Adapter 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 Adapter device driver and application program interface (API) provide the vehicle to access all hardware resources of a Red Rapids PCI adapter product from a software application written in C. This driver and API are shared across several different Red Rapids product families. The term “PCI adapter” applies to any product that includes a PCI bus as the interface to a host computer.

Features of the Adapter driver and API include:

- Unified Windows and Linux driver structure for 32-bit or 64-bit architectures.
- Support for up to eight cards in a single system.
- Support for up to four DMA channels with up to 1024 pages per channel.
- Interrupt handling for error and status indicators.

1.1 Contents and Structure

This manual includes installation instructions and a description of the features offered by the Adapter device driver and API software distribution. This publication supplements other hardware and software documents that are available for specific Red Rapids PCI adapter products.

Section	Description
Section 1	Introductory information about the manual.
Section 2	Driver installation instructions for both Windows and Linux.
Section 3	Description of the device structure associated with an Adapter handle.
Section 4	Description of the functions provided in the Adapter API.
Section 5	Application example to demonstrate the Adapter driver and API usage.

The latest product documentation and software is available for download from the Red Rapids web site (www.redrapids.com) by following links on the product page.

1.2 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

1.3 Distribution Disk (DSK-806-902-Rxx)

The Adapter device driver and API software is distributed on Red Rapids disk archive number DSK-806-902-Rxx. The directory structure of the archive is outlined below:

\example

The example subdirectory contains source code to a simple application called ScanBus that searches for Red Rapids devices attached to the host. Executable for the application can be found in the subdirectory corresponding to the specific operating system.

\include

The include subdirectory contains the header file used to access the Adapter API.

\linux_x86-32

The linux_x86-32 subdirectory contains the Adapter driver and API library for all 32-bit Linux distributions hosted on an Intel processor architecture. The driver installation shell script is available in this directory.

\linux_x86-64

The linux_x86-64 subdirectory contains the Adapter driver and API library for all 64-bit Linux distributions hosted on an Intel processor architecture. The driver installation shell script is available in this directory.

\src

The src subdirectory contains a source code template to the ISR function that is called by the Adapter library. The Adapter library is compiled using a third party driver development toolkit from Jungo. The source code is not distributed by Red Rapids due to restrictions in the Jungo licensing agreement.

\win_x86-32

The win_x86-32 subdirectory contains the Adapter driver and API library for all 32-bit Windows operating systems hosted on an Intel processor architecture. The driver installation batch file is available in this directory.

\win_x86-64

The win_x86-64 subdirectory contains the Adapter driver and API library for all 64-bit Windows operating systems hosted on an Intel processor architecture. The driver installation batch file is available in this directory.

1.4 Revision History

Version	Date	Description
R00	7/5/2011	Initial release.

2.0 Driver Installation

The Adapter driver is built on a third party cross-platform toolkit from Jungo. The Jungo WinDriver™ toolkit provides a unified Windows and Linux driver structure so that application code can move seamlessly between operating systems.

> ☞ The Adapter source code is not distributed by Red Rapids due to restrictions in the Jungo licensing agreement.

2.1 Windows Installation

> ☞ 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.

The following procedure will load drivers onto a host computer running any Windows operating system.

1. Boot the host computer without the Red Rapids hardware installed.
2. Download the DSK-806-902-Rxx distribution disk from the Red Rapids website (www.redrapids.com).
3. Extract the files in the zip archive to a local directory. The local directory will be referred to as c:\<extract path>.
4. Use Windows Explorer or a Command Prompt to navigate to the working directory designated for your operating system.

   Working directory for 32-bit operating systems:

   c:\<extract path>\DSK-806-902-Rxx\win_x86-32

   Working directory for 32-bit operating systems:

   c:\<extract path>\DSK-806-902-Rxx\win_x86-64
5. Execute the Install batch file by double clicking the file name from Windows Explorer or entering the file name at the Command Prompt.

> ☞ 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.

6. Shut down the computer and install the Red Rapids hardware as outlined in the product installation guide.
7. Boot the host computer with the Red Rapids hardware installed.
8. 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 “WinDriver” and “Red Rapids Adapter” listed under the Jungo folder.

2.2 Linux Installation

> ☞ The following procedure must be executed with root privileges.

The following procedure will load drivers onto a host system running the Linux operating system.

1. Boot the host computer without the Red Rapids hardware installed.
2. Download the DSK-806-902-Rxx distribution disk from the Red Rapids website (www.redrapids.com).
3. Extract the files in the zip archive to a local directory. The local directory will be referred to as /<extract path>.
4. Use a Linux terminal to navigate to the working directory designated for your operating system.

   Working directory for 32-bit operating systems:

   cd /<extract path>/DSK-806-902-Rxx/linux_x86-32

   Working directory for 32-bit operating systems:

   cd /<extract path>/DSK-806-902-Rxx/linux_x86-64
5. Execute the Install shell script from the terminal (./Install).

> ☞ You must install the gcc compiler and the kernel-devel package for your Linux distribution prior to loading the driver.

6. Edit the /etc/rc.local file to include commands that load the kernel module and set user permissions each time the computer reboots.

   Note: The Rxx notation below should be substituted for the current revision number of the distribution disk archive.

   Add the following two lines for 32-bit operating systems:

   /<extract path>/DSK-806-902-Rxx/linux_x86-32/redist/wdreg windrvr6 auto

   chmod 666 /dev/windrvr6

   Add the following two lines for 64-bit operating systems:

   /<extract path>/DSK-806-902-Rxx/linux_x86-64/redist/wdreg windrvr6 auto

   chmod 666 /dev/windrvr6
7. Shut down the computer and install the Red Rapids hardware as outlined in the product installation guide.
8. Boot the host computer with the Red Rapids hardware installed.

9. You can verify that the installation completed successfully by executing the following commands:

   lspci | grep 17d2

   lsmod | grep windrvr6

   The lspci command will list all Red Rapids hardware (Vendor ID 17d2) that was detected during enumeration of the PCI bus. The lsmod command will list the windrvr6 kernel module.

3.0 Device Handle (s_Adapter)

Application software communicates with each device through a handle. The device structure associated with the handle is named s_Adapter. Table 3-1 lists each member of the s_Adapter structure with a brief description of the information it conveys.

Table 3-1 s_Adapter Structure

Member Name	Data Type	Definition
DevNum	unsigned long	PCI bus device number associated with the handle. (Starts at zero.)
Bus	unsigned long	PCI bus number reported by the driver. (Starts at zero.)
Slot	unsigned long	PCI slot number reported by the driver. (Starts at zero.)
Function	unsigned long	PCI function number reported by the driver. (Starts at zero.)
UnitsFound	unsigned long	Number of Red Rapids products detected by the driver.
DMA	s_DMAConfig	Structure that conveys the desired DMA buffer configuration to the driver.
GlobalIntMask	s_INTConfig	Structure that conveys global interrupt mask information to the driver.
*User	void	Undefined pointer available to user applications.
Verbose	int	Reserved for status message reporting.

3.1 DMA Buffer Configuration (s_DMAConfig)

The s_DMAConfig structure is referenced by s_Adapter to convey DMA configuration information through the handle. The driver requests a contiguous block of memory from the operating system to create a DMA buffer. This block of memory is considered a single DMA page. The desired size of each page must be supplied to the driver through this structure.

There is both a physical DMA buffer address and virtual DMA buffer address returned by the driver. The physical address is needed by the hardware to access the buffer in host memory. The virtual address is used to access the same buffer through the application software.

Table 3-2 lists each member of the s_DMAConfig structure with a brief description of the information it conveys.

Table 3-2 s_DMAConfig Structure

Member Name	Data Type	Definition
ChannelIndex	unsigned long	Number representing the specific DMA channel to configure. (Starts at zero.) The driver supports up to four independent channels.
BurstSizeB[channel]	unsigned long	Size of each individual DMA burst transaction in bytes.
BurstCount[channel]	unsigned long	Number of bursts to store in each DMA page.
PageCount[channel]	unsigned long	Number of individual DMA pages to allocate. The API supports up to 1024 pages per channel.
PagesPerMark[channel]	unsigned long	Specifies the number of DMA pages the channel will service before the hardware issues a marker (status bit, interrupt, etc.).
*VirtDMAAdr[channel][page]	void	Virtual address of each DMA page that is allocated.
PhysDMAAdr[channel][page]	unsigned long long	Physical address of each DMA page that is allocated.

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 an individual DMA burst transaction. The burst count is the number of these 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 is 64-bit aligned, so the burst size must be a multiple of eight bytes.

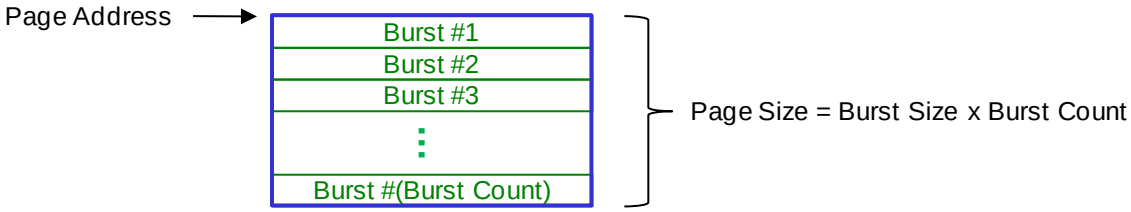

Figure 3-1 DMA Page Size Parameters

Each DMA page must occupy contiguous address space in host memory. It becomes more difficult for the operating system to allocate this space as other functions consume resources and fragment the memory. Some operating systems also place a limit on the maximum contiguous address space that can be allocated. These limitations can be overcome by allocating multiple pages per DMA channel (PageCount).

Status bits or interrupts can be used to notify an application that a DMA buffer has just been serviced by the hardware. It is convenient to set the frequency of these markers to a specified number of pages (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 every few pages.

3.2 Global Interrupt Mask Configuration (s_INTConfig)

The s_INTConfig structure is referenced by s_Adapter to convey information about the global interrupt mask through the handle. Interrupts are frequently used to report status or error conditions by the hardware. An interrupt handler manages this traffic by suppressing further interrupts from any device that already has an interrupt pending service. This is accomplished by setting the appropriate bits in the global interrupt mask register of that device.

Table 3-2 lists each member of the s_INTConfig structure with a brief description of the information it conveys.

Table 3-3 s_INTConfig Structure

Member Name	Data Type	Definition
BAR	unsigned long	Base address register (BAR) containing the global interrupt mask.
Offset	unsigned long	Address offset from the BAR that uniquely identifies the global interrupt mask register.
Disable	unsigned long long	Binary value that should be written to the global interrupt mask register to disable all interrupt generation by the hardware.

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 four 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. Service hardware interrupts.

The API also includes two utility functions to initialize the handle structure and to pause program execution for a specified period of time.

The API is pre-compiled and distributed as a dynamic link library (DLL) for Windows (RRadapter.dll) and a shared library for Linux (libRRadapter.so). The libraries are accompanied by a common header file (adapter.h).

The Windows path environment variable must include the directory containing the RRadapter.dll and wdapixxxx.dll to run any application linked to the API. These files are located in the win_x86-32\lib and win_x86-64\lib directories on the distribution disk.

4.1 Open/Close Functions

Prototype	unsigned long Adapter_Open(s_Adapter *p_Adapter);
Arguments	s_Adapter *p_Adapter – Pointer to the s_Adapter handle.
Description	Opens an instance of the s_Adapter handle and stores information about the mapped unit in the structure.
Return	0 if successful Non-zero if error

Prototype	unsigned long Adapter_Close(s_Adapter *p_Adapter);
Arguments	s_Adapter *p_Adapter – Pointer to the s_Adapter handle.
Description	Closes an open s_Adapter handle instance and de-allocates any system memory allocated to the handle.
Return	0 if successful Non-zero if error

4.2 Read/Write Functions

Prototype	unsigned long Adapter_Read64(s_Adapter *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned long long *pData);
Arguments	s_Adapter *p_Adapter – Pointer to the s_Adapter handle. BAR – Base address of the register or memory location. unsigned long Offset – Address offset from the BAR. unsigned long long *pData – Pointer to the data value read.
Description	This function performs a read operation to a register or memory location at the specified BAR and offset.
Return	0 if successful Non-zero if error

Prototype	unsigned long Adapter_Write64(s_Adapter *p_Adapter, unsigned long BAR, unsigned long Offset, unsigned long long Data);
Arguments	s_Adapter *p_Adapter – Pointer to the s_Adapter handle. BAR – Base address of the register or memory location. unsigned long Offset – Address offset from the BAR. unsigned long long Data – Data value to write.
Description	This function performs a 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	unsigned long Adapter_DMABufAllocate(s_Adapter *p_Adapter);
Arguments	s_Adapter *p_Adapter – Pointer to the s_Adapter handle.
Description	Allocates DMA buffers as specified in the s_DMAConfig structure of the s_Adapter handle.
Return	0 if successful Non-zero if error

Prototype	unsigned long Adapter_DMABufFree(s_Adapter *p_Adapter);
Arguments	s_Adapter *p_Adapter – Pointer to the s_Adapter handle.
Description	Deallocates DMA buffers as specified in the s_DMAConfig structure of the s_Adapter handle.
Return	0 if successful Non-zero if error

4.4 Interrupt Service Routine Function

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

void RRAdapter_ISR(s_Adapter *p_Adapter);

A blank stub can be used if the application software does not need to process interrupts. This stub is provided in the src directory of the distribution disk.

The RRAdapter_ISR() function creates a dynamic link library (DLL) that is called by the Adapter API in Windows. The function can be compiled and linked just as any other source in Linux.

> ☞ The Windows path environment variable must include the directory containing the RRadapter_ISR.dll to run any application linked to the API.

The RRAdapter_ISR() function is used to service interrupts. Each Red Rapids card with an open handle is associated with an interrupt handler in the API. When the API 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 already 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.

4.5 Utility Functions

Prototype	unsigned long Adapter_Zero(s_Adapter *p_Adapter);
Arguments	s_Adapter *p_Adapter – Pointer to the s_Adapter handle.
Description	Initializes all members of the s_Adapter handle.
Return	0 if successful Non-zero if error

Prototype	unsigned long Adapter_uSleep(unsigned long uSec);
Arguments	Unsigned long uSec – Number of microseconds to sleep.
Description	Causes the processor to perform a busy sleep.
Return	0 if successful Non-zero if error

5.0 ScanBus Application Example

The ScanBus application example provides a simple demonstration of the Adapter driver and API. The application searches the host computer for any Red Rapids PCI adapter products and reports the results to standard output.

The ScanBus application demonstrates several aspects of the driver and API:

- Single code base that can be compiled for Windows or Linux.
- Batch files to compile an application with Visual Studio (Windows).
- Make files to compile an application with gcc (Linux).
- Single code base that can be compiled for 32-bit or 64-bit operating systems.

The ScanBus source code can also be used as a template for more complex programs. Every application will have to open and close a Red Rapids device to configure the hardware and pass data.

5.1 Windows

There are two batch files provided to compile and link the ScanBus application using Visual Studio:

VSbuild-32.bat - Batch file for Windows 32-bit operating systems. The output files will be located in subdirectory win_x86-32.

VSbuild-64.bat - Batch file for Windows 64-bit operating systems. The output files will be located in subdirectory win_x86-64.

These batch files can be executed from any Visual Studio command prompt. Make sure to select the “x64” command prompt for 64-bit operating systems. See Figure 5-1 for a screen shot of the batch file execution in a 64-bit environment.

The batch files execute three separate Visual Studio commands to perform the required compile and link tasks:

(1) Compile the application source code (scanbus.c) and the required interrupt service routine source code (RRadapter_ISR.c).

Note: The interrupt service routine is simply a blank stub since this particular application does not generate interrupts.

(2) Link the required interrupt service routine DLL (RRadapter_ISR.dll).

(3) Link the ScanBus application executable (scanbus.exe).

The ScanBus application can be executed from any Windows command prompt, but the Adapter driver DLL directory must first be included in the path environment variable. Another batch file (SetPath.bat) is provided to easily set this variable from the command prompt. A typical command prompt session is illustrated in Figure 5-2. There was a single Red Rapids product installed in the computer when the application was executed.

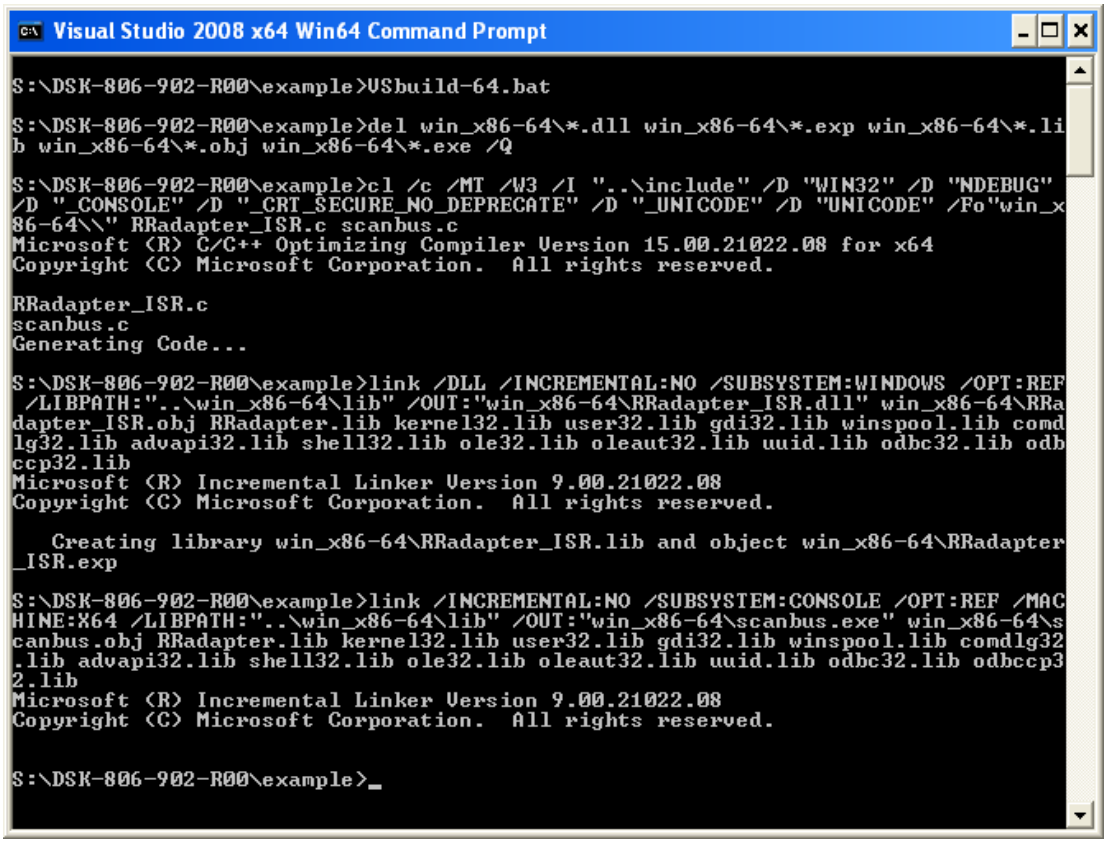

Figure 5-1 ScanBus Compile and Link in 64-bit Windows

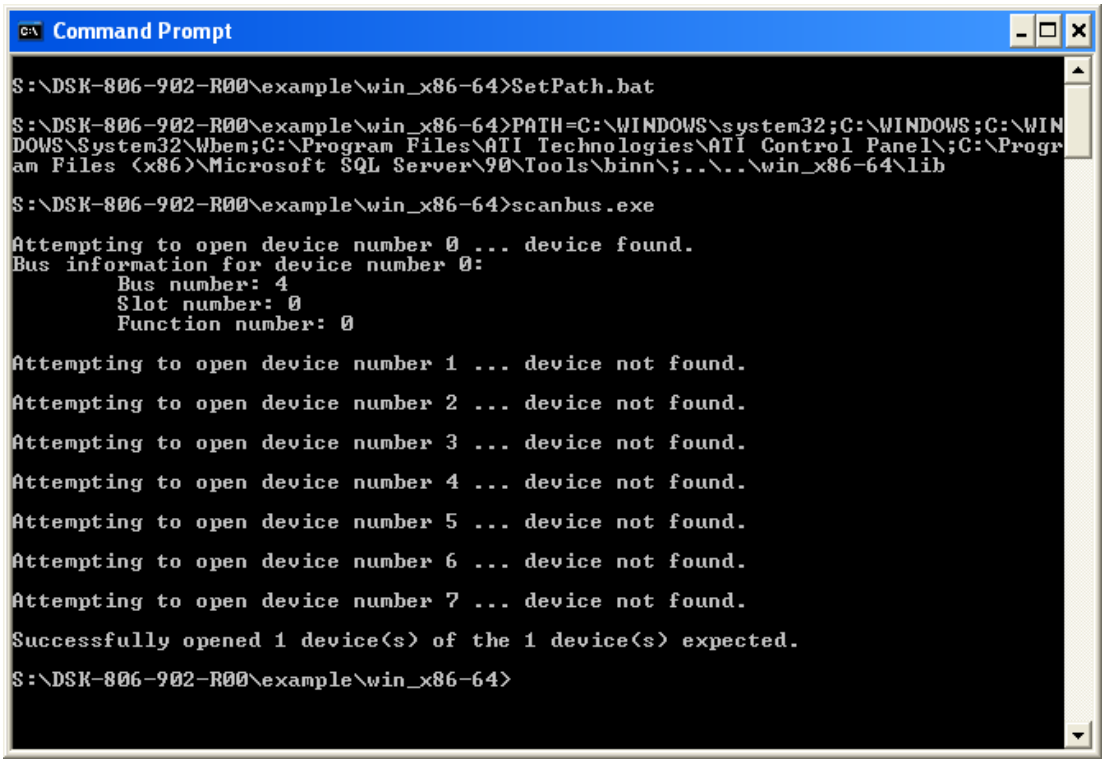

Figure 5-2 ScanBus Execution in Windows

5.2 Linux

A single Makefile is provided to compile and link the ScanBus application using gcc with either a 32-bit or 64-bit Linux distribution. The output files will be located in subdirectory linux_x86-32 or linux_x86-64, depending on the environment. See Figure 5-3 for a screen shot of the Makefile executed on a CentOS 64-bit distribution.

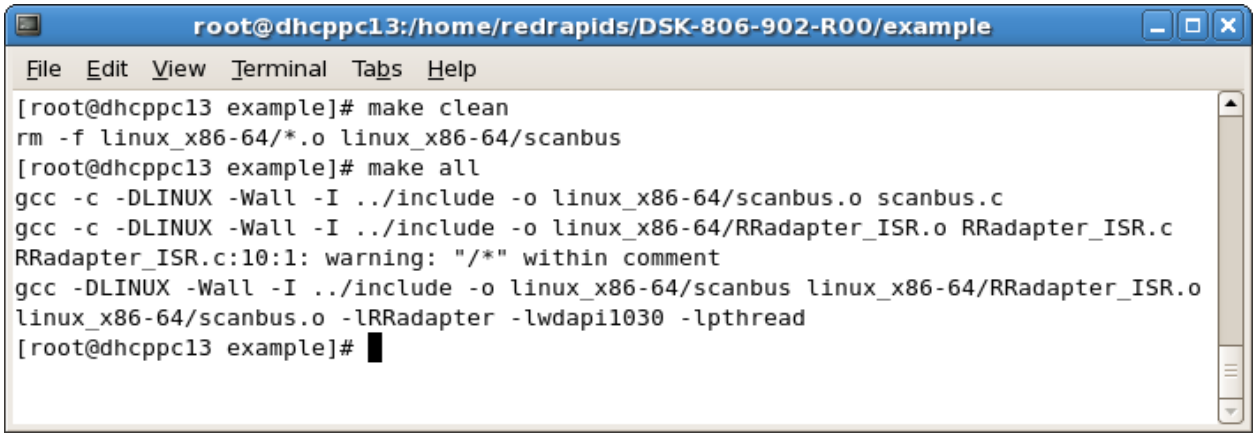

Figure 5-3 ScanBus Makefile Execution in 64-bit Linux

The ScanBus application can be executed from any Linux terminal. A typical terminal session is illustrated in Figure 5-4. There was a single Red Rapids product installed in the computer when the application was executed.

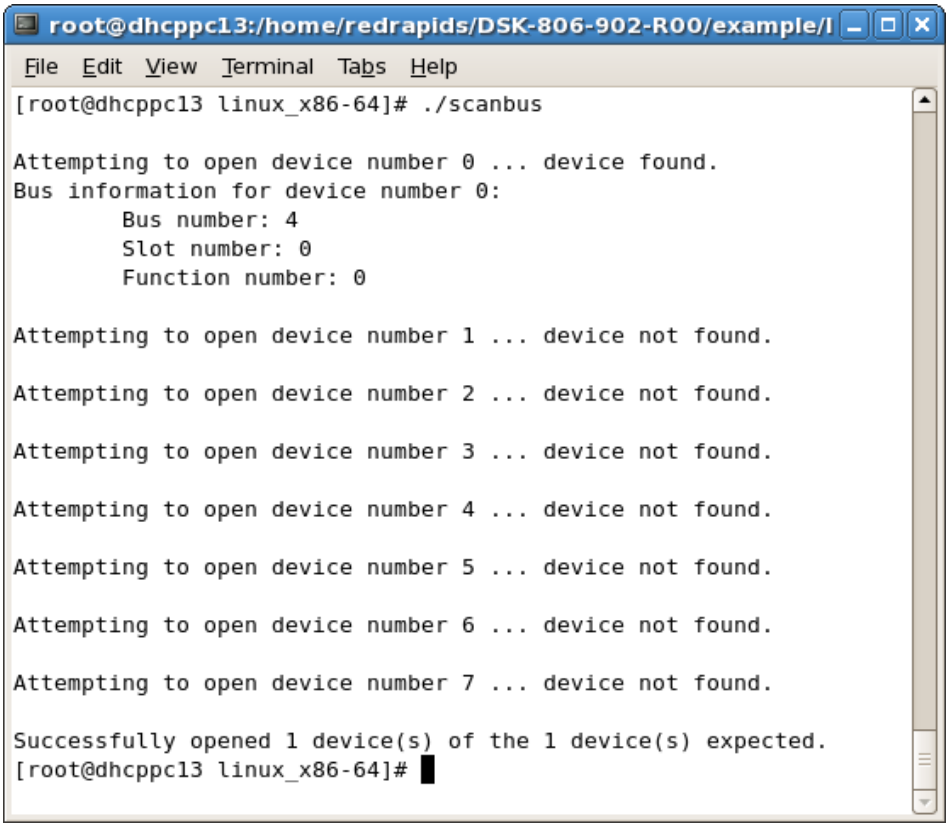

Figure 5-4 ScanBus Execution in Linux