Publication No. REF-806-910-R00

Linux Device Drivers and API Reference Manual

Red Rapids

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

Table of Contents

List of Figures

List of Tables

1.0 Introduction

The Linux device drivers and application program interface (API) provide the vehicle to access hardware resources of a Red Rapids device from a software application written in C. There are two loadable kernel modules available to support a Red Rapids device; rrdev and rrdma.

The rrdev module is required to manage basic communication tasks:

- Perform read/write transactions to memory addresses internal to the device.
- Service interrupts generated by the device.

The rrdma module is only needed if a device has direct memory access (DMA) capability. It performs two critical functions necessary to support DMA transactions:

- Allocate contiguous segments of host memory that form a DMA buffer.
- Supply the virtual and physical addresses of the DMA buffer to an application.

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.

API	Application Program Interface
BAR	Base Address Register
DEV	Device
DMA	Direct Memory Access
ISR	Interrupt Service Routine
Rxx	Any Revision Number
UIO	User (space) I/O

1.2 Tape Archive (TAR-806-91x-Rxx)

The Linux device drivers are distributed on Red Rapids tape archive number TAR-806-910-Rxx. The exact number will depend on the specific product and revision of the code.

The directory structure of the archive is outlined below:

/

The root directory contains the top level Makefile and the load shell script that will install of the drivers required to communicate with a specific Red Rapids product.

/rrdev

The rrdev subdirectory contains source code to the rrdev kernel driver. It also contains the Makefile that must be executed on the host platform to compile the loadable kernel object file.

/rrdma0 (DMA capable devices only)

The rrdma0 subdirectory contains multiple source code folders to support the specific number of data channels available on various Red Rapids products. It also contains the Makefile that must be executed on the host platform to compile the loadable kernel object file. This directory will produce the DMA buffers to support a single Red Rapids product but can be duplicated to support additional devices (rrdma1, rrdma2, etc.). The DMA_DEV_NAME identifier must be changed to uniquely identify each device as discussed in section 4.1.

/rrdma0/# (DMA capable devices only)

Each /rrdma0/# subdirectory contains source code configured to create the designated number (#) of DMA buffers. These directories are included as a convenience to quickly demonstrate the capabilities of various Red Rapids products. The code in each directory is the same except for the value assigned to the NUMBER_OF_BUFFERS identifier. This value can be modified to create any number of buffers desired as discussed in section 4.1.

1.3 Revision History

Version	Date	Description
R00	7/2/2018	Initial release.

2.0 Driver Installation

The following procedure will load the drivers required by a Red Rapids device on a host computer running either a 32-bit or 64-bit Linux operating system.

1. Boot the host computer with Red Rapids hardware installed.
2. Download the DSK-806-91x-Rxx distribution disk from the Red Rapids website (www.redrapids.com).
3. Extract the files from the tar archive to a local directory.
4. Open a Linux terminal and navigate to the local directory installed from the archive:

   ```
   $ cd /<extract path>/DSK-806-91x-Rxx
   ```

5. Run the Makefile, including the three-digit model number (xxx) of the Red Rapids product, in the root directory to compile the kernel object:

   ```
   $ make MODEL=xxx
   ```

   This Makefile will call the Makefiles in the /rrdev and /rrdma0 directories to load the drivers for a single Red Rapids product. This Makefile cannot be used if multiple DMA capable Red Rapids products are installed. Instead, the Makefile in /rrdev must be executed once for all products and individual Makefiles must be executed for each DMA capable product (/rrdma#).

   ☞ The gcc compiler and kernel source code for your Linux distribution must be installed prior to running make.

   The Makefile directories can be cleaned with the following command:

   ```
   $ make MODEL=xxx clean
   ```

6. Change the permissions on the load shell script to make it executable:

   ```
   $ chmod 755 load
   ```

7. Execute the load shell script from the terminal with superuser privileges:

   ```
   $ sudo ./load
   ```

   The load shell script needs to be run each time the computer is booted. This can be accomplished by executing the script from the /etc/rc.local file. Alternatively, the individual kernel modules that are loaded by the script can be added to the existing list of modules that are loaded at boot time. Consult the documentation for your Linux distribution for further information.

   The load shell script will need to be modified if multiple DMA capable Red Rapids products are installed. A line must be added to insert a separate /rrdma#/rrdma#.ko kernel module for each DMA capable device.

8. You can verify that the kernel modules were loaded by executing the following commands:

   ```
   $ lsmod | grep rrdev
   ```

   `$ lsmod | grep rrdma0` (DMA capable devices only)

2.1 Known Issues

rrdma: PAT Warnings

The x86 and x86-64 kernels will post a warning every time mmap() is called from user space to access memory that has been allocated by dma_alloc_coherent() in kernel space. The warning is generated by the page attribute table (PAT) extension and will have the following form:

```
map pfn RAM range req uncached-minus for [mem 0x406156000-0x406156fff], got write-back
```

The message can be eliminated by passing the “nopat” kernel command line parameter at boot time. There are several ways to edit the Linux command line, this is just one possible approach:

1. Edit the /etc/default/grub file to add the "nopat" parameter to the command line default:

   ```
   GRUB_CMDLINE_LINUX_DEFAULT="quiet splash nopat"
   ```

2. Update grub with the following command:

   ```
   $ sudo update-grub
   ```

3. Reboot the computer and verify the new parameter was added with the following command:

   ```
   $ cat /proc/cmdline
   ```

rrdma: DMA Buffer Size Limits

The Linux kernel sets a limit on the size of a contiguous memory allocation, which is equivalent to the size of a DMA page as defined by the rrdma driver. The typical limit is 4 Mbytes but may vary by distribution. Every Red Rapids product can support at least 1024 pages, allowing the creation of a 4 Gbyte scatter-gather buffer within the constraints imposed by the kernel.

3.0 Red Rapids Device (rrdev) Driver

Software applications running in user space can access a Red Rapids device through kernel space using the Linux User I/O (UIO) character device (/dev/uio#) as shown in Figure 3-1. A simple kernel space device driver (rrdev.c) supplied with the hardware is used to identify the device and respond to interrupts. A user space driver (driver_functions.c) is included with the application program interface (API) distributed with the product.

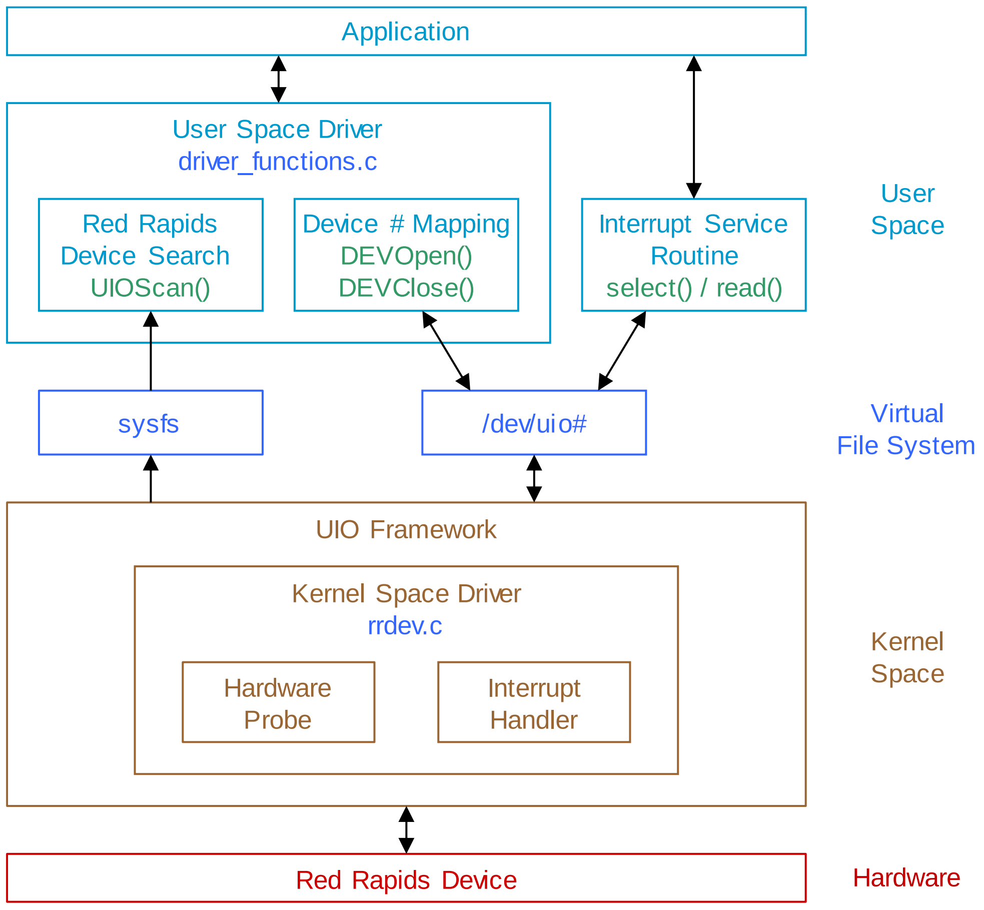

Figure 3-1 UIO Driver Structure

3.1 Kernel Space Driver

The kernel space driver consists of a hardware probe and interrupt handler. The probe function fills in the uio_info structure with information about the device. This includes hardware resources such as the internal address space and interrupt capabilities of the device. The interrupt handler simply disables further interrupts from the device until the current interrupt has been serviced. This is accomplished by writing to the global interrupt mask register of the device. The interrupt service routine running in user space must write to this same register upon completion to re-enable interrupts.

3.2 User Space Driver

The two user space driver functions are only executed when an application is first opened; UIOScan() and DEVOpen(). The UIOScan() function queries the sysfs files for attributes associated with each UIO device. This is necessary to determine which UIO devices are supplied by Red Rapids and not some other vendor. The DEVOpen() function maps the address space of the UIO device to the virtual address space of the application. This allows memory locations to be read or written with existing assignment operators. For example, a Red Rapids PCI device may have two 64-bit aligned address spaces that are each defined by a base address registers (BAR). The DEVOpen() function will provide pointers to the virtual address of each space; BAR0 and BAR2. The following code illustrates how a C assignment operator can be used to communicate directly with the UIO device:

```
unsigned long long      *BAR0;
unsigned long long      *BAR2;
unsigned long long      read_value;
unsigned long long      write_value;
// Read the first address in BAR0 space of the UIO device:
read_value = BAR0[0];
// Write the tenth address in BAR2 space of the UIO device:
BAR2[9] = write_value;
```

The DEVClose() function releases the previous mapping before closing the application.

The user space interrupt service routine also communicates directly with the UIO character device file but is not included with the driver functions since it is application dependent. The interrupt status of the Red Rapids device can be monitored with a read() call to the assigned UIO device. Unfortunately, the read() call will block until an interrupt occurs which can cause an application to hang if no interrupts are generated by the hardware. A permanent block can be avoided by using select() to determine whether the UIO device file is ready to respond to the next read() attempt. A timeout value can also be set to ensure that select() does not permanently block.

3.3 Device 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 that stores the address mapping of the device.
*DMA	s_DMABuf	Structure that conveys DMA buffer parameters obtained from the rrdma driver. This pointer is left NULL if the device is not DMA capable.
*User	void	Undefined pointer available to user applications. This pointer is initialized to NULL.

The device number (DevNum) is assigned when a Red Rapids product is discovered during the UIO scan. The numbering begins with zero and proceeds sequentially with each additional device. The DevNum will match the UIO device number (/dev/uio#) if no other hardware is using the UIO framework, otherwise they may be different.

The device (DEV) structure carries the information needed to communicate with the UIO character device (/dev/uio#). Table 3-2 lists each member of the s_DEVMap structure that is assigned to the DEV data type.

Table 3-2 s_DEVMap Structure

Member Name	Data Type	Definition
fd	int	File descriptor returned when the UIO character device file is opened.
*BAR0	unsigned long long	Pointer to BAR0 address space.
*BAR2	unsigned long long	Pointer to BAR2 address space.

Opening the UIO character device (/dev/uio#) returns a file descriptor (fd) that uniquely identifies the device for any further operations. The user space driver needs the descriptor to map the address spaces of the hardware to virtual addresses that can access the device from an application. The structure can store two of these addresses in the *BAR0 and *BAR2 pointers, although some products may have only one address space (BAR0) available. The file descriptor is also required by the interrupt service routine to read the UIO character device file.

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 4.0 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.

4.0 Red Rapids Direct Memory Access (rrdma) Driver

Products that perform DMA transactions with the host computer require access to contiguous memory that can be directly addressed by hardware. The Red Rapids DMA kernel space driver (rrdma#.c) shown in Figure 4-1 DMA Driver Structure allocates contiguous pages of memory and passes the physical and virtual addresses to user space through a character device (/dev/rrdma#). A user space driver (driver_functions.c) is included with the application program interface (API) distributed with the product.

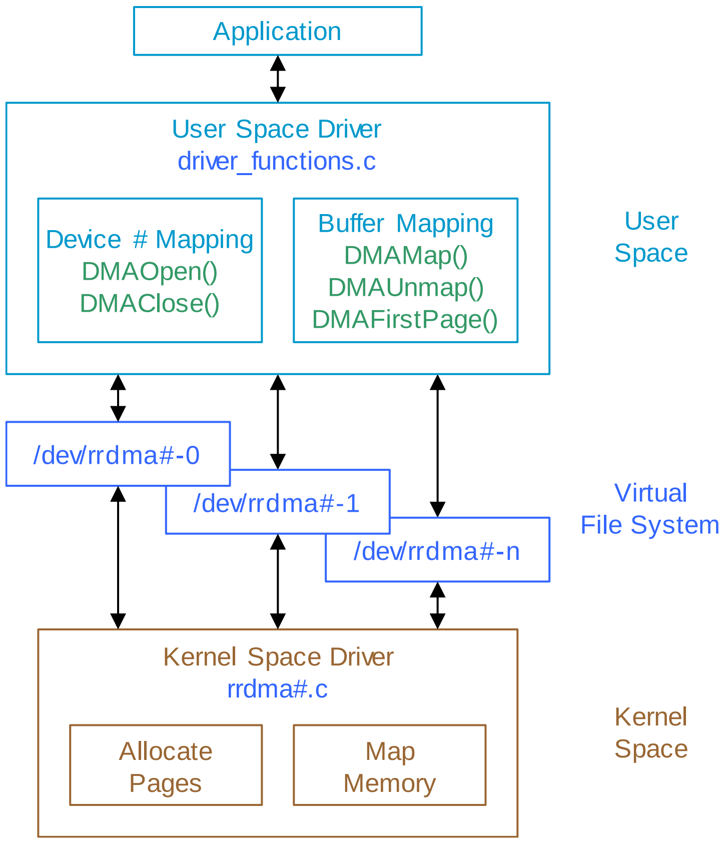

Figure 4-1 DMA Driver Structure

4.1 Kernel Space Driver

The kernel space driver does not interact with the Red Rapids hardware, it simply allocates the defined number of DMA pages that can be later assigned to a scatter-gather buffer by an application. There are two #define directives at the top of the driver source code file that establish the name of the character device and the number of buffers that get created when the kernel module is loaded:

```
#define   DMA_DEV_NAME
#define   NUMBER_OF_BUFFERS
```

The DMA character device name will convey two pieces of information; a device number and a buffer index.

The device number is embedded in the DMA_DEV_NAME string, set to zero by default. A separate character device should be created for each Red Rapids UIO device that is DMA capable. Multiple DMA kernel modules can be loaded by simply incrementing the device number to make each one unique.

The NUMBER_OF_BUFFERS identifier specifies the number of individual DMA buffers created by each DMA device. The buffers are sequentially numbered starting from zero and will appear in the character device name as an index following the device number. This results in multiple character devices of the form /dev/rrdma<DEV#>–<BUF#> created by each kernel module. For example, the character device name /dev/rrdma0–0 provides access to the first buffer supplied by DMA device zero and /dev/rrdma0–1 provides access to the second.

The remaining identifiers set default values for configuration parameters that will determine the size of the buffer. These can be changed to meet the needs of a specific application and can be unique for each buffer created by the device. Refer to the *DMA on Demand Operating Guide* for a detailed description of each parameter.

4.2 User Space Driver

The DMAOpen() user space driver function assigns DMA buffers allocated by the kernel driver to data channels managed by the application. There is minimal communication between user space and the kernel driver, so the memory allocated to each page is also used as a type of mailbox to pass descriptive information about each buffer. The kernel driver preloads six 64-bit values into each DMA page as part of the response to the DMAOpen() call. These values are stored in the s_DMA structure along with the file descriptor as shown in Table 4-1 s_DMABuf Structure.

Table 4-1 s_DMABuf Structure

Member Name	Data Type	Definition
fd	int	File descriptor returned when the DMA character device file is opened.
PageNumber	unsigned	Index to identify which page of the DMA buffer is currently mapped to the virtual address pointer (below).
PageCount	unsigned	Number of pages available in the buffer.
PagesPerMark	unsigned	Number of DMA pages the buffer will service before the hardware issues a marker (status bit, interrupt, etc.).
BurstSizeB	unsigned	Size of each DMA burst transaction in bytes.
BurstCount	unsigned	Number of bursts to store in each DMA page.
*Address	void	Pointer to store either physical address of every DMA page or the virtual address of the currently mapped page, depending on the context.

Opening the DMA character device (/dev/rrdma#–#) returns a file descriptor (fd) that uniquely identifies the device for any further operations. The user space driver references the descriptor to map each DMA page to a virtual address when the application needs to read or write buffer data.

Each page of a DMA buffer is assigned a unique identifying number when it is allocated by the kernel module. The numbering starts at zero for the first page and increments by one for each subsequent page. The page number (PageNumber) indicates which specific page is associated with the current virtual address (*Address). Both variables are updated each time a new page is mapped.

The page count (PageCount) conveys the total number of DMA pages that are allocated 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 4-2. 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. It is important to note that the DMA page size is not constrained by the Linux page size, so it can be smaller or larger. However, the kernel may limit the size of a contiguous memory allocation which is equivalent to limiting the size of a DMA page. The typical maximum value is 4 Mbytes.

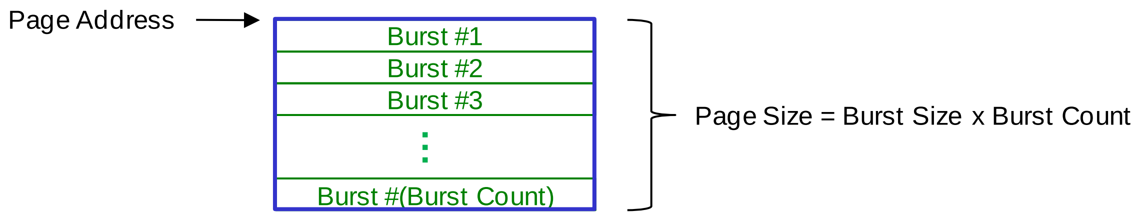

Figure 4-2 DMA Page Size Parameters

There is both a physical address and virtual address associates with each DMA page allocated by the kernel module. 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 both physical and virtual addresses, but never both at the same time. The DMAOpen() function dynamically allocates enough memory to store the physical address of each DMA page available in the open buffer. These addresses are later written to the hardware device and then discarded, freeing the pointer to store virtual addresses as needed by the application.

The DMAMap() function calls mmap() to retrieve the virtual address of the next DMA page in a buffer. The kernel driver increments the page index each time DMAMap() is called and rolls back to zero when the last page is reached. It is not possible to command the driver to a specific page from user space. Instead, multiple DMAMap() calls must be made until the desired page is retrieved. The DMAUnmap() function releases the previous mapping when the application is done processing the current page.

The DMAFirstPage() function is a convenient utility that simply sets the kernel page index back to zero so that the next DMAMap() call will retrieve the first page in the buffer.

The DMAClose() function closes the character device file opened by DMAOpen().