ALPHA DATA

ADMXRC3 API 1.8.2 Specification

Document Revision: 1.13
6 May 2016

	Head Office	US Office
Address:	4 West Silvermills Lane, Edinburgh, EH3 5BD, UK	3507 Ringsby Court Suite 105, Denver, CO 80216
Telephone:	+44 131 558 2600	(303) 954 8768
Fax:	+44 131 558 2700	(866) 820 9956 toll free
email:	sales@alpha-data.com	sales@alpha-data.com
website:	http://www.alpha-data.com	http://www.alpha-data.com

Table Of Contents

List of Tables

List of Figures

Page Intentionally left blank

1 Introduction

This document describes the ADMXRC3 Application Programming Interface, which supports a new generation of reconfigurable computing hardware from Alpha Data. The key features of this API are:

- Platform-independent (with a few exceptions).
- Functionality includes querying hardware, configuring hardware, CPU-initiated data transfer and DMA data transfers.
- Thread-safe.
- Non-blocking versions of certain functions.

1.1 New in ADMXRC3 API version 1.1.0

1.1.1 New model

The datatype ADMXRC3_MODEL_TYPE now defines the value ADMXRC3_MODEL_ADMXRC6T1.

1.1.2 Extended card information

ADMXRC3 API 1.1.0 introduces the function ADMXRC3_GetCardInfoEx in order to expose more features of Gen 3 Alpha Data reconfigurable computing hardware.

1.1.3 Hardware monitoring

ADMXRC3 API 1.1.0 introduces a set of functions and datatypes for monitoring the health of Gen 3 Alpha Data reconfigurable computing hardware. See Section 1.1.3, "Hardware monitoring" for an overview.

1.1.4 I/O personality modules

ADMXRC3 API 1.1.0 introduces the function ADMXRC3_GetModuleInfo, which returns information about what I/O personality modules (if any) are fitted to a Gen 3 Alpha Data reconfigurable computing device. Refer to Section 1.1.4, "I/O personality modules" for further details.

1.1.5 Direct-call mechanism for consuming notifications

ADMXRC3 API 1.1.0 introduces the functions ADMXRC3_StartNotificationWait and ADMXRC3_FinishNotificationWait. While these functions are intended to offer Linux applications a mechanism to consume notifications (such as a target FPGA interrupt), they are also available in Windows and Linux. Refer to Section 3.5, "Notifications" for an overview.

These functions are necessary because the GNU/Linux operating system lacks a general mechanism for registering wait objects (such as POSIX semaphores) with a kernel-mode driver. Instead of registering a wait object with the API, a Linux application uses ADMXRC3_StartNotificationWait and ADMXRC3_FinishNotificationWait.

1.1.6 VxWorks-specific API functions for consuming notifications

The functions ADMXRC3_RegisterVxwSem and ADMXRC3_UnregisterVxwSem are now available, offering equivalent functionality to ADMXRC3_RegisterWin32Event and ADMXRC3_UnregisterWin32Event.

1.2 New in ADMXRC3 API version 1.2.0

1.2.1 DMA functions for 64-bit local addresses

ADMXRC3 API 1.2.0 introduces a set of functions that accept 64-bit local addresses. These functions are named

as the existing DMA functions but with an added "Ex" suffix. For example, ADMXRC3_ReadDMAEx is the 64-bit local address counterpart of ADMXRC3_ReadDMA.

Note that some models do not use all 64 bits of the local address. The **ADMXRC3 API Hardware Addendum** defines how many DMA address bits are available for each supported model.

1.3 New in ADMXRC3 API version 1.3.0

1.3.1 Support for new models

ADMXRC3 API 1.3.0 introduces two more models: the ADM-XRC-6TGE and the ADM-XRC-6T-ADV8. The enumerated type ADMXRC3_MODEL_TYPE now includes values corresponding to those types.

1.3.2 New value for ADMXRC3_STATUS

ADMXRC3 API 1.3.0 introduces a new value for the ADMXRC3_STATUS enumerated type: **ADMXRC3_NOT_SUPPORTED**. This value is returned when the hardware does not support a particular function, or does not support the particular set of parameters passed to that function. For example, the **ADM-XRC-6T-ADV8** has a user-programmable PCI Express to OCP bridge, where some or all of the DMA engines may be configured for transferring data in one direction only (in order to conserve FPGA resources for user-created functions). Attempting a DMA transfer in a direction that is not supported by a particular DMA engine results in a return value of **ADMXRC3_NOT_SUPPORTED**.

1.4 New in ADMXRC3 API version 1.4.0

1.4.1 Support for DMA to bus addresses

ADMXRC3 API 1.4.0 introduces some new API functions which enable the DMA engines in a Gen 3 reconfigurable computing device to transfer data to arbitrary bus addresses:

- ADMXRC3_ReadDMABus
- ADMXRC3_StartReadDMABus
- ADMXRC3_StartWriteDMABus
- ADMXRC3_WriteDMABus

These functions enable bulk data transfer between peers on a bus, bypassing system memory. For example, a Gen 3 reconfigurable computing device such as the ADM-XRC-6T1 can, as bus master, read or write another PCI Express endpoint directly, provided that the PCI Express memory space address of the other endpoint is known.

1.4.2 New flag ADMXRC3_FPGA_NOTCONFIGURABLE

To better support models such as the **ADM-XRC-6T-ADV8**, that have a single user-programmable FPGA that is both the PCI Express interface and the target FPGA, the flag **ADMXRC3_FPGA_NOTCONFIGURABLE** has been added. When this flag is present in the **Flags** field of ADMXRC3_FPGA_INFO, it indicates that the FPGA in question is not reconfigurable via functions such as ADMXRC3_ConfigureFromFile. Application software can use this flag in order to determine whether or not it should attempt to configure the target FPGA.

1.4.3 New value ADMXRC3_UNIT_S for enumerated type ADMXRC3_UNIT_TYPE

A new value **ADMXRC3_UNIT_S**, meaing 'seconds', for the enumerated type ADMXRC3_UNIT_TYPE has been added.

1.5 New in ADMXRC3 API version 1.5.0

1.5.1 Common buffer support

ADMXRC3 API 1.5.0 introduces a set of API functions that expose driver-allocated common buffers to user-mode mode applications:

- ADMXRC3_GetCommonBuffer
- ADMXRC3_GetCommonBufferCount
- ADMXRC3_MapCommonBuffer
- ADMXRC3_UnmapCommonBuffer

Refer to Section 3.8.4 for a conceptual description.

1.6 New in ADMXRC3 API version 1.5.1

1.6.1 New models

New models in ADMXRC3_MODEL_TYPE:

- **ADMXRC3_MODEL_ADPEXRC6TADV_C**
- **ADMXRC3_MODEL_ADPEXRC6TADV_T**
- **ADMXRC3_MODEL_ADMXRC6TDA1**
- **ADMXRC3_MODEL_ADMXRC7K1**
- **ADMXRC3_MODEL_ADMXRC7V1**

1.6.2 ADMXRC3 header file version macros

Beginning with ADMXRC3 API 1.5.1, the macros ADMXRC3_H_VERSION_SUPER, ADMXRC3_H_VERSION_MAJOR and ADMXRC3_H_VERSION_MINOR indicate the version of the ADMXRC3 API supported by **<admxrc3.h>**, in order to enable applications to perform a compile-term version check if necessary.

1.6.3 Xilinx 7 Series support

The ADMXRC3_FAMILY_TYPE, ADMXRC3_SUBFAMILY_TYPE and ADMXRC3_FPGA_TYPE enumerated types have been augmented with new values in order to support Xilinx 7 Series devices.

1.7 New in ADMXRC3 API version 1.6.0

1.7.1 Device status API functions

The API function ADMXRC3_GetDeviceStatus returns a structure that indicates any error conditions that might be present on a particular device, and the API function ADMXRC3_ClearDeviceErrors clears error conditions on a particular device.

1.7.2 New model

New model in ADMXRC3_MODEL_TYPE: **ADMXRC3_MODEL_ADMVPX37V2**.

1.7.3 Additional ADMXRC3 header file version macros

Beginning with ADMXRC3 API 1.6.0, the macro ADMXRC3_H_VERSION provides a 24-bit number whose value is derived from the **ADMXRC3_H_VERSION_*XXX*** macros. The macro ADMXRC3_H_MAKE_VERSION is also provided, which constructs a 24-bit value that can be compared with ADMXRC3_H_VERSION in

application-defined version check code.

1.8 New in ADMXRC3 API version 1.7.0

1.8.1 New model

New model in ADMXRC3_MODEL_TYPE: **ADMXRC3_MODEL_ADMXRC6TGEL**.

1.9 New in ADMXRC3 API version 1.7.1

1.9.1 New model

New model in ADMXRC3_MODEL_TYPE: **ADMXRC3_MODEL_ADMPCIE7V3**.

1.10 New in ADMXRC3 API version 1.7.2

1.10.1 New models

New models added to ADMXRC3_MODEL_TYPE: **ADMXRC3_MODEL_ADMXRC7Z1** & **ADMXRC3_MODEL_ADMXRC7Z2**.

1.10.2 New FPGA families and devices

ADMXRC3_FAMILY_ULTRASCALE (Ultrascale) has been aded to ADMXRC3_FAMILY_TYPE.

ADMXRC3_SUBFAMILY_7Z (Zynq-7000), **ADMXRC3_SUBFAMILY_UK** (Kintex Ultrascale) & **ADMXRC3_SUBFAMILY_UV** (Virtex Ultrascale) have been aded to ADMXRC3_SUBFAMILY_TYPE.

Zynq-7000, Kintex Ultrascale and Virtex Ultrascale devices have been added to ADMXRC3_FPGA_TYPE.

2 Building C and C++ applications

2.1 Building applications for Windows

2.1.1 Compiling for Windows

C or C++ source/header files that require the ADMXRC3 API must include **<admxrc3.h>** as follows:

```
#include <admxrc3.h>
```

This header file can be found in **$(ADMXRC3_SDK)/include/**, where **$(ADMXRC3_SDK)** is where the ADMXRC3 SDK is installed. In Microsoft Visual Studio, this path can be added to the list of #include search paths in a project's properties, or to the global list of #include search paths (under **Tools->Options**).

There are also some secondary header files which define extensions to the ADMXRC3 API, listed in Table 1 below:

Header file	Extension
#include <admxrc3/combuf.h>	Common buffers
#include <admxrc3/dmabus.h>	DMA transfers with peer devices

Table 1 : Secondary header files (Windows)

2.1.2 Linking for Windows

Windows applications should be linked with one of the ADMXRC3 API import libraries from the ADMXRC3 SDK. There are several different binaries, compatible with Microsoft Visual Studio 2003 onwards and corresponding to different application platforms and configurations:

Configuration	Platform	Relative Path in ADMXRC3 SDK
Debug	x86	$(ADMXRC3_SDK)/lib/win32/msvc/admxrcd.lib
Release	x86	$(ADMXRC3_SDK)/lib/win32/msvc/admxrc.lib
Debug	x64	$(ADMXRC3_SDK)/lib64/win32/msvc/admxrcd.lib
Release	x64	$(ADMXRC3_SDK)/lib64/win32/msvc/admxrc.lib

where **$(ADMXRC3_SDK)** is where the ADMXRC3 SDK is installed.

2.2 Building applications for Linux

2.2.1 Compiling for Linux

C or C++ source/header files that require the ADMXRC3 API must include the main API header file, **<admxrc3.h>**, as follows:

```
#include <admxrc3.h>
```

This header file can be found in **$(ADMXRC3_SDK)/include/**, where **$(ADMXRC3_SDK)** is where the ADMXRC3 SDK is installed. This path can be specified using the **-I** option if using the **gcc** or **g++** compiler.

There are also some secondary header files which define extensions to the ADMXRC3 API, listed in Table 2 below:

Header file	Extension
#include <admxrc3/combuf.h>	Common buffers
#include <admxrc3/dmabus.h>	DMA transfers with peer devices

Table 2 : Secondary header files (Linux)

2.2.2 Linking for Linux

Linux applications should link with the ADMXRC3 API shared library (libadmxrc3.so).

When building an application natively (as opposed cross-building), an application normally need only specify **-ladmxrc3** on the linker command line. This **requires** that the ADMXRC3 driver has already been correctly installed so that the ADMXRC3 API shared library is in **/usr/lib** (and additionally in **/usr/lib64** in 64-bit Linux).

If cross-building, it is likely that in addition to **-ladmxrc3**, the cross-linker must be told how to find the ADMXRC3 API shared library. Recent versions of the GNU toolchain provide the **--sysroot** option, which specifies where the target machine's root filesystem is located on the development machine. In most cases, this provides sufficient information for the cross-linker.

2.3 Building applications for VxWorks

2.3.1 Compiling for VxWorks

C or C++ source/header files that require the ADMXRC3 API must include **<admxrc3.h>** as follows:

```
#include <admxrc3.h>
```

This header file can be found in **$(ADMXRC3_SDK)/include/**, where *$(ADMXRC3_SDK)* is where the ADMXRC3 SDK is installed. As there are numerous versions of VxWorks and the Tornado / Workbench IDEs, detailed instructions for setting up a VxWorks project to locate the ADMXRC3 API header files are outside of the scope of this document. Please refer to the ADMXRC3 SDK User Guide for further information.

There are also some secondary header files which define extensions to the ADMXRC3 API, listed in Table 3 below:

Header file	Extension
#include <admxrc3/combuf.h>	Common buffers
#include <admxrc3/dmabus.h>	DMA transfers with peer devices

Table 3 : Secondary header files (VxWorks)

2.3.2 Linking for VxWorks

The ADMXRC3 API functions are provided by the **ADB3 Driver for VxWorks**. Because the driver is either compiled into the VxWorks kernel image or downloaded after boot as a module, there is no explicit linking step when building a VxWorks application that uses the ADMXRC3 API.

3 Concepts

There are several concepts that must be understood in order to make full use of the ADMXRC3 API. These are:

- Hardware, devices and device handles
- Multithreading
- Non-blocking operations
- Queueing
- CPU-Initiated data transfer
- DMA and user-space buffers
- Hardware resources, like target FPGAs

3.1 Hardware, devices and device handles

A *device* is a unit of reconfigurable computing hardware. An example of a device is an Alpha Data ADM-XRC-6TL card. Before a device can be used by an application, the application must first open the device. Opening a device returns a *device handle* of type ADMXRC3_HANDLE that the application may subsequently use to perform operations on that device. A device handle identifies a particular device, and is required because there may be multiple devices in a given system.

The ADMXRC3_Open and ADMXRC3_OpenEx functions are the means by which an application obtains a device handle. These functions accept an 'index' parameter, which tells the ADMXRC3 API which device the application wants to open, and is zero in the case where the application wants to open the first or only device. Once the application obtains a device handle, it may use it to call the other functions in the ADMXRC3 API.

When finished with a device handle, a well-behaved application should close it using the ADMXRC3_Close function. This enables the API and operating system to clean up any resources that the application did not explicitly free.

Should a process terminate without closing a device handle, then the operating system will typically close the device handle and free any associated resources. This is true of Windows and Linux, but not of VxWorks. The VxWorks kernel does not in general perform automatic cleanup, so it is the application's responsibility to ensure that resources are not leaked.

Some applications, such as those that use non-blocking API calls, must open a device multiple times to obtain multiple file handles. This must be done by calling ADMXRC3_Open or ADMXRC3_OpenEx multiple times, rather than using an OS-specific mechanism such as **DuplicateHandle** (Windows) or **dup** (Linux / VxWorks). For an explanation of this issue, refer to appendix Duplicating device handles.

3.2 Multithreading

In general, the ADMXRC3 API allows any number of concurrent API calls to a given *device* at a given moment. Note that this statement is not equivalent to allowing any number of ongoing API calls to a given *device handle* at a given moment.

For the purposes of this discussion, the ADMXRC3 API can be divided into two groups of functions:

- Blocking functions, such as ADMXRC3_Read.
- Non-blocking functions, such as ADMXRC3_StartReadDMAEx. Refer to Section 3.3, "Non-blocking operations" for a discussion of non-blocking operations.

The multithreading rules for the blocking functions are different to those of the non-blocking functions, and are explained in the following subsections.

3.2.1 Multithreading with blocking API functions

The ADMXRC3 allows any number of threads to simultaneously call blocking API functions for the same device handle.

3.2.2 Multithreading with non-blocking API functions

The non-blocking functions of the ADMXRC3 API have more restrictive threading rules than the blocking functions. The rule is that each device handle may have at most one ongoing non-blocking operation at a given moment. A 'non-blocking operation' is defined to last from the instant it is initiated to the instant that it is finished. For example, the following code performs a non-blocking DMA operation (error handling omitted for clarity):

```
/* A */
status = ADMXRC3_StartReadDMAEx(hDevice, ...);
...
status = ADMXRC3_FinishDMA(hDevice, ...);
/* B */
```

The code between points A and B is a non-blocking operation. Another thread cannot use 'hDevice' to perform another non-blocking operation while the first thread is performing its non-blocking operation. Attempting to do results in undefined behavior.

Therefore, when an application must perform simultaneous non-blocking operations, it should open a device as many times as it needs, obtaining multiple device handles. It then typically uses each device handle for one thread, guaranteeing that the above rule is obeyed.

3.3 Non-blocking operations

The ADMXRC3 API has non-blocking variants of some functions. The functions eligible for having non-blocking variants are the ones that may block the calling thread for a significant length of time (or indefinitely). Non-blocking functions are provided for applications in which multithreading is inconvenient or impossible.

Performing a non-blocking operation requires at least two ADMXRC3 API calls:

1. First, call one of the non-blocking API functions. These functions have names of the form **ADMXRC3_Start*Xxx*** where **ADMXRC3_Finish*Xxx*** is its corresponding blocking counterpart. This function call will return as soon as possible, without waiting for anything. If the return value is **ADMXRC3_PENDING**, the non-blocking operation was successfully started. Otherwise, no non-blocking operation was started (likely due to invalid parameters).
2. If the non-blocking operation was successfully started, the application must eventually call one of the

ADMXRC3_Finish*Xxx* functions in order to finish it. The **ADMXRC3_Finish*Xxx*** function that is appropriate depends on which **ADMXRC3_Start*Xxx*** function was called. The return value of **ADMXRC3_Finish*Xxx*** generally indicates whether or not the operation was successful.

A device handle can support at most one unfinished non-blocking operation at a given moment. It is the application's responsibility to ensure that there is at most one unfinished non-blocking operation for a given device handle at any time. This can be achieved by opening a device multiple times using ADMXRC3_Open in order to obtain multiple device handles.

The following code fragment illustrates how to perform a non-blocking DMA transfer, with error-handling:

```
/* A */
status = ADMXRC3_StartReadDMAEx(hDevice, pTicket, ...);
if (ADMXRC3_PENDING != status) {
  /* Handle error - B1 */
  ...
} else {
  /* Started operation succcessfully */
  ... other code ...
  ... calls WaitForMultipleObjects() / poll() / select() ...
  ... other code ...
  status = ADMXRC3_FinishDMA(hDevice, pTicket, FALSE);
  if (ADMXRC3_SUCCESS != status) {
    if (ADMXRC3_INVALID_HANDLE == status) {
      /* C1 - indicates that 'hDevice' may have been corrupted */
    } else if (ADMXRC3_PENDING == status) {
      /* C2 - operation still in progress - should not happen as already waited */
    } else {
      /* B2 - handle error */
      ...
    }
  } else {
    /* B3 - success */
  }
}
```

Point A is the beginning of the non-blocking operation. Points B1, B2 and B3 are possible finish points at which the non-blocking operation can be considered finished. The program should never reach points C1 or C2, but if it does, it indicates that 'hDevice' may have been corrupted somehow and the status of the non-blocking operation is indeterminate. A program may not be able to recover from such an error except by opening the device again, possibly leaking resources.

The **bWait** argument of ADMXRC3_FinishDMA is FALSE in this example because the application itself waits for completion using an operating system specific wait function such as **WaitForMultipleObjects**, **poll** or **select**.

3.3.1 Multithreading and non-blocking operations

When a single-threaded application wishes to perform many concurrent non-blocking operations, a practical strategy is as follows:

1. Initiate as many non-blocking operations as desired, using multiple device handles and multiple tickets.
2. Use an OS-specific function to wait for some or all of the non-blocking operations to complete:
   a. In Windows, ADMXRC3_HANDLE is a typedef of **HANDLE**, so an array of ADMXRC3_HANDLE can be passed to **WaitForMultipleObjects**.
   b. In Linux, an ADMXRC3_HANDLE is a file descriptor, so it may be used to initialize a **pollfd** struct for use with the **poll** system call. When initializing a **pollfd** struct, set the **fd** field to the value of an ADMXRC3_HANDLE and set the **events** field to the value **POLLPRI**.
   c. In VxWorks, an ADMXRC3_HANDLE is a file descriptor, so it may be used to initialize an **fd_set** for use with the **select** system call. When calling **select**, the file descriptor for each ADMXRC3_HANDLE should be present in the **fd_set** passed as the 2nd parameter to **select**.
3. Finish each completed non-blocking operation using one of the **ADMXRC3_Finish*Xxx*** functions, passing

FALSE for the bWait parameter.

3.3.2 Tickets

In order for the ADMXRC3 API to be able to keep track of each non-blocking operation, an application must pass an initialized object of type ADMXRC3_TICKET to all **ADMXRC3_Start*Xxx*** and **ADMXRC3_Finish*Xxx*** functions. A given ticket object must be valid from start to finish of a non-blocking operation, and must not be used in more than one concurrent non-blocking operation. The behavior resulting from violating this rule is undefined.

The ADMXRC3 API cannot completely hide OS-specific implementation details of non-blocking operations, so the initialization that must be performed for a ticket depends upon the operating system. It is necessary to initialize a ticket only once, although a ticket may be reinitialized provided it is not being used in a non-blocking operation at that moment. The following sections describe how to initialize a ticket on each supported operating system.

3.3.2.1 Tickets in Windows

An ADMXRC3_TICKET object should be first initialized with a call to ADMXRC3_InitializeTicket, and the **Overlapped.hEvent** member must then be set to a valid non-auto-reset Win32 Event handle. As with a ticket, the Win32 Event must be valid from start to finish of a non-blocking operation, and must not be used in more than one concurrent non-blocking operation. The behavior resulting from violating this rule is undefined.

3.3.2.2 Tickets in Linux and VxWorks

An ADMXRC3_TICKET object should be initialized with a call to ADMXRC3_InitializeTicket.

3.4 Queueing

Some ADMXRC3 API functions place calling threads into a queue. In abstract terms, if several threads attempt to perform an operation on the same resource at the same time, the threads will be queued on a first-come, first-served basis (a FIFO queue). Each thread's operation will be performed in its entirety before another thread is dequeued and allowed access to the resource.

For example, if several threads each call ADMXRC3_WriteDMAEx at the same time, and each thread specifies the same DMA channel, then the API automatically queues them. Once a thread's DMA transfer is started, it will be performed in its entirety, until the entire byte count is satisfied or an error occurs, before another thread is dequeued and its DMA transfer performed.

Resources are generally independent of each other, meaning that a queue exists for each resource. Thus, there is a queue for DMA channel 0, another independent queue for DMA channel 1, and so on.

Queueing is performed transparently by the ADMXRC3 API where appropriate, but for some API functions, flags exist that can modify queueing behavior.

3.5 Notifications

The ADMXRC3 API provides a set of functions to enable applications to consume notifications from a Gen 3 reconfigurable computing device. A notification is some event that occurs asynchronously within a device that applications may wish to know about. Types of notification include target FPGA interrupts and overtemperature alerts. There are two sets of functions provided for consuming notifications, as described in the following subsections.

3.5.1 Event / Semaphore registration

In Windows, it is possible to register a Win32 Event handle (type **HANDLE**) for a particular event, using ADMXRC3_RegisterWin32Event. When the specified event occurs, the Win32 Event is signalled.

In VxWorks, it is possible to register a semaphore **SEM_ID** for a particular event, using ADMXRC3_RegisterVxwSem. When the specified event occurs, the semaphore is signalled.

In Linux, there is no function equivalent to ADMXRC3_RegisterWin32Event or ADMXRC3_RegisterVxwSem, but the next subsection describes how equivalent functionality is provided in the form of direct-call notification.

3.5.2 Direct-call notification

The functions ADMXRC3_StartNotificationWait and ADMXRC3_FinishNotificationWait provide a direct-call mechanism for applications to consume notifications from a device. Unlike the methods described in the previous subsection, these functions exist for all supported operating systems.

A thread uses these functions as follows:

1. A thread initializes a ticket suitable for non-blocking operations, using ADMXRC3_InitializeTicket.
2. A thread calls ADMXRC3_StartNotificationWait, specifying the type of notification that it wishes to wait for.
3. The thread then performs some other task, such as starting an operation on the hardware.
4. The thread then waits for the requested notification, which can be done in one of several ways:
   (a) The thread can call ADMXRC3_FinishNotificationWait specifying TRUE for the bWait argument. This works on all supported operating systems.
   (b) In Windows, the thread can call **WaitForSingleObject** or **WaitForMultipleObjects** using the event handle in the ticket. It can then call ADMXRC3_FinishNotificationWait specifying FALSE for the bWait argument.
   (c) In Linux, the thread can call **poll** using the device handle used in ADMXRC3_StartNotificationWait. It can then call ADMXRC3_FinishNotificationWait specifying FALSE for the bWait argument.
   (d) In VxWorks, the thread can call **select** using the device handle used in ADMXRC3_StartNotificationWait. It can then call ADMXRC3_FinishNotificationWait specifying FALSE for the bWait argument.
5. After calling ADMXRC3_FinishNotificationWait, the thread takes appropriate action in order to handle the notification. If the thread is running in a loop, it returns to step 2 in order to wait for another notification.

3.6 Endian issues

The ADMXRC3 API does not perform any endian-conversion in functions that transfer blocks of data, such as ADMXRC3_Write, ADMXRC3_ReadVPD, ADMXRC3_StartReadDMAEx etc. There are two possible approaches to handling endian-conversion issues:

1. Make the FPGA design capable of accepting data of the same endianness as the CPU. For example, if the CPU is a PowerPC operating in big-endian mode, then the FPGA design should expect big-endian data.
2. Make the FPGA design little-endian, but use endian-conversion macros in software:
   - Linux has a set of macros **__cpu_to_le32** etc. in the header file <asm/byteorder.h>.
   - VxWorks has the **LONGSWAP** and **WORDSWAP** macros. The **_BYTE_ORDER** macro indicates endianness of the CPU for which the code is being compiled, so a set of macros equivalent to **__cpu_to_le32** in Linux etc. can be defined.
   - In Windows, it is possible to verify that the CPU for which the code is being compiled is little-endian, using predefined macros such as **_M_IX86**. A set of trivial macros equivalent to **__cpu_to_le32** in Linux etc. can then be defined, which return the argument unmodified.

3.7 String encoding issues

The ADMXRC3 API has support for applications that use **char** or **wchar_t** strings. API functions that return or expect **char** strings have names that end in *-A*, whereas functions that return or expect **wchar_t** strings have names that end in *-W*. The encoding of strings in the **char** case is slightly different between Windows and Linux, and the following table summarizes these differences:

Operating system	Encoding of **char** strings
Windows	ANSI, using the default codepage
Linux	UTF-8
VxWorks	Depends on how the host interprets **char** strings.

Table 4 : char string encoding in ADMXRC3 API by operating system

For convenience, the ADMXRC3 API defines a number of macros for portability, depending on whether or not the **_UNICODE** preprocessor symbol is defined. An example is the macro ADMXRC3_FPGA_INFO:

- When **_UNICODE** is not defined, ADMXRC3_FPGA_INFO is an alias for ADMXRC3_FPGA_INFOA. The strings in the ADMXRC3_FPGA_INFOA structure are NUL-terminated **char** strings.
- When **_UNICODE** is defined, ADMXRC3_FPGA_INFO is an alias for ADMXRC3_FPGA_INFOW. The strings in the ADMXRC3_FPGA_INFOW structure are NUL-terminated **wchar_t** strings.

Note that the **_UNICODE** preprocessor symbol, which is conventionally defined in Win32 applications that use **wchar_t** strings, really means "UTF-16 encoding". In VxWorks, there is no support for wide-character strings to speak of, so the **_UNICODE** symbol should not be defined.

3.8 Hardware features

3.8.1 Target FPGAs

A reconfigurable computing device has one or more user-programmable FPGAs, and the ADMXRC3 API provides functions for configuring them. The term "target FPGA" shall be used in place of "user-programmable FPGA" for the remainder of this document. Functions are provided for:

- Obtaining information about target FPGAs on a device, e.g. ADMXRC3_GetFpgaInfo
- Loading .BIT files into memory and unloading them, e.g. ADMXRC3_LoadBitstream
- Configuring a target FPGA with a .BIT (bitstream) file, e.g. ADMXRC3_ConfigureFromFile
- Configuring a target FPGA with in-memory configuration frame data, e.g. ADMXRC3_ConfigureFromBuffer
- Unconfiguring a target FPGA, e.g. ADMXRC3_Unconfigure

3.8.1.1 Full reconfiguration

By default, functions such as ADMXRC3_ConfigureFromFile perform a full reconfiguration of a target FPGA. This means that the target FPGA is first cleared, destroying any existing configuration, and the FPGA is then completely configured with a new bitstream. In this mode of operation, the ADMXRC3 API performs the following steps:

1. Assert PROG# on the target FPGA.
2. Wait until the target FPGA asserts INIT# and deasserts DONE.
3. Deassert PROG# on the target FPGA.
4. Wait until the target FPGA deasserts INIT#. At this point, the target FPGA is now ready to accept a bitstream.
5. Write the configuration frame data, either from a .BIT file or an in-memory buffer, to the target FPGA's SelectMap interface.
6. Wait until one of the following events occurs:
   (a) DONE is asserted; in this case, proceed to the next step.
   (b) DONE is deasserted and INIT# is asserted, which indicates that the target FPGA detected an error in the configuration frame data; in this case, return an error code and don't perform the following steps.

7 Unless the ADMXRC3_CONFIGURE_NOCHECK flag is passed, wait until OCP communication with the target FPGA is established; this is done in a model-specific way. For example, on the ADM-XRC-6T1, the ADMXRC3 API waits for the link between the PCI-E-to-OCP Bridge and the target FPGA to achieve lock.

8 Return ADMXRC3_SUCCESS.

In all of the above steps that wait, a timeout error is possible. If a timeout error occurs at a particular step, the procedure is considered unsuccessful and is terminated without performing the remaining steps.

If any error (timeout or unexpected INIT# assertion) occurs during the above procedure, the ADMXRC3 API unconfigures the target FPGA (whose state is now indeterminate) as a safety measure in case an invalid bitstream had somehow been used.

3.8.1.2 Partial reconfiguration

The ADMXRC3 API also allows a target FPGA to be partially reconfigured. In this mode of operation, the ADMXRC3 API simply downloads configuration frame data, either from a .BIT file or an in-memory buffer, to the target FPGA's SelectMap interface without asserting PROG# and without monitoring the DONE and INIT# pins of the target FPGA. Unlike full reconfiguration, the ADMXRC3_CONFIGURE_NOCHECK flag has no effect for partial reconfiguration.

3.8.1.3 Unconfiguration

A target FPGA can be unconfigured by calling ADMXRC3_Unconfigure. This might be useful for reducing power consumption when an application knows that there is nothing to do. The unconfiguration sequence is the same as the first four steps of the full reconfiguration sequence, namely:

1 Assert PROG# on the target FPGA.

2 Wait until the target FPGA asserts INIT# and deasserts DONE.

3 Deassert PROG# on the target FPGA.

4 Wait until the target FPGA deasserts INIT#.

It should be noted that at this point, the target FPGA is unconfigured, but also in a state where it can receive configuration data.

3.8.1.4 Target FPGA ownership

In order to reduce the risk of one process inadvertently reconfiguring the target FPGA that another process is using, the ADMXRC3 API has an ownership mechanism for target FPGAs. An application may choose to opt-out of this mechanism by passing the flag **ADMXRC3_CONFIGURE_SHARE** to functions such as ADMXRC3_ConfigureFromBuffer.

Assuming that an application opts into the ownership mechanism, a device handle that is successfully used to configure a target FPGA becomes the owner of that target FPGA. The owning device handle must voluntarily relinquish a target FPGA before a different device handle can be used to reconfigure the same target FPGA. The rules governing the ownership mechanism are as follows:

- When a function such as ADMXRC3_ConfigureFromBuffer is called, the ADMXRC3 API verifies that the target FPGA is either (i) free, or (ii) owned by the device handle that is calling ADMXRC3_ConfigureFromBuffer. If this check fails, the function returns an error. If the check suceeds, then configuration operation is allowed to proceed. Additionally, the target FPGA changes ownership to the device handle used in the call to ADMXRC3_ConfigureFromBuffer provided that all of the following conditions are met:
  - The target FPGA is free
  - The flags parameter **does not** contain the flag **ADMXRC3_CONFIGURE_SHARE**.
  - The configuration operation is successful

  As implied above, an application can pass **ADMXRC3_CONFIGURE_SHARE** in the flags parameter in

order to avoid its device handle becoming the new owner of a target FPGA.

- When ADMXRC3_Unconfigure is called, the ADMXRC3 API verifies that the target FPGA is owned by the device handle passed to ADMXRC3_Unconfigure. If this check fails, the function returns an error. If the check suceeds, the target FPGA becomes free unless the **ADMXRC3_CONFIGURE_SHARE** flag is passed.
- If ADMXRC3_Close is called for the device handle that owns a particular target FPGA, that target FPGA automatically becomes free.

The following state machine illustrates the ownership mechanism, from the perspective of a target FPGA:

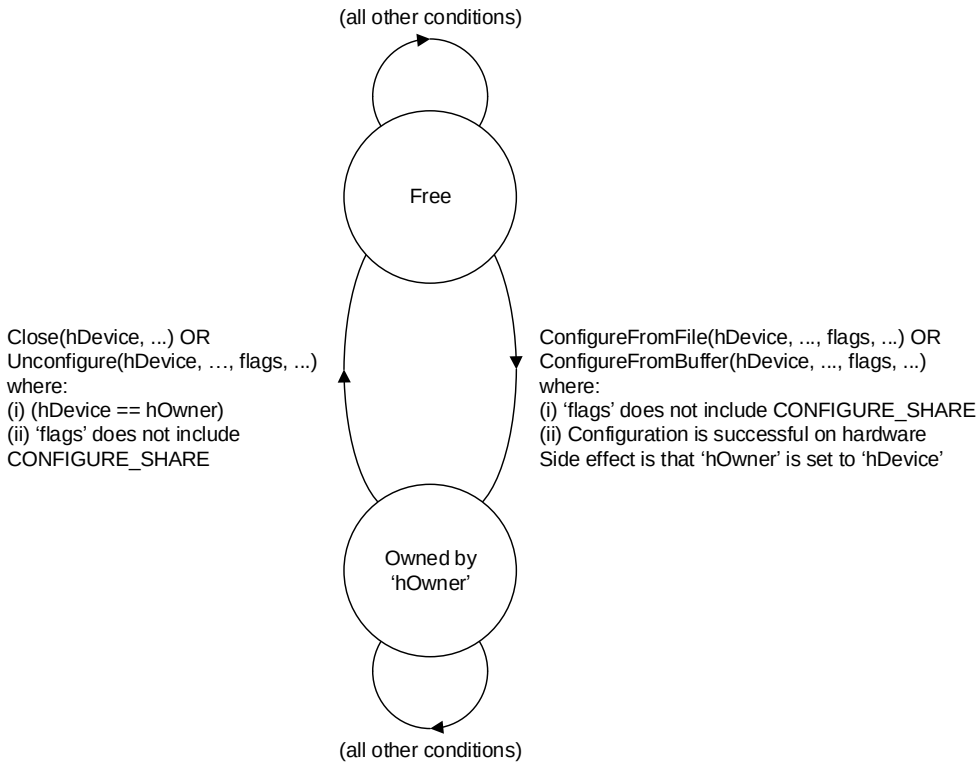


Figure 1 : Target FPGA ownership state machine

3.8.2 Memory windows

The ADMXRC3 API exposes a set of *memory windows* for each reconfigurable computing device in the system. These memory windows are an abstraction of the device's presence on the host computer's I/O bus. This abstraction permits the ADMXRC3 API to be I/O-bus-agnostic, so that applications need not be aware of the what kind of I/O bus is used in the system (PCI, PCI-X, PCI Express, HyperTransport etc.).

For example, in a PCI Express system, a PCI Express device has one or more Base Address Registers (BARs) which reside in the device's configuration space. System-level firmware (e.g. BIOS on an x86 PC) configures the BARs at boot time, which places the device at a definite address on the host system's I/O bus. There is a mapping between PCI Express BARs and *memory windows* in the ADMXRC3 API. This mapping is model-specific, and not necessarily one-to-one, for reasons related to the technicalities of the various I/O bus standards. The actual mapping for each supported model is defined in the **ADMXRC3 API Hardware Addendum**. Figure 2 shows how a reconfigurable computing device's BARs, each corresponding to a memory window (or possibly more than one memory window), might be located in the CPU's physical address space:

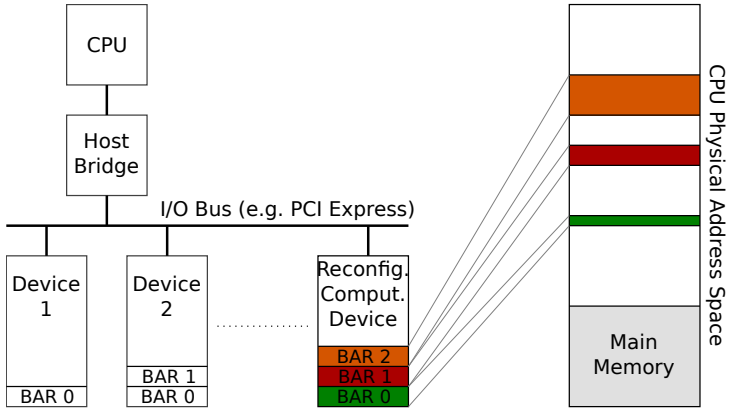


Figure 2 : BARs in a reconfigurable computing device

The ADMXRC3 API permits an application to access a reconfigurable computing device via its memory windows. Each memory window on a device serves a different purpose - for example, one window may provide access to the target FPGA in the device, and another window may allow an application to manipulate registers in the device, if necessary. Note that manipulating registers in the device is normally done by the driver and not by an application, but occasionally an application may need to read or write device registers for debugging or troubleshooting. The **ADMXRC3 API Hardware Addendum** defines the memory windows and their purposes for each model supported by the API.

The API function ADMXRC3_GetWindowInfo allows an application to enumerate and obtain information about each memory window in a device.

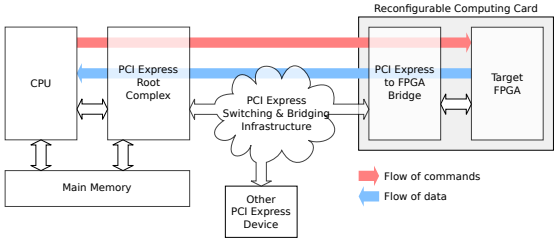
3.8.2.1 Mapping memory windows into user-space

In order to avoid the recurring overhead of calling the ADMXRC3 API and underlying driver when accessing a device, the ADMXRC3_MapWindow function is provided for mapping some or all of a memory window into the address space of a process. The ADMXRC3_MapWindow function returns a user-space pointer to where the window (or window region) is mapped, and the window can then be accessed by dereferencing the pointer. ADMXRC3_UnmapWindow performs the inverse, deleting the mapping and invalidating the pointer.

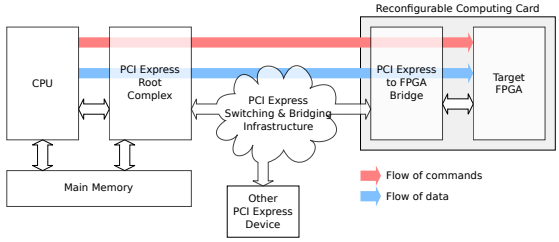
A device typically advertises several register windows to applications via the ADMXRC3_GetWindowInfo function, of which one (or more) provides access to the target FPGA (or target FPGAs). Other windows correspond to device-specific registers that should normally only be manipulated by the driver, but it is possible for an application to access those registers for debug or diagnostic purposes.

3.8.3 FPGA data transfer

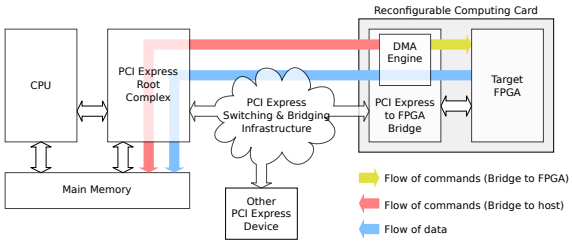
There are several ways to transfer data between the host CPU/memory and an FPGA in a reconfigurable computing device. Figure 3 illustrates the 4 main modes available in the ADMXRC3 API:



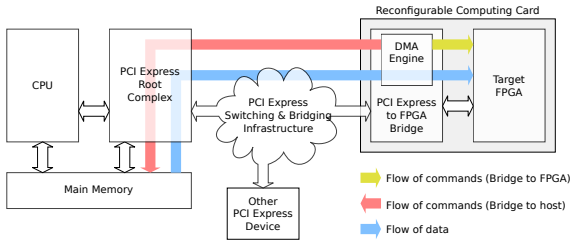
(b) CPU-initiated read of FPGA



(b) CPU-initiated write to FPGA



(c) DMA read of FPGA



(d) DMA write to FPGA

Figure 3 : Methods of FPGA data transfer

In CPU-initiated data transfer, the CPU drives the transfer of data, whereas in DMA transfers, a DMA engine in the reconfigurable computing device drives the data transfer.

CPU-initiated data transfer is generally appropriate for random access to FPGA registers, due to its relatively low latency. However, CPU-initiated data transfer yields poor throughput for bulk data transfer on most platforms. DMA transfers offer higher average throughput for blocks of data larger than a certain threshold, with the additional advantage of lower CPU utilization. Table 5 summarizes the available methods of data transfer:

Method	Throughput	CPU utilization	Software overhead
CPU-initiated read/write, via mapped pointer	poor	1 CPU core	negligible
CPU-initiated read/write, via API	poor	1 CPU core	low
DMA read/write	high	<< 1 CPU core	moderate

Table 5 : Methods of FPGA data transfer

The above table indicates the relative characteristics of the various methods of data transfer; it should be noted that for DMA transfers, block size strongly influences throughput, CPU utilization and overhead. Using DMA transfers for small blocks of data (e.g. 128 byte blocks) is highly inefficient.

3.8.3.1 CPU-initiated transfers

The functions ADMXRC3_Read and ADMXRC3_Write read and write a particular memory window using the CPU to drive the data transfer, as illustrated by Figure 4 and Figure 5:

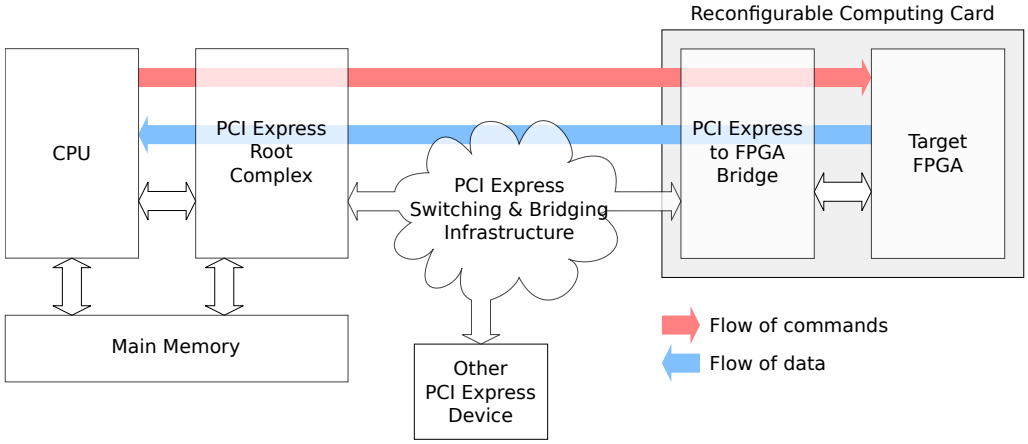


Figure 4 : CPU-initiated read of a target FPGA

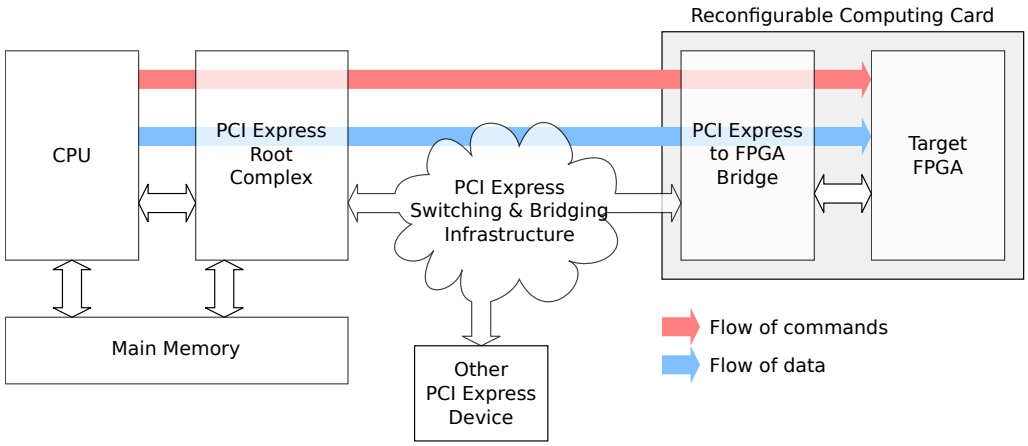


Figure 5 : CPU-initiated write of a target FPGA

Some points to bear in mind when using these functions for bulk data transfer are:

- As mentioned previously, a CPU-initiated data transfer will in general consume an entire CPU core for the duration of the transfer.
- It's possible that the performance of other CPU cores on the same die (besides the one used for the transfer) may be negatively affected if they may share the same bus interface unit.
- Writes generally transfer data quicker than reads because writes are "fire and forget". In other words, when the CPU executes a store instruction, it does not need to wait for any confirmation at the destination that the write has been completed. Thus, subject to available write buffer space, the next write can be issued immediately. In addition, on some platforms, the CPU and/or any write buffers in the datapath may also be intelligent enough to coalesce sequential writes into a burst.
- Read performance is generally extremely poor due to two factors:

  1 When the CPU executes a load instruction, it issues a read command via its bus interface. Then CPU has to wait for tread read command to reach the destination, for the destination to fetch the data, and finally for the data to propagate back through any buffers along the datapath. These latencies are additive and can be in the order of a microsecond on modern PCI Express platforms.

    Many CPUs will not issue the next load instruction until the data from the current load instruction has been returned by the system.

For these reasons, it is strongly recommended that CPU-initiated data transfer be avoided for bulk data transfer, and DMA transfer used instead.

3.8.3.2 DMA transfers with host memory

The ADMXRC3 API includes a set of functions for rapidly transferring data between host memory and the target FPGA(s) on a device using DMA transfers. Figure 6 and Figure 7 illustrate DMA transfers:

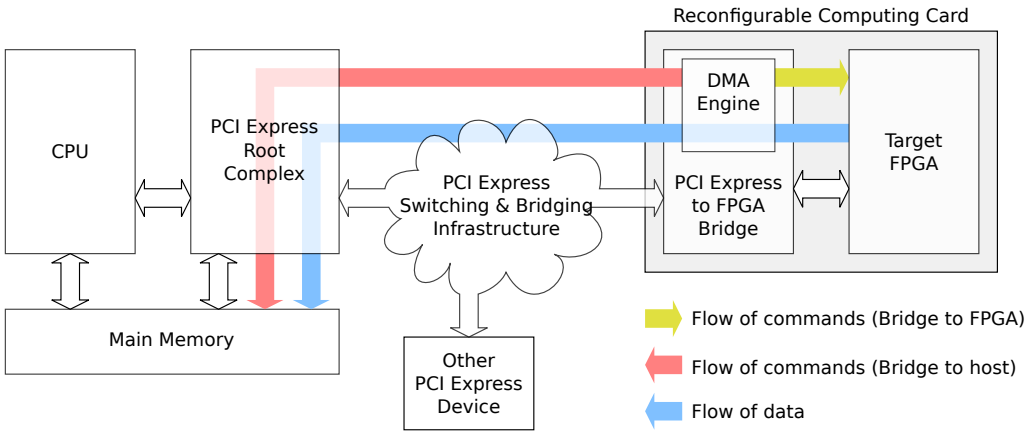


Figure 6 : DMA read of a target FPGA

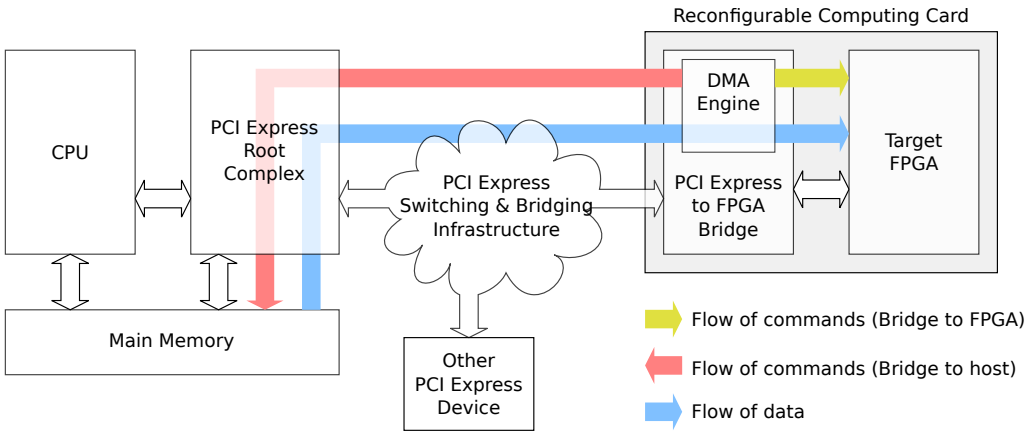


Figure 7 : DMA write of a target FPGA

Depending on the runtime platform, DMA transfers have a throughput that is typically one to two orders of magnitude faster than CPU-initiated transfers. The disadvantage of DMA transfers is that each transfer must be 'set up', and later 'torn down'. Although set-up and tear-down are performed transparently by the API, the overhead and latency in execution time may be noticeable for small DMA transfers. DMA transfers, therefore, are best used for bulk data transfer. CPU-initiated transfers are best used for random reads and writes of registers in the target FPGA(s), or for transferring small blocks of data.

The ADMXRC3 API provides two groups of functions for performing DMA transfers:

- 'Unlocked' functions, such as ADMXRC3_WriteDMAEx and ADMXRC3_StartReadDMA.
- 'Locked' functions, such as ADMXRC3_StartReadDMALockedEx and ADMXRC3_WriteDMALocked.

The following subsections describe the above two groups of DMA functions and their pros and cons.

3.8.3.2.1 Unlocked DMA functions

The Unlocked DMA functions in the ADMXRC3 API accept a description of a user-space buffer that consists simply of (i) a pointer to a buffer and (ii) a byte count. The set-up performed by these functions consists mainly of

locking the user-space buffer in memory (so that it cannot be paged out by the operating system), while tear-down consists mainly of unlocking the user-space buffer so that the operating system is once again free to page it in and out of memory.

The advantage of these functions is ease-of-use; there is no need for an application to explicitly lock a user-space buffer prior to calling them. The disadvantage is that, compared to the Locked DMA functions, there is a greater overhead per DMA transfer.

3.8.3.2.2 Locked DMA functions

The Locked DMA functions accept a description of a user-space buffer that consists of (i) a handle to an already-locked buffer, (ii) an offset into that buffer and (iii) a byte count. Locking a user-space buffer is done via the ADMXRC3_Lock function, which (if successful) returns a handle to the locked buffer. The buffer handle can then be passed to functions such as ADMXRC3_ReadDMALockedEx. Once a user-space buffer is locked, the operating system cannot page the buffer out of main memory during a DMA transfer, which is necessary to ensure that corruption of main memory cannot occur. Handles to locked buffers are global to the system, so that if another process can determine the handle of a locked buffer, it can use that buffer, even with a different device.

The inverse of ADMXRC3_Lock is ADMXRC3_Unlock, which allows the operating system to once again page the buffer in and out of main memory. Although buffer handles are global to the system, only the device handle that was used to lock a buffer can be used to unlock it. If an application attempts to unlock a user-space buffer while a DMA transfer is in progress, the call to ADMXRC3_Unlock is successful, but the buffer remains locked (via a reference counting mechanism) until the DMA transfer finishes. This avoids the possibility of memory corruption.

The advantage of using the Locked DMA functions is that the overhead of set-up and tear-down is considerably reduced on most platforms, in comparison to the Unlocked DMA functions. The Locked DMA functions should be used in performance-critical or latency-critical applications. The disadvantage is ease-of-use; an application must first lock any user-space buffers that it wishes to use prior to performing Locked DMA transfers.

3.8.3.3 DMA transfers with peer devices

Note
The API functions for performing DMA transfers with peer devices are not defined in the main API header file, **<admxrc3.h>**. In order to import the prototypes of these functions and associated datatypes, use

```
#include <admxrc3.h>
#include <admxrc3/dmabus.h>
```

As of revision 1.4.0, the ADMXRC3 API includes a set of functions for rapidly transferring data between a peer device and the target FPGA(s) on a device using DMA transfers. In order to use these functions, the bus address of the peer device must be known. Figure 8 and Figure 9 illustrate DMA transfers between a target FPGA on a reconfigurable computing device and a peer PCI Express device:

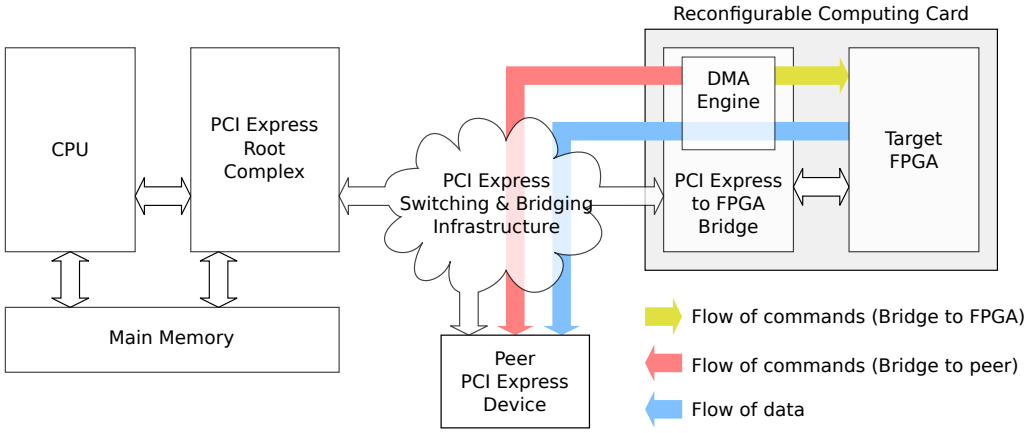


Figure 8 : DMA transfer from a target FPGA to a peer device

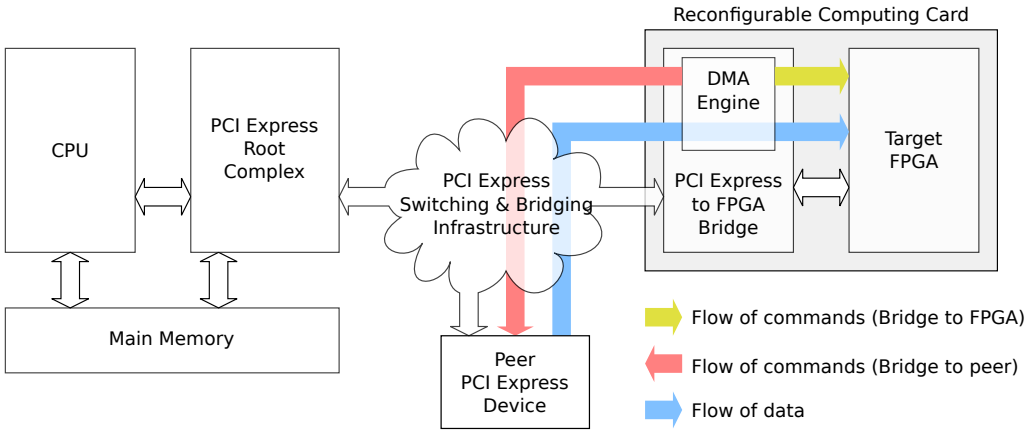


Figure 9 : DMA transfer from a peer device to a target FPGA

Depending on the runtime platform, peer DMA transfers have a throughput that is typically one to two orders of magnitude faster than CPU-initiated transfers. The disadvantage of peer DMA transfers is that each transfer must be 'set up', and later 'torn down'. Compared to host memory DMA transfer, the overhead for peer DMA transfers is relatively low, because no virtual memory management is performed. The available peer DMA transfer functions in the ADMXRC3 API are:

- ADMXRC3_ReadDMABus
- ADMXRC3_StartReadDMABus
- ADMXRC3_StartWriteDMABus
- ADMXRC3_WriteDMABus

3.8.4 Common buffers

Note
The API functions for using common buffers are not defined in the main API header file, **<admxrc3.h>**. In order to import the prototypes of these functions and associated datatypes, use

```
#include <admxrc3.h>
#include <admxrc3/combuf.h>
```

As of revision 1.5.0, the ADMXRC3 API includes a set of functions that expose driver-allocated "common buffers" to user-mode applications. The characteristics of common buffers are:

- They persist from when the driver is started until the driver is stopped.
- They are guaranteed aligned to a specified power-of-2 address boundary.
- They are allocated from the appropriate pool of memory in order to be contiguous and visible to bus-master devices.
- They can be mapped into the virtual address space of user-mode process via the ADMXRC3 API.

For a bus-mastering I/O interface, where the sizes and arrival times of packets of data are not known in advance by an application (e.g. a network interface), a common buffer can be used as a ring buffer into which new packets are placed by the I/O interface.

The number of common buffers available for a particular device is returned by the function ADMXRC3_GetCommonBufferCount. Information about a common buffer's address and size is returned by the function ADMXRC3_GetCommonBuffer. Mapping and unmapping a common buffer from a user-mode process is performed by ADMXRC3_MapCommonBuffer and ADMXRC3_UnmapCommonBuffer.

NOTE: in order for the driver to allocate one of more common buffers at startup, the driver must be started with the appropriate parameters; otherwise, no common buffers will be created (the default behaviour of the driver). The driver parameters relevant to common buffers are documented in the release notes of the ADB3 Driver for each supported operating system.

3.8.4.1 Common buffers and DMA transfers

Since ADMXRC3_GetCommonBuffer returns the bus address of a common buffer, the ADMXRC3 API functions for performing DMA transfers with peer devices can target common buffers. In such a case, the "peer device" happens to be the host machine's memory. Refer to Section 3.8.3.3 for details of these functions.

3.8.4.2 Common buffers and Direct Master transfers

Since ADMXRC3_GetCommonBuffer returns the bus address of a common buffer, advanced FPGA designs that make use of Direct Master transfers can target common buffers. The fact that common buffers persist for as long as the driver is running reduces the potential for inadvertent corruption of host memory by a Direct Master FPGA design.

3.8.5 Clock generators

Most models in Alpha Data's reconfigurable computing range contain one or more programmable clock generators. These are software-programmable and able to generate a large number of discrete frequencies in order to approximate a continuous frequency range. The output of a clock generator is usually routed to a target FPGA, but may also be connected to other functions on a card. The number of clock generators on a card and the purpose of each is model-specific. Consult the User Guide for the model in question to learn details of the available clock generators.

The ADMXRC3 API provides the following functions for manipulating clock generators:

- ADMXRC3_GetClockFrequency
- ADMXRC3_SetClockFrequency

3.8.6 Flash memory

A device may contain one or more banks of Flash (nonvolatile) memory that is used to store bitstreams for the target FPGA. The address map of each bank of Flash memory is specific to each model, so the User Guide for a particular model should be consulted when writing applications that access Flash memory.

The ADMXRC3 API provides several functions for programming Flash memory:

- ADMXRC3_GetFlashInfo
- ADMXRC3_GetFlashBlockInfo
- ADMXRC3_EraseFlash
- ADMXRC3_ReadFlash
- ADMXRC3_SyncFlash
- ADMXRC3_WriteFlash

3.8.6.1 Flash memory caching

Since most Flash devices are block-oriented, modifying a single byte results in a read-merge-erase-write cycle being performed for the block containing the location being written. To hide the latency of block erase operations, and to enable applications to treat a Flash memory bank as a uniform array of bytes, the ADMXRC3 API implements a cache for each Flash memory bank. Consequently, in order to ensure that the hardware has been synchronized with the cache, an application must call ADMXRC3_SyncFlash when appropriate.

Alternatively, an application may effectively disable the cache for a Flash memory bank by passing the flag ADMXRC3_FLASH_SYNC to those API functions that accept it. The effect of this flag is to perform synchronization before the function returns, as if ADMXRC3_SyncFlash were called.

A side of effect of having a cache means that calling ADMXRC3_ReadFlash may sometimes require block-erase and block-write operations to be performed when a dirty block must be written back to the hardware. This means that if the hardware is faulty, error codes that would normally not be expected for a read operation may be returned.

3.8.7 Vital Product Data

Alpha Data reconfigurable computing hardware stores its Vital Product Data (VPD) in a nonvolatile memory. This memory may be Flash, EEPROM or something else. The ADMXRC3 API provides functions for reading and writing VPD:

- ADMXRC3_ReadVPD
- ADMXRC3_WriteVPD

The data structures in the VPD memory are model-specific in general. Information about a particular model's VPD structures is available as application note.

The VPD memory can always be read, but in order to write to the VPD memory, the VPD write-protection mechanism must first be disabled. The VPD write-protection mechanism is operating-system dependent; refer to the release notes for the ADB3 driver specific to your operating system for details.

3.8.8 Hardware monitoring

Gen 3 Alpha Data reconfigurable computing hardware typically includes a number of on-board sensors that measure supply voltages, currents, temperatures etc. Version 1.1.0 and later of the ADMXRC3 API provides functions for reading these sensors:

- ADMXRC3_GetSensorInfo
- ADMXRC3_ReadSensor

The number of sensors in a device and the type of each is model-specific. The functions ADMXRC3_GetCardInfoEx and ADMXRC3_GetSensorInfo can be used together to enumerate the sensors and obtain information about them.

3.8.9 I/O personality modules

Gen 3 Alpha Data reconfigurable computing hardware typically has at least one site for fitting an I/O personality module. Version 1.1.0 and later of the ADMXRC3 API provides the function ADMXRC3_GetModuleInfo for

obtaining information about what module, if any, is fitted to a module site.

The number of I/O module sites is model-specific, and is 1 on most models that have a small to medium-sized form factor such as a PCI Express plug-in card, PMC or XMC. However, a device that comes in larger form factor such as VME or CompactPCI may feature multiple I/O module sites. The functions ADMXRC3_GetCardInfoEx and ADMXRC3_GetModuleInfo can be used together to enumerate the module sites and obtain information about what is fitted.

4 ADMXRC3 API Reference

4.1 ADMXRC3 API constants and macros

4.1.1 ADMXRC3_BITSTREAM

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_BITSTREAMW, if _**UNICODE** is defined
- ADMXRC3_BITSTREAMA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.2 ADMXRC3_ConfigureFromFile

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_ConfigureFromFileW, if _**UNICODE** is defined
- ADMXRC3_ConfigureFromFileA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.3 ADMXRC3_FLASH_INFO

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_FLASH_INFOW, if _**UNICODE** is defined
- ADMXRC3_FLASH_INFOA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.4 ADMXRC3_FPGA_INFO

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_FPGA_INFOW, if _**UNICODE** is defined
- ADMXRC3_FPGA_INFOA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.5 ADMXRC3_GetFlashInfo

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_GetFlashInfoW, if _**UNICODE** is defined
- ADMXRC3_GetFlashInfoA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.6 ADMXRC3_GetFpgaInfo

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_GetFpgaInfoW, if _**UNICODE** is defined
- ADMXRC3_GetFpgaInfoA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.7 ADMXRC3_GetModuleInfo

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_GetModuleInfoW, if _**UNICODE** is defined
- ADMXRC3_GetModuleInfoA, otherwise

Remarks

This macro is available in ADMXRC3 API version 1.1.0 and later.

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.8 ADMXRC3_GetSensorInfo

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_GetSensorInfoW, if _**UNICODE** is defined
- ADMXRC3_GetSensorInfoA, otherwise

Remarks

This macro is available in ADMXRC3 API version 1.1.0 and later.

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.9 ADMXRC3_GetStatusString

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_GetStatusStringW, if _**UNICODE** is defined
- ADMXRC3_GetStatusStringA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.10 ADMXRC3_H_MAKE_VERSION

Description

A macro of 3 arguments, which constructs a 24-bit version number from super, major and minor numbers (in that order). The 24-bit version number returned can be directly compared with the ADMXRC3_H_VERSION macro, should an application need to implement a compile-time version-check of the ADMXRC3 header file <admxrc3.h>.

Remarks

This macro is available in ADMXRC3 API 1.6.0 and later.

4.1.11 ADMXRC3_H_VERSION

Description

A macro whose value is a 24-bit number that represents the version of the ADMXRC3 API that is supported by <admxrc3.h>. The low 8 bits have the value of ADMXRC3_H_VERSION_MINOR, bits [15:8] have the value of ADMXRC3_H_VERSION_MAJOR and bits [23:16] have the value of

ADMXRC3_H_VERSION_SUPER.

For example, if <admxrc3.h> defines **ADMXRC3_H_VERSION** to be 67072 (0x10600), then the ADMXRC3 API version supported is 1.6.0. In other words, the functions, macros, constants and flags defined by that particular <admxrc3.h> correspond to ADMXRC3 API 1.6.0.

Application code that relies on features of the ADMXRC3 API that were introduced in a particular version of the ADMXRC3 API can use this macro to perform a compile-time sanity check, in order to ensure that it is being compiled against a suitably up-to-date version of <admxrc3.h>.

Remarks

This macro is available in ADMXRC3 API 1.6.0 and later.

4.1.12 ADMXRC3_H_VERSION_MAJOR

Description

A macro whose value is the major version number of the ADMXRC3 API. If the major version number changes without a change to the super version number, this generally represents a significant change to the ADMXRC3 API, for example the addition of a new API function.

Remarks

This macro is available in ADMXRC3 API 1.5.1 and later.

4.1.13 ADMXRC3_H_VERSION_MINOR

Description

A macro whose value is the minor version number of the ADMXRC3 API. If the minor version number (alone) changes, this generally represents a small change to the ADMXRC3 API, for example the addition of a new flag.

Remarks

This macro is available in ADMXRC3 API 1.5.1 and later.

4.1.14 ADMXRC3_H_VERSION_SUPER

Description

A macro whose value is the super version number of the ADMXRC3 API. When the super version number changes, it generally represents a large-scale change to the API.

Remarks

This macro is available in ADMXRC3 API 1.5.1 and later.

4.1.15 ADMXRC3_HANDLE_INVALID_VALUE

Description

Value that represents an invalid device handle, typically used to initialize a variable of type ADMXRC3_HANDLE.

4.1.16 ADMXRC3_LoadBitstream

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows)

and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_LoadBitstreamW, if _**UNICODE** is defined
- ADMXRC3_LoadBitstreamA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.17 ADMXRC3_MODULE_INFO

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_MODULE_INFOW, if _**UNICODE** is defined
- ADMXRC3_MODULE_INFOA, otherwise

Remarks

This macro is available in ADMXRC3 API version 1.1.0 and later.

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.18 ADMXRC3_SENSOR_INFO

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_SENSOR_INFOW, if _**UNICODE** is defined
- ADMXRC3_SENSOR_INFOA, otherwise

Remarks

This macro is available in ADMXRC3 API version 1.1.0 and later.

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.1.19 ADMXRC3_UnloadBitstream

Description

A macro that can be used in code that is intended to be portable between UTF-8 (Linux), ANSI (Windows) and Unicode versions of an application. The value of this macro depends on whether or not the _**UNICODE** preprocessor symbol is defined, as follows:

- ADMXRC3_UnloadBitstreamW, if _**UNICODE** is defined
- ADMXRC3_UnloadBitstreamA, otherwise

Remarks

Refer to Section 3.7, "String encoding issues" for an explanation of how the ADMXRC3 API handles string encodings.

4.2 ADMXRC3 API datatypes

4.2.1 ADMXRC3_BANK_INFO

Declaration

```
typedef struct ... {
    uint64_t     MaximumFrequency;
    uint64_t     MinimumFrequency;
    uint64_t     PhysicalSize;
    uint16_t     PhysicalDataWidth;
    uint16_t     PhysicalECCWidth;
    uint16_t     PhysicalWidth;
    uint16_t     Reserved1;
    uint32_t     TypeMask;
    uint32_t     ConnectivityMask;
    boolean_t    Present;
} ADMXRC3_BANK_INFO;
```

The members of this structure are as follows:

MaximumFrequency

Maximum operating frequency of the memory devices, in Hz.

If this value is 0, then both the maximum and minimum operating frequency of the device must be considered unknown. In this case, software must either use an alternative method of determining operating frequency, or avoid making assumptions about operating frequency.

MinimumFrequency

Minimum operating frequency of the memory devices, in Hz. This member must be considered invalid if MaximumFrequency is 0.

PhysicalSize

Physical size of the memory bank, in physical words.

PhysicalDataWidth

Number of physical data bits in the data bus of the memory bank, excluding error correction bits.

PhysicalECCWidth

Number of physical error correction bits in the data bus of the memory bank. Error correction bits can take the form of parity bits, ECC bits or other error correction schemes. For most types of memory, error correction bits can be used either as additional data bits, or for the purpose of error detection and correction.

PhysicalWidth

Number of physical data and error correction bits in the data bus of the memory bank. This member is always the sum of the PhysicalDataWidth and PhysicalECCWidth members.

Reserved1

Must be ignored by application software.

TypeMask

Bitwise-OR of flags that represent the operating modes that the memory bank can support, of which at least one bit must be 1. Certain types of memory, notably ZBT SSRAM, can operate in more than one mode. Supported operating modes are:

- ADMXRC3_BANK_ZBT_FT is flowthrough ZBT SSRAM
- ADMXRC3_BANK_ZBT_P is pipelined ZBT SSRAM
- ADMXRC3_BANK_SDRAM_DDR is DDR SDRAM
- ADMXRC3_BANK_SSRAM_DDR2 is DDR2 SSRAM
- ADMXRC3_BANK_SDRAM_SDR is single data rate SDRAM
- ADMXRC3_BANK_SDRAM_DDR2 is double data rate SDRAM
- ADMXRC3_BANK_SSRAM_QDR is QDR SSRAM
- ADMXRC3_BANK_SDRAM_DDR3 is DDR3 SDRAM
- ADMXRC3_BANK_SDRAM_DDR4 is DDR4 SDRAM

ConnectivityMask

This member is a bitmask indicating which target FPGA(s) the memory bank is connected to. In most cases, a single bit is set. However, some models may connect a memory bank to more than one target FPGA, and in such cases more than one bit is set. Examples:

- Assume that the NumFpga member of ADMXRC3_CARD_INFOEX is 1, indicating that the device has one target FPGA whose index is 0. The ConnectivityMask for all memory banks must therefore be 0x1, since bit 0 corresponds to target FPGA 0.
- Assume that the NumFpga member of ADMXRC3_CARD_INFOEX is 2, indicating that the device has two target FPGAs whose indices are 0 and 1. A ConnectivityMask of 0x1 indicates that the memory bank is connected to target FPGA 0, while a connectivity mask of 0x2 indicates that the memory bank is connected to target FPGA 1. A ConnectivityMask of 0x3 indicates that the memory bank is connected to both target FPGAs.

Present

If this member is nonzero, it indicates that the bank is fully populated with memory devices. If zero, the bank is unpopulated. Memory banks are normally fully populated, but may be unpopulated for two reasons:

- The target FPGA may not have sufficient bonded I/O pins in order to be able to use all memory banks. This can occur for the smaller devices in an FPGA family.
- On request by a customer, Alpha Data can manufacture cards with a subset of memory banks populated, as cost-saving measure for volume orders.

Description

A structure that describes a memory bank, obtained by call to ADMXRC3_GetBankInfo.

Remarks

The value **ADMXRC3_BANK_SDRAM_DDR4** is available in ADMXRC3 API version 1.8.0 and later.

4.2.2 ADMXRC3_BITSTREAMA

Declaration

```
typedef struct ... {
    char        Identifier[32];
    uint32_t    Length;
    uint8_t     Data[4];
} ADMXRC3_BITSTREAMA;
```

The members of this structure are as follows:

Identifier

A NUL-terminated **char** string that identifies an FPGA device and package, in the same format as in the Identifier member of ADMXRC3_FPGA_INFOA. This string is typically used to verify that a bitstream (.BIT)

file matches the target FPGA in a device, by comparing it with the Identifier member of ADMXRC3_FPGA_INFOA.

Length

The number of bytes in the variable-length array that begins with the Data member.

Data

The bitstream data, as a variable-length array. Although this member is defined to be an array of length 4, the actual number of bytes in the array is given by the Length member. This variable length array is suitable for passing to ADMXRC3_ConfigureFromBuffer.

Description

This structure represents a bitstream suitable for configuring a target FPGA, and is typically allocated and initialized by a call to ADMXRC3_LoadBitstreamA. Although applying the **sizeof** operator to this structure returns a value of a few tens of bytes, the structure should be regarded as the beginning of an object that could be many megabytes in size (for the latest FPGAs). The inverse function is ADMXRC3_UnloadBitstreamA, which deallocates the memory used by the object.

Remarks

This is the ANSI / UTF-8 version of the ADMXRC3_BITSTREAM structure. ADMXRC3_BITSTREAM is actually a macro defined to be either ADMXRC3_BITSTREAMW if the _**UNICODE** preprocessor symbol is defined, or else ADMXRC3_BITSTREAMA.

4.2.3 ADMXRC3_BITSTREAMW

Declaration

```
typedef struct ... {
    wchar_t     Identifier[32];
    uint32_t    Length;
    uint8_t     Data[4];
} ADMXRC3_BITSTREAMW;
```

The members of this structure are as follows:

Identifier

A NUL-terminated **wchar_t** string that identifies an FPGA device and package, in the same format as in the Identifier member of ADMXRC3_FPGA_INFOW. This string is typically used to verify that a bitstream (.BIT) file matches the target FPGA in a device, by comparing it with the Identifier member of ADMXRC3_FPGA_INFOW.

Length

The number of bytes in the variable-length array that begins with the Data member.

Data

The bitstream data, as a variable-length array. Although this member is defined to be an array of length 4, the actual number of bytes in the array is given by the Length member. This variable length array is suitable for passing to ADMXRC3_ConfigureFromBuffer.

Description

This structure represents a bitstream suitable for configuring a target FPGA, and is typically allocated and initialized by a call to ADMXRC3_LoadBitstreamW. Although applying the **sizeof** operator to this structure returns a value of a few tens of bytes, the structure should be regarded as the beginning of an object that could be many megabytes in size (for the latest FPGAs). The inverse function is ADMXRC3_UnloadBitstreamW, which deallocates the memory used by the object.

Remarks

This is the Unicode version of the ADMXRC3_BITSTREAM structure. ADMXRC3_BITSTREAM is actually a macro defined to be either ADMXRC3_BITSTREAMW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_BITSTREAMA.

4.2.4 ADMXRC3_BUFFER_HANDLE

Declaration

```
typedef ... ADMXRC3_BUFFER_HANDLE;
```

Description

This type is a handle representing a buffer that is locked into physical memory. A buffer handle is obtained by a call to ADMXRC3_Lock, and invalidated by a call to the inverse function, ADMXRC3_Unlock.

The following functions accept a parameter of this type, which identifies the buffer that a DMA transfer is to target:

- ADMXRC3_StartReadDMALocked
- ADMXRC3_StartWriteDMALocked
- ADMXRC3_ReadDMALocked
- ADMXRC3_WriteDMALocked

A value of zero is always invalid for a variable of type ADMXRC3_BUFFER_HANDLE. Buffer handles returned by ADMXRC3_Lock are always greater than zero.

Buffer handles are global to the system, so that if one process can lock a buffer and communicate the buffer handle to a second process, the second process can use it to perform DMA transfers.

Remarks

It is good practice to initialize variables of this type to zero, and to set such variables to zero after calling ADMXRC3_Unlock.

This type is a 32-bit unsigned integer. In Windows, it is a typedef of UINT32, while in Linux it is a typedef of uint32_t.

4.2.5 ADMXRC3_CARD_INFO

Declaration

```
typedef struct ... {
    uint32_t              SerialNumber;
    uint32_t              Reserved1;
    ADMXRC3_MODEL_TYPE    Model;
    unsigned int          NumClockGen;
    unsigned int          NumDmaChannel;
    unsigned int          NumFlashBank;
    unsigned int          NumMemoryBank;
    unsigned int          NumTargetFpga;
    unsigned int          NumWindow;
    uint32_t              MemoryBankPresent;
} ADMXRC3_CARD_INFO;
```

The members of this structure are as follows:

SerialNumber

SerialNumber is the serial number of device. Alpha Data guarantees serial numbers to be unique only for a particular model; it is possible for two devices that are different models to have the same serial number.

Reserved1

Must be ignored by application software.

Model

Model identifies the model or product, and is of type ADMXRC3_MODEL_TYPE.

NumClockGen

NumClockGen is the number of independently programmable clock generators in the device. The clockIndex parameter of the clock generator programming functions ADMXRC3_GetClockFrequency and ADMXRC3_SetClockFrequency must be less than this value.

NumDmaChannel

NumDmaChannel is the number of independently programmable DMA channels in the device. The dmaChannel parameter of the DMA functions such as ADMXRC3_ReadDMA must be less than this value.

NumFlashBank

NumFlashBank is the number of independent Flash memory banks in the device. The flashIndex parameter of Flash functions such as ADMXRC3_GetFlashInfo and ADMXRC3_GetFlashBlockInfo must be less than this value.

NumMemoryBank

NumMemoryBank is the number of independent memory banks in the device. The bankIndex parameter of ADMXRC3_GetBankInfo must be less than this value.

NumTargetFpga

NumFpga is the number of target FPGAs in the device. The 'targetIndex' parameter of the target FPGA configuration functions such as ADMXRC3_ConfigureFromBuffer must be less than this value.

NumWindow

NumWindow is the number of independent register/memory access windows in the device. The windowIndex parameter of functions that allow access to target FPGA registers (such as ADMXRC3_GetWindowInfo must be less than this value.

MemoryBankPresent

MemoryBankPresent is a bitmask where each bit that is 1 indicates that the corresponding memory bank is present. For example, if MemoryBankPresent were 0x0000000F and NumMemoryBank were 6, it would indicate that banks 0, 1, 2 and 3 are present whilst banks 4 and 5 are not present.

Description

A structure that describes a device, obtained from a call to ADMXRC3_GetCardInfo.

4.2.6 ADMXRC3_CARD_INFOEX

Declaration

```
typedef struct ... {
    uint32_t              SerialNumber;
    uint32_t              Reserved1;
    ADMXRC3_MODEL_TYPE    Model;
    unsigned int          NumClockGen;
    unsigned int          NumDmaChannel;
    unsigned int          NumFlashBank;
    unsigned int          NumMemoryBank;
    unsigned int          NumTargetFpga;
    unsigned int          NumWindow;
    uint32_t              MemoryBankPresent;
    unsigned int          NumSensor;
    unsigned int          NumModuleSite;
} ADMXRC3_CARD_INFOEX;
```

The members of this structure are as follows:

SerialNumber

SerialNumber is the serial number of device. Alpha Data guarantees serial numbers to be unique only for a particular model; it is possible for two devices that are different models to have the same serial number.

Reserved1

Must be ignored by application software.

Model

Model identifies the model or product, and is of type ADMXRC3_MODEL_TYPE.

NumClockGen

NumClockGen is the number of independently programmable clock generators in the device. The clockIndex parameter of the clock generator programming functions ADMXRC3_GetClockFrequency and ADMXRC3_SetClockFrequency must be less than this value.

NumDmaChannel

NumDmaChannel is the number of independently programmable DMA channels in the device. The dmaChannel parameter of the DMA functions such as ADMXRC3_ReadDMA must be less than this value.

NumFlashBank

NumFlashBank is the number of independent Flash memory banks in the device. The flashIndex parameter of ADMXRC3_GetFlashInfo and ADMXRC3_GetFlashBlockInfo must be less than this value.

NumMemoryBank

NumMemoryBank is the number of independent (non-Flash) memory banks in the device. The bankIndex parameter of ADMXRC3_GetBankInfo must be less than this value.

NumTargetFpga

NumFpga is the number of target FPGAs in the device. The 'targetIndex' parameter of the target FPGA configuration functions such as ADMXRC3_ConfigureFromBuffer must be less than this value.

NumWindow

NumWindow is the number of independent register/memory access windows in the device. The windowIndex parameter of functions that allow access to target FPGA registers (such as ADMXRC3_GetWindowInfo must be less than this value.

MemoryBankPresent

MemoryBankPresent is a bitmask where each bit that is 1 indicates that the corresponding memory bank is present. For example, if MemoryBankPresent were 0x0000000F and NumMemoryBank were 6, it would indicate that banks 0, 1, 2 and 3 are present whilst banks 4 and 5 are not present.

NumSensor

NumSensor is the number of sensors (for voltage, current, temperature etc.) in the device. The sensorIndex parameter of ADMXRC3_GetSensorInfo and ADMXRC3_ReadSensor must be less than this value.

NumModuleSite

NumModuleSite is the number of I/O personality module sites in the device. The moduleIndex parameter of ADMXRC3_GetModuleInfo must be less than this value.

Description

A structure that describes a device, obtained from a call to ADMXRC3_GetCardInfoEx.

Remarks

The information in this structure is a superset of that of ADMXRC3_CARD_INFO, and can therefore be used

instead of ADMXRC3_CARD_INFO in most situations.

4.2.7 ADMXRC3_DATA_TYPE

Declaration

```
typedef enum ... {
    ADMXRC3_DATA_BOOL          = 0,          /* BOOL / int */
    ADMXRC3_DATA_DOUBLE        = 1,          /* double */
    ADMXRC3_DATA_INT32         = 2,          /* INT32 / int32_t */
    ADMXRC3_DATA_UINT32        = 3,          /* UINT32 / uint32_t */
    ADMXRC3_DATA_LASTVALUE     = 4,          /* Reserved for future expansion */
   ADMXRC3_DATA_FORCE32BITS   = 0x7FFFFFFF  /* Force type to be at least 32 bits */
} ADMXRC3_DATA_TYPE;
```

Description

An enumerated type used to represent the type of data returned by the ADMXRC3_ReadSensor function. Values **ADMXRC3_DATA_LASTVALUE** and greater are reserved for adding new datatypes in future revisions of the API.

In the comments above, where a comment lists two possible types such as **BOOL** / **int**, the first type is the datatype in Windows and the second type is the datatype in Linux or VxWorks.

Remarks

This datatype is available in ADMXRC3 API version 1.1.0 and later.

4.2.8 ADMXRC3_DEVICE_STATUS

Declaration

```
typedef struct ... {
    uint32_t    Window;
    uint32_t    DMA;
    uint32_t    DirectMaster;
    uint32_t    Reserved1[3];
    uint32_t    ModelLow;
    uint32_t    ModelHigh;
} ADMXRC3_DEVICE_STATUS;
```

The members of this structure are as follows:

Window

Indicates error conditions for the Memory Windows in a reconfigurable computing device. See **ADMXRC3_DS_*XXX*** flags below.

DMA

Indicates error conditions for the DMA engines in a reconfigurable computing device. See **ADMXRC3_DMA_*XXX*** flags below.

DirectMaster

Indicates error conditions for the Direct Master OCP port in a reconfigurable computing device. See **ADMXRC3_DM_*XXX*** flags below.

Reserved1

Must be ignored by application software.

ModelLow

Indicates model-specific error conditions. Currently, this field has no definitions.

ModelHigh

Indicates additional model-specific error conditions. Currently, this field has no definitions.

Description

This structure provides high-level information about non-transient error states (i.e. errors that do not self-clear) in which a reconfigurable configurable computing device may be. The values returned in the fields of this structure are generally the bitwise OR of a number of possible error conditions.

Error conditions that apply to Memory Windows are represented by flags whose names are of the form **ADMXRC3_DS_*XXX***:

- ADMXRC3_DS_LOCAL_TIMEOUT(n) where *n* is the index of a memory window
  This flag indicates that an OCP transaction on memory window *n* timed out before it could be completed.

Error conditions that apply to DMA engines are represented by flags whose names are of the form **ADMXRC3_DMA_*XXX***:

- ADMXRC3_DMA_BUS_TIMEOUT(n) where *n* is the index of a DMA engine
  This flag indicates that a transaction initiated by DMA engine *n* on the host system's I/O bus timed out before it could be completed.

Error conditions that apply to the Direct Master channel are represented by flags whose names are of the form **ADMXRC3_DM_*XXX***:

- ADMXRC3_DM_BUS_TIMEOUT
  This flag indicates that a transaction initiated by the Direct Master channel on the host system's I/O bus timed out before it could be completed.
- ADMXRC3_DM_BAD_TRANSACTION
  This flag indicates that an OCP command submitted to the Direct Master channel's OCP port in the FPGA design had an unsupported/illegal command or length.

Remarks

This datatype is available in ADMXRC3 API version 1.6.0 and later.

4.2.9 ADMXRC3_FAMILY_TYPE

Declaration

```
typedef enum ... {
    ADMXRC3_FAMILY_4K            = 0,          /* XC4XXX */
    ADMXRC3_FAMILY_VIRTEX        = 1,          /* Virtex, Virtex-E or Virtex-EM */
    ADMXRC3_FAMILY_VIRTEX2       = 2,          /* Virtex-II */
    ADMXRC3_FAMILY_VIRTEX2P      = 3,          /* Virtex-II Pro */
    ADMXRC3_FAMILY_VIRTEX4       = 4,          /* Virtex-4 */
    ADMXRC3_FAMILY_VIRTEX5       = 5,          /* Virtex-5 */
    ADMXRC3_FAMILY_VIRTEX6       = 6,          /* Virtex-6 */
    ADMXRC3_FAMILY_7SERIES       = 7,          /* 7 Series */
    ADMXRC3_FAMILY_ULTRASCALE    = 8,          /* Ultrascale */
   ADMXRC3_FAMILY_FORCE32BITS  = 0x7FFFFFFF  /* Force type to be at least 32 bits */
} ADMXRC3_FAMILY_TYPE;
```

Description

An enumerated type used to represent an FPGA device family. Values ADMXRC3_FPGA_LASTVALUE and greater are reserved for adding support for future models.

Remarks

The value **ADMXRC3_FAMILY_ULTRASCALE** is available in ADMXRC3 API version 1.7.2 and later.

4.2.10 ADMXRC3_FLASH_INFOA

Declaration

```
typedef struct ... {
    char        Identifier[32];
    uint64_t    Size;
    uint64_t    UseableStart;
    uint64_t    UseableLength;
} ADMXRC3_FLASH_INFOA;
```

The members of this structure are as follows:

Identifier

A NUL-terminated **char** string containing the name of the Flash device, provided primarily for diagnostic purposes.

Size

The total size of the Flash memory bank, in bytes.

UseableStart

The starting address within the Flash memory bank of the region that is available for use by applications.

UseableLength

The length, in bytes, of the region that is available for use by applications.

Description

This structure provides high-level information about a Flash memory bank. It is returned by a call to ADMXRC3_GetFlashInfo.

On some models, only a part of a Flash memory bank is available for use by applications. The UseableStart and UseableLength members indicate the bounds of this region. Attempting to erase, read or write data outside of this region using ADMXRC3_EraseFlash etc. fails.

Remarks

This is the ANSI / UTF-8 version of the ADMXRC3_FLASH_INFO structure. ADMXRC3_FLASH_INFO is actually a macro defined to be either ADMXRC3_FLASH_INFOW if the _**UNICODE** preprocessor symbol is defined, or else ADMXRC3_FLASH_INFOA.

4.2.11 ADMXRC3_FLASH_INFOW

Declaration

```
typedef struct ... {
    wchar_t     Identifier[32];
    uint64_t    Size;
    uint64_t    UseableStart;
    uint64_t    UseableLength;
} ADMXRC3_FLASH_INFOW;
```

The members of this structure are as follows:

Identifier

A NUL-terminated **wchar_t** string containing the name of the Flash device, provided primarily for diagnostic purposes.

Size

The total size, in bytes, of the Flash memory bank.

UseableStart

The starting byte address within the Flash memory bank of the region that is available for use by applications.

UseableLength

The length, in bytes, of the region that is available for use by applications.

Description

This structure provides high-level information about a Flash memory bank. It is returned by a call to ADMXRC3_GetFlashInfo.

On some models, only a part of a Flash memory bank is available for use by applications. The UseableStart and UseableLength members indicate the bounds of this region. Attempting to erase, read or write data outside of this region using ADMXRC3_EraseFlash etc. fails.

Remarks

This is the Unicode version of the ADMXRC3_FLASH_INFO structure. ADMXRC3_FLASH_INFO is actually a macro defined to be either ADMXRC3_FLASH_INFOW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_FLASH_INFOA.

4.2.12 ADMXRC3_FLASHBLOCK_INFO

Declaration

```
typedef struct ... {
    uint64_t    Address;
    uint64_t    Length;
    uint32_t    Flags;
} ADMXRC3_FLASHBLOCK_INFO;
```

The members of this structure are as follows:

Address

The starting address, in bytes, of the Flash block.

Length

The length, in bytes, of the Flash block.

Flags

Bitwise-OR of flags that represent attributes of the Flash block. Currently, the following flags are defined:

- ADMXRC3_FLASHBLOCK_BOOT
  Indicates that the block is a "boot block".

Description

This structure specifies the bounds of a Flash block within a Flash memory bank. An application determines which Flash block contains a particular location by calling ADMXRC3_GetFlashBlockInfo, which returns this structure.

Remarks

Flash devices that do not have a block-oriented architecture simply define a single block that is the entire Flash device.

4.2.13 ADMXRC3_FPGA_INFOA

Declaration

```
typedef struct ... {
    char                    Identifier[32];
    ADMXRC3_FPGA_TYPE       DeviceCode;
```

```
    ADMXRC3_FAMILY_TYPE       FamilyCode;
    ADMXRC3_SUBFAMILY_TYPE    SubfamilyCode;
    ADMXRC3_PACKAGE_TYPE      PackageCode;
    char                      Device[16];
    char                      Package[16];
    uint32_t                  Flags;
    char                      SpeedGrade[8];
    char                      Stepping[8];
    boolean_t                 Present;
} ADMXRC3_FPGA_INFOA;
```

The members of this structure are as follows:

Identifier

A NUL-terminated **char** string that identifies an FPGA die and package. The string is in a format compatible with that of a bitstream (.BIT) file, and thus may be usefully compared with the Identifier member of ADMXRC3_BITSTREAMA to determine whether or not a target FPGA matches the bitstream file. An example string is "5vsx240tff1738".

DeviceCode

Identifies the FPGA family and die size of the target FPGA, but yields no information about the package. This member is one of the values of ADMXRC3_FPGA_TYPE.

FamilyCode

Identifies the FPGA family to which the target FPGA belongs. This member is one of the values of ADMXRC3_FAMILY_TYPE.

SubfamilyCode

Identifies the FPGA subfamily to which the target FPGA belongs. This member is one of the values of ADMXRC3_SUBFAMILY_TYPE.

PackageCode

The package of the FPGA is encoded in this member. Refer to ADMXRC3_PACKAGE_TYPE for information on how a package is encoded.

Device

A NUL-terminated **char** string that identifies an FPGA die, excluding the package. For example, "5VLX220T".

Package

A NUL-terminated **char** string that identifies an FPGA package. For example, "FG456".

Flags

A bitwise-OR of flags that indicate which of the following members are valid. Currently, the following flags are defined:

- ADMXRC3_FPGA_SPEEDVALID
  Indicates that the SpeedGrade member is valid.
- ADMXRC3_FPGA_STEPPINGVALID
  Indicates that the Stepping member is valid.
- ADMXRC3_FPGA_NOTCONFIGURABLE
  Indicates that FPGA in question is not configurable via functions such as ADMXRC3_ConfigureFromFile. This flag is available in ADMXRC3 API version 1.4.0 and later; see remarks below.

SpeedGrade

A NUL-terminated **char** string that identifies the speed grade and temperature grade. If the Flags member

(see above) contains ADMXRC3_FPGA_SPEEDVALID, then this member is valid. Otherwise, it is not valid, and speed grade information is not available for the target FPGA.

The last character of the string represents the temperature grade, which can be 'I' for industrial or 'C' for commercial. Example strings are are: "4C" (Virtex-II commercial), "1C" (Virtex-5 commercial), and "10I" (Virtex-4 industrial). Correct interpretation of speed grade strings requires knowledge of the FPGA family and subfamily.

Stepping

A NUL-terminated **char** string that identifies the stepping level. If the Flags member (see above) contains ADMXRC3_FPGA_STEPPINGVALID, then this member is valid. Otherwise, it is not valid, and stepping level information is not available for the target FPGA.

Example strings are "ES", "0", "1", etc. Correct interpretation of stepping level strings requires knowledge of the FPGA family and subfamily.

Present

TRUE if the site for the target FPGA is physically populated with an FPGA, otherwise FALSE. If this member is FALSE, none of the above members are valid.

Description

This structure describes a target FPGA and is returned by ADMXRC3_GetFpgaInfoA.

Remarks

This is the ANSI / UTF-8 version of the ADMXRC3_FPGA_INFO structure. ADMXRC3_FPGA_INFO is actually a macro defined to be either ADMXRC3_FPGA_INFOW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_FPGA_INFO.

The most common reason for the **ADMXRC3_FPGA_NOTCONFIGURABLE** flag being present is that the FPGA in question is both the PCI Express interface and the target FPGA. Reconfiguring it would destroy all PCI-E configuration state, rendering it unusable until the next system reset. An FPGA for which this flag is present uses an alternative method of programming such as (i) JTAG or (ii) by programming a bitstream into a nonvolatile memory, from which the FPGA is automatically configured at power-up.

4.2.14 ADMXRC3_FPGA_INFOW

Declaration

```
typedef struct ... {
    wchar_t                 Identifier[32];
    ADMXRC3_FPGA_TYPE       DeviceCode;
    ADMXRC3_FAMILY_TYPE     FamilyCode;
    ADMXRC3_SUBFAMILY_TYPE  SubfamilyCode;
    ADMXRC3_PACKAGE_TYPE    PackageCode;
    wchar_t                 Device[16];
    wchar_t                 Package[16];
    uint32_t                Flags;
    wchar_t                 SpeedGrade[8];
    wchar_t                 Stepping[8];
    boolean_t               Present;
} ADMXRC3_FPGA_INFOW;
```

The members of this structure are as follows:

Identifier

A NUL-terminated **wchar_t** string that identifies an FPGA die and package. The string is in a format compatible with that of a bitstream (.BIT) file, and thus may be usefully compared with the Identifier member of ADMXRC3_BITSTREAMA to determine whether or not a target FPGA matches the bitstream

file. An example string is "5vsx240tff1738".

DeviceCode

Identifies the FPGA family and die size of the target FPGA, but yields no information about the package. This member is one of the values of ADMXRC3_FPGA_TYPE.

FamilyCode

Identifies the FPGA family to which the target FPGA belongs. This member is one of the values of ADMXRC3_FAMILY_TYPE.

SubfamilyCode

Identifies the FPGA subfamily to which the target FPGA belongs. This member is one of the values of ADMXRC3_SUBFAMILY_TYPE.

PackageCode

The package of the FPGA is encoded in this member. Refer to ADMXRC3_PACKAGE_TYPE for information on how a package is encoded.

Device

A NUL-terminated **wchar_t** string that identifies an FPGA die, excluding the package. For example, "5VLX220T".

Package

A NUL-terminated **wchar_t** string that identifies an FPGA package. For example, "FG456".

Flags

A bitwise-OR of flags that indicate which of the following members are valid. Currently, the following flags are defined:

- ADMXRC3_FPGA_SPEEDVALID
  Indicates that the SpeedGrade member is valid.
- ADMXRC3_FPGA_STEPPINGVALID
  Indicates that the Stepping member is valid.
- ADMXRC3_FPGA_NOTCONFIGURABLE
  Indicates that FPGA in question is not configurable via functions such as ADMXRC3_ConfigureFromFile. This flag is available in ADMXRC3 API version 1.4.0 and later; see remarks below.

SpeedGrade

A NUL-terminated **wchar_t** string that identifies the speed grade and temperature grade. If the Flags member (see above) contains ADMXRC3_FPGA_SPEEDVALID, then this member is valid. Otherwise, it is not valid, and speed grade information is not available for the target FPGA.

The last character of the string represents the temperature grade, which can be 'I' for industrial or 'C' for commercial. Example strings are are: "4C" (Virtex-II commercial), "1C" (Virtex-5 commercial), and "10I" (Virtex-4 industrial). Correct interpretation of speed grade strings requires knowledge of the FPGA family and subfamily.

Stepping

A NUL-terminated **wchar_t** string that identifies the stepping level. If the Flags member (see above) contains ADMXRC3_FPGA_STEPPINGVALID, then this member is valid. Otherwise, it is not valid, and stepping level information is not available for the target FPGA.

Example strings are "ES", "0", "1", etc. Correct interpretation of stepping level strings requires knowledge of the FPGA family and subfamily.

Present

TRUE if the site for the target FPGA is physically populated with an FPGA, otherwise FALSE. If this member is FALSE, none of the above members are valid.

Description

This structure describes a target FPGA and is returned by ADMXRC3_GetFpgaInfoW.

Remarks

This is the Unicode version of the ADMXRC3_FPGA_INFO structure. ADMXRC3_FPGA_INFO is actually a macro defined to be either ADMXRC3_FPGA_INFOW if the _**UNICODE** preprocessor symbol is defined, or else ADMXRC3_FPGA_INFO.

The most common reason for the **ADMXRC3_FPGA_NOTCONFIGURABLE** flag being present is that the FPGA in question is both the PCI Express interface and the target FPGA. Reconfiguring it would destroy all PCI-E configuration state, rendering it unusable until the next system reset. An FPGA for which this flag is present uses an alternative method of programming such as (i) JTAG or (ii) by programming a bitstream into a nonvolatile memory, from which the FPGA is automatically configured at power-up.

4.2.15 ADMXRC3_FPGA_TYPE

Declaration

```
typedef enum ... {
    ADMXRC3_FPGA_V1000          = 4,          /* XCV1000 (Virtex) */
    ADMXRC3_FPGA_V400           = 5,          /* XCV400 (Virtex) */
    ADMXRC3_FPGA_V600           = 6,          /* XCV600 (Virtex) */
    ADMXRC3_FPGA_V800           = 7,          /* XCV800 (Virtex) */
    ADMXRC3_FPGA_V2000E         = 8,          /* XCV2000E (Virtex-E) */
    ADMXRC3_FPGA_V1000E         = 9,          /* XCV1000E (Virtex-E) */
    ADMXRC3_FPGA_V1600E         = 10,         /* XCV1600E (Virtex-E) */
    ADMXRC3_FPGA_V3200E         = 11,         /* XCV3200E (Virtex-E) */
    ADMXRC3_FPGA_V812E          = 12,         /* XCV812E (Virtex-EM) */
    ADMXRC3_FPGA_V405E          = 13,         /* XCV405E (Virtex-EM) */
    ADMXRC3_FPGA_2V1000         = 32,         /* XC2V1000 (Virtex-II) */
    ADMXRC3_FPGA_2V1500         = 33,         /* XC2V1500 (Virtex-II) */
    ADMXRC3_FPGA_2V2000         = 34,         /* XC2V2000 (Virtex-II) */
    ADMXRC3_FPGA_2V3000         = 35,         /* XC2V3000 (Virtex-II) */
    ADMXRC3_FPGA_2V4000         = 36,         /* XC2V4000 (Virtex-II) */
    ADMXRC3_FPGA_2V6000         = 37,         /* XC2V6000 (Virtex-II) */
    ADMXRC3_FPGA_2V8000         = 38,         /* XC2V8000 (Virtex-II) */
    ADMXRC3_FPGA_2V10000        = 39,         /* XC2V10000 (Virtex-II) */
    ADMXRC3_FPGA_2VP2           = 64,         /* XC2VP2 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP4           = 65,         /* XC2VP4 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP7           = 66,         /* XC2VP7 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP20          = 67,         /* XC2VP20 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP30          = 68,         /* XC2VP30 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP40          = 69,         /* XC2VP40 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP50          = 70,         /* XC2VP50 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP100         = 71,         /* XC2VP100 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP125         = 72,         /* XC2VP125 (Virtex-II Pro) */
    ADMXRC3_FPGA_2VP70          = 73,         /* XC2VP70 (Virtex-II Pro) */
    ADMXRC3_FPGA_4VLX15         = 96,         /* XC4VLX15 (Virtex-4 LX) */
    ADMXRC3_FPGA_4VLX25         = 97,         /* XC4VLX25 (Virtex-4 LX) */
    ADMXRC3_FPGA_4VLX40         = 98,         /* XC4VLX40 (Virtex-4 LX) */
    ADMXRC3_FPGA_4VLX60         = 99,         /* XC4VLX60 (Virtex-4 LX) */
    ADMXRC3_FPGA_4VLX100        = 100,        /* XC4VLX100 (Virtex-4 LX) */
    ADMXRC3_FPGA_4VLX160        = 101,        /* XC4VLX160 (Virtex-4 LX) */
    ADMXRC3_FPGA_4VLX200        = 102,        /* XC4VLX200 (Virtex-4 LX) */
```

```
    ADMXRC3_FPGA_4VLX80         = 103,      /* XC4VLX80 (Virtex-4 LX) */
    ADMXRC3_FPGA_4VSX25         = 104,      /* XC4VSX25 (Virtex-4 SX) */
    ADMXRC3_FPGA_4VSX35         = 105,      /* XC4VSX35 (Virtex-4 SX) */
    ADMXRC3_FPGA_4VSX55         = 106,      /* XC4VSX55 (Virtex-4 SX) */
    ADMXRC3_FPGA_4VFX12         = 112,      /* XC4VFX12 (Virtex-4 FX) */
    ADMXRC3_FPGA_4VFX20         = 113,      /* XC4VFX20 (Virtex-4 FX) */
    ADMXRC3_FPGA_4VFX40         = 114,      /* XC4VFX40 (Virtex-4 FX) */
    ADMXRC3_FPGA_4VFX60         = 115,      /* XC4VFX60 (Virtex-4 FX) */
    ADMXRC3_FPGA_4VFX100        = 116,      /* XC4VFX100 (Virtex-4 FX) */
    ADMXRC3_FPGA_4VFX140        = 117,      /* XC4VFX140 (Virtex-4 FX) */
    ADMXRC3_FPGA_5VLX30         = 128,      /* XC5VLX30 (Virtex-5 LX) */
    ADMXRC3_FPGA_5VLX50         = 129,      /* XC5VLX50 (Virtex-5 LX) */
    ADMXRC3_FPGA_5VLX85         = 130,      /* XC5VLX85 (Virtex-5 LX) */
    ADMXRC3_FPGA_5VLX110        = 131,      /* XC5VLX110 (Virtex-5 LX) */
    ADMXRC3_FPGA_5VLX220        = 132,      /* XC5VLX220 (Virtex-5 LX) */
    ADMXRC3_FPGA_5VLX330        = 133,      /* XC5VLX330 (Virtex-5 LX) */
    ADMXRC3_FPGA_5VLX30T        = 136,      /* XC5VLX30T (Virtex-5 LXT) */
    ADMXRC3_FPGA_5VLX50T        = 137,      /* XC5VLX50T (Virtex-5 LXT) */
    ADMXRC3_FPGA_5VLX85T        = 138,      /* XC5VLX85T (Virtex-5 LXT) */
    ADMXRC3_FPGA_5VLX110T       = 139,      /* XC5VLX110T (Virtex-5 LXT) */
    ADMXRC3_FPGA_5VLX330T       = 140,      /* XC5VLX330T (Virtex-5 LXT) */
    ADMXRC3_FPGA_5VLX220T       = 141,      /* XC5VLX22T (Virtex-5 LXT) */
    ADMXRC3_FPGA_5VLX155T       = 142,      /* XC5VLX155T (Virtex-5 LXT) */
    ADMXRC3_FPGA_5VSX35T        = 144,      /* XC5VSX35T (Virtex-5 SXT) */
    ADMXRC3_FPGA_5VSX50T        = 145,      /* XC5VSX50T (Virtex-5 SXT) */
    ADMXRC3_FPGA_5VSX95T        = 146,      /* XC5VSX95T (Virtex-5 SXT) */
    ADMXRC3_FPGA_5VSX240T       = 147,      /* XC5VSX240T (Virtex-5 SXT) */
    ADMXRC3_FPGA_5VFX100T       = 152,      /* XC5VFX100T (Virtex-5 FXT) */
    ADMXRC3_FPGA_5VFX130T       = 153,      /* XC5VFX130T (Virtex-5 FXT) */
    ADMXRC3_FPGA_5VFX200T       = 154,      /* XC5VFX200T (Virtex-5 FXT) */
    ADMXRC3_FPGA_5VFX30T        = 155,      /* XC5VFX30T (Virtex-5 FXT) */
    ADMXRC3_FPGA_5VFX70T        = 156,      /* XC5VFX70T (Virtex-5 FXT) */
    ADMXRC3_FPGA_6VLX760        = 160,      /* XC6VLX760 (Virtex-6 LX) */
    ADMXRC3_FPGA_6VLX75T        = 168,      /* XC6VLX75T (Virtex-6 LXT) */
    ADMXRC3_FPGA_6VLX130T       = 169,      /* XC6VLX130T (Virtex-6 LXT) */
    ADMXRC3_FPGA_6VLX195T       = 170,      /* XC6VLX195T (Virtex-6 LXT) */
    ADMXRC3_FPGA_6VLX240T       = 171,      /* XC6VLX240T (Virtex-6 LXT) */
    ADMXRC3_FPGA_6VLX365T       = 172,      /* XC6VLX365T (Virtex-6 LXT) */
    ADMXRC3_FPGA_6VLX550T       = 173,      /* XC6VLX550T (Virtex-6 LXT) */
    ADMXRC3_FPGA_6VSX315T       = 176,      /* XC6VLX315T (Virtex-6 SXT) */
    ADMXRC3_FPGA_6VSX475T       = 177,      /* XC6VLX475T (Virtex-6 SXT) */
    ADMXRC3_FPGA_6VHX250T       = 184,      /* XC6VHX250T (Virtex-6 HXT) */
    ADMXRC3_FPGA_6VHX255T       = 185,      /* XC6VHX255T (Virtex-6 HXT) */
    ADMXRC3_FPGA_6VHX385T       = 186,      /* XC6VHX385T (Virtex-6 HXT) */
    ADMXRC3_FPGA_6VHX565T       = 187,      /* XC6VHX565T (Virtex-6 HXT) */
    ADMXRC3_FPGA_7A100T         = 224,      /* XC7A100T (Artix-7) */
    ADMXRC3_FPGA_7A200T         = 225,      /* XC7A200T (Artix-7) */
    ADMXRC3_FPGA_7A350T         = 226,      /* XC7A350T (Artix-7) */
    ADMXRC3_FPGA_7K70T          = 232,      /* XC7K70T (Kintex-7T) */
    ADMXRC3_FPGA_7K160T         = 233,      /* XC7K160T (Kintex-7T) */
    ADMXRC3_FPGA_7K325T         = 234,      /* XC7K325T (Kintex-7T) */
    ADMXRC3_FPGA_7K355T         = 235,      /* XC7K355T (Kintex-7T) */
    ADMXRC3_FPGA_7K410T         = 236,      /* XC7K410T (Kintex-7T) */
    ADMXRC3_FPGA_7K420T         = 237,      /* XC7K420T (Kintex-7T) */
    ADMXRC3_FPGA_7K480T         = 238,      /* XC7K480T (Kintex-7T) */
    ADMXRC3_FPGA_7V585T         = 248,      /* XC7V585T (Virtex-7T) */
    ADMXRC3_FPGA_7V1500T        = 249,      /* XC7V1500T (Virtex-7T) */
    ADMXRC3_FPGA_7V2000T        = 250,      /* XC7V2000T (Virtex-7T) */
```

```
    ADMXRC3_FPGA_7VX330T       = 256,        /* XC7VX330T (Virtex-7XT) */
    ADMXRC3_FPGA_7VX415T       = 257,        /* XC7VX415T (Virtex-7XT) */
    ADMXRC3_FPGA_7VX485T       = 258,        /* XC7VX485T (Virtex-7XT) */
    ADMXRC3_FPGA_7VX550T       = 259,        /* XC7VX550T (Virtex-7XT) */
    ADMXRC3_FPGA_7VX690T       = 260,        /* XC7VX690T (Virtex-7XT) */
    ADMXRC3_FPGA_7VX980T       = 261,        /* XC7VX980T (Virtex-7XT) */
    ADMXRC3_FPGA_7VX1140T      = 262,        /* XC7VX1140T (Virtex-7XT) */
    ADMXRC3_FPGA_7VH290T       = 272,        /* XC7VH290T (Virtex-7HT) */
    ADMXRC3_FPGA_7VH580T       = 273,        /* XC7VH580T (Virtex-7HT) */
    ADMXRC3_FPGA_7VH870T       = 274,        /* XC7VH870T (Virtex-7HT) */
    ADMXRC3_FPGA_7Z010         = 288,        /* XC7Z010 (Zynq-7000) */
    ADMXRC3_FPGA_7Z015         = 289,        /* XC7Z015 (Zynq-7000) */
    ADMXRC3_FPGA_7Z020         = 290,        /* XC7Z020 (Zynq-7000) */
    ADMXRC3_FPGA_7Z030         = 291,        /* XC7Z030 (Zynq-7000) */
    ADMXRC3_FPGA_7Z045         = 292,        /* XC7Z045 (Zynq-7000) */
    ADMXRC3_FPGA_7Z100         = 293,        /* XC7Z100 (Zynq-7000) */
    ADMXRC3_FPGA_KU035         = 321,        /* XCKU035 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU040         = 322,        /* XCKU040 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU060         = 323,        /* XCKU060 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU075         = 324,        /* XCKU075 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU100         = 325,        /* XCKU100 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU115         = 326,        /* XCKU115 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU025         = 327,        /* XCKU025 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU085         = 328,        /* XCKU085 (Kintex Ultrascale) */
    ADMXRC3_FPGA_KU095         = 329,        /* XCKU095 (Kintex Ultrascale) */
    ADMXRC3_FPGA_VU065         = 333,        /* XCKU115 (Virtex Ultrascale) */
    ADMXRC3_FPGA_VU080         = 334,        /* XCVU080 (Virtex Ultrascale) */
    ADMXRC3_FPGA_VU095         = 335,        /* XCVU095 (Virtex Ultrascale) */
    ADMXRC3_FPGA_VU125         = 336,        /* XCVU125 (Virtex Ultrascale) */
    ADMXRC3_FPGA_VU145         = 337,        /* XCVU145 (Virtex Ultrascale) */
    ADMXRC3_FPGA_VU160         = 338,        /* XCVU160 (Virtex Ultrascale) */
    ADMXRC3_FPGA_VU440         = 339,        /* XCVU440 (Virtex Ultrascale) */
    ADMXRC3_FPGA_VU190         = 340,        /* XCVU190 (Virtex Ultrascale) */
    ADMXRC3_FPGA_KU3P          = 348,        /* XCKU3P (Kintex Ultrascale+) */
    ADMXRC3_FPGA_KU5P          = 349,        /* XCKU5P (Kintex Ultrascale+) */
    ADMXRC3_FPGA_KU9P          = 351,        /* XCKU9P (Kintex Ultrascale+) */
    ADMXRC3_FPGA_KU11P         = 352,        /* XCKU11P (Kintex Ultrascale+) */
    ADMXRC3_FPGA_KU13P         = 353,        /* XCKU13P (Kintex Ultrascale+) */
    ADMXRC3_FPGA_KU15P         = 354,        /* XCKU15P (Kintex Ultrascale+) */
    ADMXRC3_FPGA_VU3P          = 361,        /* XCVU3P (Virtex Ultrascale+) */
    ADMXRC3_FPGA_VU5P          = 362,        /* XCVU5P (Virtex Ultrascale+) */
    ADMXRC3_FPGA_VU7P          = 363,        /* XCVU7P (Virtex Ultrascale+) */
    ADMXRC3_FPGA_VU9P          = 364,        /* XCVU9P (Virtex Ultrascale+) */
    ADMXRC3_FPGA_VU11P         = 365,        /* XCVU11P (Virtex Ultrascale+) */
    ADMXRC3_FPGA_VU13P         = 366,        /* XCVU13P (Virtex Ultrascale+) */
    ADMXRC3_FPGA_ZU2EG         = 377,        /* XCZU2EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU3EG         = 378,        /* XCZU3EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU4EV         = 379,        /* XCZU4EV (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU5EV         = 380,        /* XCZU5EV (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU6EG         = 381,        /* XCZU6EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU7EV         = 382,        /* XCZU7EV (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU9EG         = 384,        /* XCZU9EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU11EG        = 386,        /* XCZU11EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU15EG        = 390,        /* XCZU15EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU17EG        = 392,        /* XCZU17EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_ZU19EG        = 394,        /* XCZU19EG (Zynq Ultrascale+) */
    ADMXRC3_FPGA_LASTVALUE     = 400,        /* Reserved for future products */
  ADMXRC3_FPGA_FORCE32BITS   = 0x7FFFFFFF  /* Force type to be at least 32 bits */
```

```
} ADMXRC3_FPGA_TYPE;
```

Description

An enumerated type used to represent an FPGA device, excluding package and speed grade information. Values beginning with ADMXRC3_FPGA_LASTVALUE are reserved for adding support for future models.

Remarks

The values **ADMXRC3_FPGA_7Z*XXX***, **ADMXRC3_FPGA_KU*XXX*** (excluding _KU025, _KU085 & _KU095) & **ADMXRC3_FPGA_VU*XXX*** are available in ADMXRC3 API version 1.7.2 and later.

The values **ADMXRC3_FPGA_KU025**, **ADMXRC3_FPGA_KU085** & **ADMXRC3_FPGA_KU095**, **ADMXRC3_FPGA_KU*XX*P**, **ADMXRC3_FPGA_VU*XX*P**, **ADMXRC3_FPGA_ZU*XX*EG** & **ADMXRC3_FPGA_ZU*XX*EV** are available in ADMXRC3 API version 1.8.1 and later.

4.2.16 ADMXRC3_HANDLE

Declaration

```
typedef ... ADMXRC3_HANDLE;
```

Description

A handle type used to represent a device handle. Most functions in the ADMXRC3 API require a valid device handle of type ADMXRC3_HANDLE as the first parameter. A valid device handle is obtained via a call to ADMXRC3_Open. When an application has finished with a device handle, it should close it by calling ADMXRC3_Close.

Remarks

It is good practice to initialize device handles (i.e. variables of type ADMXRC3_HANDLE) to the value ADMXRC3_HANDLE_INVALID_VALUE, and to always set such variables to ADMXRC3_HANDLE_INVALID_VALUE after closing them with ADMXRC3_Close. This enables an application to avoid attempting to close invalid device handles.

In Windows, this type is a typedef of **HANDLE**. A value of type ADMXRC3_HANDLE can therefore be usefully passed to certain functions that accept a **HANDLE**, notably **WaitForSingleObject** and **WaitForMultipleObjects**. **CloseHandle** should not be used to close a device handle, as this prevents the user-mode part of the ADMXRC3 API from performing cleanup operations, resulting in resource leaks. Instead, ADMXRC3_Close should be used.

In Linux and VxWorks, this type is a typedef of a file descriptor, i.e. **int**. A value of type ADMXRC3_HANDLE can therefore be usefully passed to certain functions that accept file descriptors, such as **poll**. The **api** system call should not be used to close a device handle, as this prevents the user-mode part of the ADMXRC3 API from performing cleanup operations, resulting in resource leaks. Instead, ADMXRC3_Close should be used.

4.2.17 ADMXRC3_MODEL_TYPE

Declaration

```
typedef enum ... {
    ADMXRC3_MODEL_GENERIC          = 0,          /* Unknown model */
    ADMXRC3_MODEL_ADMXRC2          = 4,          /* ADM-XRC-II */
    ADMXRC3_MODEL_ADPEXRC5T        = 0x100,      /* ADPe-XRC-5T */
    ADMXRC3_MODEL_ADMXRC6TL        = 0x101,      /* ADM-XRC-6TL */
    ADMXRC3_MODEL_ADMXRC6T1        = 0x102,      /* ADM-XRC-6T1 */
    ADMXRC3_MODEL_ADMXRC6TGE       = 0x103,      /* ADM-XRC-6TGE */
    ADMXRC3_MODEL_ADMXRC6TADV8     = 0x104,      /* ADM-XRC-6T-ADV8 */
    ADMXRC3_MODEL_ADPEXRC6T        = 0x105,      /* ADPE-XRC-6T */
    ADMXRC3_MODEL_ADPEXRC6TL       = 0x106,      /* ADPE-XRC-6T-L */
```

```
    ADMXRC3_MODEL_ADPEXRC6TADV_C   = 0x107,      /* ADPE-XRC-6T-ADV (controller) */
     ADMXRC3_MODEL_ADPEXRC6TADV_T   = 0x108,      /* ADPE-XRC-6T-ADV (target) */
     ADMXRC3_MODEL_ADMXRC7K1        = 0x109,      /* ADM-XRC-7K1 */
     ADMXRC3_MODEL_ADMXRC6TDA1      = 0x10A,      /* ADM-XRC-6T-DA1 */
     ADMXRC3_MODEL_ADMXRC7V1        = 0x10B,      /* ADM-XRC-7V1 */
     ADMXRC3_MODEL_ADMVPX37V2       = 0x10C,      /* ADM-VPX3-7V2 */
     ADMXRC3_MODEL_ADMXRC6TGEL      = 0x10D,      /* ADM-XRC-6TGEL */
     ADMXRC3_MODEL_ADMPCIE7V3       = 0x10E,      /* ADM-PCIE-7V3 */
     ADMXRC3_MODEL_ADMXRC7Z1        = 0x10F,      /* ADM-XRC-7Z1 */
     ADMXRC3_MODEL_ADMXRC7Z2        = 0x110,      /* ADM-XRC-7Z2 */
     ADMXRC3_MODEL_ADMXRCKU1        = 0x111,      /* ADM-XRC-KU1 */
     ADMXRC3_MODEL_ADMPCIEKU3       = 0x113,      /* ADM-PCIE-KU3 */
     ADMXRC3_MODEL_ADMXRCKU1_P5     = 0x114,      /* ADM-XRC-KU1 (P5 endpoint) */
     ADMXRC3_MODEL_ADMXRCKU1_P6     = 0x115,      /* ADM-XRC-KU1 (P6 endpoint) */
     ADMXRC3_MODEL_ADMPCIE8V3       = 0x116,      /* ADM-PCIE-8V3 */
     ADMXRC3_MODEL_ADMPCIE8K5       = 0x117,      /* ADM-PCIE-8K5 */
     ADMXRC3_MODEL_LASTVALUE,                     /* Reserved for future models */
   ADMXRC3_MODEL_FORCE32BITS    = 0x7FFFFFFF  /* Force type to be at least 32 bits */
} ADMXRC3_MODEL_TYPE;
```

Description

A datatype used to represent a model or product in Alpha Data's reconfigurable computing range. Legacy hardware is represented by a value that is nonzero and less than 0x100. The API does not return the value ADMXRC3_MODEL_GENERIC under normal circumstances, but if it does, it may indicate that the device is experimental or development hardware. Values beginning with ADMXRC3_MODEL_LASTVALUE are reserved for future models.

Remarks

The values **ADMXRC3_MODEL_ADMXRC6TGE** and **ADMXRC3_MODEL_ADMXRC6TADV8** are available in ADMXRC3 API version 1.3.0 and later.

The values **ADMXRC3_MODEL_ADPEXRC6T**, **ADMXRC3_MODEL_ADPEXRC6TL**, **ADMXRC3_MODEL_ADPEXRC6TADV_C**, **ADMXRC3_MODEL_ADPEXRC6TADV_T**, **ADMXRC3_MODEL_ADMXRC7K1**, **ADMXRC3_MODEL_ADMXRC6TDA1** and **ADMXRC3_MODEL_ADMXRC7V1** are available in ADMXRC3 API version 1.5.1 and later.

The value **ADMXRC3_MODEL_ADMVPX37V2** is available in ADMXRC3 API version 1.6.0 and later.

The value **ADMXRC3_MODEL_ADMXRC6TGEL** is available in ADMXRC3 API version 1.7.0 and later.

The value **ADMXRC3_MODEL_ADMPCIE7V3** is available in ADMXRC3 API version 1.7.1 and later.

The values **ADMXRC3_MODEL_ADMXRC7Z1** and **ADMXRC3_MODEL_ADMXRC7Z2** are available in ADMXRC3 API version 1.7.2 and later.

The value **ADMXRC3_MODEL_ADMPCIEKU3** is available in ADMXRC3 API version 1.7.3 and later.

The value **ADMXRC3_MODEL_ADMXRCKU1** is available in ADMXRC3 API version 1.8.0 and later.

The value **ADMXRC3_MODEL_ADMPCIE8V3** is available in ADMXRC3 API version 1.8.1 and later.

The value **ADMXRC3_MODEL_ADMPCIE8K5** is available in ADMXRC3 API version 1.8.2 and later.

4.2.18 ADMXRC3_MODULE_INFOA

Declaration

```
typedef struct ... {
    char        Product[64];
    char        PartNumber[64];
```

```
    char        Manufacturer[64];
    char        SerialNumber[64];
    double      IoVoltage;
    uint32_t    ManufactureDate;
    uint32_t    Flags;
    bool_t      Present;
} ADMXRC3_MODULE_INFOA;
```

The members of this structure are as follows:

Product

A NUL-terminated **char** string that identifies the product name of the module, for example "XRM-OPT". See description below about validity.

PartNumber

A NUL-terminated **char** string that identifies the part number of the module, for example "XRM2-OPT/1/LNK-ST11/125". See description below about validity.

Manufacturer

A NUL-terminated **char** string that identifies the manufacturer of the module, for example "Alpha Data". See description below about validity.

SerialNumber

A NUL-terminated **char** string that is the serial number of the module, for example "340". The string is not restricted to numeric digits, and may contain letters and dashes, for example. See description below about validity.

IoVoltage

The nominal I/O voltage required by the module, measured in Volts. See description below about validity.

ManufactureDate

This value is the number of minutes since 00:00 1/1/1996 when the module was manufactured. See description below about validity.

Flags

This value is a bitmask of properties that may apply to the module. The value consists of zero or more of the following flags, ORed together:

- ADMXRC3_MODULE_LEGACYXRM
  Indicates that the module fitted is a legacy XRM, which does not posess a FRU ROM. If this flag is present, then none of the Product, PartNumber, Manufacturer, SerialNumber or ManufactureDate members are valid.
- ADMXRC3_MODULE_FORCE2V5
  Indicates that the module fitted asserts the FORCE2V5 signal, indicating that it always requires a 2.5V I/O voltage supply. This flag typically occurs for legacy XRMs, as they have no other means of indicating the required I/O voltage.
- ADMXRC3_MODULE_ROMERROR
  Indicates that the driver detected an error in the data structure of the FRU ROM for the module. As a result, the data in the first six members of this structure should be considered unreliable, and may be incomplete, undefined or corrupted.
- ADMXRC3_MODULE_CSUMERROR
  Indicates that the driver detected an error in the checksum of at least one area in the data structure of the FRU ROM for the module. As a result, the data in the first six members of this structure should be considered unreliable, and may be incomplete, undefined or corrupted.

See description below about validity.

Present

Indicates whether or not a module is fitted to the specified module site. If FALSE, then none of the other members of this structure are valid.

Description

This structure is returned by ADMXRC3_GetModuleInfoA and contains information about what, if anything, is fitted to an I/O module site.

When inspecting this structure, the following algorithm is suggested:

1. If the Present member is FALSE, then nothing was detected in the site and none of the other members of the structure are valid.
2. Otherwise, the Flags member is inspected. If the ADMXRC3_MODULE_LEGACYXRM flag is present, then none of the Product, PartNumber, Manufacturer, SerialNumber, IoVoltage and ManufactureDate members are valid.
3. Otherwise, if either the ADMXRC3_MODULE_ROMERROR or ADMXRC3_MODULE_CSUMERROR flags are present, then the data in the Product, PartNumber, Manufacturer, SerialNumber, IoVoltage and ManufactureDate members should be considered unreliable.
4. Otherwise, the Product, PartNumber, Manufacturer, SerialNumber, IoVoltage and ManufactureDate members are valid.

Remarks

This datatype is available in ADMXRC3 API version 1.1.0 and later.

This is the ANSI / UTF-8 version of the ADMXRC3_MODULE_INFO structure. ADMXRC3_MODULE_INFO is actually a macro defined to be either ADMXRC3_MODULE_INFOW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_MODULE_INFOA.

4.2.19 ADMXRC3_MODULE_INFOW

Declaration

```
typedef struct ... {
    wchar_t     Product[64];
    wchar_t     PartNumber[64];
    wchar_t     Manufacturer[64];
    wchar_t     SerialNumber[64];
    double      IoVoltage;
    uint32_t    ManufactureDate;
    uint32_t    Flags;
    bool_t      Present;
} ADMXRC3_MODULE_INFOW;
```

The members of this structure are as follows:

Product

A NUL-terminated **wchar_t** string that identifies the product name of the module, for example "XRM-OPT". See description below about validity.

PartNumber

A NUL-terminated **wchar_t** string that identifies the part number of the module, for example "XRM2-OPT/1/LNK-ST11/125". See description below about validity.

Manufacturer

A NUL-terminated **wchar_t** string that identifies the manufacturer of the module, for example "Alpha Data". See description below about validity.

SerialNumber

A NUL-terminated **wchar_t** string that is the serial number of the module, for example "340". The string is

not restricted to numeric digits, and may contain letters and dashes, for example. See description below about validity.

IoVoltage

The nominal I/O voltage required by the module, measured in Volts. See description below about validity.

ManufactureDate

This value is the number of minutes since 00:00 1/1/1996 when the module was manufactured. See description below about validity.

Flags

This value is a bitmask of properties that may apply to the module. The value consists of zero or more of the following flags, ORed together:

- ADMXRC3_MODULE_LEGACYXRM
  Indicates that the module fitted is a legacy XRM, which does not posess a FRU ROM. If this flag is present, then none of the Product, PartNumber, Manufacturer, SerialNumber or ManufactureDate members are valid.
- ADMXRC3_MODULE_FORCE2V5
  Indicates that the module fitted asserts the FORCE2V5 signal, indicating that it always requires a 2.5V I/O voltage supply. This flag typically occurs for legacy XRMs, as they have no other means of indicating the required I/O voltage.
- ADMXRC3_MODULE_ROMERROR
  Indicates that the driver detected an error in the data structure of the FRU ROM for the module. As a result, the data in the first six members of this structure should be considered unreliable, and may be incomplete, undefined or corrupted.
- ADMXRC3_MODULE_CSUMERROR
  Indicates that the driver detected an error in the checksum of at least one area in the data structure of the FRU ROM for the module. As a result, the data in the first six members of this structure should be considered unreliable, and may be incomplete, undefined or corrupted.

See description below about validity.

Present

Indicates whether or not a module is fitted to the specified module site. If FALSE, then none of the other members of this structure are valid.

Description

This structure is returned by ADMXRC3_GetModuleInfoW and contains information about what, if anything, is fitted to an I/O module site.

When inspecting this structure, the following algorithm is suggested:

1. If the Present member is FALSE, then nothing is fitted to the site and none of the other members of the structure are valid.
2. Otherwise, the Flags member is inspected. If the ADMXRC3_MODULE_LEGACYXRM flag is present, then none of the Product, PartNumber, Manufacturer, SerialNumber, IoVoltage and ManufactureDate members are valid.
3. Otherwise, if either the ADMXRC3_MODULE_ROMERROR or ADMXRC3_MODULE_CSUMERROR flags are present, then the data in the Product, PartNumber, Manufacturer, SerialNumber, IoVoltage and ManufactureDate members should be considered unreliable.
4. Otherwise, the Product, PartNumber, Manufacturer, SerialNumber, IoVoltage and ManufactureDate members are valid.

Remarks

This datatype is available in ADMXRC3 API version 1.1.0 and later.

This is the Unicode version of the ADMXRC3_MODULE_INFO structure. ADMXRC3_MODULE_INFO is actually a macro defined to be either ADMXRC3_MODULE_INFOW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_MODULE_INFOA.

4.2.20 ADMXRC3_PACKAGE_TYPE

Declaration

```
typedef ... ADMXRC3_PACKAGE_TYPE;
```

Description

A type used to represent the package code of an FPGA device. It is a 32-bit unsigned integer composed of three bit-fields:

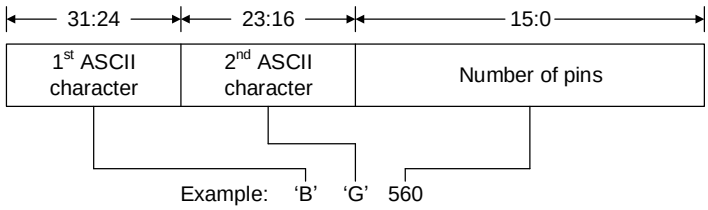


Figure 10 : ADMXRC3_PACKAGE_TYPE bit fields

The following macros are available for manipulating values of type ADMXRC3_PACKAGE_TYPE:

- ADMXRC3_PACKAGE_GETPINS(n) returns the number of pins in the package, as a 16-bit unsigned integer value. For example, ADMXRC3_PACKAGE_GETTYPE1(ADMXRC3_PACKAGE_MAKEFG (456)) returns 456.
- ADMXRC3_PACKAGE_GETTYPE0(n) returns the first package type character. For example, ADMXRC3_PACKAGE_GETTYPE0(ADMXRC3_PACKAGE_MAKEFG(456)) returns the ASCII character 'F'.
- ADMXRC3_PACKAGE_GETTYPE1(n) returns the second package type character. For example, ADMXRC3_PACKAGE_GETTYPE1(ADMXRC3_PACKAGE_MAKEFG(456)) returns the ASCII character 'G'.
- ADMXRC3_PACKAGE_MAKE(t0, t1, n) constructs a package code, given the ASCII type characters *t0*, *t1* and a pin count *n*. For example, ADMXRC3_PACKAGE_MAKE('B', 'G', 560) constructs a value corresponding to the BG560 package.
- ADMXRC3_PACKAGE_MAKEBG(n) constructs a code representing a BG package, where *n* is the number of pins. For example, ADMXRC3_PACKAGE_MAKEBG(560) constructs a value corresponding to the BG560 package.
- ADMXRC3_PACKAGE_MAKEFF(n) constructs a code representing an FF package, where *n* is the number of pins. For example, ADMXRC3_PACKAGE_MAKEFF(1738) constructs a value corresponding to the FF1738 package.
- ADMXRC3_PACKAGE_MAKEFG(n) constructs a code representing an FG package, where *n* is the number of pins. For example, ADMXRC3_PACKAGE_MAKEFG(456) constructs a value corresponding to the FG456 package.

Remarks

This datatype is a 32-bit unsigned integer. In Windows, this type is a typedef of UINT32, while in Linux it is a typedef of uint32_t.

4.2.21 ADMXRC3_SENSOR_INFOA

Declaration

```
typedef struct ... {
```

```
    char                Description[28];
    uint32_t            Capabilities;
    uint32_t            Error;
    ADMXRC3_DATA_TYPE   DataType;
    ADMXRC3_UNIT_TYPE   Unit;
    int                 Exponent;
} ADMXRC3_SENSOR_INFOA;
```

The members of this structure are as follows:

Description

A NUL-terminated **char** string that describes the sensor, for example "1.5V supply rail".

Capabilities

A bitmask of flags that indicate the sensor's capabilities. The following flags are defined:

- ADMXRC3_SENSOR_SWVALUE
  The sensor does not represent a physical sensor, and the values are generated in software. Such a sensor might be used for returning statistics about something.

Error

A value that indicates the accuracy of the sensor, as the maximum positive or negative deviation from the actual value. Error values are scaled in the same way as values read from the sensor using ADMXRC3_ReadSensor.

DataType

A value of type ADMXRC3_DATA_TYPE that indicates the type of the data (boolean, double, integer etc.) returned by the sensor.

Unit

A value of type ADMXRC3_UNIT_TYPE that indicates the unit of measurement (amperes, volts, degrees Celsius etc.) of the data returned by the sensor.

Exponent

A value that indicates the exponent (power of 10) by which the Error member and values read from the sensor are scaled.

Description

This structure is returned by ADMXRC3_GetSensorInfoA and contains information about a sensor.

Example:

Assume that for a given sensor, Error is 0.4, DataType is ADMXRC3_UNIT_DOUBLE, Unit is ADMXRC3_UNIT_V and Exponent is -3. Assume also that ADMXRC3_ReadSensor returns 9.5. The unit is therefore mV, and the absolute value read from the sensor is 9.5 mV. Furthermore, the error is ± 0.4 mV.

Remarks

This datatype is available in ADMXRC3 API version 1.1.0 and later.

For boolean values (where DataType is ADMXRC3_DATA_BOOL), the Error and Exponent members are not valid and should be ignored. The Unit member is always ADMXRCD_UNIT_NONE.

This is the ANSI / UTF-8 version of the ADMXRC3_SENSOR_INFO structure. ADMXRC3_SENSOR_INFO is actually a macro defined to be either ADMXRC3_SENSOR_INFOW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_SENSOR_INFOA.

4.2.22 ADMXRC3_SENSOR_INFOW

Declaration

```
typedef struct ... {
    wchar_t              Description[28];
    uint32_t             Capabilities;
    uint32_t             Error;
    ADMXRC3_DATA_TYPE    DataType;
    ADMXRC3_UNIT_TYPE    Unit;
    int                  Exponent;
} ADMXRC3_SENSOR_INFOW;
```

The members of this structure are as follows:

Description

A NUL-terminated **wchar_t** string that describes the sensor, for example "1.5V supply rail".

Capabilities

A bitmask of flags that indicate the sensor's capabilities. The following flags are defined:

- ADMXRC3_SENSOR_SWVALUE
  The sensor does not represent a physical sensor, and the values are generated in software. Such a sensor might be used for returning statistics about something.

Error

A value that indicates the accuracy of the sensor, as the maximum positive or negative deviation from the actual value. Error values are scaled in the same way as values read from the sensor using ADMXRC3_ReadSensor.

DataType

A value of type ADMXRC3_DATA_TYPE that indicates the type of the data (boolean, double, integer etc.) returned by the sensor.

Unit

A value of type ADMXRC3_UNIT_TYPE that indicates the unit of measurement (amperes, volts, degrees Celsius etc.) of the data returned by the sensor.

Exponent

A value that indicates the exponent (power of 10) by which the Error member and values read from the sensor are scaled.

Description

This structure is returned by ADMXRC3_GetSensorInfoW and contains information about a sensor.

Example:

Assume that for a given sensor, Error is 0.4, DataType is ADMXRC3_UNIT_DOUBLE, Unit is ADMXRC3_UNIT_V and Exponent is -3. Assume also that ADMXRC3_ReadSensor returns 9.5. The unit is therefore mV, and the absolute value read from the sensor is 9.5 mV. Furthermore, the error is ± 0.4 mV.

Remarks

This datatype is available in ADMXRC3 API version 1.1.0 and later.

For boolean values (where DataType is ADMXRC3_DATA_BOOL), the Error and Exponent members are not valid and should be ignored. The Unit member is always ADMXRCD_UNIT_NONE.

This is the Unicode version of the ADMXRC3_SENSOR_INFO structure. ADMXRC3_SENSOR_INFO is actually a macro defined to be either ADMXRC3_SENSOR_INFOW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_SENSOR_INFOA.

4.2.23 ADMXRC3_SENSOR_VALUE

Declaration

```
typedef union ... {
    bool_t      Bool;
    double      Double;
    int32_t     Int32;
    uint32_t    UInt32;
} ADMXRC3_SENSOR_VALUE;
```

The members of this union are as follows:

Bool

The current reading from the sensor.

Double

The current reading from the sensor.

Int32

The current reading from the sensor.

UInt32

The current reading from the sensor.

Description

This union is returned by ADMXRC3_ReadSensor and contains the current readings from the sensor.

As this datatype is a union, interpretation of its value depends on the DataType member of the ADMXRC3_SENSOR_INFO structure for the sensor in question, as returned by ADMXRC3_GetSensorInfo. It is the application's responsibility to correctly interpret the value by using the appropriate member from the union (Bool, Double, etc.).

Remarks

This datatype is available in ADMXRC3 API version 1.1.0 and later.

4.2.24 ADMXRC3_STATUS

Declaration

```
typedef enum ... {
    ADMXRC3_SUCCESS                 = 0,          /* Success */
   ADMXRC3_INTERNAL_ERROR       = 0x100,    /* An error in the API logic occurred */
    ADMXRC3_UNEXPECTED_ERROR        = 0x101,      /* An unexpected error caused
the operation to fail */
  ADMXRC3_BAD_DRIVER        = 0x102,   /* The driver may not be correctly installed */
    ADMXRC2_NO_MEMORY               = 0x103,      /* Couldn't allocate memory r
equired to complete operation */
    ADMXRC3_ACCESS_DENIED           = 0x104,      /* The calling process does n
ot have permission to open the device */
    ADMXRC3_DEVICE_NOT_FOUND        = 0x105,      /* Failed to open the device
with the specified index */
    ADMXRC3_FILE_NOT_FOUND          = 0x106,      /* Failed to open the .BIT fi
le with the specified filename */
  ADMXRC3_HARDWARE_ERROR      = 0x107,    /* An error in the hardware was detected */
    ADMXRC3_FPGA_MISMATCH           = 0x108,      /* The FPGA on the card did n
ot match that of the bitstream file */
  ADMXRC3_INVALID_BITSTREAM    = 0x109,    /* The .BIT file appeared to be corrupt */
    ADMXRC3_INVALID_BUFFER          = 0x10A,      /* The supplied buffer was in
valid and could not be read/written */
   ADMXRC3_INVALID_FLAG        = 0x10B,    /* A flag was invalid or not recognized */
```

```
    ADMXRC3_INVALID_FREQUENCY       = 0x10C,      /* The frequency was not poss
ible for the specified clock generator */
    ADMXRC3_INVALID_HANDLE         = 0x10D,     /* The device handle was invalid */
    ADMXRC3_INVALID_INDEX         = 0x10E,     /* The index parameter was invalid */
    ADMXRC3_INVALID_REGION          = 0x10F,      /* The offset and/or length p
arameters were invalid */
    ADMXRC3_NULL_POINTER            = 0x110,      /* A NULL pointer was passed
where non-NULL was required */
    ADMXRC3_BUSY                    = 0x111,      /* There is already an operat
ion in progress for the specified device */
    ADMXRC3_CANCEL_FAILED           = 0x112,     /* There is nothing to cancel */
    ADMXRC3_CANCELLED              = 0x113,     /* The operation was cancelled */
    ADMXRC3_RESOURCE_LIMIT         = 0x114,     /* A resource limit was reached */
    ADMXRC3_REGION_TOO_LARGE        = 0x115,      /* The specified region was t
oo large for a single operation */
    ADMXRC3_MUST_WAIT               = 0x116,      /* The specified operation co
uldn't be started immediately */
    ADMXRC3_NOT_OWNER               = 0x117,      /* The buffer handle belongs
to a different device handle */
    ADMXRC3_PENDING                 = 0x118,      /* The non-blocking operation
is not yet complete */
    ADMXRC3_NONBLOCK_IDLE           = 0x119,      /* There is no non-blocking o
peration to finish */
   ADMXRC3_INVALID_BUFFER_HANDLE   = 0x11A,     /* The buffer handle was invalid */
    ADMXRC3_INVALID_LOCAL_REGION    = 0x11B,      /* The local address region w
as invalid */
    ADMXRC3_DATATYPE_MISMATCH       = 0x11C,      /* The datatype was invalid f
or the specified sensor */
    ADMXRC3_NOT_SUPPORTED           = 0x11D,      /* The hardware does not supp
ort the requested operation */
  ADMXRC3_STATUS_FORCE32BITS    = 0x7FFFFFFF  /* Force type to be at least 32 bits */
} ADMXRC3_STATUS;
```

Description

An enumerated type used as the return value from most ADMXRC3 API functions. A value of zero corresponds to ADMXRC3_SUCCESS, and indicates success for most functions. However, the **ADMXRC3_Start*Xxx*** functions such as ADMXRC3_StartReadDMA return ADMXRC3_PENDING in order to indicate that a non-blocking operation was successfully started.

4.2.25 ADMXRC3_SUBFAMILY_TYPE

Declaration

```
typedef enum ... {
    ADMXRC3_SUBFAMILY_NONE          = 0,          /* Not in a subfamily */
    ADMXRC3_SUBFAMILY_E             = 16,         /* Virtex-E */
    ADMXRC3_SUBFAMILY_EM            = 17,         /* Virtex-EM */
    ADMXRC3_SUBFAMILY_2PROX         = 48,         /* Virtex-II Pro-X */
    ADMXRC3_SUBFAMILY_4FX           = 64,         /* Virtex-4 FX */
    ADMXRC3_SUBFAMILY_4LX           = 65,         /* Virtex-4 LX */
    ADMXRC3_SUBFAMILY_4SX           = 66,         /* Virtex-4 SX */
    ADMXRC3_SUBFAMILY_5LX           = 80,         /* Virtex-5 LX */
    ADMXRC3_SUBFAMILY_5FXT          = 84,         /* Virtex-5 FXT */
    ADMXRC3_SUBFAMILY_5LXT          = 85,         /* Virtex-5 LXT */
    ADMXRC3_SUBFAMILY_5SXT          = 86,         /* Virtex-5 SXT */
    ADMXRC3_SUBFAMILY_6LX           = 96,         /* Virtex-6 LX */
    ADMXRC3_SUBFAMILY_6HXT          = 97,         /* Virtex-6 HXT */
```

```
    ADMXRC3_SUBFAMILY_6LXT          = 98,         /* Virtex-6 LXT */
    ADMXRC3_SUBFAMILY_6SXT          = 99,         /* Virtex-6 SXT */
    ADMXRC3_SUBFAMILY_7AT           = 112,        /* Artix-7T */
    ADMXRC3_SUBFAMILY_7KT           = 113,        /* Kintex-7T */
    ADMXRC3_SUBFAMILY_7VT           = 114,        /* Virtex-7T */
    ADMXRC3_SUBFAMILY_7VXT          = 115,        /* Virtex-7XT */
    ADMXRC3_SUBFAMILY_7VHT          = 116,        /* Virtex-7HT */
    ADMXRC3_SUBFAMILY_7Z            = 117,        /* Zynq-7000 */
    ADMXRC3_SUBFAMILY_UK            = 128,        /* Kintex Ultrascale */
    ADMXRC3_SUBFAMILY_UV            = 129,        /* Virtex Ultrascale */
    ADMXRC3_SUBFAMILY_UKP           = 130,        /* Kintex Ultrascale+ */
    ADMXRC3_SUBFAMILY_UVP           = 131,        /* Virtex Ultrascale+ */
    ADMXRC3_SUBFAMILY_UZP           = 132,        /* Zynq Ultrascale+ */
  ADMXRC3_SUBFAMILY_FORCE32BITS  = 0x7FFFFFFF /* Force type to be at least 32 bits */
} ADMXRC3_SUBFAMILY_TYPE;
```

Description

An enumerated type used to represent a subfamily within an FPGA device family.

4.2.26 ADMXRC3_TICKET

Declaration

```
typedef ... ADMXRC3_TICKET;
```

Description

A ticket used to keep track of a non-blocking operation. Every non-blocking operation requires a ticket, which must be unique to that operation and valid from when the operation is initiated until it is finished. A ticket may be reused for further non-blocking operations once the current non-blocking operation has finished.

In Windows, this is a structure that should be considered partially opaque by applications, and contains an **OVERLAPPED** structure. It is initialized by first calling ADMXRC3_InitializeTicket and then setting the **Overlapped.hEvent** member to a **HANDLE** value representing a non-auto-reset Win32 event. The event must be used in at most one non-blocking operation at any given moment.

In Linux or VxWorks, this type should be considered fully opaque. It is initialized by ADMXRC3_InitializeTicket.

On all platforms, a ticket must be initialized at least once before its first use, regardless of how many times it is used. Reinitializing a ticket (and/or changing the event in the case of Windows) between non-blocking operations is permitted.

A ticket is required by the following functions:

- ADMXRC3_FinishDMA
- ADMXRC3_FinishNotificationWait
- ADMXRC3_StartNotificationWait
- ADMXRC3_StartReadDMA
- ADMXRC3_StartReadDMALocked
- ADMXRC3_StartWriteDMA
- ADMXRC3_StartWriteDMALocked

4.2.27 ADMXRC3_UNIT_TYPE

Declaration

```
typedef enum ... {
```

```
    ADMXRC3_UNIT_NONE          = 0,          /* Unitless data */
    ADMXRC3_UNIT_A             = 1,          /* Ampere */
    ADMXRC3_UNIT_V             = 2,          /* Volt */
    ADMXRC3_UNIT_C             = 3,          /* Degrees Celsius */
    ADMXRC3_UNIT_HZ            = 4,          /* Frequency, in Hz */
    ADMXRC3_UNIT_RPM           = 5,          /* Revolutions per minute */
    ADMXRC3_UNIT_S             = 6,          /* Seconds */
    ADMXRC3_UNIT_LASTVALUE     = 7,          /* Reserved for adding new units */
   ADMXRC3_UNIT_FORCE32BITS   = 0x7FFFFFFF  /* Force type to be at least 32 bits */
} ADMXRC3_UNIT_TYPE;
```

Description

An enumerated type used to indicate the unit of measurement of the values returned by the ADMXRC3_ReadSensor function. Values beginning with ADMXRC3_UNIT_LASTVALUE are reserved for adding new units in future revisions of the API.

Remarks

This datatype is available in ADMXRC3 API version 1.1.0 and later. The **ADMXRC3_UNIT_S** value is available in ADMXRC3 API version 1.4.0 and later.

4.2.28 ADMXRC3_VERSION_INFO

Declaration

```
typedef struct ... {
    struct ... {
        uint16_t    Major;
        uint16_t    Minor;
        uint16_t    Bugfix;
    } Api;
    struct ... {
        uint16_t    Major;
        uint16_t    Minor;
        uint16_t    Bugfix;
    } Driver;
} ADMXRC3_VERSION_INFO;
```

The members of this structure are as follows:

Api

Major

Major version number of the ADMXRC3 API library.

Minor

Minor version number of the ADMXRC3 API library.

Bugfix

Bugfix version number of the ADMXRC3 API library.

Driver

Major

Major version number of the underlying device driver for the device.

Minor

Minor version number of the underlying device driver for the device.

Bugfix

Bugfix version number of the underlying device driver for the device.

Description

This structure is returned by ADMXRC3_GetVersionInfo and indicates the version numbers of the components of the ADMXRC3 API.

Remarks

The Major and Minor members may be incremented when functionality is added, such as support for new models, or new API functions. The Bugfix member is incremented when defects are corrected, but is also reset to zero when either the Major or Minor members are incremented.

4.2.29 ADMXRC3_WINDOW_INFO

Declaration

```
typedef struct ... {
    uint64_t    BusSize;
    uint64_t    BusBase;
    uint64_t    LocalSize;
    uint64_t    LocalBase;
    uint64_t    VirtualSize;
} ADMXRC3_WINDOW_INFO;
```

The members of this structure are as follows:

BusSize

The size, in bytes, of the window on its device's bus (typically PCI, PCI-X or PCI Express).

BusBase

The base address, in bytes, of the window on its device's bus (typically PCI, PCI-X or PCI Express).

LocalSize

The size, in bytes, of the region of local address space that the window occupies.

If this member is nonzero, it is typically a window onto a target FPGA in the device. A target FPGA decodes the local address in order to determine what function within the target FPGA is being accessed.

If this member is zero, it means that the window terminates within the bus interface of the device and does not occupy any local address space. Such windows provide access to device registers in the bus interface of a reconfigurable computing card.

LocalBase

The absolute base address, in bytes, of the region of local address space that the window occupies. This member is not valid if the LocalSize member is zero.

VirtualSize

The size, in bytes, of the window if it were mapped in its entirety into a process' virtual address space. This is the same as the BusSize member on most CPU architectures, but may be larger for a CPU architecture that uses *sparse I/O addressing*. The topic of *sparse I/O addressing* is outside the scope of this document, but in such architectures, attributes of load and store operations (alignment, byte mask, etc.) are encoded in the addresses used to perform such loads and stores.

Description

This structure is returned by ADMXRC3_GetWindowInfo and indicates the size and base addresses, in various address spaces, of a window provided by a device. An application may use this information to avoid errors when calling ADMXRC3_MapWindow.

Remarks

In general, each model in Alpha Data's reconfigurable computing range provides a different set of windows, and the purpose of each window can vary between models. In order to know which windows to map, an application requires knowledge of the model in use. The model that is in use can be obtained by calling ADMXRC3_GetCardInfoEx and inspecting the Model member of the returned ADMXRC3_CARD_INFOEX structure.

4.3 ADMXRC3 API functions

This section describes the functions available in the ADMXRC3 API. The semantics of "direction" for a function parameter are as follows

- (in) means that the parameter is used to pass data to a function, either by value or by reference to a data structure that is read by the function.
- (out) means that the parameter is used to return data from a function, by reference to a data structure that is filled in by the function.
- (inout) means that the parameter is used to both pass data to and return data from a function, by reference to a data structure that is read and potentially modified by the function.

4.3.1 ADMXRC3_Cancel

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Cancel(
    __in  ADMXRC3_HANDLE  hDevice);
```

The parameter(s) of this function are as follows:

hDevice (in)

The device handle for which any ongoing API calls should cancelled.

Description

The ADMXRC3_Cancel function forces any ongoing ADMXRC3 API calls for a given device handle to return as soon as possible. Function calls that return early due to a call to ADMXRC3_Cancel return a status value of ADMXRC3_CANCELLED. However, a function call that happens to be completed, whether successfully or unsuccessfully, at the same time as ADMXRC3_Cancel is called may return a status value other than ADMXRC3_CANCELLED. The purpose of ADMXRC3_Cancel is to force any threads that are blocked inside ADMXRC3 API functions to return early, so that error recovery mechanisms such as timeouts and watchdogs can be implemented by an application.

ADMXRC3_Cancel itself does not wait for ongoing ADMXRC3 API calls to return; it returns as soon as action has been taken to ensure that all other ongoing ADMXRC3 API calls for the specified handle will return.

ADMXRC3_Cancel affects *only* function calls made using the specified device handle. Function calls made using other device handles are entirely unaffected.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_CANCEL_FAILED	There was nothing to cancel.

Remarks

ADMXRC3_Cancel does not affect ADMXRC3 API functions that do not block the calling thread for a significant length of time. An example of such a function is ADMXRC3_GetCardInfoEx. In general, if an API function is guaranteed not to block the caller for a user-perceptible length of time, ADMXRC3_Cancel has no effect on it.

ADMXRC3_Cancel can cancel only those ADMXRC3 API function calls that occur before ADMXRC3_Cancel is called. For example, if thread A calls ADMXRC3_Cancel, and thread B calls ADMXRC3_ReadDMAEx, a degree of uncertainty exists about whether ADMXRC3_Cancel occurs before or after ADMXRC3_ReadDMAEx, because either thread may be preemptively descheduled in most operating systems. Applications should therefore not rely on cancelled ADMXRC3 API calls always returning ADMXRC3_CANCELLED, because this represents a race condition.

4.3.2 ADMXRC3_ClearDeviceErrors

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ClearDeviceErrors(
    __in  ADMXRC3_HANDLE  hDevice);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device of interest.

Description

This function attempts to clear any non-transient error states (i.e. errors that do not self-clear) that might be present for a particular device. This function is generally called as a result of determining that error states are present by a call to ADMXRC3_GetDeviceStatus.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.

Remarks

This function is available in ADMXRC3 API version 1.6.0 and later.

4.3.3 ADMXRC3_Close

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Close(
    __in  ADMXRC3_HANDLE  hDevice);
```

The parameter(s) of this function are as follows:

hDevice (in)

The device handle that is to be closed.

Description

This function closes a device handle, invalidating it. It should be called when an application has finished with a device handle.

If a thread closes a device handle using ADMXRC3_Close when other threads are executing inside ADMXRC3 API functions using the same device handle or there is an ongoing non-blocking operation using the same device handle, then the thread calling ADMXRC3_Close may either:

(a) Block until all of the ADMXRC3 API calls of other threads have returned and any non-blocking operations are finished, OR

(b) Return without blocking, but cause the ADMXRC3 API calls of other threads to return early with an error status value such ADMXRC3_CANCELLED, as if ADMXRC3_Cancel were called.

Which of these two behaviors occurs is operating-system dependent.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.

Remarks

Well-behaved applications should clean up by calling ADMXRC3_Close when finished with a device handle. Failure to do so may lead to resource leaks, although leaked resources are reclaimed by the driver / operating system when an application terminates (with the exception of VxWorks, which performs very little automatic cleanup). The following cleanup is performed by ADMXRC3_Close:

- Mapped memory windows (that were mapped using the same device handle) are unmapped, as if ADMXRC3_UnmapWindow were called.
- Locked user-space buffers (that were locked using the same the device handle) are unlocked, as if ADMXRC3_Unlock were called.
- If the device handle owns any target FPGAs, those target FPGAs become free again.
- For each Flash memory bank in the device, if there exists a dirty block as a result of the device handle being used to call ADMXRC3_EraseFlash or ADMXRC3_WriteFlash, the Flash memory bank is synchronized as if ADMXRC3_SyncFlash were called. In other words, explicitly calling ADMXRC3_SyncFlash before ADMXRC3_Close avoids a noticeable delay in the execution of ADMXRC3_Close.
- In Windows, any Win32 Events registered with the device handle are unregistered, as if ADMXRC3_UnregisterWin32Event were called.
- In VxWorks, any semaphores registered with the device handle are unregistered, as if ADMXRC3_UnregisterVxwSem were called.

It is good practice to set variables of type ADMXRC3_HANDLE to ADMXRC3_HANDLE_INVALID_VALUE after closing them with ADMXRC3_Close.

4.3.4 ADMXRC3_ConfigureFromBuffer

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ConfigureFromBuffer(
```

```
__in  ADMXRC3_HANDLE  hDevice,
__in  unsigned int    targetIndex,
__in  uint32_t        flags,
__in  const void*     pBuffer,
__in  size_t          length);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the target FPGA to be configured.

targetIndex (in)

The zero-based index of the target FPGA to be configured. This value must be less than the **NumTargetFpga** member of ADMXRC3_CARD_INFOEX.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_CONFIGURE_NOCHECK
  Causes the post-configuration check, that OCP communications with the target FPGA have been established, to be skipped. This flag is appropriate when a target FPGA design does not have a host interface at all. Ignored if ADMXRC3_CONFIGURE_PARTIAL is also passed.
- ADMXRC3_CONFIGURE_PARTIAL
  Causes a partial reconfiguration to be performed. A partial reconfiguration does not clear the target FPGA's existing configuration before downloading the bitstream. If this flag is omitted, a full reconfiguration is performed.
- ADMXRC3_CONFIGURE_SHARE
  Causes the device handle passed in hDevice **not** to take ownership of the target FPGA if configuration is successful.

pBuffer (in)

Points to the bitstream data to be downloaded to the target FPGA.

length (in)

The number of bytes of bitstream data to be downloaded to the target FPGA.

Description

This function downloads the bitstream data in a buffer to the specified device. The bitstream data is typically obtained from the Data member of an ADMXRC3_BITSTREAM object, returned by ADMXRC3_LoadBitstream, but an application may elect to use its own .BIT file loader to obtain the data. See the remarks below about the format of the data.

The default behavior is that full reconfiguration of the target FPGA is performed, where the bitstream download is preceded by unconfiguring the target FPGA. Full reconfiguration is described in Section 3.8.1.1, "Full reconfiguration".

Partial reconfiguration of the target FPGA, as described in Section 3.8.1.2, "Partial reconfiguration", is performed if the flags parameter contains ADMXRC3_CONFIGURE_PARTIAL.

In order to avoid the possibility that a target FPGA is inadvertently reconfigured (perhaps by another process) whilst it is in use, an ownership mechanism exists for target FPGAs. This ownership mechanism is described in Section 3.8.1.4, "Target FPGA ownership".

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	Configuring the target FPGA with the bitstream was not successful due to a hardware error.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The targetIndex parameter is out of range.
ADMXRC3_NOT_OWNER	The specified device handle is not the owner of the target FPGA.
ADMXRC3_NULL_POINTER	The pBuffer parameter was a NULL pointer.

Remarks

As already mentioned, an application can load a .BIT file into memory using ADMXRC3_LoadBitstream, or by using its own loader functions. If using its own loader, an application should parse the .BIT file, keeping only the configuration frame data from the .BIT file. The record structure in the header of the .BIT file should not be in the data passed to ADMXRC3_ConfigureFromBuffer. The configuration frame data inside a .BIT file is a stream of bytes, where each byte maps to the SelectMap data bits D0..D7 as follows:

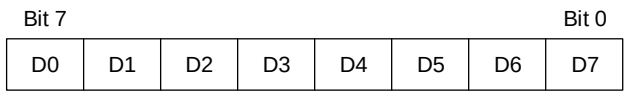


Figure 11 : SelectMap D0..D7 byte mapping

As can be seen from the above figure, SelectMap D0 is found in bit 7 of each byte and SelectMap D7 in bit 0. Despite this, it is **not** necessary for an application to reorder the bits within each byte of the configuration frame data, because ADMXRC3_ConfigureFromBuffer expects bitstream data in the above format, exactly as read from a .BIT file. If necessary for the model in use, the data is automatically re-ordered by ADMXRC3_ConfigureFromBuffer.

Passing the ADMXRC3_CONFIGURE_PARTIAL flag with a length of zero can be used to take ownership of a take FPGA without actually changing its configuration state.

An important difference between ADMXRC3_LoadBitstream and the legacy **ADMXRC2_LoadBitstream** function is that **ADMXRC2_LoadBitstream may** reorder bits within each byte of the configuration frame data for some models, and not for others. By contrast, ADMXRC3_LoadBitstream **never** reorders; it returns the configuration frame data as it is found in the .BIT file.

4.3.5 ADMXRC3_ConfigureFromFileA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ConfigureFromFileA(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    targetIndex,
    __in  uint32_t        flags,
    __in  const char*     pFilename);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the target FPGA to be configured.

targetIndex (in)

The zero-based index of the target FPGA to be configured. This value must be less than the **NumTargetFpga** member of ADMXRC3_CARD_INFOEX.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_CONFIGURE_NOCHECK
  Causes the post-configuration check, that OCP communications with the target FPGA have been established, to be skipped. This flag is appropriate when a target FPGA design does not have a host interface at all. Ignored if ADMXRC3_CONFIGURE_PARTIAL is also passed.
- ADMXRC3_CONFIGURE_PARTIAL
  Causes a partial reconfiguration to be performed. A partial reconfiguration does not clear the target FPGA's existing configuration before downloading the bitstream. If this flag is omitted, a full reconfiguration is performed.
- ADMXRC3_CONFIGURE_SHARE
  Causes the device handle passed in hDevice **not** to take ownership of the target FPGA if configuration is successful.
- ADMXRC3_CONFIGURE_IGNOREMISMATCH
  Causes any mismatch between the device identifier string that is embedded in the .bit file and the target FPGA's to be ignored, preventing the function from failing with the return value **ADMXRC3_FPGA_MISMATCH**.

pFilename (in)

Specifies the name of a .BIT file to be downloaded to the target FPGA.

Description

This function downloads the data in a bitstream (.BIT) file to the specified device. As a .BIT file contains a record that specifies for what kind of FPGA it was generated, this function verifies that the .BIT file FPGA type matches that of the device, and returns an error if there is a mismatch.

The default behavior is that full reconfiguration of the target FPGA is performed, where the bitstream download is preceded by unconfiguring the target FPGA. Full reconfiguration is described in Section 3.8.1.1, "Full reconfiguration".

Partial reconfiguration of the target FPGA, as described in Section 3.8.1.2, "Partial reconfiguration", is performed if the flags parameter contains ADMXRC3_CONFIGURE_PARTIAL.

In order to avoid the possibility that a target FPGA is inadvertently reconfigured (perhaps by another process) whilst it is in use, an ownership mechanism exists for target FPGAs. This ownership mechanism is described in Section 3.8.1.4, "Target FPGA ownership".

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_FILE_NOT_FOUND	The file identified by the pFilename parameter could not be opened.
ADMXRC3_FPGA_MISMATCH	The FPGA in the .BIT file header does not match the target FPGA in the specified device.
ADMXRC3_HARDWARE_ERROR	Configuring the target FPGA with the bitstream was not successful due to a hardware error.
ADMXRC3_INVALID_BITSTREAM	The file identified by the pFilename parameter is not a valid .BIT file.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The targetIndex parameter is out of range.
ADMXRC3_NOT_OWNER	The specified device handle is not the owner of the target FPGA.
ADMXRC3_NO_MEMORY	Memory to hold the bitstream data could not be allocated.
ADMXRC3_NULL_POINTER	The pFilename parameter was a NULL pointer.

Remarks

This is the ANSI / UTF-8 version of ADMXRC3_ConfigureFromFile. ADMXRC3_ConfigureFromFile is actually a macro defined to be either ADMXRC3_ConfigureFromFileW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_ConfigureFromFileA.

The flag **ADMXRC3_CONFIGURE_IGNOREMISMATCH** is available in ADMXRC3 API version 1.8.0 and later.

4.3.6 ADMXRC3_ConfigureFromFileW

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ConfigureFromFileW(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    targetIndex,
    __in  uint32_t        flags,
    __in  const wchar_t*  pFilename);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the target FPGA to be configured.

targetIndex (in)

The zero-based index of the target FPGA to be configured. This value must be less than the **NumTargetFpga** member of ADMXRC3_CARD_INFOEX.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_CONFIGURE_NOCHECK
  Causes the post-configuration check, that OCP communications with the target FPGA have been established, to be skipped. This flag is appropriate when a target FPGA design does not have a host interface at all. Ignored if ADMXRC3_CONFIGURE_PARTIAL is also passed.
- ADMXRC3_CONFIGURE_PARTIAL
  Causes a partial reconfiguration to be performed. A partial reconfiguration does not clear the target FPGA's existing configuration before downloading the bitstream. If this flag is omitted, a full reconfiguration is performed.
- ADMXRC3_CONFIGURE_SHARE
  Causes the device handle passed in hDevice **not** to take ownership of the target FPGA if configuration is successful.
- ADMXRC3_CONFIGURE_IGNOREMISMATCH
  Causes any mismatch between the device identifier string that is embedded in the .bit file and the target FPGA's to be ignored, preventing the function from failing with the return value **ADMXRC3_FPGA_MISMATCH**.

pFilename (in)

Specifies the name of a .BIT file to be downloaded to the target FPGA.

Description

This function downloads the data in a bitstream (.BIT) file to the specified device. As a .BIT file contains a record that specifies for what kind of FPGA it was generated, this function verifies that the .BIT file FPGA type matches that of the device, and returns an error if there is a mismatch.

The default behavior is that full reconfiguration of the target FPGA is performed, where the bitstream download is preceded by unconfiguring the target FPGA. Full reconfiguration is described in Section 3.8.1.1, "Full reconfiguration".

Partial reconfiguration of the target FPGA, as described in Section 3.8.1.2, "Partial reconfiguration", is performed if the flags parameter contains ADMXRC3_CONFIGURE_PARTIAL.

In order to avoid the possibility that a target FPGA is inadvertently reconfigured (perhaps by another process) whilst it is in use, an ownership mechanism exists for target FPGAs. This ownership mechanism is described in Section 3.8.1.4, "Target FPGA ownership".

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_FILE_NOT_FOUND	The file identified by the pFilename parameter could not be opened.

Return Value	Description
ADMXRC3_FPGA_MISMATCH	The FPGA in the .BIT file header does not match the target FPGA in the specified device.
ADMXRC3_HARDWARE_ERROR	Configuring the target FPGA with the bitstream was not successful due to a hardware error.
ADMXRC3_INVALID_BITSTREAM	The file identified by the pFilename parameter is not a valid .BIT file.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The targetIndex parameter is out of range.
ADMXRC3_NOT_OWNER	The specified device handle is not the owner of the target FPGA.
ADMXRC3_NO_MEMORY	Memory to hold the bitstream data could not be allocated.
ADMXRC3_NULL_POINTER	The pFilename parameter was a NULL pointer.

Remarks

This is the Unicode version of ADMXRC3_ConfigureFromFile. ADMXRC3_ConfigureFromFile is actually a macro defined to be either ADMXRC3_ConfigureFromFileW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_ConfigureFromFileA.

The flag **ADMXRC3_CONFIGURE_IGNOREMISMATCH** is available in ADMXRC3 API version 1.8.0 and later.

4.3.7 ADMXRC3_EraseFlash

Declaration

```
ADMXRC3_STATUS
ADMXRC3_EraseFlash(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    flashIndex,
    __in  uint32_t        flags,
    __in  uint64_t        offset,
    __in  uint64_t        length);
```

The parameter(s) of this function are as follows:

hDevice (in)

The device handle of the device that contains the Flash memory bank of interest.

flashIndex (in)

Identifies the Flash memory bank of interest, within the specified device.

flags (in)

Flags that modify how the operation is performed. Currently, the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_FLASH_SYNC
  Causes this erase operation, as well as any previously pending erase and write operations, to be committed to hardware before the function returns.

offset (in)

Offset, in bytes, into the Flash memory bank of the beginning of region that is to be erased.

length (in)

Length, in bytes, of the region that is to be erased.

Description

The ADMXRC3_EraseFlash function erases a region of a Flash memory bank.

Depending on the model, not all locations in a Flash memory bank may be erased using this function. This is required in order to prevent inadvertent corruption of VPD, firmware etc. on some models. To determine the region of a Flash memory bank that is modifiable, an application calls ADMXRC3_GetFlashInfo. Attempts to modify any location outside of the modifiable region will fail.

ADMXRC3_EraseFlash performs a read-erase-write cycle when the region specified by offset and length does not start and end exactly on block boundaries. Applications are therefore not required to have knowledge of the block-architecture of Flash chips, and can treat a Flash memory bank as an array of bytes.

The ADMXRC3 API implements a caching mechanism for each Flash memory bank. Passing the ADMXRC3_FLASH_SYNC flag ensures that the cache for the specified Flash memory bank is synchronized with the hardware before the function returns. An alternative way of ensuring synchronization is to call ADMXRC3_SyncFlash.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	Erasing the Flash region was not successful due to a hardware error.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The flashIndex parameter is out of range.
ADMXRC3_INVALID_REGION	The offset and length parameters specify a region that is partially or wholly out of bounds of the specified Flash bank.

Remarks

Erasing a region of Flash memory results in all bytes in the erased region returning 0xFF when subsequently read.

Depending on the model, a Flash memory bank may hold several kinds of data:

1. Vital Product Data (VPD)
2. "Soft jumper" information, such as default clock frequencies
3. Non-target FPGA firmware images
4. Target FPGA bitstreams

Of these types of data, the first three cannot be erased using ADMXRC3_EraseFlash, because they always

lie outside of the user-programmable region in the Flash memory bank. Attempting to do so results in a error being returned. This is done as a safety measure to protect against inadvertent corruption of data that is vital to the operation of the hardware. The fourth type of data, target FPGA bitstreams, is usually stored in the user-programmable area and can therefore be erased or written.

For information about a particular model's Flash address map, refer to the User Guide for that model.

4.3.8 ADMXRC3_FinishDMA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_FinishDMA(
    __in  ADMXRC3_HANDLE   hDevice,
    __in  ADMXRC3_TICKET*  pTicket,
    __in  bool_t           bWait);
```

The parameter(s) of this function are as follows:

hDevice (in)

The device handle for which a non-blocking DMA operation is pending.

pTicket (in)

Points to the ticket used for the non-blocking operation. Ticket structures must be initialized according to Section 3.3.2, "Tickets".

bWait (in)

Specifies whether to wait for completion, or poll for completion.

Description

This function can be used to finish a previously-started non-blocking DMA operation, optionally waiting for it to be completed in the hardware (if bWait is TRUE). If bWait is FALSE, the function polls for whether or not the operation has been completed, returning ADMXRC3_PENDING if not.

This function is suitable for use with the following functions that initiate non-blocking operations:

- ADMXRC3_StartReadDMA
- ADMXRC3_StartReadDMABus
- ADMXRC3_StartReadDMAEx
- ADMXRC3_StartReadDMALocked
- ADMXRC3_StartReadDMALockedEx
- ADMXRC3_StartWriteDMA
- ADMXRC3_StartWriteDMABus
- ADMXRC3_StartWriteDMAEx
- ADMXRC3_StartWriteDMALocked
- ADMXRC3_StartWriteDMALockedEx

Every non-blocking operation that is successfully initiated on a given device handle using one of the above set of functions (that is, ADMXRC3_PENDING was returned from one of the above set of functions) must eventually be finished by a call to ADMXRC3_FinishDMA. Failure to obey this rule means that attempting to start further non-blocking operations using the same device handle results in undefined behavior.

Return Value

A value of ADMXRC3_SUCCESS indicates that the non-blocking operation was completed successfully. Otherwise, the following values may be returned:

Return Value	Description
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	The DMA transfer failed due to a hardware error.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NONBLOCK_IDLE	There was no non-blocking operation to finish.
ADMXRC3_PENDING	The non-blocking operation is ongoing. This value should never be returned if bWait is TRUE.

Remarks

Refer to Section 3.3, "Non-blocking operations" for an overview of the non-blocking functions in the ADMXRC3 API.

4.3.9 ADMXRC3_FinishNotificationWait

Declaration

```
ADMXRC3_STATUS
ADMXRC3_FinishNotificationWait(
    __in  ADMXRC3_HANDLE   hDevice,
    __in  ADMXRC3_TICKET*  pTicket,
    __in  bool_t           bWait);
```

The parameter(s) of this function are as follows:

hDevice (in)

The device handle for which a non-blocking operation is pending.

pTicket (in)

Points to the ticket used for the non-blocking operation. Ticket structures must be initialized according to Section 3.3.2, "Tickets".

bWait (in)

Specifies whether to wait for completion, or poll for completion.

Description

This function is the counterpart of ADMXRC3_StartNotificationWait. It must be called to finish a non-blocking operation that was successfully started with ADMXRC3_StartNotificationWait (i.e. ADMXRC3_PENDING was returned).

Return Value

A value of ADMXRC3_SUCCESS indicates that the non-blocking operation was completed successfully. Otherwise, the following values may be returned:

Return Value	Description
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NONBLOCK_IDLE	There was no non-blocking operation to finish.
ADMXRC3_PENDING	The non-blocking operation is ongoing. This value should never be returned if bWait is TRUE.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

Refer to Section 3.3, "Non-blocking operations" for an overview of the non-blocking functions in the ADMXRC3 API.

4.3.10 ADMXRC3_GetBankInfo

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetBankInfo(
    __in  ADMXRC3_HANDLE      hDevice,
    __in  unsigned int        bankIndex,
    __out ADMXRC3_BANK_INFO*  pBankInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the memory bank for which information is required.

bankIndex (in)

Specifies the memory bank within the device for which information is required. This is a zero-based index that must be less than the **NumMemoryBank** member of ADMXRC3_CARD_INFOEX.

pBankInfo (out)

Points to an object of type ADMXRC3_BANK_INFO in which to return information about the memory bank.

Description

This function returns information about a memory bank in a device.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved due to an error reading the device's Vital Product Data.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The bankIndex parameter is out of range.

Return Value	Description
ADMXRC3_NULL_POINTER	The pBankInfo parameter is NULL.

4.3.11 ADMXRC3_GetCardInfo

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetCardInfo(
    __in  ADMXRC3_HANDLE      hDevice,
    __out ADMXRC3_CARD_INFO*  pCardInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device for which information is required.

pCardInfo (out)

Points to an object of type ADMXRC3_CARD_INFO in which to return information about the device.

Description

This function returns information about a device.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved due to an error reading the device's Vital Product Data.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NULL_POINTER	The pCardInfo parameter is NULL.

Remarks

ADMXRC3_GetCardInfo has been superseded by ADMXRC3_GetCardInfoEx, as the latter returns a superset of the information returned by ADMXRC3_GetCardInfo.

4.3.12 ADMXRC3_GetCardInfoEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetCardInfoEx(
    __in  ADMXRC3_HANDLE        hDevice,
    __out ADMXRC3_CARD_INFOEX*  pCardInfoEx);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device for which information is required.

pCardInfoEx (out)

Points to an object of type ADMXRC3_CARD_INFOEX in which to return information about the device.

Description

This function returns information about a device.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved due to an error reading the device's Vital Product Data.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NULL_POINTER	The pCardInfoEx parameter is NULL.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later, and supersedes ADMXRC3_GetCardInfo. The information returned by ADMXRC3_GetCardInfoEx is a superset of that of ADMXRC3_GetCardInfo.

4.3.13 ADMXRC3_GetClockFrequency

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetClockFrequency(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    clockIndex,
    __out double*         pActualFrequency);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the clock generator whose current frequency is to be returned.

clockIndex (in)

Specifies the clock generator within the device whose current frequency is to be returned. This is a zero-based index that must be less than the **NumClockGen** member of ADMXRC3_CARD_INFOEX.

pActualFrequency (out)

Points to an object of type **double** in which to return the current frequency of the clock generator.

Description

This function returns the current frequency being output by a clock generator in a device.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The clockIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pActualFrequency parameter is NULL.

4.3.14 ADMXRC3_GetCommonBuffer

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetCommonBuffer(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    bufferIndex,
    __out size_t*         pSizeInBytes,
    __out uint64_t*       pBusAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the common buffer of interest.

bufferIndex (in)

Specifies the zero-based index of the common buffer of interest; must be less than the common buffer count returned by ADMXRC3_GetCommonBufferCount.

pSizeInBytes (out)

Points to a variable of type **size_t** in which to return the size of the common buffer (in bytes); may be NULL if the size is not of interest.

pBusAddress (out)

Points to a 64-bit unsigned integer variable in which to return the base bus address of the common buffer; may be NULL if the base bus address is not of interest.

Description

This function returns the size and/or base bus address of a common buffer that belongs to a reconfigurable computing device.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The bufferIndex parameter is out of range.

Remarks

This function is available in ADMXRC3 API version 1.5.0 and later via the header file <admxrc3/combuf.h>.

4.3.15 ADMXRC3_GetCommonBufferCount

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetCommonBufferCount(
    __in  ADMXRC3_HANDLE  hDevice,
    __out unsigned int*   pCount);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the common buffer of interest.

pCount (out)

Points to a variable of type **unsigned int** in which to return the number of available common buffers.

Description

This function returns the number of common buffers available for the specified reconfigurable computing device.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NULL_POINTER	The pCount parameter is out of range.

Remarks

This function is available in ADMXRC3 API version 1.5.0 and later via the header file <admxrc3/combuf.h>.

4.3.16 ADMXRC3_GetDeviceStatus

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetDeviceStatus(
    __in  ADMXRC3_HANDLE          hDevice,
    __in  uint32_t                flags,
    __out ADMXRC3_DEVICE_STATUS*  pDeviceStatus,
    __out bool_t*                 pbAnyError);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device of interest.

flags (in)

Flags that modify how the operation is performed. Currently, there are no flags defined and this parameter must be zero.

pDeviceStatus (out)

Points to an object of type ADMXRC3_DEVICE_STATUS in which to return the device's status.

pbAnyError (out)

Indicates whether or not the specified device has at least one error condition.

Description

This function returns high-level information about any error conditions that might be present for a particular device. The **pDeviceStatus** parameter must point to a variable of type ADMXRC3_DEVICE_STATUS that is to be filled in with device status information. The **pbAnyError** parameter may point to a variable that is set to TRUE if at least one error condition exists in the specified device, or FALSE otherwise. Alternatively, **pbAnyError** may be NULL if not of interest.

The structure ADMXRC3_DEVICE_STATUS consists of members whose values can be the bitwise OR of various flags. Please refer to ADMXRC3_DEVICE_STATUS for details of those flags.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NULL_POINTER	The pModuleInfo parameter is NULL.

Remarks

This function is available in ADMXRC3 API version 1.6.0 and later.

4.3.17 ADMXRC3_GetFlashBlockInfo

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetFlashBlockInfo(
    __in  ADMXRC3_HANDLE            hDevice,
    __in  unsigned int              flashIndex,
    __in  uint64_t                  address,
    __out ADMXRC3_FLASHBLOCK_INFO*  pFlashBlockInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the Flash memory bank of interest.

flashIndex (in)

Identifies the Flash memory bank of interest, within the device. This is a zero-based index that must be less than the **NumFlashBank** member of ADMXRC3_CARD_INFOEX.

address (in)

An address, in bytes, within the Flash memory bank of interest.

pFlashBlockInfo (out)

Points to an object of type ADMXRC3_FLASHBLOCK_INFO in which to return information about the Flash block.

Description

This function returns information about a Flash block, given any address within that block.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved because the Flash device is malfunctioning or not present.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The flashIndex parameter is out of range.
ADMXRC3_INVALID_REGION	The address parameter is out of range.
ADMXRC3_NULL_POINTER	The pFlashBlockInfo parameter is NULL.

4.3.18 ADMXRC3_GetFlashInfoA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetFlashInfoA(
    __in  ADMXRC3_HANDLE        hDevice,
    __in  unsigned int          flashIndex,
    __out ADMXRC3_FLASH_INFOA*  pFlashInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the Flash memory bank of interest.

flashIndex (in)

Identifies the Flash memory bank of interest, within the device. This is a zero-based index that must be less than the **NumFlashBank** member of ADMXRC3_CARD_INFOEX.

pFlashInfo (out)

Points to an object of type ADMXRC3_FLASH_INFOA in which to return information about the Flash memory bank.

Description

This function returns information about a Flash memory bank.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved because the Flash device is malfunctioning or not present.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The flashIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pFlashInfo parameter is NULL.

Remarks

This is the ANSI / UTF-8 version of ADMXRC3_GetFlashInfo. ADMXRC3_GetFlashInfo is actually a macro defined to be either ADMXRC3_GetFlashInfoW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetFlashInfoA.

4.3.19 ADMXRC3_GetFlashInfoW

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetFlashInfoW(
    __in  ADMXRC3_HANDLE        hDevice,
    __in  unsigned int          flashIndex,
    __out ADMXRC3_FLASH_INFOW*  pFlashInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the Flash memory bank of interest.

flashIndex (in)

Identifies the Flash memory bank of interest, within the device. This is a zero-based index that must be less than the **NumFlashBank** member of ADMXRC3_CARD_INFOEX.

pFlashInfo (out)

Points to an object of type ADMXRC3_FLASH_INFOW in which to return information about the Flash memory bank.

Description

This function returns information about a Flash memory bank.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved because the Flash device is malfunctioning or not present.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The flashIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pFlashInfo parameter is NULL.

Remarks

This is the Unicode version of ADMXRC3_GetFlashInfo. ADMXRC3_GetFlashInfo is actually a macro defined to be either ADMXRC3_GetFlashInfoW if the _**UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetFlashInfoA.

4.3.20 ADMXRC3_GetFpgaInfoA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetFpgaInfoA(
    __in  ADMXRC3_HANDLE       hDevice,
    __in  unsigned int         targetIndex,
    __out ADMXRC3_FPGA_INFOA*  pFpgaInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the target FPGA of interest.

targetIndex (in)

Identifies the target FPGA of interest, within the device. This is a zero-based index that must be less than the **NumTargetFpga** member of ADMXRC3_CARD_INFOEX.

pFpgaInfo (out)

Points to an object of type ADMXRC3_FPGA_INFOA in which to return information about the target FPGA.

Description

This function returns information about a target FPGA.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved due to an error reading the device's Vital Product Data.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The targetIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pFpgaInfo parameter is NULL.

Remarks

This is the ANSI / UTF-8 version of ADMXRC3_GetFpgaInfo. ADMXRC3_GetFpgaInfo is actually a macro defined to be either ADMXRC3_GetFpgaInfoW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetFpgaInfoA.

4.3.21 ADMXRC3_GetFpgaInfoW

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetFpgaInfoW(
    __in  ADMXRC3_HANDLE       hDevice,
    __in  unsigned int         targetIndex,
    __out ADMXRC3_FPGA_INFOW*  pFpgaInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the target FPGA of interest.

targetIndex (in)

Identifies the target FPGA of interest, within the device. This is a zero-based index that must be less than the **NumTargetFpga** member of ADMXRC3_CARD_INFOEX.

pFpgaInfo (out)

Points to an object of type ADMXRC3_FPGA_INFOW in which to return information about the target FPGA.

Description

This function returns information about a target FPGA.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	The information could not be retrieved due to an error reading the device's Vital Product Data.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The targetIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pFpgaInfo parameter is NULL.

Remarks

This is the Unicode version of ADMXRC3_GetFpgaInfo. ADMXRC3_GetFpgaInfo is actually a macro defined to be either ADMXRC3_GetFpgaInfoW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetFpgaInfoA.

4.3.22 ADMXRC3_GetModuleInfoA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetModuleInfoA(
    __in  ADMXRC3_HANDLE          hDevice,
    __in  unsigned int            moduleIndex,
    __out ADMXRC3_MODULE_INFOA*   pModuleInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the module site of interest.

moduleIndex (in)

Identifies the module site of interest, within the device. This is a zero-based index that must be less than the **NumModuleSite** member of ADMXRC3_CARD_INFOEX.

pModuleInfo (out)

Points to an object of type ADMXRC3_MODULE_INFOA in which to return information about the module.

Description

This function returns information about an I/O personality module site.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The moduleIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pModuleInfo parameter is NULL.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

This is the ANSI / UTF-8 version of ADMXRC3_GetModuleInfo. ADMXRC3_GetModuleInfo is actually a

macro defined to be either ADMXRC3_GetModuleInfoW if the _**UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetModuleInfoA.

4.3.23 ADMXRC3_GetModuleInfoW

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetModuleInfoW(
    __in  ADMXRC3_HANDLE          hDevice,
    __in  unsigned int            moduleIndex,
    __out ADMXRC3_MODULE_INFOW*   pModuleInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the module site of interest.

moduleIndex (in)

Identifies the module site of interest, within the device. This is a zero-based index that must be less than the **NumModuleSite** member of ADMXRC3_CARD_INFOEX.

pModuleInfo (out)

Points to an object of type ADMXRC3_MODULE_INFOW in which to return information about the module.

Description

This function returns information about an I/O personality module site.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The moduleIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pModuleInfo parameter is NULL.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

This is the Unicode version of ADMXRC3_GetModuleInfo. ADMXRC3_GetModuleInfo is actually a macro defined to be either ADMXRC3_GetModuleInfoW if the _**UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetModuleInfoA.

4.3.24 ADMXRC3_GetSensorInfoA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetSensorInfoA(
    __in  ADMXRC3_HANDLE          hDevice,
    __in  unsigned int            sensorIndex,
    __out ADMXRC3_SENSOR_INFOA*   pSensorInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the sensor of interest.

sensorIndex (in)

Identifies the sensor of interest, within the device. This is a zero-based index that must be less than the **NumSensor** member of ADMXRC3_CARD_INFOEX.

pSensorInfo (out)

Points to an object of type ADMXRC3_SENSOR_INFOA in which to return information about the sensor.

Description

This function returns information about a sensor.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The sensorIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pSensorInfo parameter is NULL.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

This is the ANSI / UTF-8 version of ADMXRC3_GetSensorInfo. ADMXRC3_GetSensorInfo is actually a macro defined to be either ADMXRC3_GetSensorInfoW if the _**UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetSensorInfoA.

4.3.25 ADMXRC3_GetSensorInfoW

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetSensorInfoW(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  unsigned int           sensorIndex,
    __out ADMXRC3_SENSOR_INFOW*  pSensorInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the sensor of interest.

sensorIndex (in)

Identifies the sensor of interest, within the device. This is a zero-based index that must be less than the **NumSensor** member of ADMXRC3_CARD_INFOEX.

pSensorInfo (out)

Points to an object of type ADMXRC3_SENSOR_INFOW in which to return information about the sensor.

Description

This function returns information about a sensor.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error

occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The sensorIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pSensorInfo parameter is NULL.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

This is the Unicode version of ADMXRC3_GetSensorInfo. ADMXRC3_GetSensorInfo is actually a macro defined to be either ADMXRC3_GetSensorInfoW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetSensorInfoA.

4.3.26 ADMXRC3_GetStatusStringA

Declaration

```
const char*
ADMXRC3_GetStatusStringA(
    __in  ADMXRC3_STATUS  code,
    __in  boolean_t       bShort);
```

The parameter(s) of this function are as follows:

code (in)

Identifies the device that contains the target FPGA of interest.

bShort (in)

Specifies whether the description should be the full description, or merely a string of the corresponding symbolic value as defined by the ADMXRC3 API.

Description

This function returns a textual description of an ADMXRC3_STATUS code. The following example illustrates how the 'code' parameter affects the returned string:

FALSE

```
"The offset and/or length parameter(s) were invalid"
```

TRUE

```
"ADMXRC3_INVALID_REGION"
```

Return Value

The return value is a pointer to a NUL-terminated **char** string that describes the error code.

Remarks

The returned string is statically allocated, so an application must treat it as read-only.

This is the ANSI / UTF-8 version of ADMXRC3_GetStatusString. ADMXRC3_GetStatusString is actually a macro defined to be either ADMXRC3_GetStatusStringW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetStatusStringA.

4.3.27 ADMXRC3_GetStatusStringW

Declaration

```
const wchar_t*
ADMXRC3_GetStatusStringW(
    __in  ADMXRC3_STATUS  code,
    __in  boolean_t       bShort);
```

The parameter(s) of this function are as follows:

code (in)

Identifies the device that contains the target FPGA of interest.

bShort (in)

Specifies whether the description should be the full description, or merely a string of the corresponding symbolic value as defined by the ADMXRC3 API.

Description

This function returns a textual description of an ADMXRC3_STATUS code. The following example illustrates how the 'code' parameter affects the returned string:

FALSE

```
"The offset and/or length parameter(s) were invalid"
```

TRUE

```
"ADMXRC3_INVALID_REGION"
```

Return Value

The return value is a pointer to a NUL-terminated **wchar_t** string that describes the error code.

Remarks

The returned string is statically allocated, so an application must treat it as read-only.

This is the Unicode version of ADMXRC3_GetStatusString. ADMXRC3_GetStatusString is actually a macro defined to be either ADMXRC3_GetStatusStringW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_GetStatusStringA.

4.3.28 ADMXRC3_GetVersionInfo

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetVersionInfo(
    __in  ADMXRC3_HANDLE          hDevice,
    __out ADMXRC3_VERSION_INFO*   pVersionInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device handle to use to retrieve version information.

pVersionInfo (out)

Points to an object of type ADMXRC3_VERSION_INFO in which to return version information.

Description

This function returns version information for the ADMXRC3 API library and ADMXRC3 driver.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NULL_POINTER	The pVersionInfo parameter is NULL.

4.3.29 ADMXRC3_GetWindowInfo

Declaration

```
ADMXRC3_STATUS
ADMXRC3_GetWindowInfo(
    __in  ADMXRC3_HANDLE        hDevice,
    __in  unsigned int          windowIndex,
    __out ADMXRC3_WINDOW_INFO*  pWindowInfo);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the memory window of interest.

windowIndex (in)

Identifies the memory window of interest, within the device. This is a zero-based index that must be less than the **NumWindow** member of ADMXRC3_CARD_INFOEX.

pWindowInfo (out)

Points to an object of type ADMXRC3_WINDOW_INFO in which to return information about the memory window.

Description

This function returns information about a memory window that can be used to access device registers and/or target FPGAs.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The windowIndex parameter is out of range.
ADMXRC3_NULL_POINTER	The pWindowInfo parameter is NULL.

4.3.30 ADMXRC3_InitializeTicket

Declaration

```
ADMXRC3_InitializeTicket(
    __out ADMXRC3_TICKET*  pTicket);
```

The parameter(s) of this function are as follows:

pTicket (out)

Points to the ticket to be initialized.

Description

This function initializes an ADMXRC3_TICKET structure, and must be called in order to initialize a ticket.

In Windows, after calling ADMXRC3_InitializeTicket, the **Overlapped.hEvent** member of the ticket must also be set to a valid event handle, as described in ADMXRC3_TICKET.

Attempting to use an uninitialized ticket for a non-blocking operation may result in an error being returned, or undefined behavior.

Return Value

Remarks

In some operating systems, this function is a macro that does nothing.

4.3.31 ADMXRC3_LoadBitstreamA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_LoadBitstreamA(
    __in  const char*           pFilename,
    __out ADMXRC3_BITSTREAMA**  ppBitstream);
```

The parameter(s) of this function are as follows:

pFilename (in)

NUL-terminated **char** string that specifies the name of bitstream (.BIT) file.

ppBitstream (out)

Points to a variable of type ADMXRC3_BITSTREAMA*, which is initialized to point to an ADMXRC3_BITSTREAMA object.

Description

This function loads a bitstream (.BIT) file into memory, returing a pointer to an object of type ADMXRC3_BITSTREAMA via the ppBitstream parameter. The Data member of the ADMXRC3_BITSTREAMA object represents the beginning of an array containing configuration frame data suitable for passing to ADMXRC3_ConfigureFromBuffer.

ADMXRC3_LoadBitstreamW allocates memory to hold the entire bitstream, and an application eventually must free the object using ADMXRC3_UnloadBitstreamW.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_FILE_NOT_FOUND	The file identified by the pFilename parameter could not be opened.
ADMXRC3_INVALID_BITSTREAM	The file identified by the pFilename parameter is not a valid .BIT file.
ADMXRC3_NO_MEMORY	Memory to hold the bitstream data could not be allocated.
ADMXRC3_NULL_POINTER	The pFilename and / or ppBitstream parameter is a NULL pointer.

Remarks

This is the ANSI / UTF-8 version of ADMXRC3_LoadBitstream. ADMXRC3_LoadBitstream is actually a macro defined to be either ADMXRC3_LoadBitstreamW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_LoadBitstreamA.

4.3.32 ADMXRC3_LoadBitstreamW

Declaration

```
ADMXRC3_STATUS
ADMXRC3_LoadBitstreamW(
    __in  const wchar_t*        pFilename,
    __out ADMXRC3_BITSTREAMW**  ppBitstream);
```

The parameter(s) of this function are as follows:

pFilename (in)

NUL-terminated **wchar_t** string that specifies the name of bitstream (.BIT) file.

ppBitstream (out)

Points to a variable of type ADMXRC3_BITSTREAMW*, which is initialized to point to an ADMXRC3_BITSTREAMW object.

Description

This function loads a bitstream (.BIT) file into memory, returing a pointer to an object of type ADMXRC3_BITSTREAMW via the ppBitstream parameter. The Data member of the ADMXRC3_BITSTREAMW object represents the beginning of an array containing configuration frame data suitable for passing to ADMXRC3_ConfigureFromBuffer.

ADMXRC3_LoadBitstreamW allocates memory to hold the entire bitstream, and an application eventually must free the object using ADMXRC3_UnloadBitstreamW.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_FILE_NOT_FOUND	The file identified by the pFilename parameter could not be opened.
ADMXRC3_INVALID_BITSTREAM	The file identified by the pFilename parameter is not a valid .BIT file.
ADMXRC3_NO_MEMORY	Memory to hold the bitstream data could not be allocated.
ADMXRC3_NULL_POINTER	The pFilename and / or ppBitstream parameter is a NULL pointer.

Remarks

This is the Unicode version of ADMXRC3_LoadBitstream. ADMXRC3_LoadBitstream is actually a macro defined to be either ADMXRC3_LoadBitstreamW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_LoadBitstreamA.

4.3.33 ADMXRC3_Lock

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Lock(
    __in  ADMXRC3_HANDLE          hDevice,
    __in  const void*             pBuffer,
    __in  size_t                  length,
    __out ADMXRC3_BUFFER_HANDLE*  phBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

The device handle that is to be the owner of the locked buffer.

pBuffer (in)

Points to the buffer to be locked.

length (in)

The size of the buffer to be locked, in bytes.

phBuffer (out)

Points to a variable of type ADMXRC3_BUFFER_HANDLE that is to receive the handle to the locked buffer.

Description

This function locks a user-space buffer into memory so that the operating system cannot swap it out to backing store. If successful, then the function returns a handle of type ADMXRC3_BUFFER_HANDLE, and guarantees that the buffer will be wholly resident in physical memory until:

- The application unlocks the buffer with an inverse call to ADMXRC3_Unlock, or
- The application closes the device handle with a call to ADMXRC3_Close, or
- The application terminates without cleaning up; in this case, the buffer is unlocked when the system automatically closes any open device handles. However, this does not happen in VxWorks because it does not automatically close device handles when a task exits.

Use of this function reduces the overhead incurred in performing DMA transfers by eliminating the need to lock and unlock a user-space buffer for every DMA transfer. A user-space buffer can be locked once when an application initializes, used in an arbitrary number of DMA transfers, and unlocked when the application terminates. The following DMA functions target a locked user-space buffer:

- ADMXRC3_StartReadDMALocked
- ADMXRC3_StartReadDMALockedEx
- ADMXRC3_StartWriteDMALocked
- ADMXRC3_StartWriteDMALockedEx
- ADMXRC3_ReadDMALocked
- ADMXRC3_ReadDMALockedEx
- ADMXRC3_WriteDMALocked
- ADMXRC3_WriteDMALockedEx

ADMXRC3_BUFFER_HANDLE values are global to the system. This enables one process to lock a buffer, and if it can somehow communicate the buffer handle to another process, the other process can also use the buffer handle to perform DMA transfers. However, every buffer handle has an owner, which is the device handle that was used to create it. Only the owning device handle used can be used to successfully call ADMXRC3_Unlock.

A user-space buffer can be wholly or partially locked multiple times if necessary. If a user-space buffer is locked several times, yielding several buffer handles, it will only become swappable after every one of the buffer handles has been passed to ADMXRC3_Unlock.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_NO_MEMORY	Memory for keeping track of the locked buffer could not be allocated.
ADMXRC3_NULL_POINTER	The phBuffer parameter was a NULL pointer.
ADMXRC3_RESOURCE_LIMIT	The maximum number of locked buffers was already reached.

Remarks

Applications should avoid locking so much memory that the system no longer has enough physical memory left for the working sets of other processes in the system, such as applications and system services.

In order to avoid potential system memory corruption due to a badly behaved application, locked buffers have a reference counting mechanism. A locked user-space buffer begins with a reference count of 1. Calling ADMXRC3_Unlock invalidates the buffer handle and decrements the reference count. Each DMA transfer performed using the buffer increments the reference count when it begins and decrements the reference count when complete. When the reference count reaches zero, the user-space buffer is made swappable again. Thus, while a DMA transfer is in progress on a buffer, that buffer is not swappable, even if ADMXRC3_Unlock is called during the DMA transfer.

Locking is implemented with a page granuality by the operating system, where each user-space page has a lock count (separate to that of the reference count of buffer). Therefore, to be precise, the effect of this function is to iterate over each page of the user-space buffer and increment its lock count by 1. The implication of this is that any part of a user-space buffer can be locked multiple times, regardless of whether or not those parts are disjoint or overlapping. As long as there are matching calls to ADMXRC3_Unlock, the lock counts of all pages eventually return to 0.

4.3.34 ADMXRC3_MapCommonBuffer

Declaration

```
ADMXRC3_STATUS
ADMXRC3_MapCommonBuffer(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  uint64_t        busAddress,
    __in  size_t          offset,
    __in  size_t          length,
    __out void**          ppVirtualBase);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to which the common buffer of interest belongs.

busAddress (in)

The base bus address of the common buffer of interest within the device, obtained by a call to ADMXRC3_GetCommonBuffer.

offset (in)

The byte offset in the common buffer of interest of the region to be mapped.

length (in)

The length of the region to be mapped, in bytes.

ppVirtualBase (out)

Points to a variable of type **void*** that is to receive the address of where the region is mapped in the caller's address space.

Description

This function maps a region of a common buffer into the caller's address space. If successful, the returned pointer can then be dereferenced in order to read and write the mapped region. The region remains mapped until one of the following occurs:

- The region is unmapped by a call to ADMXRC3_UnmapCommonBuffer.
- The device handle used to map the common buffer is closed by a call to ADMXRC3_Close.
- The process terminates without cleaning up, in which case the mapping is lost as the device handle is automatically closed by the operating system and the process is destroyed.

A device's common buffers can be enumerated by calling ADMXRC3_GetCommonBuffer in order to avoid errors when calling ADMXRC3_MapCommonBuffer. It is legal to wholly or partially map a common buffer more than once.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The bufferIndex parameter specifies a nonexistent common buffer.
ADMXRC3_INVALID_REGION	The offset and length parameters specify an invalid region of the specified common buffer.
ADMXRC3_NO_MEMORY	Memory could not be allocated for keeping track of the mapping.
ADMXRC3_NULL_POINTER	The ppVirtualBase parameter was a NULL pointer.
ADMXRC3_REGION_TOO_LARGE	The region specified by the offset and length parameters was too large to be mapped in a single operation. See remarks below.

Remarks

The mapped region will generally use one of two strategies for avoiding cache-coherency problems:

1. In systems which automatically take care of cache invalidation and flushing for DMA transfers, the mapped region is cacheable. When a bus-master device accesses the region, the appropriate flushing and/or invalidation of cache lines is performed by the hardware in order to avoid stale data being read by either the CPU or the bus-master device.
2. In systems which do not automatically take care of cache invalidation and flushing for DMA transfers, the mapped region is noncacheable, avoiding any cache coherency issues altogether.

If **ADMXRC3_REGION_TOO_LARGE** is returned, the specified region is larger than the OS-dependent limit for a single map operation. The region should be split into two or more smaller regions which should then be mapped separately. This OS-dependent limit, if it exists, is generally much larger than the maximum size of common buffer that can reliably be allocated by the driver so is usually not a source of problems.

In VxWorks, this function does very little other than return a pointer to the region to be mapped, as VxWorks typically has a single global address space shared by the kernel and tasks alike.

This function is available in ADMXRC3 API version 1.5.0 and later via the header file <admxrc3/combuf.h>.

4.3.35 ADMXRC3_MapWindow

Declaration

```
ADMXRC3_STATUS
ADMXRC3_MapWindow(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    windowIndex,
    __in  uint64_t        offset,
    __in  uint64_t        length,
    __out void**          ppVirtualBase);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the window of interest.

windowIndex (in)

Identifies the window of interest in the device.

offset (in)

The byte offset in the window of interest of the region to be mapped.

length (in)

The length of the region to be mapped, in bytes.

ppVirtualBase (out)

Points to a variable of type **void*** that is to receive the address of where the region is mapped in the caller's address space.

Description

This function maps a region of a memory window into the caller's address space. If successful, the returned pointer can then be dereferenced in order to read and write the mapped region. The region remains mapped until one of the following occurs:

- The region is unmapped by a call to ADMXRC3_UnmapWindow.
- The device handle used to map the window is closed by a call to ADMXRC3_Close.
- The process terminates without cleaning up, in which case the mapping is lost as the device handle is automatically closed by the operating system and the process is destroyed.

A device's windows can be enumerated by calling ADMXRC3_GetWindowInfo in order to avoid errors when calling ADMXRC3_MapWindow. It is legal to wholly or partially map a window more than once.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The windowIndex parameter specifies a nonexistent window.
ADMXRC3_INVALID_REGION	The offset and length parameters specify an invalid region of the specified window.
ADMXRC3_NO_MEMORY	Memory could not be allocated for keeping track of the mapping.
ADMXRC3_NULL_POINTER	The ppVirtualBase parameter was a NULL pointer.
ADMXRC3_REGION_TOO_LARGE	The region specified by the offset and length parameters was too large to be mapped in a single operation. See remarks below.

Remarks

This function is useful for mapping a target FPGA's registers in a caller's address space. As described in Section 3.8.2.1, "Mapping memory windows into user-space", reading and writing using pointer dereference avoids the overhead of calling the ADMXRC3 API that is incurred with ADMXRC3_Read and ADMXRC3_Write.

There are a number of issues to be aware of when dereferencing the returned pointer to perform reads and writes, notably write buffers, load / store ordering and barriers. A discussion of these issues is outside the scope of this document, but applications must take account of them in order to guarantee the desired behavior.

The mapped region will generally be noncacheable by the CPU, as memory windows usually correspond to device registers or target FPGAs.

If **ADMXRC3_REGION_TOO_LARGE** is returned, the specified region is larger than the OS-dependent limit for a single map operation. The region should be split into two or more smaller regions which should then be mapped separately.

In VxWorks, this function does very little other than return a pointer to the region to be mapped, as VxWorks typically has a single global address space shared by the kernel and tasks alike.

4.3.36 ADMXRC3_Open

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Open(
    __in  unsigned int     index,
    __out ADMXRC3_HANDLE*  phCard);
```

The parameter(s) of this function are as follows:

index (in)

Identifies the device to open.

phCard (out)

Points to a variable of type ADMXRC3_HANDLE that is to receive the handle to the opened device.

Description

This function opens a device and returns a handle to it that can be used in subsequent ADMXRC3 API calls. The handle remains valid until ADMXRC3_Close is called. A given device can be opened multiple times, by the same process or different processes.

It is equivalent to calling ADMXRC3_OpenEx where the bPassive parameter is FALSE and the cooperativeLevel parameter is 0. Therefore, assuming the call succeeds, there are no restrictions on what API functions can be used with the returned device handle.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The calling process does not have sufficient privileges to open the device.
ADMXRC3_DEVICE_NOT_FOUND	The index parameter specifies a device that does not exist in the system.
ADMXRC3_NO_MEMORY	Memory could not be allocated for keeping track of a new device handle.

Remarks

With the exception of VxWorks, the operating system automatically closes any open handles if a process terminates without cleaning up.

4.3.37 ADMXRC3_OpenEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_OpenEx(
    __in  unsigned int     index,
    __in  bool_t           bPassive,
    __in  uint32_t         cooperativeLevel,
    __out ADMXRC3_HANDLE*  phCard);
```

The parameter(s) of this function are as follows:

index (in)

Identifies the device to open.

bPassive (in)

Specifies how the device is to be opened.

cooperativeLevel (in)

Must currently be zero.

phCard (out)

Points to a variable of type ADMXRC3_HANDLE that is to receive the handle to the opened device.

Description

This function opens a device and returns a handle to it that can be used in subsequent ADMXRC3 API calls. The handle remains valid until ADMXRC3_Close is called. A given device can be opened multiple times, by the same process or different processes.

The bPassive parameter controls how the device is opened:

- If the bPassive parameter is TRUE, the device is opened in "passive" mode. This allows unprivileged processes to open the device, but any subsequent calls using the returned device handle that modify device state will fail with the error ADMXRC3_ACCESS_DENIED. Most informational functions, such as ADMXRC3_GetCardInfoEx are callable, along with the diagnostic functions such as ADMXRC3_ReadSensor.
- If the bPassive parameter is FALSE, the device is opened in "active" mode. Only privileged processes can open a device in this manner, but unlike the passive case, there is no restriction on what API functions can be used with the returned device handle. If an unprivileged process attempts to call ADMXRC3_OpenEx in active mode, the call will fail and return an error code of ADMXRC3_ACCESS_DENIED.

This function is useful for monitoring applications that do not modify device state. The following is the list of functions that can be used with a device handle that has been opened in passive mode:

- ADMXRC3_Cancel
- ADMXRC3_Close
- ADMXRC3_FinishNotificationWait
- ADMXRC3_GetBankInfo
- ADMXRC3_GetCardInfo
- ADMXRC3_GetCardInfoEx
- ADMXRC3_GetClockFrequency
- ADMXRC3_GetFlashBlockInfo
- ADMXRC3_GetFlashInfoA
- ADMXRC3_GetFlashInfoW
- ADMXRC3_GetFpgaInfoA
- ADMXRC3_GetFpgaInfoW
- ADMXRC3_GetSensorInfoA
- ADMXRC3_GetSensorInfoW
- ADMXRC3_GetVersionInfo
- ADMXRC3_GetWindowInfo
- ADMXRC3_ReadSensor
- ADMXRC3_RegisterWin32Event
- ADMXRC3_RegisterVxwSem
- ADMXRC3_StartNotificationWait
- ADMXRC3_UnregisterWin32Event
- ADMXRC3_UnregisterVxwSem

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The calling process does not have sufficient privileges to open the device.
ADMXRC3_DEVICE_NOT_FOUND	The index parameter specifies a device that does not exist in the system.
ADMXRC3_NO_MEMORY	Memory could not be allocated for keeping track of a new device handle.

Remarks

With the exception of VxWorks, the operating system automatically closes any open handles if a process terminates without cleaning up.

4.3.38 ADMXRC3_Read

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Read(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    windowIndex,
    __in  uint32_t        flags,
    __in  size_t          offset,
    __in  size_t          length,
    __out void*           pBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the window to be read.

windowIndex (in)

Identifies the window within the device to be read.

flags (in)

Flags that modify the behavior of the function. There are no flags currently defined, and so this parameter must be zero.

offset (in)

The byte offset into the window at which reading is to begin.

length (in)

The number of bytes to read.

pBuffer (out)

Points to the buffer to receive the the data read from the window.

Description

This function reads data from a memory window in a device, starting at the specified offset within the window. The data transfer is performed by the CPU, and is therefore CPU intensive for large blocks of data.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The windowIndex parameter specifies a nonexistent window.
ADMXRC3_INVALID_REGION	The offset and length parameters specify an invalid region of the specified window.

Remarks

Calling ADMXRC3_Read incurs a certain overhead, but is acceptable for tasks that are not performance-critical, such as reading registers during initialization of an application. As described in Section 3.8.2.1, "Mapping memory windows into user-space", mapping a memory window into the caller's address space and using pointer dereferencing is generally faster when many register reads must be performed.

For large blocks of data, consider using DMA transfers, as described in Section 3.8.3.2, "DMA transfers with host memory".

If this function is used to read data from a target FPGA, the FPGA design must be able to cope with arbitrary byte enables being presented during reads, and side-effects on reads are best avoided. This is because the operand size used for load instructions in the data copying routines used by the ADMXRC3 API may vary depending upon CPU architecture, buffer alignment, etc. and even upon timing.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.39 ADMXRC3_ReadDMA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadDMA(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __out void*           pBuffer,
    __in  size_t          length,
    __in  uint32_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (out)

Points to the buffer that is to receive the data read from the device.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications, ADMXRC3_ReadDMALocked may be more appropriate than ADMXRC3_ReadDMA.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.40 ADMXRC3_ReadDMABus

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadDMABus(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  uint64_t        busAddress,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

busAddress (in)

The starting bus address at which data read from the device is stored.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function is intended for scenarios where data must be transferred directly between two peers on a bus,

where at least one of the devices is a Gen 3 reconfigurable computing device. It reads a block of data from a device, starting at the specified local address, and writes it beginning at the specified bus address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.3, "DMA transfers with peer devices".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Unlike ADMXRC3_ReadDMA and ADMXRC3_ReadDMAEx, this function does not lock anything into memory because the data is read directly from the device specified by **hDevice** and written to the bus address specified by **busAddress**. Thus, in some CPU architectures and platforms, there can be significantly less overhead in calling **ADMXRC3_ReadDMABus** compared to ADMXRC3_ReadDMA or ADMXRC3_ReadDMAEx.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.4.0 and later via the header file <admxrc3/dmabus.h>.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

The way in which the **busAddress** parameter is interpreted depends upon the I/O bus standard used by the device specified by **hDevice**. For example, if **hDevice** refers to an ADM-XRC-6T1, which has a PCI Express host interface, **busAddress** is a PCI Express memory space address.

In models that have a PCI Express host interface, it is not possible to determine when no PCI Express endpoint claims the address specified by **busAddress**. This is because PCI Express does not provide a standard mechanism for this type of error to be reported to the bus master. In such cases, the data that was read from the device is silently discarded and (assuming no other detectable errors occur) the return value is

ADMXRC3_SUCCESS.

4.3.41 ADMXRC3_ReadDMAEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadDMAEx(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __out void*           pBuffer,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (out)

Points to the buffer that is to receive the data read from the device.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications, ADMXRC3_ReadDMALockedEx may be more appropriate than ADMXRC3_ReadDMAEx.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.2.0 and later.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.42 ADMXRC3_ReadDMALocked

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadDMALocked(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint32_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be

bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer where data read from the device should be placed.

offset (in)

Offset into the locked buffer where data read from the device should be placed.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a locked buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_ReadDMALocked has less overhead than ADMXRC3_ReadDMA, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.

Return Value	Description
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.43 ADMXRC3_ReadDMALockedEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadDMALockedEx(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint64_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer where data read from the device should be placed.

offset (in)

Offset into the locked buffer where data read from the device should be placed.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a locked buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_ReadDMALockedEx has less overhead than ADMXRC3_ReadDMAEx, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.44 ADMXRC3_ReadFlash

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadFlash(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    flashIndex,
    __in  uint32_t        flags,
    __in  size_t          address,
    __in  size_t          length,
    __out void*           pBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device containing the Flash memory bank to be read.

flashIndex (in)

Specifies which Flash memory bank in the device is to be read.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_FLASH_SYNC
  Causes the cache for the Flash memory bank to be synchronized with the hardware before the function returns. For a description of the caching mechanism, refer to Section 3.8.6.1, "Flash memory caching"

address (in)

The byte address in the Flash memory bank at which to begin reading.

length (in)

Number of bytes to read from the Flash memory bank.

pBuffer (out)

The buffer that should receive the data read from the Flash memory bank.

Description

This function reads a block of data from Flash memory bank in a device. The data transfer is performed by the CPU, so is CPU intensive for large blocks.

The region of the Flash memory bank that is read, which is specified by the address and length parameters, can be of arbitrary alignment and does not (for example) need to be aligned to block boundaries. The region must be inside the user-programmable region, the bounds of which can be determined by calling ADMXRC3_GetFlashInfo.

The ADMXRC3 API implements a caching mechanism for each Flash memory bank. Passing the ADMXRC3_FLASH_SYNC flag ensures that the cache for the specified Flash memory bank is synchronized with the hardware before the function returns.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred while reading the Flash memory bank.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The flashIndex parameter specifies a nonexistent Flash memory bank.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the Flash memory bank.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the Flash read operation.

Remarks

Although the region specified by the address and length parameters need not bear any relationship to Flash block boundaries, an application can call ADMXRC3_GetFlashBlockInfo in order to determine which block contains a particular address. An application can enumerate every block in a Flash memory bank by beginning at address 0 and repeatedly calling ADMXRC3_GetFlashBlockInfo until the end of the bank is reached.

Although this function reads a Flash memory bank, in the event of an error, the returned error code may imply that a write or erase operation failed. This can occur because the caching mechanism may need to write a dirty block back to the hardware before another block is read.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.45 ADMXRC3_ReadSensor

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadSensor(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  unsigned int           sensorIndex,
    __out ADMXRC3_SENSOR_VALUE*  pValue);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the sensor of interest.

sensorIndex (in)

Identifies the sensor of interest within the device. This is a zero-based index that must be less than the

NumSensor member of ADMXRC3_CARD_INFOEX.

pValue (out)

Points to an object of type ADMXRC3_SENSOR_VALUE in which to return the sensor reading.

Description

This function reads a sensor, returning the current reading via the union ADMXRC3_SENSOR_VALUE.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_HARDWARE_ERROR	A hardware error occurred while reading the sensor.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The sensorIndex parameter specifies a nonexistent sensor.
ADMXRC3_NULL_POINTER	The pValue parameter was a NULL pointer.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

The datatype of the sensor, as returned by ADMXRC3_GetSensorInfo, determines how an application should interpret a value of type ADMXRC3_SENSOR_VALUE. It is the application's responsibility to be aware of a sensor's datatype and correctly interpret the returned sensor value.

4.3.46 ADMXRC3_ReadVPD

Declaration

```
ADMXRC3_STATUS
ADMXRC3_ReadVPD(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  uint32_t        flags,
    __in  size_t          offset,
    __in  size_t          length,
    __out void*           pBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device from which to read the VPD.

flags (in)

Flags that modify how the operation is performed. Currently no flags are defined, so this parameter must be zero.

offset (in)

The byte offset into the VPD memory at which to begin reading.

length (in)

Number of bytes to read from the VPD memory.

pBuffer (out)

The buffer that should receive the data read from the VPD memory.

Description

This function reads a block of Vital Product Data (VPD) from a device. The data transfer is performed by the CPU, so is CPU intensive for large blocks. For an overview, refer to Section 3.8.7, "Vital Product Data".

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred while reading the VPD memory.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the VPD memory.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the VPD read operation.

Remarks

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.47 ADMXRC3_RegisterWin32Event

Declaration

```
ADMXRC3_STATUS
ADMXRC3_RegisterWin32Event(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  uint32_t        notification,
    __in  HANDLE          hEvent);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device for which the Win32 Event should be registered.

notification (in)

The type of notification for which the Win32 Event should be registered.

hEvent (in)

The handle to the Win32 Event that is to be registered.

Description

This Windows-specific function registers a Win32 Event that is signalled whenever a certain event occurs within a device. The prototype for this function exists only for Windows; it is not defined for other operating systems.

The type of notification must be one of the following values:

- ADMXRC3_EVENT_FPGAALERT(targetIndex)
  Notification of overtemperature alerts for a target FPGA, where 'targetIndex' is the index of the target FPGA.
- ADMXRC3_EVENT_FPGAINTERRUPT(targetIndex)
  Notification of interrupts generated by a target FPGA, where 'targetIndex' is the index of the target FPGA.

A Win32 Event that is registered using this function is associated with the device handle used to register it. It is legal to register the same Win32 Event with several different devices, although the application will not be able to determine why the Win32 Event was signalled unless it checks each device in some application-specific manner. Multiple events may be registered with the same device handle for the same notification type, if necessary.

To unregister a Win32 Event so that it will no longer be signalled, an application calls ADMXRC3_UnregisterWin32Event using the same device handle that was used to register it. Attempting to unregister a Win32 Event using a different device handle to the one used to register it will fail.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle, or the hEvent parameter is not a valid Win32 Event handle.
ADMXRC3_INVALID_INDEX	The notification parameter specifies a type of notification that the API does not recognize.
ADMXRC3_NO_MEMORY	Memory could not be allocated for keeping track of the registered Win32 Event.

Remarks

When a notification is delivered, a Win32 Event registered for that notification is signalled regardless of whether or not it is already in a signalled state.

There is no queueing or counting of notifications in the ADMXRC3 API. A notification that occurs when there is no Win32 Event registered for it is lost. If a second notification is delivered via a Win32 Event before the application responds to the first notification, the application will see only one notification.

There are no restrictions on the parameters to the Win32 **CreateEvent** function when creating an event for use with ADMXRC3_RegisterWin32Event.

The ADMXRC3 API automatically unregisters any Win32 Events when a device handle is closed. This can occur either because of call to ADMXRC3_Close, or when a device handle is automatically closed by the operating system, as the result of a process terminating without cleaning up.

4.3.48 ADMXRC3_RegisterVxwSem

Declaration

```
ADMXRC3_STATUS
ADMXRC3_RegisterVxwSem(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  uint32_t        notification,
    __in  SEM_ID          semId);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device for which the semaphore should be registered.

notification (in)

The type of notification for which the semaphore should be registered.

semId (in)

The ID of the VxWorks semaphore that is to be registered.

Description

This VxWorks-specific function registers a semaphore that is signalled whenever a certain event occurs within a device. The prototype for this function exists only for VxWorks; it is not defined for other operating systems.

The type of notification must be one of the following values:

- ADMXRC3_EVENT_FPGAALERT(targetIndex)
  Notification of overtemperature alerts for a target FPGA, where 'targetIndex' is the index of the target FPGA.
- ADMXRC3_EVENT_FPGAINTERRUPT(targetIndex)
  Notification of interrupts generated by a target FPGA, where 'targetIndex' is the index of the target FPGA.

A VxWorks semaphore that is registered using this function is associated with the device handle used to register it. It is legal to register the same semaphore with several different devices, although the application will not be able to determine why the semaphore was signalled unless it checks each device in some application-specific manner. Multiple semaphores may be registered with the same device handle for the same notification type, if necessary.

To unregister a semaphore so that it will no longer be signalled, an application calls ADMXRC3_UnregisterVxwSem using the same device handle that was used to register it. Attempting to unregister a semaphore using a different device handle to the one used to register it will fail.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle, or the semId parameter is not a valid VxWorks semaphore ID.
ADMXRC3_INVALID_INDEX	The notification parameter specifies a type of notification that the API does not recognize.
ADMXRC3_NO_MEMORY	Memory could not be allocated for keeping track of the registered VxWorks semaphore.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

When a notification is delivered, a semaphore registered for that notification is signalled regardless of whether or not it is already in a signalled state.

There is no queueing or counting of notifications in the ADMXRC3 API. A notification that occurs when there is no semaphore registered for it is lost. If a second notification is delivered via a semaphore before the application responds to the first notification, the application will see only one notification unless a counting semaphore is used.

There are no restrictions on the type of VxWorks semaphore (binary, counting etc.) when creating a semaphore for use with ADMXRC3_RegisterVxwSem.

The ADMXRC3 API automatically unregisters any semaphores when a device handle is closed.

4.3.49 ADMXRC3_SetClockFrequency

Declaration

```
ADMXRC3_STATUS
ADMXRC3_SetClockFrequency(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    clockIndex,
    __in  uint32_t        flags,
    __in  double          frequency,
    __out double*         pActualFrequency);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the clock generator of interest.

clockIndex (in)

Specifies which clock generator within the device is of interest. This is a zero-based index that must be less than the **NumClockGen** member of ADMXRC3_CARD_INFOEX.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_SETCLOCK_TESTONLY
  Causes the actual frequency to be calculated, but does not actually program the hardware.
- ADMXRC3_SETCLOCK_MAXIMUM
  Causes the algorithm that generates the programming information for the clock generator to return an actual frequency that is not greater than the requested frequency.
- ADMXRC3_SETCLOCK_MINIMUM
  Causes the algorithm that generates the programming information for the clock generator to return an actual frequency that is not less than the requested frequency.

frequency (in)

Specifies the requested frequency, in Hz.

pActualFrequency (out)

Points to an object of type **double** in which to return the actual frequency (in Hz) that is (or would be) programmed into the clock generator. This parameter may be NULL if an application does not require the actual programmed frequency.

Description

This function programs a clock generator in a device to output a clock signal at a specified frequency.

Clock generator chips are typically implemented as a phase-locked loop with programmable division and

multiplication factors. This means that a clock generator does not have a continuous frequency range, but is capable generating a large number of discrete frequencies. Thus ADMXRC3_SetClockFrequency returns the achievable frequency that is closest to the requested frequency.

The ADMXRC3_SETCLOCK_TESTONLY flag can be passed when an application wishes to determine how close the actual frequency will be to the requested frequency, before actually programming the hardware. It causes the function to return without actually programming the hardware.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred while programming the clock generator.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_FREQUENCY	The requested frequency could not be programmed. See remarks below.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The clockIndex parameter specifies a nonexistent clock generator.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the clock programming operation.

Remarks

Depending on the model, this function may take up 50 milliseconds to execute, as some clock generator chips require a significant length of time to lock to the new frequency. During the transition period, the clock signal does not exhibit discontinuities or glitches.

Passing both the ADMXRC3_SETCLOCK_MAXIMUM and ADMXRC3_SETCLOCK_MINIMUM flags together means that if the clock generator cannot generate the exact requested frequency, it returns an error.

If the function returns ADMXRC3_INVALID_FREQUENCY, it may be because the flags were too restrictive (e.g. ADMXRC3_SETCLOCK_MINIMUM and ADMXRC3_SETCLOCK_MAXIMUM passed together, requiring the frequency to be "exact"), or because the requested frequency is too high or low for the clock generator in question.

4.3.50 ADMXRC3_StartNotificationWait

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartNotificationWait(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  ADMXRC3_TICKET  pTicket,
    __in  uint32_t        notification);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device for which to perform the wait.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

notification (in)

The type of notification for which to wait.

Description

This function begins a non-blocking operation that waits for a notification from a device. The type of notification must be one of the following values:

- ADMXRC3_EVENT_FPGAALERT(targetIndex)
  Notification of overtemperature alerts for a target FPGA, where 'targetIndex' is the index of the target FPGA.
- ADMXRC3_EVENT_FPGAINTERRUPT(targetIndex)
  Notification of interrupts generated by a target FPGA, where 'targetIndex' is the index of the target FPGA.

The counterpart to this function is ADMXRC3_FinishNotificationWait, which finishes the non-blocking operation. Every call to ADMXRC3_StartNotificationWait that succeeds (i.e. returns ADMXRC3_PENDING) must be finished with a call to ADMXRC3_FinishNotificationWait.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The notification parameter specifies a type of notification that the API does not recognize.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the notification wait operation.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later, and provides a mechanism that is broadly equivalent to ADMXRC3_RegisterWin32Event or ADMXRC3_RegisterVxwSem, except that instead of registering a wait object with the API, it is a direct-call mechanism.

There is no queueing of notifications. If some event occurs in a device and no thread is waiting for that notification, the event is lost.

4.3.51 ADMXRC3_StartReadDMA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartReadDMA(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  ADMXRC3_TICKET  pTicket,
```

```
__in  unsigned int    dmaChannel,
__in  uint32_t        flags,
__out void*           pBuffer,
__in  size_t          length,
__in  uint32_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (out)

Points to the buffer that is to receive the data read from the device.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_ReadDMA, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications, ADMXRC3_StartReadDMALocked may be more appropriate than ADMXRC3_StartReadDMA.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking

operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.52 ADMXRC3_StartReadDMABus

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartReadDMABus(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  ADMXRC3_TICKET  pTicket,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  uint64_t        busAddress,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

busAddress (in)

The starting bus address at which data read from the device is stored.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function is intended for scenarios where data must be transferred directly between two peers on a bus, where at least one of the devices is a Gen 3 reconfigurable computing device. It reads a block of data from a device, starting at the specified local address, and writes it beginning at the specified bus address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.3, "DMA transfers with peer devices".

This is the non-blocking version of ADMXRC3_ReadDMABus, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Unlike ADMXRC3_StartReadDMA and ADMXRC3_StartReadDMAEx, this function does not lock anything into memory because the data is read directly from the device specified by **hDevice** and written to the bus address specified by **busAddress**. Thus, in some CPU architectures and platforms, there can be significantly less overhead in calling **ADMXRC3_StartReadDMABus** compared to ADMXRC3_StartReadDMA or ADMXRC3_StartReadDMAEx.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.

Return Value	Description
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.4.0 and later via the header file <admxrc3/dmabus.h>.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

The way in which the **busAddress** parameter is interpreted depends upon the I/O bus standard used by the device specified by **hDevice**. For example, if **hDevice** refers to an ADM-XRC-6T1, which has a PCI Express host interface, **busAddress** is a PCI Express memory space address.

In models that have a PCI Express host interface, it is not possible to determine when no PCI Express endpoint claims the address specified by **busAddress**. This is because PCI Express does not provide a standard mechanism for this type of error to be reported to the bus master. In such cases, the data that was read from the device is silently discarded and (assuming no other detectable errors occur) the return value from ADMXRC3_FinishDMA is **ADMXRC3_SUCCESS**.

4.3.53 ADMXRC3_StartReadDMAEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartReadDMAEx(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  ADMXRC3_TICKET  pTicket,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __out void*           pBuffer,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE

Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (out)

Points to the buffer that is to receive the data read from the device.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_ReadDMAEx, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications, ADMXRC3_StartReadDMALockedEx may be more appropriate than ADMXRC3_StartReadDMAEx.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.

Return Value	Description
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.2.0 and later.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.54 ADMXRC3_StartReadDMALocked

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartReadDMALocked(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  ADMXRC3_TICKET         pTicket,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint32_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer where data read from the device should be placed.

offset (in)

Offset into the locked buffer where data read from the device should be placed.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a locked buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_ReadDMALocked, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_StartReadDMALocked has less overhead than ADMXRC3_StartReadDMA, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.55 ADMXRC3_StartReadDMALockedEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartReadDMALockedEx(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  ADMXRC3_TICKET         pTicket,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint64_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer where data read from the device should be placed.

offset (in)

Offset into the locked buffer where data read from the device should be placed.

length (in)

Number of bytes to read from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin reading.

Description

This function reads a block of data from a device into a locked buffer, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_ReadDMALockedEx, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in

Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_StartReadDMALockedEx has less overhead than ADMXRC3_StartReadDMAEx, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.56 ADMXRC3_StartWriteDMA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartWriteDMA(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  ADMXRC3_TICKET  pTicket,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  const void*     pBuffer,
    __in  size_t          length,
    __in  uint32_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (in)

Points to the buffer that contains data to write to the device.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from a buffer into a device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_WriteDMA, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications, ADMXRC3_StartWriteDMALocked may be more appropriate than ADMXRC3_StartWriteDMA.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.57 ADMXRC3_StartWriteDMABus

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartWriteDMABus(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  ADMXRC3_TICKET  pTicket,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  uint64_t        busAddress,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.

- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

busAddress (in)

The starting bus address from which data is read.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function is intended for scenarios where data must be transferred directly between two peers on a bus, where at least one of the devices is a Gen 3 reconfigurable computing device. It reads a block of data, starting at the specified bus address, and writes it to the specified device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.3, "DMA transfers with peer devices".

This is the non-blocking version of ADMXRC3_WriteDMABus, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Unlike ADMXRC3_StartWriteDMA and ADMXRC3_StartWriteDMAEx, this function does not lock anything into memory because the data is read directly from the bus address specified by **busAddress** and written to the the device specified by **hDevice**. Thus, in some CPU architectures and platforms, there can be significantly less overhead in calling **ADMXRC3_StartWriteDMABus** compared to ADMXRC3_StartWriteDMA or ADMXRC3_StartWriteDMAEx.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.

Return Value	Description
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.4.0 and later via the header file <admxrc3/dmabus.h>.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

The way in which the **busAddress** parameter is interpreted depends upon the I/O bus standard used by the device specified by **hDevice**. For example, if **hDevice** refers to an ADM-XRC-6T1, which has a PCI Express host interface, **busAddress** is a PCI Express memory space address.

When no PCI Express endpoint claims the address specified by **busAddress**, the DMA engine will not receive any data and will time out within a few tens of microseconds of beginning the DMA transfer. In such cases, the return value from ADMXRC3_FinishDMA is **ADMXRC3_HARDWARE_ERROR**.

4.3.58 ADMXRC3_StartWriteDMAEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartWriteDMAEx(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  ADMXRC3_TICKET  pTicket,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  const void*     pBuffer,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (in)

Points to the buffer that contains data to write to the device.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from a buffer into a device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_WriteDMAEx, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications, ADMXRC3_StartWriteDMALockedEx may be more appropriate than ADMXRC3_StartWriteDMAEx.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.

Return Value	Description
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.2.0 and later.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.59 ADMXRC3_StartWriteDMALocked

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartWriteDMALocked(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  ADMXRC3_TICKET         pTicket,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint32_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer that contains the data to be written to the device.

offset (in)

Offset into the locked buffer where the data to be written to the device is located.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from a locked buffer into a device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_WriteDMALocked, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_StartWriteDMALocked has less overhead than ADMXRC3_StartWriteDMA, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.60 ADMXRC3_StartWriteDMALockedEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_StartWriteDMALockedEx(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  ADMXRC3_TICKET         pTicket,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint64_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

pTicket (in)

A ticket used to keep track of the non-blocking operation.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer that contains the data to be written to the device.

offset (in)

Offset into the locked buffer where the data to be written to the device is located.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from a locked buffer into a device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

This is the non-blocking version of ADMXRC3_WriteDMALockedEx, and the pTicket parameter must follow the rules laid out in Section 3.3, "Non-blocking operations". To finish the non-blocking operation and determine the success or failure of the DMA transfer, call ADMXRC3_FinishDMA.

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in

Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_StartWriteDMALockedEx has less overhead than ADMXRC3_StartWriteDMAEx, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_PENDING indicates that the function executed successfully and that a non-blocking operation was started. Otherwise, if an error occurs, a non-blocking operation was not started and the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.61 ADMXRC3_SyncFlash

Declaration

```
ADMXRC3_STATUS
ADMXRC3_SyncFlash(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    flashIndex,
    __in  uint32_t        flags);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device containing the Flash memory bank to be synchronized.

flashIndex (in)

Specifies which Flash memory bank in the device is to synchronized.

flags (in)

Flags that modify how the operation is performed. Currently there are no flags defined, so this parameter must be zero.

Description

This function ensures that the cache for a Flash memory bank is "clean" before returning. Applications can call it to ensure that write or erase operations have been committed to the hardware. The caching mechanism for Flash memory banks is described in Section 3.8.6.1, "Flash memory caching".

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred while synchronizing the Flash memory bank.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The flashIndex parameter specifies a nonexistent Flash memory bank.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the Flash synchronization operation.

Remarks

For Flash memory banks that use chips with a block-oriented architecture, calling this function may result in noticeable delays in execution, because some Flash chips have noticeable block erase and programming periods.

4.3.62 ADMXRC3_Unconfigure

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Unconfigure(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    flashIndex,
    __in  uint32_t        flags);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device containing the target FPGA of interest.

flashIndex (in)

Specifies which target FPGA within a device is of interest.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_CONFIGURE_PARTIAL
  Causes the target FPGA not to be unconfigured.
- ADMXRC3_CONFIGURE_SHARE
  Causes ownership to be unchanged.

Description

This function unconfigures a target FPGA and/or relinquishes ownership.

Passing the ADMXRC3_CONFIGURE_PARTIAL flag causes an unconfiguration sequence, as described in Section 3.8.1.3, "Unconfiguration", not to be performed on the target FPGA.

Passing the ADMXRC3_CONFIGURE_SHARE flag causes the target FPGA's ownership state to be unchanged.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	Unconfiguring the target FPGA was not successful. See remarks below.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter is not a valid device handle.
ADMXRC3_INVALID_INDEX	The targetIndex parameter is out of range.
ADMXRC3_NOT_OWNER	The specified device handle is not the owner of the target FPGA.

Remarks

Passing only the ADMXRC3_CONFIGURE_PARTIAL flag is how a device handle can relinquish ownership of a target FPGA without unconfiguring it.

Passing only the ADMXRC3_CONFIGURE_SHARE flag is how a device handle can unconfigure a target FPGA without relinquishing ownership.

Passing both the ADMXRC3_CONFIGURE_PARTIAL and ADMXRC3_CONFIGURE_SHARE flags is legal but results in a no-operation.

If the function returns ADMXRC3_HARDWARE_ERROR, the target FPGA will be in an indeterminate state (may or may not be configured).

4.3.63 ADMXRC3_UnloadBitstreamA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_UnloadBitstreamA(
    __inout ADMXRC3_BITSTREAMA*  pBitstream);
```

The parameter(s) of this function are as follows:

pBitstream (inout)

Points to the object of type ADMXRC3_BITSTREAMA that is to be freed.

Description

This function unloads a bitstream (.BIT) file, freeing the memory used by it. It is the inverse of ADMXRC3_LoadBitstreamA.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_NULL_POINTER	The pBitstream parameter was a NULL pointer.

Remarks

This is the ANSI / UTF-8 version of ADMXRC3_UnloadBitstream. ADMXRC3_UnloadBitstream is actually a macro defined to be either ADMXRC3_UnloadBitstreamW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_UnloadBitstreamA.

This function must be used to deallocate an object of type ADMXRC3_BITSTREAMA, rather than **free**. In Windows, attempting to use **free** results in heap corruption because memory must be freed using the same heap that was used to allocate it.

4.3.64 ADMXRC3_UnloadBitstreamW

Declaration

```
ADMXRC3_STATUS
ADMXRC3_UnloadBitstreamW(
    __inout ADMXRC3_BITSTREAMW*  pBitstream);
```

The parameter(s) of this function are as follows:

pBitstream (inout)

Points to the object of type ADMXRC3_BITSTREAMW that is to be freed.

Description

This function unloads a bitstream (.BIT) file, freeing the memory used by it. It is the inverse of ADMXRC3_LoadBitstreamW.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_NULL_POINTER	The pBitstream parameter was a NULL pointer.

Remarks

This is the ANSI / UTF-8 version of ADMXRC3_UnloadBitstream. ADMXRC3_UnloadBitstream is actually a macro defined to be either ADMXRC3_UnloadBitstreamW if the **_UNICODE** preprocessor symbol is defined, or else ADMXRC3_UnloadBitstreamA.

This function must be used to deallocate an object of type ADMXRC3_BITSTREAMW, rather than **free**. In Windows, attempting to use **free** results in heap corruption because memory must be freed using the same heap that was used to allocate it.

4.3.65 ADMXRC3_Unlock

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Unlock(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

The device handle that was used to create the locked buffer.

hBuffer (in)

Identifies the locked buffer that is to be unlocked.

Description

This function unlocks a locked user-space buffer so that the operating system can again swap it out to backing store. An application should assume that an operating system may begin swapping the buffer out as soon as this function is called. ADMXRC3_Unlock is the inverse of ADMXRC3_Lock.

ADMXRC3_BUFFER_HANDLE values are global to the system. However, every buffer handle has an owner, which is the device handle that was used to create it. Only the device handle used to lock a buffer can be used to successfully call ADMXRC3_Unlock.

A user-space buffer can be locked multiple times if necessary. If a user-space buffer is locked several times, yielding several buffer handles, it will only become swappable once more after every one of the locked buffer handles has been passed to ADMXRC3_Unlock.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_BUFFER_INVALID_HANDLE	The hBuffer parameter was not a valid handle to a locked buffer.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_NOT_OWNER	The locked buffer was created using a different device handle.

Remarks

In order to avoid potential system memory corruption due to a badly behaved application, locked buffers have

a reference counting mechanism. A locked user-space buffer begins with a reference count of 1. Calling ADMXRC3_Unlock invalidates the buffer handle and decrements the reference count. Each DMA transfer performed using the buffer increments the reference count when it begins and decrements the reference count when complete. When the reference count reaches zero, the user-space buffer is made swappable again. Thus, while a DMA transfer is in progress on a buffer, that buffer is not swappable, even if ADMXRC3_Unlock is called during the DMA transfer.

Locking is implemented with a page granuality by the operating system, where each user-space page has a lock count. Therefore, to be precise, this function iterates through each page of the user-space buffer and decrements its lock count by 1. The implication of this is that any part of a user-space buffer can be locked multiple times, regardless of whether or not those parts are disjoint or overlapping. As long as there are matching calls to ADMXRC3_Unlock, the lock counts of all pages eventually return to 0.

4.3.66 ADMXRC3_UnmapCommonBuffer

Declaration

```
ADMXRC3_STATUS
ADMXRC3_UnmapCommonBuffer(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  void*           pVirtualBase);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to which the common buffer of interest belongs.

pVirtualBase (in)

The base virtual address of a region of a common buffer that was mapped using ADMXRC3_MapCommonBuffer.

Description

This function unmaps a region of a common buffer from the caller's address space, and is the inverse of ADMXRC3_MapCommonBuffer. An application must consider the region that **pVirtualBase** points to invalid as soon as this function is called.

Partially unmapping a region is not possible, so the **pVirtualBase** parameter must be the value previously returned by ADMXRC3_MapCommonBuffer, and cannot be offset by some amount.

If an application terminates without unmapping any regions that it mapped, automatic cleanup is performed by the operating system (except in VxWorks).

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_REGION	The pVirtualBase parameter is not recognized as the base address of a mapped region of a common buffer.
ADMXRC3_NULL_POINTER	The pVirtualBase parameter was a NULL pointer.

Remarks

In VxWorks, this function is a essentially a no-operation, as VxWorks typically has a single global address

space shared by the kernel and tasks alike.

4.3.67 ADMXRC3_UnmapWindow

Declaration

```
ADMXRC3_STATUS
ADMXRC3_UnmapWindow(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  void*           pVirtualBase);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the window of interest.

pVirtualBase (in)

The base virtual address of a region of a window that was mapped using ADMXRC3_MapWindow.

Description

This function unmaps a region of a memory window from the caller's address space, and is the inverse of ADMXRC3_MapWindow. An application must consider the region that **pVirtualBase** points to invalid as soon as this function is called.

Partially unmapping a region is not possible, so the **pVirtualBase** parameter must be the value previously returned by ADMXRC3_MapWindow, and cannot be offset by some amount.

If an application terminates without unmapping any regions that it mapped, automatic cleanup is performed by the operating system (except in VxWorks).

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_REGION	The pVirtualBase parameter is not recognized as the base address of a mapped region of device memory.
ADMXRC3_NULL_POINTER	The pVirtualBase parameter was a NULL pointer.

Remarks

In VxWorks, this function is a essentially a no-operation, as VxWorks typically has a single global address space shared by the kernel and tasks alike.

4.3.68 ADMXRC3_UnregisterWin32Event

Declaration

```
ADMXRC3_STATUS
ADMXRC3_UnregisterWin32Event(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  uint32_t        notification,
    __in  HANDLE          hEvent);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device for which the Win32 Event should no longer be registered.

notification (in)

The type of notification for which the Win32 Event should no longer be registered.

hEvent (in)

The handle to the Win32 Event that is to be unregistered.

Description

This Windows-specific function unregisters a Win32 Event that was previously registered using ADMXRC3_RegisterWin32Event. The prototype for this function exists only for Windows; it is not defined for other operating systems.

The type of notification must be one of the following values:

- ADMXRC3_EVENT_FPGAALERT(targetIndex)
  Notification of overtemperature alerts for a target FPGA, where 'targetIndex' is the index of the target FPGA.
- ADMXRC3_EVENT_FPGAINTERRUPT(targetIndex)
  Notification of interrupts generated by a target FPGA, where 'targetIndex' is the index of the target FPGA.

Unregistering a Win32 Event fails if the Win32 Event is not currently registered for the specified notification for the specified device handle.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle, or the hEvent was not recognized as a valid Win32 Event registered with the specified device handle.
ADMXRC3_INVALID_INDEX	The notification parameter specifies a type of notification that the API does not recognize.

Remarks

The ADMXRC3 API automatically unregisters any Win32 Events when a device handle is closed. This can occur either because of call to ADMXRC3_Close, or when a device handle is automatically closed by the operating system, as the result of a process terminating without cleaning up.

4.3.69 ADMXRC3_UnregisterVxwSem

Declaration

```
ADMXRC3_STATUS
ADMXRC3_UnregisterVxwSem(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  uint32_t        notification,
    __in  SEM_ID          semId);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device for which the semaphore should no longer be registered.

notification (in)

The type of notification for which the semaphore should no longer be registered.

semId (in)

Identifies the semaphore that is to be unregistered.

Description

This VxWorks-specific function unregisters a VxWorks semaphore that was previously registered using ADMXRC3_RegisterVxwSem. The prototype for this function exists only for VxWorks; it is not defined for other operating systems.

The type of notification must be one of the following values:

- ADMXRC3_EVENT_FPGAALERT(targetIndex)
  Notification of overtemperature alerts for a target FPGA, where 'targetIndex' is the index of the target FPGA.
- ADMXRC3_EVENT_FPGAINTERRUPT(targetIndex)
  Notification of interrupts generated by a target FPGA, where 'targetIndex' is the index of the target FPGA.

Unregistering a semaphore fails if the semaphore is not currently registered for the specified notification type for the specified device handle.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle, or the semId parameter was not recognized as a valid VxWorks semaphore registered with the specified device handle.
ADMXRC3_INVALID_INDEX	The notification parameter specifies a type of notification that the API does not recognize.

Remarks

This function is available in ADMXRC3 API version 1.1.0 and later.

The ADMXRC3 API automatically unregisters any semaphores when a device handle is closed.

4.3.70 ADMXRC3_Write

Declaration

```
ADMXRC3_STATUS
ADMXRC3_Write(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    windowIndex,
    __in  uint32_t        flags,
    __in  size_t          offset,
    __in  size_t          length,
    __in  const void*     pBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device that contains the window to be written.

windowIndex (in)

Identifies the window within the device to be written.

flags (in)

Flags that modify the behavior of the function. There are no flags currently defined, and so this parameter must be zero.

offset (in)

The byte offset into the window at which writing is to begin.

length (in)

The number of bytes to write.

pBuffer (in)

Points to the buffer containing the data to be written to the window.

Description

This function writes data from a buffer to a memory window in a device, starting at the specified offset within the window. The data transfer is performed by the CPU, and is therefore CPU intensive for large blocks of data.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The windowIndex parameter specifies a nonexistent window.
ADMXRC3_INVALID_REGION	The offset and length parameters specify an invalid region of the specified window.

Remarks

Calling ADMXRC3_Write incurs a certain overhead, but is acceptable for tasks that are not performance-critical, such as writing to registers during initialization of an application. As described in Section 3.8.2.1, "Mapping memory windows into user-space", mapping a memory window into the caller's address space and using pointer dereferencing is generally faster when many register writes must be performed.

For large blocks of data, consider using DMA transfers, as described in Section 3.8.3.2, "DMA transfers with host memory".

If this function is used to write data to a target FPGA, the FPGA design must be able to cope with arbitrary byte enables being presented during writes. Side-effects during writes are allowed, but should always be qualified by byte enables. This is because the operand size used for store instructions in the data copying routines used by the ADMXRC3 API may vary depending upon CPU architecture, buffer alignment, etc. and

even upon timing.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.71 ADMXRC3_WriteDMA

Declaration

```
ADMXRC3_STATUS
ADMXRC3_WriteDMA(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  const void*     pBuffer,
    __in  size_t          length,
    __in  uint32_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (in)

Points to the buffer that contains the data to write to the device.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from buffer into a device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications,

ADMXRC3_WriteDMALocked may be more appropriate than ADMXRC3_WriteDMA.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.72 ADMXRC3_WriteDMABus

Declaration

```
ADMXRC3_STATUS
ADMXRC3_WriteDMABus(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  uint64_t        busAddress,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

busAddress (in)

The starting bus address from which data is read.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function is intended for scenarios where data must be transferred directly between two peers on a bus, where at least one of the devices is a Gen 3 reconfigurable computing device. It reads a block of data, starting at the specified bus address, and writes it to the specified device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.3, "DMA transfers with peer devices".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Unlike ADMXRC3_WriteDMA and ADMXRC3_WriteDMAEx, this function does not lock anything into memory because the data is read directly from the bus address specified by **busAddress** and written to the the device specified by **hDevice**. Thus, in some CPU architectures and platforms, there can be significantly less overhead in calling **ADMXRC3_WriteDMABus** compared to ADMXRC3_WriteDMA or ADMXRC3_WriteDMAEx.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.

Return Value	Description
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.4.0 and later via the header file <admxrc3/dmabus.h>.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

The way in which the **busAddress** parameter is interpreted depends upon the I/O bus standard used by the device specified by **hDevice**. For example, if **hDevice** refers to an ADM-XRC-6T1, which has a PCI Express host interface, **busAddress** is a PCI Express memory space address.

When no PCI Express endpoint claims the address specified by **busAddress**, the DMA engine will not receive any data and will time out within a few tens of microseconds of beginning the DMA transfer. In such cases, the return value is **ADMXRC3_HARDWARE_ERROR**.

4.3.73 ADMXRC3_WriteDMAEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_WriteDMAEx(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    dmaChannel,
    __in  uint32_t        flags,
    __in  const void*     pBuffer,
    __in  size_t          length,
    __in  uint64_t        localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

pBuffer (in)

Points to the buffer that contains the data to write to the device.

length (in)

Number of bytes to write to the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from buffer into a device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

Because buffers in user-space may be wholly or partially swapped out to backing store at any time in most operating systems, this function locks the buffer into physical memory before starting the DMA transfer proper. When the DMA transfer is finished, the buffer is unlocked and made swappable again. Locking and unlocking a user-space buffer incurs a certain overhead, so for performance-critical applications, ADMXRC3_WriteDMALockedEx may be more appropriate than ADMXRC3_WriteDMAEx.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

This function is available in ADMXRC3 API version 1.2.0 and later.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.74 ADMXRC3_WriteDMALocked

Declaration

```
ADMXRC3_STATUS
ADMXRC3_WriteDMALocked(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint32_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer that contains the data to be written to the device.

offset (in)

Offset into the locked buffer where the data to be written to the device is located.

length (in)

Number of bytes to write from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from a locked buffer into device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_WriteDMALocked has less overhead than ADMXRC3_WriteDMA, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.75 ADMXRC3_WriteDMALockedEx

Declaration

```
ADMXRC3_STATUS
ADMXRC3_WriteDMALockedEx(
    __in  ADMXRC3_HANDLE         hDevice,
    __in  unsigned int           dmaChannel,
    __in  uint32_t               flags,
    __in  ADMXRC3_BUFFER_HANDLE  hBuffer,
    __in  size_t                 offset,
    __in  size_t                 length,
    __in  uint64_t               localAddress);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device to perform the DMA transfer.

dmaChannel (in)

Specifies which DMA channel in the device is to be used to perform the DMA transfer.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_DMA_FIXEDLOCAL
  Causes the local address to be held constant throughout the entire DMA transfer.
- ADMXRC3_DMA_DONOTQUEUE
  Causes the function to return an error immediately, rather than wait in a queue, if the DMA transfer cannot be started immediately due to another DMA transfer ongoing on the same DMA channel.

hBuffer (in)

Handle to the locked buffer that contains the data to be written to the device.

offset (in)

Offset into the locked buffer where the data to be written to the device is located.

length (in)

Number of bytes to write from the device.

localAddress (in)

The local address (in bytes) in the device at which to begin writing.

Description

This function writes a block of data from a locked buffer into device, starting at the specified local address. The data transfer is performed by a DMA engine within the device. DMA transfers in the ADMXRC3 API are described in general terms in Section 3.8.3.2, "DMA transfers with host memory".

DMA transfers are subject to a queueing mechanism unless the ADMXRC3_DMA_DONOTQUEUE flag is passed. This flag causes the function to give up immediately and return an error if it finds that another DMA transfer is in progress on the same DMA channel. The queueing mechanism is described in general terms in Section 3.4, "Queueing".

This function must be passed a handle to an already locked buffer, obtained by calling ADMXRC3_Lock. ADMXRC3_WriteDMALockedEx has less overhead than ADMXRC3_WriteDMAEx, since it does not need to lock anything in physical memory before starting the DMA transfer.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred during the DMA transfer.
ADMXRC3_INVALID_BUFFER_HANDLE	The hBuffer parameters is not a valid handle to a locked buffer.

Return Value	Description
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The dmaChannel parameter specifies a nonexistent DMA channel.
ADMXRC3_INVALID_LOCAL_REGION	The localAddress and length parameters specify a region of local bus address space that is invalid.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the locked buffer.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the DMA transfer.

Remarks

Before starting DMA transfer in the hardware, this function increments the reference count of the locked buffer so that it cannot be inadvertently unlocked if a badly-behaved application calls ADMXRC3_Unlock before the DMA transfer finishes. When the DMA transfer finishes, the locked buffer's reference count is decremented.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.76 ADMXRC3_WriteFlash

Declaration

```
ADMXRC3_STATUS
ADMXRC3_WriteFlash(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  unsigned int    flashIndex,
    __in  uint32_t        flags,
    __in  size_t          address,
    __in  size_t          length,
    __in  const void*     pBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device containing the Flash memory bank to be written.

flashIndex (in)

Specifies which Flash memory bank in the device is to be written.

flags (in)

Flags that modify how the operation is performed. Currently the following flags are defined, which may be bitwise ORed together:

- ADMXRC3_FLASH_SYNC
  Causes the cache for the Flash memory bank to be synchronized with the hardware before the function returns. For a description of the caching mechanism, refer to Section 3.8.6.1, "Flash memory caching"

address (in)

The byte address in the Flash memory bank at which to begin writing.

length (in)

Number of bytes to write to the Flash memory bank.

pBuffer (in)

The buffer that contains the data to be written to the the Flash memory bank.

Description

This function writes a block of data from a buffer into a Flash memory bank in a device. The data transfer is performed by the CPU, so is CPU intensive for large blocks.

Depending on the model, not all locations in Flash memory bank may be writable using this function. This is required in order to prevent inadvertent corruption of VPD, firmware etc. on some models. To determine the region of a Flash memory bank that is modifiable, an application calls ADMXRC3_GetFlashInfo. Attempts to modify any location outside of the modifiable region will fail.

The region of the Flash memory bank that is written, which is specified by the address and length parameters, can be of arbitrary alignment and does not (for example) need to be aligned to block boundaries. The region must be inside the user-programmable region, the bounds of which can be determined by calling ADMXRC3_GetFlashInfo.

The ADMXRC3 API implements a caching mechanism for each Flash memory bank. Passing the ADMXRC3_FLASH_SYNC flag ensures that the cache for the specified Flash memory bank is synchronized with the hardware before the function returns. An alternative way of ensuring synchronization is to call ADMXRC3_SyncFlash.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred while writing to the Flash memory bank.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_INDEX	The flashIndex parameter specifies a nonexistent Flash memory bank.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the Flash memory bank.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the Flash write operation.

Remarks

Although the region specified by the address and length parameters need not bear any relationship to a Flash

block boundaries, an application can call ADMXRC3_GetFlashBlockInfo in order to determine which block contains a particular address. An application can enumerate every block in a Flash memory bank by beginning at address 0 and repeatedly calling ADMXRC3_GetFlashBlockInfo until the end of the bank is reached.

For Flash memory banks that use chips with block-oriented architectures, calling this function may result in noticeable delays in execution because of the caching mechanism and block erase and programming delays.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

4.3.77 ADMXRC3_WriteVPD

Declaration

```
ADMXRC3_STATUS
ADMXRC3_WriteVPD(
    __in  ADMXRC3_HANDLE  hDevice,
    __in  uint32_t        flags,
    __in  size_t          offset,
    __in  size_t          length,
    __in  const void*     pBuffer);
```

The parameter(s) of this function are as follows:

hDevice (in)

Identifies the device whose VPD is to be written.

flags (in)

Flags that modify how the operation is performed. Currently no flags are defined, so this parameter must be zero.

offset (in)

The byte offset into the VPD memory at which to begin writing.

length (in)

Number of bytes to write to the VPD memory.

pBuffer (in)

The buffer that contains the data to be written to the VPD memory.

Description

This function writes a block of Vital Product Data (VPD) to a device. The data transfer is performed by the CPU, so is CPU intensive for large blocks. For an overview, refer to Section 3.8.7, "Vital Product Data".

VPD memory area is normally read-only and protected by a write-protection mechanism. The VPD write-protection mechanism is operating-system dependent; refer to the release notes for the ADB3 driver specific to your operating system for details.

Return Value

A value of ADMXRC3_SUCCESS indicates that the function executed successfully. Otherwise, if an error occurs, the following values may be returned:

Return Value	Description
ADMXRC3_ACCESS_DENIED	The device handle was opened with insufficient privileges for modifying device state.
ADMXRC3_CANCELLED	During the operation, another thread called ADMXRC3_Cancel on the device handle, or the device handle was closed.
ADMXRC3_HARDWARE_ERROR	A hardware error occurred while writing to the VPD memory. This may indicate that the write-protection mechanism has not been disabled.
ADMXRC3_INVALID_BUFFER	The pBuffer and length parameters represent a buffer that is not valid in the caller's address space.
ADMXRC3_INVALID_FLAG	The flags parameter contains an unrecognized flag.
ADMXRC3_INVALID_HANDLE	The hDevice parameter was not a valid device handle.
ADMXRC3_INVALID_REGION	The offset and length parameters represent a region that exceeds the bounds of the VPD memory.
ADMXRC3_NO_MEMORY	A control block could not be allocated for keeping track of the VPD write operation.

Remarks

VPD may be stored in a Flash device, an EEPROM device, or something else, depending on the model. For some types of nonvolatile memory that have block-oriented architectures, calling this function may result in noticeable delays in execution due to block erase and block programming delays.

This function does not perform any endian-conversion on the data. Refer to Section 3.6, "Endian issues" for a discussion of how to make an application portable between big- and little-endian CPU architectures.

Page Intentionally left blank

Appendix A: Duplicating device handles

Most operating systems provide a means of duplicating a file handle. Windows has **DuplicateHandle**, whilst Linux and VxWorks have the **dup** and **dup2** system calls. These functions are not suitable as a means of duplicating a device handle of type ADMXRC3_HANDLE. The reason is that at the kernel level, a device driver is unable to distinguish between the duplicated handle and the original one. The following example illustrates this.

First, a Linux or VxWorks application opens a device using ADMXRC3_Open. After the call returns, the device handle can be envisaged as follows:

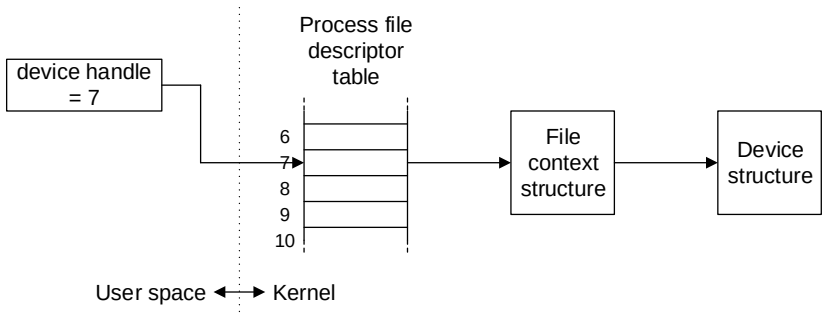


Figure 12 : An open device handle

Next, the application duplicates the handle using **dup**. After the call returns, the device handles can be envisaged as follows:

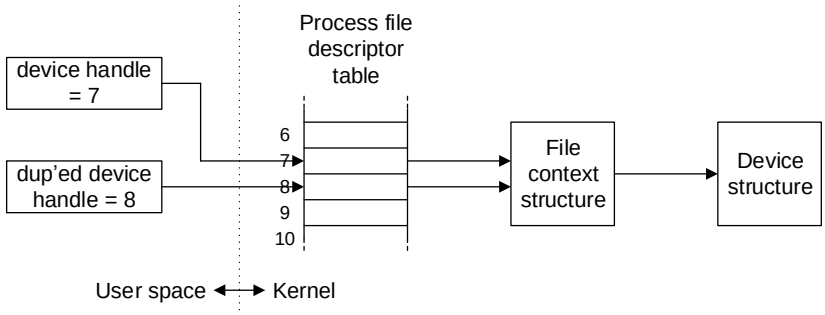


Figure 13 : After duplicating a device handle

This should be contrasted with the result of calling ADMXRC3_Open again instead of **dup**:

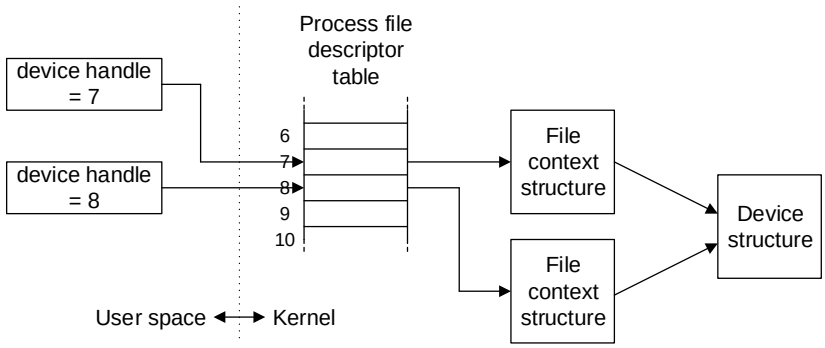


Figure 14 : After opening a device twice

In Windows the terminology is different, but the same principle applies. The device driver maintains per-device-handle state in the file context structure, so it is essential for ADMXRC3_Open or ADMXRC3_OpenEx to be used in order for the driver to be able to distinguish between two device handles. This is important for non-blocking operations, as described in Section 3.3, "Non-blocking operations".

Revision History

Date	Revision	Nature of Change
14 April 2010	1.0	Initial version.
9 July 2010	1.1	General corrections and clarifications. Added commonly-encountered error codes for ADMXRC API functions. Documented changes in ADMXRC3 API version 1.1.0. Added subsection about string encoding issues and changed references to ASCII strings to ANSI / UTF-8. Corrected NumMemoryBank and NumTargetFpga shown in the wrong order in the section for ADMXRC3_CARD_INFO.
21 September 2010	1.2	Added VxWorks-specific API functions ADMXRC3_RegisterVxwSem and ADMXRC3_UnregisterVxwSem Clarified information about linking VxWorks applications that use the ADMXRC3 API.
4 March 2011	1.3	Added ADMXRC3_*DMA*Ex functions. Documented changes in ADMXRC3 API version 1.2.0. Corrected datatype for localAddress parameters of non-Ex DMA functions; was incorrectly documented as uint64_t but is actually uint32_t. Reorganized the sections about hardware features and added figures illustrating memory windows and methods of data transfer.
24 June 2011	1.4	Documented changes in ADMXRC3 API version 1.3.0. Added new enum values for models ADM-XRC-6TGE and ADM-XRC-6T-ADV8. Added new value ADMXRC3_NOT_SUPPORTED for ADMXRC3_STATUS enumerated type.
5 August 2011	1.5	Documented changes in ADMXRC3 API version 1.4.0: Added new API functions ADMXRC3_ReadDMABus, ADMXRC3_StartReadMABus, ADMXRC3_StartWriteDMABus and ADMXRC3_WriteDMABus. Added new flag ADMXRC3_FPGA_NOTCONFIGURABLE. Added new value ADMXRC3_UNIT_S to ADMXRC3_UNIT_TYPE enumerated type.
13 March 2012	1.6	Added the missing flag ADMXRC3_CONFIGURE_NOCHECK to the descriptions of the ADMXRC3_Configure* API functions; this flag has been useable in all versions of the ADMXRC3 API, but was erroneously omitted from the documentation until now. Documented changes in ADMXRC3 API version 1.5.0: Added new API functions ADMXRC3_GetCommonBuffer, ADMXRC3_GetCommonBufferCount, ADMXRC3_MapCommonBuffer and ADMXRC3_UnmapCommonBuffer. Corrected the text "The pVirtualBase parameter is recognized as ..." to read "The pVirtualBase parameter is not recognized as ..." in the description of ADMXRC3_UnmapWindow.

Date	Revision	Nature of Change
8 August 2012	1.7	Documented changes in ADMXRC3 API version 1.5.1: Added new models to ADMXRC3_MODEL_TYPE: ADPE-XRC-6T-ADV, ADM-XRC-6T-DA1, ADPE-XRC-6T-ADV (Controller & Target), ADM-XRC-7K1, ADM-XRC-7V1. Added new FPGA families to ADMXRC3_FAMILY_TYPE & _ADMXRC3_SUBFAMILY_TYPE, along with new devices to ADMXRC3_FPGA_TYPE, in order to support Xilinx Series 7 FPGAs.
18 Feb 2013	1.8	Documented changes in ADMXRC3 API version 1.6.0: Added new model to ADMXRC3_MODEL_TYPE: ADM-VPX3-7V2. Added new API functions: ADMXRC3_GetDeviceStatus & ADMXRC3_ClearDeviceErrors. Added new structure: ADMXRC3_DEVICE_STATUS. Added new flags: ADMXRC3_DS_LOCAL_TIMEOUT, ADMXRC3_DMA_BUS_TIMEOUT, ADMXRC3_DM_BUS_TIMEOUT & ADMXRC3_DM_BAD_TRANSACTION.
24 May 2013	1.9	Documented changes in ADMXRC3 API version 1.7.0: Added new model to ADMXRC3_MODEL_TYPE: ADM-XRC-6TGEL.
3 Dec 2013	1.10	Provided more detail on initializing pollfd struct in Linux for non-blocking operations. Provided more detail on using select in VxWorks for non-blocking operations.
27 Mar 2014	1.11	Documented changes in ADMXRC3 API version 1.7.1: Added ADMXRC3_MODEL_ADMPCIE7V3 to ADMXRC3_MODEL_TYPE Documented changes in ADMXRC3 API version 1.7.2: Added models ADMXRC3_MODEL_ADMXRC7Z1 & ADMXRC3_MODEL_ADMXRC7Z2 to ADMXRC3_MODEL_TYPE. Added Zynq-7000, Kintex Ultrascale & Virtex Ultrascale devices to ADMXRC3_FPGA_TYPE. Added ADMXRC3_FAMILY_ULTRASCALE to ADMXRC3_FAMILY_TYPE. Added ADMXRC3_FAMILY_7Z, ADMXRC3_FAMILY_UK & ADMXRC3_FAMILY_UV to ADMXRC3_SUBFAMILY_TYPE.
24 Aug 2015	1.12	Added tables to section "Building C and C++ applications" that explicitly list secondary header files for ADMXRC3 API extensions such as peer-to-peer DMA functions and common buffer functions. Updated sections "DMA transfers with peer devices" and "Common Buffers" to also explicitly list secondary header files.

Date	Revision	Nature of Change
6 May 2016	1.13	Documented changes in ADMXRC3 API version 1.7.3: Added ADMXRC3_MODEL_ADMPCIEKU3 to ADMXRC3_MODEL_TYPE. Documented changes in ADMXRC3 API version 1.8.0: Added ADMXRC3_MODEL_ADMXRCKU1 to ADMXRC3_MODEL_TYPE. Added new flag ADMXRC3_CONFIGURE_IGNOREMISMATCH for functions ADMXRC3_ConfigureFromFileA & ADMXRC3_ConfigureFromFileW. Documented changes in ADMXRC3 API version 1.8.1: Added ADMXRC3_MODEL_ADMPCIE8V3 to ADMXRC3_MODEL_TYPE. Added values for Kintex, Virtex & Zynq Ultrascale+ devices to ADMXRC3_FPGA_TYPE. Added values for Kintex, Virtex & Zynq Ultrascale+ to ADMXRC3_SUBFAMILY_TYPE. Documented changes in ADMXRC3 API version 1.8.2: Added ADMXRC3_MODEL_ADMPCIE8K5 to ADMXRC3_MODEL_TYPE.

Page Intentionally left blank

Address: 4 West Silvermills Lane,
Edinburgh, EH3 5BD, UK
Telephone: +44 131 558 2600
Fax: +44 131 558 2700
email: sales@alpha-data.com
website: http://www.alpha-data.com

Address: 3507 Ringsby Court Suite 105,
Denver, CO 80216
Telephone: (303) 954 8768
Fax: (866) 820 9956 toll free
email: sales@alpha-data.com
website: http://www.alpha-data.com