-supplied activate code.

These default arrangements can be customized by editing the project's .LDF file. Refer to the *VisualDSP++ 4.0 Linker and Utilities Manual* for information on segmenting your code. Refer to the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for SHARC Processors* for information about defining additional heaps.

Register Usage

The ADSP-21x6x processor devices provide a set of alternate (or background) registers for both the computation units and the DAGs. VDK does not save and restore these alternate registers during context switches but instead employs them to accelerate entry to and exit from the scheduler interrupt by providing it with a dedicated C run-time environment.

This means that the following alternate registers (Table A-1) contain fixed values and must not be changed by user code:

Table A-1. ADSP-21x6x DSP Alternate Registers

Register	Value
M5’, M13’	0
M6’, M14’	1
M7’, M15’	-1
B6’, B7’	System Stack base
L6’, L7’	System Stack length
L2’-L5’, L10’-L15’	0

Since these are used at Interrupt Level (which may be entered at any time), it is not sufficient to save these registers and restore them after use, except within a critical region.

The following alternate registers are not used by VDK and are available for use by user code:

```
I0’, M0’, B0’, L0’,
I1’, M1’, B1’, L1’,
I8’, M8’, B8’, L8’,
I9’, M9’, B9’, L9’
```

Note that these registers do not form part of the thread context and, hence, are not context-switched between threads. Therefore, their most appropriate use is in implementing fast I/O interrupt handlers for devices where DMA is not available.

The remaining alternate registers:

```
R0’-R15’,
I2’-I7’, I10’-I15’,
M2’-M4’, M10’-M12’,
B2’-B5’, B10’-B15’
```

are used by the scheduler ISR and, hence, may change at any time (unless inside a critical region). If these registers are to be used in a user-defined ISR, then their contents must be saved and restored before returning from the interrupt, as for the primary registers.

Note that the background multiplier result register, `MRB`, is not considered to be an alternate register in the above discussion, and that both `MRB` and `MRF` are context-switched between threads.

40-bit Register Usage

The ALU registers of the ADSP-21x6x processor are 40 bits in width in order to support the 40-bit extended precision floating-point mode. However, the C/C++ run-time environment does not support this mode and, instead, runs with 32-bit rounding enabled (bit `RND32=1` in the `MODE1` register). The eight additional Least Significant Bits (LSBs) of the mantissa are, therefore, not used in C/C++ applications.

VDK only preserves the 32-bit form of the data registers during a context switch. In addition, both the VDK `Timer` ISR and any user ISR which calls the VDK ISR API macros, may cause the eight additional LSBs of `F0–F3` to be cleared. Neither of these restrictions cause any difficulty for C/C++ code or for assembly code which runs with `RND32=1` (the default), but code that requires the use of the 40-bit extended precision mode must observe the following restrictions:

- It must be written in assembly
- It must clear `RND32` explicitly and set it again when the 40-bit computation is complete.
- It must execute within an unscheduled region.
- It must avoid the use of `F0`, `F1`, `F2`, and `F3`, unless all interrupts are disabled (`F4–F15` are safe to use with interrupts enabled).

Interrupt Nesting

VDK fully supports nested interrupts:

- The skeleton Interrupt Service Routines, which are generated by the VisualDSP++ 4.0 IDDE for user-defined interrupt handlers, enable nesting by re-enabling interrupts after entry.
- The ISR API macros are re-entrant between different interrupt levels.

Note that the skeleton ISRs in the VisualDSP++ 2.0 release did not enable nesting, but a comment indicates that this should be done before returning from the ISR. When converting existing VisualDSP++ 2.0 projects, it is, therefore, necessary to manually move the instruction to re-enable interrupts to an earlier point in existing ISRs in order to obtain the benefits of interrupt nesting. Conversely, if nesting is not required in VisualDSP++ 4.0 projects, it is acceptable to manually delete these instructions from the ISRs.

Interrupt Latency

Every effort has been made to minimize the duration of the intervals where interrupts are disabled by VDK. Interrupts are disabled only where necessary for synchronization with Interrupt Level, and then for the shortest feasible number of instructions. The instruction sequences executed during these interrupts-off periods are deterministic.

Within VDK itself, synchronization between Thread Level and Kernel Level is achieved by selectively masking the Kernel Level interrupt, while leaving the higher priority interrupts unmasked.

Multiprocessor Messaging

The ADSP-21060, ADSP-21062, ADSP-21160, and ADSP-21161 processors are the only processors in the SHARC processor family for which out-of-the-box multiprocessor messaging support is provided (these being the SHARC processor variants that include link port hardware). Device drivers that use the link port hardware for communication between processors are included under the examples directories in the VisualDSP++ 4.0 installation.

In the example projects, device driver instances (boot I/O objects) are provided for at least two link ports. The selection of which devices to use for message routing can be configured by the user according to the routing topology that best fits their hardware, but the presence of two links per node allows (at least) construction of a ring topology containing any number of nodes. Note that it is possible to use link ports bi-directionally, so as to carry both incoming and outgoing messages over the same port. Where two ports are available, however, the unidirectional mode may be slightly more efficient as the link hardware never needs to switch direction.

Support for the cluster bus address space is provided via a standard marshalling type (`ClusterBusMarshaller`), which automatically translates local payload addresses into multiprocessor space addresses prior to transmission in a message, and translates incoming payload addresses to local addresses whenever the portion of the multiprocessor space corresponds to the local processor. In an appropriately designed application, this can allow payloads to be passed by reference, rather than by copying, even between processors. However, the overall efficiency of such an approach (that is, accessing payload contents in-place across the cluster bus rather than copying to local memory) is an issue for the application designer to consider. It is also necessary to ensure that payload memory blocks are freed only by the processor that originally allocated them.

B MIGRATING DEVICE DRIVERS

The device driver architecture has fundamentally changed between VisualDSP++ 2.0 and VisualDSP++ 4.0. Device drivers have become part of the I/O interface and are now class-based. While the underlying changes to device drivers have a minimal effect from the usage perspective, device drivers created under VisualDSP++ 2.0 are incompatible with the device drivers of VisualDSP++ 4.0.

This appendix describes how to convert device drivers of VisualDSP++ 2.0 for use in projects built using VisualDSP++ 4.0.

The converting procedure includes:

- "Step 1: Saving Existing Sources" on page B-2
- "Step 2: Revising Properties" on page B-2
- "Step 3: Revising Sources" on page B-3
- "Step 4: Creating Boot Objects" on page B-4

Step 1: Saving Existing Sources

Make backup copies of all existing VisualDSP++2.0 device driver sources and header files.

Step 2: Revising Properties

Open the existing VisualDSP++ 2.0 device driver project in the VisualDSP++ 4.0 IDDE. The following changes pertaining to device drivers can be observed in the kernel window:

a. The VisualDSP++ 2.0 **Device drivers** node can now be found under the **I/O Interface** node in the **Kernel** window. Due to the class-based model of device drivers in VisualDSP++4.0, there is also the **Boot I/O Objects** node under the **I/O Interface** node.

b. Under the **I/O Interface** node, there is a newly added property **Max number of I/O Objects**. Set this property to the number of device drivers to be used.

c. Device flags are no longer associated directly with a specific device driver. In VisualDSP++ 4.0, there is a separate **Device Flags** node in the **Kernel** window.

   Create a device flag with the same name under the **Device Flags** node for each device flag present in the original VisualDSP++ 2.0 project.

Step 3: Revising Sources

To change the existing device driver source and header files to the new model:

a. Delete all the existing **Device Drivers** that have been imported from the original VisualDSP++ 2.0 project under the **I/O Interface**->**Device Drivers** node in the kernel window. Remove the original device driver source and header files from the project directory.

b. Create new device drivers from the **I/O Interface**->**Device Drivers** node with the same names as those in the original VisualDSP++ 2.0 project. The automatically-generated source files use the VisualDSP++ 4.0 templates.

c. Copy any code added to the VisualDSP++ 2.0 device driver header files to the analogous location in the equivalent header files generated from the VisualDSP++4.0 templates. Changes may include additional variable declarations and include files.

d. Copy any code added to the VisualDSP++ 2.0 device driver source files to the analogous location in the equivalent source files generated from the VisualDSP++4.0 templates.

   Note that the dispatch function cases are renamed from `kDD::xxx` to `kIO::xxx` in C++ device drivers and `kDD_xxx` to `kIO_xxx` in C device drivers.

e. Check all project sources and header files for references to the dispatch functions `VDK::kDD_xxx` (C++ device drivers) or `VDK_kDD_xxx` (C device drivers) and replace them with `VDK::kIO_xxx` or `VDK_kIO_xxx`, respectively.

For C++ device drivers, the dispatch function is scoped under the device driver class in VisualDSP++ 4.0. For C device drivers, the dispatch function is identical to that in VisualDSP++ 2.0, apart from the renaming of cases.

f. Modify the error checking code.
The errors thrown by VDK functions have changed from `VDK::kDDxxx` (C++ device drivers) or `VDK_kDDxxx` (C device drivers) to `VDK::kxxxx` or `VDK_kxxxx`, respectively.

The `DeviceDispatchUnion` data type has been renamed `DispatchUnion` in VisualDSP++ 4.0 and any variables of that type must be renamed accordingly.

Step 4: Creating Boot Objects

Create boot I/O objects for each device driver:

In VisualDSP++ 4.0, the created device drivers are merely templates. In order to instantiate a device driver so it can be used at boot, an I/O object must be created using the device driver template.

For each device driver created under the **I/O Interface**->**Device Drivers** node, create a boot I/O object of that type from under **I/O Interface**->**Boot I/O Objects**.

Note that the name of the boot I/O object must be different from that of the device driver template.

I INDEX

Numerics

K

L

M

N

O

P

Q

R