Windows Device Driver and API Reference Manual

797 North Grove Rd, Suite 101
Richardson, TX 75081
Phone: (972) 671-9570
www.redrapids.com

Red Rapids reserves the right to alter product specifications or discontinue any product without notice. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment. This product is not designed, authorized, or warranted for use in a life-support system or other critical application.

All trademark and registered trademarks are the property of their respective owners.

Table of Contents

List of Figures

List of Tables

1.0 Introduction

The Windows device driver and application program interface (API) provide the vehicle to access all hardware resources of a Red Rapids PCI adapter (RRadapter) from a software application written in C. The RRadapter driver and API are shared across several different Red Rapids product families.

Features of the RRadapter driver and API include:

- Common API for 32-bit and 64-bit operating systems.
- Communication with up to four devices (more upon request) in a single system.
- Support for any number of DMA buffers with any number of pages.
- Interrupt handling for error and status indicators.

The RRadapter driver is built on the Windriver™ cross-platform development toolkit from Jungo (www.jungo.com).

1.1 Conventions

This manual uses the following conventions:

- Hexadecimal numbers are prefixed by “0x” (e.g. 0x00058C).
- *Italic* font is used for names of registers.
- Blue font is used for names of directories, files, and OS commands.
- Green font is used to designate source code.

☞	Text in this format highlights useful or important information.

!	Text shown in this format is a warning. It describes a situation that could potentially damage your equipment. Please read each warning carefully.

The following are some of the acronyms and abbreviations used in this manual.

- **-32** 32-bit Operating System
- **-64** 64-bit Operating System
- **API** Application Program Interface
- **BAR** Base Address Register
- **DLL** Dynamic Link Library
- **DMA** Direct Memory Access
- **ISR** Interrupt Service Routine
- **Rxx** Any Revision Number
- **x86** Intel Processor Architecture

1.2 Revision History

Version	Date	Description
R00	11/28/2018	Initial release.

2.0 Driver Installation

The RRadapter driver is built on the Windriver™ cross-platform toolkit from Jungo. The source code is not distributed by Red Rapids due to restrictions in the Jungo licensing agreement.

2.1 Directory Structure

The RRadapter device driver is distributed with the Windows Software Development Kit (SDK) available for each Red Rapids PCI product. The software is located under the \RRadapter directory of the SDK and contains the following subdirectory structure:

\src

The src subdirectory contains a source code template to the ISR function that is called by the RRadapter library.

\win_x86-32

The win_x86-32 subdirectory contains the RRadapter driver and API library for all 32-bit Windows operating systems hosted on an Intel processor architecture.

\win_x86-32\lib

The lib folder contains the 32-bit RRadapter driver library, RRadapter DLL, and the Jungo Windriver™ DLL. The two DLL files are required to run any executable created with the library. They must either be copied to the directory containing the executable or the lib directory must be added to the PATH environment variable so that it is included in the search at runtime.

\win_x86-32\win7

The win7 folder contains the installation batch file and Jungo Windriver™ code needed to install the RRadapter driver on a 32-bit Windows 7 operating system.

\win_x86-32\win8

The win8 folder contains the installation batch file and Jungo Windriver™ code needed to install the RRadapter driver on a 32-bit Windows 8/8.1 operating system.

\win_x86-32\win10

The win10 folder contains the installation batch file and Jungo Windriver™ code needed to install the RRadapter driver on a 32-bit Windows 10 operating system.

\win_x86-64

The win_x86-64 subdirectory contains the RRadapter driver and API library for all 64-bit Windows operating systems hosted on an Intel processor architecture.

\win_x86-64\lib

The lib folder contains the 64-bit RRadapter driver library, RRadapter DLL, and the Jungo Windriver™ DLL. The two DLL files are required to run any executable created with the library. They must either be copied to the directory containing the executable or the lib directory must be added to the PATH environment variable so that it is included in the search at runtime.

\win_x86-64\win7

The win7 folder contains the installation batch file and Jungo Windriver™ code needed to install the RRadapter driver on a 64-bit Windows 7 operating system.

\win_x86-64\win8

The win8 folder contains the installation batch file and Jungo Windriver™ code needed to install the RRadapter driver on a 64-bit Windows 8/8.1 operating system.

\win_x86-64\win10

The win10 folder contains the installation batch file and Jungo Windriver™ code needed to install the RRadapter driver on a 64-bit Windows 10 operating system.

2.2 Installation Procedure

The following procedure will load drivers onto a host computer running a supported version of the Windows operating system.

> ☞ The following procedure must be executed with Administrator privileges. You may also have to adjust the User Account Control (UAC) settings from the User Accounts option in the Control Panel to overcome security restrictions.

1. Boot the host computer with the Red Rapids hardware installed.
2. Download the product SDK from the Red Rapids website (www.redrapids.com).
3. Extract the files in the SDK zip archive to a local directory. The directory created by the SDK will be referred to as c:\<extract path>.
4. Use a Command Prompt (Admin) or PowerShell (Admin) to navigate to the working directory designated for your operating system.

   Working directory for 32-bit operating systems:

   c:\<extract path>\RRadapter \win_x86-32

   Working directory for 64-bit operating systems:

   c:\<extract path>\RRadapter \win_x86-64
5. Navigate to the subdirectory indicated for the specific version of Windows running on the host:

   Windows 7: win7

   Windows 8/8.1: win8

   Windows 10: win10
6. Execute the Install batch file from the Command Prompt or PowerShell.

> ☞ You may encounter multiple prompts asking you to confirm the installation. You might also receive a warning that the driver may not have installed correctly. These messages do not indicate a problem.

You can verify that the installation completed successfully by opening the Windows Device Manager. A folder named “Jungo” should appear as one of the device types. You should find “RRadapter” and “Red Rapids Device” listed under the Jungo folder.

3.0 Device Handle (s_Handle)

Each device is assigned a handle that contains the information necessary to communicate with the hardware. Table 3-1 lists each member of the s_Handle structure with a brief description of the information it conveys.

Table 3-1 s_Handle Structure

Member Name	Data Type	Definition
DevNum	unsigned long	Device number associated with the handle.
DEV	s_DEVMap	Structure to convey identification information about the device.
DMA	s_DMABuf	Structure to convey the desired DMA buffer configuration to the driver.
*User	void	Undefined pointer available to user applications.

The device number (DevNum) is assigned when a Red Rapids product is discovered during the driver scan of the PCI bus. The numbering begins with zero and proceeds sequentially with each additional device.

The device (DEV) structure carries PCI identification information for the specific device number. Refer to section 3.1 for further details.

The direct memory access (DMA) structure is used to assign memory buffers to devices that have DMA capability. There can be multiple buffers attached to this pointer if needed by the device. See section 0 for detailed information.

The User pointer allows software developers to attach any application specific information to the device handle. This is a convenient vehicle to pass a user defined structure to multiple functions or an interrupt service routine.

3.1 Device Identification (s_DEVMap)

The s_DEVMap structure is referenced by s_Handle to convey PCI device identifiers obtained by the driver. Table 3-2 lists each member of the s_DEVMap structure with a brief description of the information it conveys.

Table 3-2 s_DEVMap Structure

Member Name	Data Type	Definition
SubsystemID	unsigned	PCI subsystem identification assigned to the device.
UnitsFound	unsigned long	Number of Red Rapids devices detected by the driver.
Bus	unsigned long	PCI bus number reported by the driver.
Slot	unsigned long	PCI slot number reported by the driver.
Function	unsigned long	PCI function number reported by the driver.

Every PCI device includes a subsystem identification (SubsystemID) value embedded in the Configuration Space Header. This information helps distinguish between different devices from the same vendor when multiple products are connected to the bus.

The remaining device information is not typically needed by application software but is included for completeness. The driver scans the bus for any Red Rapids product and records the number of units found (UnitsFound). The physical location of every device on a PCI bus is identified by the Bus and Slot numbers. Each physical slot can contain more than one logical device which are identified by Function numbers.

3.2 DMA Buffer Configuration (s_DMABuf)

The s_DMABuf structure is referenced by s_Handle to convey DMA configuration information through the handle. A separate structure is assigned to each DMA buffer requested by a device and most Red Rapids products require multiple buffers. The buffers are accessed as an array, starting from zero. Each DMA buffer may consist of multiple pages as described in the *DMA on Demand Operating Guide*.

Table 3-3 lists each member of the s_DMABuf structure with a brief description of the information it conveys.

Table 3-3 s_DMABuf Structure

Member Name	Data Type	Definition
PageNumber	unsigned	Number of the active DMA page.
PageCount	unsigned	Number of DMA pages assigned to the buffer.
PagesPerMark	unsigned	Number of DMA pages to transfer between completion marker flags.
BurstSizeB	unsigned	Size of each individual DMA burst transaction in bytes.
BurstCount	unsigned	Number of bursts to store in each DMA page.
**Address	unsigned long long	Virtual address of each DMA page that is allocated.

Each page of a DMA buffer is assigned a unique identifying number when it is allocated by the driver. The numbering starts at zero for the first page and increments by one for each subsequent page. The page number (PageNumber) is used to access the virtual address (*Address) for a specific DMA page.

The page count (PageCount) conveys the total number of DMA pages assigned to the buffer.

Status registers or interrupts can be used to notify an application that a DMA buffer has just been accessed by the device. It is convenient to set the frequency of these markers to some number of pages, defined as pages per mark (PagesPerMark). For instance, a signal acquisition product may stream raw data samples at a very high rate through a single DMA channel. There may be several hundred pages of DMA buffer storage assigned to that channel, but the application may want to be notified of updates only every hundred pages to throttle interrupt frequency.

The driver sizes each DMA page in relation to the specified burst size (BurstSizeB) and burst count (BurstCount) as shown in Figure 3-1. The burst size is the number of bytes that will be transferred in a single DMA transaction. The burst count is the number of

transactions that can be performed on a single page. Consequently, the page size in bytes is simply the burst size multiplied by the burst count. Each DMA page must occupy contiguous address space in host memory. It becomes more difficult for the operating system to allocate large pages as other functions consume resources and fragment the memory.

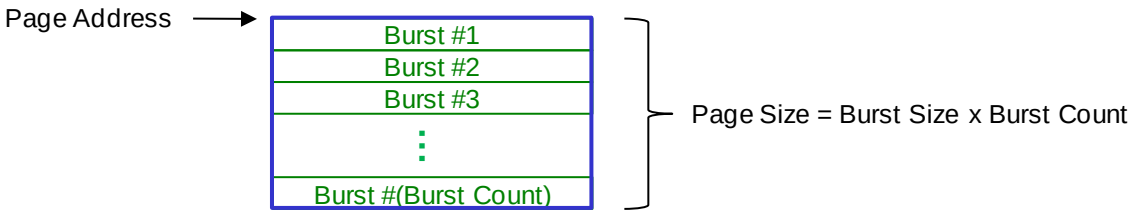

Figure 3-1 DMA Page Size Parameters

There is both a physical address and virtual address associated with each DMA page allocated by the driver. The physical address is needed to access the buffer from the hardware device. The virtual address is used to access the same buffer through the application software. The address pointer (**Address) is used to store the array of virtual addresses assigned to each page of the buffer.

4.0 Application Program Interface (API)

The Adapter API consists of a library of functions that provide access to the basic features of the driver. There are five primary operations performed through the API:

1. Open and close the hardware device.
2. Read and write hardware register or memory contents.
3. Allocate and free DMA buffers.
4. Set a timer to pause execution.
5. Service hardware interrupts.

The API is pre-compiled and distributed as a dynamic link library (RRadapter.dll) and a common header file (RRadapter.h).

4.1 Open/Close Functions

Prototype	int DriverName(const char* Name);
Arguments	Name: Name of the driver, usually “RRadapter”.
Description	Set the name of the driver kernel module. This function must be called only once before attempting to communicate with the driver.
Return	0 if successful Non-zero if error

Prototype	int HandleInit(s_Handle *p_Adapter);
Arguments	*p_Adapter: Pointer to the device handle.
Description	Initialize all members of the device handle.
Return	0 if successful Non-zero if error

Prototype	int BusOpen(s_Handle *p_Adapter);
Arguments	*p_Adapter: Pointer to the device handle.
Description	Open an instance of the device handle and store information about the mapped unit in the structure.
Return	0 if successful Non-zero if error

Prototype	int BusClose(s_Handle *p_Adapter);
Arguments	*p_Adapter: Pointer to the device handle.
Description	Close an open device handle instance and de-allocate any system memory allocated to the handle.
Return	0 if successful Non-zero if error

4.2 Read/Write Functions

Prototype	int Read8(s_Handle *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned char *pData);
Arguments	*p_Adapter: Pointer to the device handle. BAR: Base address of the register or memory location. Offset: Address offset from the BAR. *pData: Pointer to the data value returned.
Description	Perform an 8-bit read operation to a register or memory location at the specified BAR and offset.
Return	0 if successful Non-zero if error

Prototype	int Read32(s_Handle *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned *pData);
Arguments	*p_Adapter: Pointer to the device handle. BAR: Base address of the register or memory location. Offset: Address offset from the BAR. *pData: Pointer to the data value returned.
Description	Perform a 32-bit read operation to a register or memory location at the specified BAR and offset.
Return	0 if successful Non-zero if error

Prototype	int Read64(s_Handle *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned long long *pData);
Arguments	*p_Adapter: Pointer to the device handle. BAR: Base address of the register or memory location. Offset: Address offset from the BAR. *pData: Pointer to the data value returned.
Description	Perform a 64-bit read operation to a register or memory location at the specified BAR and offset.
Return	0 if successful Non-zero if error

Prototype	int Write8(s_Handle *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned char Data);
Arguments	*p_Adapter: Pointer to the device handle. BAR: Base address of the register or memory location. Offset: Address offset from the BAR. Data: Data value to write.
Description	Perform an 8-bit write operation to a register or memory location at the specified BAR and offset.
Return	0 if successful Non-zero if error

Prototype	int Write32(s_Handle *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned Data);
Arguments	*p_Adapter: Pointer to the device handle. BAR: Base address of the register or memory location. Offset: Address offset from the BAR. Data: Data value to write.
Description	Perform a 32-bit write operation to a register or memory location at the specified BAR and offset.
Return	0 if successful Non-zero if error

Prototype	int Write64(s_Handle *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned long long Data);
Arguments	*p_Adapter: Pointer to the device handle. BAR: Base address of the register or memory location. Offset: Address offset from the BAR. Data: Data value to write.
Description	Perform a 64-bit write operation to a register or memory location at the specified BAR and offset.
Return	0 if successful Non-zero if error

4.3 DMA Buffer Functions

Prototype	int DMABufAllocate(s_Handle *p_Adapter);
Arguments	*p_Adapter: Pointer to the device handle.
Description	Allocate DMA buffers as specified in the s_DMABuf structure of the device handle.
Return	0 if successful Non-zero if error

Prototype	int DMABufFree(s_Handle *p_Adapter);
Arguments	*p_Adapter: Pointer to the device handle.
Description	Deallocate DMA buffers as specified in the s_DMABuf structure of the device handle.
Return	0 if successful Non-zero if error

4.1 TImer Function

Prototype	int MicrosecondTimer(unsigned long uSec);
Arguments	uSeconds: Number of microseconds to pause.
Description	Pause execution for the specified time in microseconds.
Return	0 if successful Non-zero if error

4.2 Interrupt Service Routine Function

The following function must be included in all applications that call the Adapter API:

void RRAdapter_ISR(s_Handle *p_Adapter);

The RRAdapter_ISR() function creates a dynamic link library (DLL) that is called by the Adapter API. A blank stub can be used for any application that does not need to process interrupts. This stub is provided in the src directory of the driver distribution.

The RRAdapter_ISR() function is used to service interrupts. Each Red Rapids device with an open handle is associated with an interrupt handler in the driver. When the driver detects an interrupt, it first determines whether it originated from a Red Rapids product.

The RRAdapter_ISR() is called only in the application that has an open handle to the hardware producing that interrupt. The device number is passed to the ISR through the handle so that the specific device generating the interrupt can be identified. This is important in applications that have multiple Red Rapids devices open.

The interrupt handler in the API will disable further interrupts from any device that has an interrupt pending service. This is accomplished by setting the appropriate bits in the global interrupt mask register of that device. The RRAdapter_ISR() function must unmask the global interrupt just before exiting to reverse this action and allow the device to generate interrupts again.

The user application code must service any interrupt status registers and interrupt mask registers from within RRAdapter_ISR(). A typical procedure might involve masking bits associated with persistent error conditions, reading the status register, taking corrective action, then unmasking bits that have been cleared. The specific actions required to service an interrupt will vary by application.