LINDRV

GRLIB Linux Drivers User's Manual

USER MANUAL

RELEASED NOVEMBER 2026

Frontgrade Gaisler AB | Kungsgatan 12 | SE-411 19 | Göteborg | Sweden | frontgrade.com/gaisler

Table of Contents

1. Introduction 5
1.1. Drivers included in the package 5
1.2. Requirements 5
1.2.1. Hardware support and limitations 6
1.2.2. DMA memory 6
1.3. Installing 6
1.4. Device tree bindings 6
1.5. Device node numbering 6
2. GRSPW SpaceWire Driver 8
2.1. Introduction 8
2.1.1. Sources 8
2.1.2. Using the driver 8
2.1.3. Interface rules 9
2.1.4. Examples 9
2.2. Control Interface 9
2.2.1. Overview 9
2.2.2. Data structures and constants 14
2.3. Packet Transfer Interface 18
2.3.1. Packet Reception 19
2.3.2. Packet Transmission 21
2.4. User-space access routines 23
3. GRSPW Kernel Library driver 24
3.1. Introduction 24
3.1.1. Hardware Support 24
3.1.2. Driver sources 24
3.1.3. Examples 24
3.1.4. Known driver limitations 24
3.2. Software design overview 24
3.2.1. Overview 24
3.2.2. Initialization 25
3.2.3. Link control 25
3.2.4. Time Code support 26
3.2.5. RMAP support 26
3.2.6. Port support 26
3.2.7. SpaceWire node address configuration 27
3.2.8. User DMA buffer handling 27
3.2.9. Driver DMA buffer handling 27
3.2.10. Polling and blocking mode 29
3.2.11. Interrupt and work queue 29
3.2.12. Starting and stopping DMA 30
3.2.13. Thread concurrency 30
3.2.14. SMP Support 31
3.2.15. User space support 32
3.3. Device Interface 32
3.3.1. Opening and closing device 32
3.3.2. Kernel users 33
3.3.3. Hardware capabilities 34
3.3.4. Link Control 35
3.3.5. Node address configuration 37
3.3.6. Time Code support 39
3.3.7. Port Control 40
3.3.8. RMAP Control 41
3.3.9. Statistics 42
3.4. DMA interface 43
3.4.1. Opening and closing DMA channels 43
3.4.2. Starting and stopping DMA operation 45
3.4.3. Packet buffer description 47

3.4.4. Blocking/Waiting on DMA activity 48
3.4.5. Sending packets 49
3.4.6. Receiving packets 52
3.4.7. Transmission queue status 55
3.4.8. Statistics 56
3.4.9. DMA channel configuration 57
3.4.10. DMA event callback 59
3.5. API reference 60
3.5.1. Data structures 60
3.5.2. Device functions 60
3.5.3. DMA functions 61
4. SpaceWire Router APB Register Driver 63
4.1. Introduction 63
4.1.1. Sources 63
4.1.2. Using the driver 63
4.1.3. Examples 63
4.2. The spwrtr command line tool 63
4.2.1. Showing the router 64
4.2.2. Changing the router 64
4.2.3. Configuration files 64
4.2.4. Examples 65
4.3. Control Interface 65
4.3.1. Overview 65
4.3.2. Data structures and constants 66
5. GRSPFI SpaceFibre Driver 68
5.1. Introduction 68
5.1.1. Sources 68
5.1.2. Using the driver 68
5.1.3. Examples 69
5.2. ioctl Reference 69
5.2.1. Overview 69
5.2.2. GRSPFI_IOCTL_GET_CAPABILITIES 70
5.2.3. GRSPFI_IOCTL_CONFIG 71
5.2.4. GRSPFI_IOCTL_VC_CONFIG 72
5.2.5. GRSPFI_IOCTL_BC_CONFIG 73
5.2.6. GRSPFI_IOCTL_TIME_CONFIG 74
5.2.7. GRSPFI_IOCTL_TIME_SET 74
5.2.8. GRSPFI_IOCTL_TIME_GET 75
5.2.9. GRSPFI_IOCTL_DMA_CONFIG 75
5.2.10. Packet transfer 76
5.2.11. GRSPFI_IOCTL_BC_SEND 77
5.2.12. GRSPFI_IOCTL_BC_RECEIVE 78
5.2.13. GRSPFI_IOCTL_TX_WAIT, GRSPFI_IOCTL_RX_WAIT and GRSPFI_IOCTL_BCRX_WAIT 78
6. GRSPFI Kernel Library Driver 80
6.1. Introduction 80
6.1.1. Driver sources 80
6.1.2. Examples 80
6.2. Software design overview 80
6.2.1. Overview 80
6.2.2. Key Data Structures 80
6.2.3. Module Registration 81
6.2.4. Device Lifecycle 81
6.2.5. Link Layer Control 82
6.2.6. Time-slot Management 83
6.2.7. Virtual Channel API 83
6.2.8. Broadcast Channel API 86
6.2.9. DMA Channel API 87
6.2.10. Status Registers 87

6.2.11. Interrupt Management 88
6.3. Device tree 89
6.3.1. Bindings 89

1. Introduction

The purpose of the GRLIB Driver package is to provide Linux drivers for GRLIB cores that does not really benefit from being part of the official kernel tree or for other reasons not part of the official kernel tree. SpaceWire for example does not have a generic driver model in Linux.

Drivers can be built outside of the kernel source tree as modules or within the kernel by installing the drivers into the kernel sources tree.

After installing the package into the kernel source tree a menu named "GRLIB Drivers" will appear in the bottom of the "Device Drivers" directory in the kernel configuration GUI. The Kernel Configuration GUI is invoked as usual (for non-LEON system see Table 1.1):

```
[linux/]$ make ARCH=sparc CROSS_COMPILE=sparc-linux- xconfig
```

If the drivers are built outside of the kernel tree and installed into the filesystem for loading during runtime, the building process is as follows (for non-LEON systems select target from Table 1.2):

```
[kernel/drivers/grlib/]$ make KERNELDIR=/path/to/kernel/linux/sources leon
```

This builds all driver modules: grspw.ko, grspw_user.ko, grspw_router.ko, grspfi_drv.ko and grspfi.ko. The kernel must be configured with CONFIG_DMA_SHARED_BUFFER, which the in-tree build selects automatically.

The user-space examples are built from the `user` directory with the same targets, for example:

```
[user/]$ make leon
```

Note that the kernel sources provides a way to install modules using the make target modules_install together with INSTALL_MOD_PATH=/path/to/rootfs/.

Table 1.1. Linux kernel configuration and build settings

Processor	Kernel environment settings
NOEL-V 64-bit	ARCH=riscv CROSS_COMPILE=riscv64-linux-
NOEL-V 32-bit	ARCH=riscv CROSS_COMPILE=riscv32-linux-
LEON	ARCH=sparc CROSS_COMPILE=sparc-linux-

Table 1.2. GRLIB Driver Package Make targets

Processor	Make target
NOEL-V 64-bit	noel64
NOEL-V 32-bit	noel32
LEON	leon

1.1. Drivers included in the package

Below is a list of which drivers are currently distributed in the GRLIB Linux driver package.

- GRSPW2 Kernel Library (for custom kernel driver, or GRSPW Driver)
- GRSPW2 Driver (Char device accessible from Linux User space)
- GRSPW-ROUTER APB Register Driver
- GRSPFI Kernel Library (for custom kernel driver, or GRSPFI Driver)
- GRSPFI Driver (Char device accessible from Linux User space)

1.2. Requirements

The GRLIB Drivers package is built against one specific Linux release, it is expected that drivers may fail to build or does not function properly if used under another Linux version. The kernel that must be used is taken from

www.kernel.org and may require patching using the Frontgrade Gaisler "unofficial patches" distributed until they are included in the official kernel tree.

Please check which GIT version is required used in the VERSION file.

1.2.1. Hardware support and limitations

The following processor platforms are supported:

- LEON 3/4/5
- NOEL-V 32-bit
- NOEL-V 64-bit
- Limited to 32-bit address bus

The following GRLIB IPs are supported by this package:

- GRSPW2
- SpaceWire Router AMBA Ports (GRSPW_SPW2_DMA)
- SpaceWire Router APB control registers interface
- GRSPFI SpaceFibre

1.2.2. DMA memory

The GRSPW and GRSPFI user-space drivers take their packet buffers from a CMA dma-buf heap, which provides physically contiguous memory. The kernel must support CMA and be built with:

- CONFIG_DMA_CMA=y
- CONFIG_DMABUF_HEAPS=y
- CONFIG_DMABUF_HEAPS_CMA=y

The CMA area must be large enough for the buffers of the application. Its size is set with CONFIG_CMA_SIZE_MBYTES, the `cma=` kernel boot parameter, or a reusable `shared-dma-pool` reserved-memory node with `linux,cma-default` in the device tree.

The heap appears under `/dev/dma_heap/`, named `default_cma_region` from Linux 6.16, and `reserved` or after its device tree node (`linux,cma`…) before that. The `system` heap cannot be used, since its buffers are not physically contiguous.

1.3. Installing

Please see the README file included in the driver package for installation instructions.

1.4. Device tree bindings

The drivers requires device tree bindings. On a LEON based system the bindings are typically provided by MKLINUXIMG, but for NOEL based system the bindings needs to be declared in a Device Tree Source (DTS) file.

Documentation about the bindings can be found in the driver package under `kernel/Documentation/devicetree/bindings/grlib`

1.5. Device node numbering

The GRLIB drivers dynamically assigns major numbers, typically within the range 234-254 (from the “LOCAL/EXPERIMENTAL USE” series). More information on device node numbering can be found in `linux/Documentation/admin-guide/devices.txt`

Device nodes are created in `/dev` in the local file system.

```
# ls -l grspw* spwrouter* grspfi*
crw-------    1 root     root      250,   0 Apr  29  2025 grspw0
crw-------    1 root     root      250,   1 Apr  29  2025 grspw1
crw-------    1 root     root      250,   2 Apr  29  2025 grspw2
crw-------    1 root     root      250,   3 Apr  29  2025 grspw3
crw-------    1 root     root      249,   0 Apr  29  2025 spwrouter0
crw-------    1 root     root      248,   0 Apr  29  2025 grspfi0
crw-------    1 root     root      248,   1 Apr  29  2025 grspfi0_0
```

```
crw-------    1 root     root      248,   2 Apr  29  2025 grspfi0_1
```

The GRSPFI VC device nodes (`grspfi`*`N_V`*) are created when the control device is opened the first time.

2. GRSPW SpaceWire Driver

2.1. Introduction

This section describes the Linux GRSPW driver. It provides user space applications with a SpaceWire packet nad link control interface. The driver is implemented using the GRSPW Kernel library (described in Chapter 3) for GRSPW device control and DMA transfer, and it imports a dma-buf allocated from the kernel's CMA dma-heap (a device under `/dev/dma_heap/`, allocated via `DMA_HEAP_IOCTL_ALLOC`) for physically continuous device memory (DMA memory) shared with user-space. The driver supports the GRSPW, GRSPW2 and the DMA interface of the Frontgrade Gaisler SpaceWire Router.

By splitting the GRSPW SpaceWire support into a kernel library and a kernel driver it is possible to reuse specific parts of the driver source. The GRSPW kernel library does not depend on the GRSPW Kernel driver, this makes it possible to create a custom GRSPW kernel module without the involvement of user space using the kernel library only.

The driver provides two different types of interfaces through the standard UNIX access routines (`open, close, ioctl`), one GRSPW device control interface and one packet transfer interface, both accessed using `ioctl`. The actual packet data transferred on SpaceWire is not copied by the packet transfer `ioctl` commands, instead offsets to the data and header are interchanged between kernel space (the driver) and user space (the application). Transferring only addresses to data/header allows the driver to be zero-copy all the way from user-space to actually sending the packet over SpaceWire, however some care must be taken to what memory is used. For example even though memory seems to be linear i user space it might not be linear in physical address space due to the memory management unit (MMU) setup, and when the GRSPW core is doing direct memory access (DMA) only linear addresses can be used. This is solved by allocating packet buffers from a CMA (Contiguous Memory Allocator) pool exposed as a dma-buf through the kernel's dma-heap framework, which guarantees the allocated memory is physically contiguous.

If the SpaceWire router DMA interface is the underlying hardware, some of the parts described here does not affect the hardware at all. For example the link controlling options are of course not implemented at the DMA interface. One can control the SpaceWire router's link by using the SpaceWire router driver instead.

2.1.1. Sources

The GRSPW driver sources are provided under the GPL license, they are available in the GRLIB driver package as described in the table below. Applications include only the user space interface header. All files are relative to `kernel/drivers/grlib/` in the driver package.

Table 2.1. GRSPW driver sources

Location	Description
`spw/grspw.c`	GRSPW Kernel library
`spw/grspw_user.c`	GRSPW Kernel Driver
`include/uapi/linux/grlib/grspw_user.h`	GRSPW Kernel Driver user space interface header
`include/linux/grlib/grspw_user.h`	GRSPW Kernel Driver header, kernel only (includes the user space interface header)
`include/linux/grlib/grspw.h`	GRSPW Kernel library header, kernel only
`include/linux/grlib/dmapool.h`	dma-buf import helper used by the GRSPW Kernel Driver

2.1.2. Using the driver

Applications wanting to access GRSPW devices from user-space include `<linux/grlib/grspw_user.h>`. The header is self-contained: it needs only `<linux/types.h>` and `<linux/ioctl.h>`, not the GRSPW kernel library header. In the driver package it is in `kernel/drivers/grlib/include/uapi`, so a build adds `-I`*`drvpkg`*`/kernel/drivers/grlib/include/uapi` to the compiler flags. A distribution may install it in `/usr/include/linux/grlib`. The headers in `include/linux/grlib` are for the kernel only, and stop a user space build with `#error`.

The user space interface header is licensed `GPL-2.0-or-later WITH Linux-syscall-note`, as the kernel's own user space headers are, so applications built with it are not covered by the GPL.

As mentioned above the user is also responsible for allocating packet buffer memory from a CMA dma-heap and passing the resulting dma-buf file descriptor to the driver, see the `BUFCFG` ioctl below.

Debug output is available through the `/proc/kmsg` interface, and additional debug output can be enabled by defining GRSPWU_DEBUG in the driver sources `grspw_user.c`.

Each GRSPW core is accessed using a single major/minor number, regardless of how many DMA channels the core has. The Major/Minor numbers are determined by the driver package configuration, see Section 1.5.

A GRSPW device can only be opened by one process at a time, a second `open` fails with errno set to EBUSY. The GRSPW device may at the same time be used by other kernel drivers through the GRSPW Kernel Library, as long as they use other DMA channels. A DMA channel another kernel driver has open cannot be used by this driver; HWSUP reports the channels this device may use in *`chan_avail_mask`*. DMA channel 0 is opened by `open`, which fails with EBUSY if another driver has it open. ENOMEM is returned if the GRSPW device could not be opened.

2.1.3. Interface rules

The rules below are part of the user space interface. They are the same for grspw_user, the SpaceWire router driver and the GRSPFI driver.

All GRLIB drivers use the ioctl magic `'J'` (`GRLIB_IOCTL_MAGIC`), with a range of command numbers each. A command sent to another driver's device, and an unknown command, fail with ENOTTY.

Table 2.2. GRLIB ioctl number ranges (magic 'J')

Numbers	Driver
0x00-0x3f	grspw_user (this driver)
0x40-0x5f	grspw_router
0x60-0x7f	Other GRLIB drivers
0x80-0xbf	grspfi
0xc0-0xff	Other GRLIB drivers

- All structs have a fixed layout, the same for every ABI: every field has a fixed-size type (__u8, __u32, __u64, ...) and every hole is a reserved field. The same structs therefore work for 32-bit user space on a 64-bit kernel.
- A struct never changes size. A larger version gets a new ioctl number.
- User space sets input reserved fields to zero and ignores output reserved fields. A nonzero input reserved field gives EINVAL.
- Unknown flags, options and values give EINVAL; they are not masked or mapped. Fields named *`flags`* with no flags defined must be 0.
- Every input is checked before anything is changed, so a call that fails with EINVAL has no effect.
- HWSUP reports the interface version in *`api_version`*, which is `GRSPW_API_VERSION` (2) for the interface described here, and optional driver features in *`features`*, none defined yet (0).

2.1.4. Examples

Within the GRLIB driver package there is a user space example of how this driver can be used. The example uses the user-space API used to call the driver's ioctl interface.

2.2. Control Interface

2.2.1. Overview

The Control interface provides information about the GRSPW hardware, configuration of the driver, reading current statistics, link control and status, selecting port if two ports are available, handling time code transmission, starting/stopping DMA channels and waiting for DMA operations to complete by blocking. The Packet Transfer Interface can not be used unless the DMA channel has been started, the link state is independent of starting/stop-

ping DMA channels. The link state will of course have an impact on what is transferred over SpaceWire, it will affect all DMA channels. Since SpaceWire supports "flow-control" packets may buffer up when the link state goes from run-state to any other state. The user is expected to handle the link and its state.

The control interface is accessed using the standard UNIX `ioctl` routine.

In the table below all currently supported `ioctl` commands, their numbers and their argument type are listed. The data structures and constants are declared in the `grspw_user.h` header file and described in Section 2.2.2. All GRSPW commands starts with GRSPW_IOCTL_ which has to be added to the command name given in the table below. The data direction below indicates in which direction data is transferred to the kernel:

- Input: Argument is an address. The driver reads data from the given address.
- Output: Argument is an address. The driver writes data to the given address.
- Input/Output: both above cases.
- Argument: an integer passed in place of the address, no memory transferred between kernel/user.
- None: Argument ignored.

Any command that takes an address fails with EFAULT if the address is not valid.

Table 2.3. `ioctl` commands supported by the GRSPW Kernel driver.

Command (number)	Data Direction	Argument Type	Description
HWSUP (0x01)	Output	struct grspw_hwsup *	Copy the interface version and the hardware configuration of the GRSPW core: the DMA channels this device may use (*`chan_avail_mask`*), the number of DMA channels in hardware, if RMAP/RMAP-CRC is supported by the core, the number of SpW ports, etc.
BUFCFG (0x02)	Input	struct grspw_bufcfg *	Set up packet buffers. The user is responsible for allocating packet data/header memory from a CMA dma-heap and passing its dma-buf file descriptor via `pool_dmabuf_fd`. The driver attaches to it here and keeps it until the next BUFCFG or until the device is closed, so the file descriptor may be closed after this call. Packet buffer references are given as byte offsets into that dma-buf, rather than user-space pointers. If `pool_dmabuf_fd` is not a dma-buf this command fails with EBADF or EINVAL, and the previous configuration is kept. EINVAL is also returned if `rx_pkts` or `tx_pkts` is zero or above 65536, or `reserved` is not zero, and EBUSY if DMA is started. A successful BUFCFG puts the driver in stopped state: packets still in the driver after STOP are dropped, and RX_RECV and TX_RECLAIM fail with EBUSY until the next START. On systems without cache-coherent DMA (NOEL-V), RX buffers should start and end on a cache line boundary: data in the same cache line as an RX buffer may be lost while the driver owns the buffer. The driver must also allocate structures for packet handling. The packet structures stores the packet state, data/header offsets, *`user_data`* etc. This command specifies how many RX packets (`rx_pkts`) and TX packets (`tx_pkts`) maximally can simultaneously be buffered internally by the driver, the default is 1024 of each. The packet structures are shared between all DMA channels. The packet structures are allocated when the START command is issued, ENOMEM is returned if driver was not able to allocate as many packet structures as requested.
CONFIG_SET (0x03)	Input	struct grspw_config *	Configure driver according to input. One can configure promiscuous mode, node addresses, which DMA channels will be used, DMA channel configuration,

Command (number)	Data Direction	Argument Type	Description
			time code RX/TX enable and RMAP options (destination key, RMAP enable, RMAP buffer). The node address configuration of all DMA channels is written, except for DMA channels another driver has open (not in *chan_avail_mask* of HWSUP), whatever their *node_en*. The available channels are determined again on every HWSUP, CONFIG_READ and CONFIG_SET, as another driver may open a channel after open. GRSPW_TCOPTS_EN_RXIRQ in *tc_cfg* is read only and ignored, so a configuration read with CONFIG_READ can always be written again. Packets still in a DMA channel that CONFIG_SET disables are dropped. The whole configuration, including RMAP support in hardware, is checked before anything is changed. On failure errno is set as follows: • EINVAL: a reserved field is not zero, *promiscuous* is not 0 or 1, *node_en* is not 0 or 1, *rmap_cfg* or *tc_cfg* contains unknown bits, a DMA channel configuration has unknown flags, an enabled channel's *rxmaxlen* is not 1 to 0xffffff, or *enable_chan_mask* selects a DMA channel the hardware lacks. Nothing is changed. • EIO: RMAP is enabled but not implemented in hardware. Nothing is changed. • EBUSY: DMA is started, *enable_chan_mask* enables a channel another driver has open. Nothing is changed. • EAGAIN: a DMA channel to disable could not be closed because a thread woken from RX_WAIT or TX_WAIT has not left it yet. Channels may have been opened or closed, but no register is written. Call it again.
CONFIG_READ (0x04)	Output	struct grspw_config *	Copies the current configuration to the address given by the argument. DMA Channel configuration will only be copied for previously enabled channels, for other channels the data is zero. The node addresses of all channels are reported, also of those another driver has open.
STATS_READ (0x10)	Output	struct grspw_stats *	The driver gather statistics both globally and for respective DMA channel. All gathered statistics are copied to the user provided buffer. The statistics of DMA channels that are not enabled are zero.
STATS_CLR (0x11)	None	N/A	Clears the current gathered statistics. Resets all counters.
LINKCTRL (0x20)	Input	struct grspw_link_ctrl *	Set SpaceWire transfer speed (clock division factor) and control the link start, link disable, link auto start, IRQ on link error and disable link on error functions of the GRSPW core. See GRSPW_LINKOPTS_* options. The clock division factors are always written. The *stscfg* field selects which link status bits (GRSPW_LINKSTS_* options) the driver clears when it handles them; the others are left for STATUS_READ and STATUS_CLR. EINVAL is returned if *ctrl* or *stscfg* contains unknown bits, or a reserved field is not zero.

Command (number)	Data Direction	Argument Type	Description
PORTCTRL (0x21)	Argument	integer	Select SpaceWire port configuration. The GRSPW core may have have support for two SpaceWire ports, the port select behavior of the core can be controlled by using this command. The argument is the value, not a pointer: • `GRSPW_PORT0` (0): Port0 always selected. • `GRSPW_PORT1` (1): Port1 always selected. • `GRSPW_PORT_AUTO` (3): Both Port0 and Port1, core selects between them. EINVAL is returned for any other value, and EIO if Port1 is selected but not implemented in hardware.
LINKSTATE (0x22)	Output	struct grspw_link_state *	Copies the current link state of the GRSPW core to the provided buffer. The current link configuration, Clock division factors (start and run), the link state, port configuration and which port is currently active is copied.
TC_SEND (0x23)	Argument	integer	The argument is the value, not a pointer. This command sets the TCTRL and TIMECNT bits of the GRSPW core if bit 8 (`GRSPW_TC_SEND_OPTION_SET_TCTRL`) is set to one. The TCTRL and TIMECNT values are taken from the low 8-bits of the argument (`GRSPW_TC_SEND_OPTION_TCTRL`). After (optionally) setting the TCTRL:TIMECNT a Tick-In is generated if bit 9 (`GRSPW_TC_SEND_OPTION_TCTX`) is set to one. EINVAL is returned if any other bit is set.
TC_READ (0x24)	Output	__u32 *	This command stores the current value of the GRSPW core TCTRL:TIMECNT bits (TCTRL bits 7..6, TIMECNT bits 5..0) to the address given by the argument.
QPKTCNT (0x25)	Output	struct grspw_qpktcnt *	Reads the current number of packets in all TX/RX queues of all enabled DMA channels. This can be used for debugging of the RX/TX process in an application, it can also be used to determine the number of packets currently buffered by the driver. The counters of DMA channels that are not enabled are zero.
STATUS_READ (0x26)	Output	__u32 *	Reads the current value of the GRSPW STATUS register and copies it back to the user provided buffer. From this value the link error flags can be read.
STATUS_CLR (0x27)	Input	__u32 *	Clears one or more bits in the GRSPW STATUS register. The user controls which bits are cleared by setting respective bit to a one. Bits that are zero does not affect the GRSPW STATUS register bits. This functionality is typcially used in combination with STATUS_READ and configuring the GRSPW_LINKSTS_* options to allow the user to customly control the link. The standard behaviour is to let the driver's interrupt handler clear the status bits and count statistics on errors instead. This command fails with EBUSY while DMA is started.
START (0x30)	None	N/A	Start all DMA activity on all enabled DMA channels. The receiver is enabled however packet buffers must be prepared in order to actually receive anything. After starting the packet transfer commands of the driver are available. See the Packet Transfer Interface on how packets are sent/received. After start the BUFCFG and CONFIG_SET `ioctl` commands fail with EBUSY until stopped again.

Command (number)	Data Direction	Argument Type	Description
			Packets still in the driver from before a STOP are dropped. On failure errno is set as follows: • EBUSY: DMA is already started. • ENOMEM: the packet structures could not be allocated, due to either not enough memory or too many requested. • EINVAL: no dma-buf pool has been given with the `BUFCFG` command. • EAGAIN: a thread woken by a previous STOP has not yet returned from RX_WAIT or TX_WAIT. The command can be retried. • EIO: a DMA channel could not be started.
STOP (0x31)	None	N/A	Stops DMA operation, this will disable the receiver and transmitter of the GRSPW core, and wakes the threads in RX_WAIT and TX_WAIT, which return EIO. After the driver has been stopped TX(SEND) and RX(PREPARE) operation will result in error EBUSY, but the RX(RECEIVE) and TX(RECLAIM) operation will still be working so that the user can read out all packet buffers. By setting the appropriate flags in the packet information it is possible to determine if a packet has been received/transmitted or not. STOP when not started has no effect.
RX_WAIT (0x34)	Input	struct grspw_rx_wait_chan *	Blocks the caller until the RX queue packet counters conditions of one DMA channel are fulfilled. The conditions and timeout are described by the input data structure, see struct grspw_rx_wait_chan for usage. The timeout is given in milliseconds by *`timeout_ms`*, zero waits without timeout. The counts and *`timeout_ms`* must be at most INT_MAX. The wait takes no lock that other commands need. Only one thread at a time may wait per DMA channel and direction. The driver only checks the conditions on DMA interrupts, so the packets or the channel need interrupts (`GRSPW_RXPKT_IE`, `GRSPW_TXPKT_IE`, *`rx_irq_en_cnt`*, *`tx_irq_en_cnt`*, `GRSPW_DMAFLAG_RXIE` or `GRSPW_DMAFLAG_TXIE`), or the wait needs a timeout. On failure errno is set as follows: • ETIMEDOUT: the timeout expired before the conditions were fulfilled. • EINTR: a signal arrived, with or without a timeout. • EIO: DMA is not started, or was stopped during the wait (STOP, or a link or AHB error). • EBUSY: the DMA channel is not enabled, or another thread is already waiting on RX of this DMA channel. • EINVAL: *`chan`* is not a DMA channel present in hardware, *`flags`* is not zero, *`op`* is not `GRSPW_OP_AND` or `GRSPW_OP_OR`, or a count or *`timeout_ms`* is above INT_MAX.
TX_WAIT (0x35)	Input	struct grspw_tx_wait_chan *	Blocks the caller until the TX queue packet counters conditions of one DMA channel are fulfilled. The conditions and timeout are described by the input data structure, see struct grspw_tx_wait_chan for usage. The timeout is giv-

Command (number)	Data Direction	Argument Type	Description
			en in milliseconds by *timeout_ms*, zero waits without timeout. The counts and *timeout_ms* must be at most INT_MAX. The wait takes no lock that other commands need. Only one thread at a time may wait per DMA channel and direction. The driver only checks the conditions on DMA interrupts, so the packets or the channel need interrupts (GRSPW_RXPKT_IE, GRSPW_TXPKT_IE, *rx_irq_en_cnt*, *tx_irq_en_cnt*, GRSPW_DMAFLAG_RXIE or GRSPW_DMAFLAG_TXIE), or the wait needs a timeout. On failure errno is set as follows: • ETIMEDOUT: the timeout expired before the conditions were fulfilled. • EINTR: a signal arrived, with or without a timeout. • EIO: DMA is not started, or was stopped during the wait (STOP, or a link or AHB error). • EBUSY: the DMA channel is not enabled, or another thread is already waiting on TX of this DMA channel. • EINVAL: *chan* is not a DMA channel present in hardware, *flags* is not zero, *op* is not GRSPW_OP_AND or GRSPW_OP_OR, or a count or *timeout_ms* is above INT_MAX.

The packet transfer commands RX_PREPARE (0x38), RX_RECV (0x39), TX_SEND (0x3a) and TX_RECLAIM (0x3b) are described in the Packet Transfer Interface section.

2.2.2. Data structures and constants

The structures and constants of the control interface are listed below as declared in grspw_user.h. Fields named *reserved* must be zero on input and are to be ignored on output.

2.2.2.1. Hardware support (HWSUP)

hw_version holds the device ID in bits 31..16 and the core version in bits 15..0. *ndma_chans* is the number of DMA channels in hardware, while *chan_avail_mask* leaves out the channels another kernel driver has open.

```
#define GRSPW_API_VERSION 2

struct grspw_hwsup {
 __u32 api_version; /* GRSPW_API_VERSION of the driver */
 __u32 features;  /* Optional driver features, none yet */
 __u32 hw_version; /* Device ID (GRSPW_HW_*) << 16 | version */
 __u32 chan_avail_mask; /* DMA channels this device may use: the
     * hardware's, less those another kernel
     * driver uses
     */
 __u8 rmap;  /* RMAP target available */
 __u8 rmap_crc;  /* RMAP CRC available */
 __u8 rx_unalign; /* RX to unaligned (byte) addresses */
 __u8 nports;  /* Number of ports, 1 or 2 */
 __u8 ndma_chans; /* Number of DMA channels in hardware, 1..4 */
 __u8 strip_adr;  /* Hardware can strip the address byte */
 __u8 strip_pid;  /* Hardware can strip the protocol ID */
 __u8 reserved0;
 __u32 reserved[4];
};

/* Device IDs in hw_version >> 16 */
#define GRSPW_HW_SPW1  0x01f /* GRSPW */
#define GRSPW_HW_SPW2  0x029 /* GRSPW2 */
#define GRSPW_HW_SPW2_DMA 0x08a /* AMBA port of a GRSPW router */
```

2.2.2.2. Buffer configuration (BUFCFG)

```
struct grspw_bufcfg {
 __u32 rx_pkts;  /* RX packet structures, shared by all channels,
     * 1 to 65536 (1024 by default)
     */
 __u32 tx_pkts;  /* TX packet structures, as rx_pkts */
 __s32 pool_dmabuf_fd; /* dma-buf the packet offsets refer to */
 __u32 reserved;
};
```

2.2.2.3. Configuration (CONFIG_SET, CONFIG_READ)

chan of struct grspw_addrcfg holds the node address of each DMA channel, and *chan* of struct grspw_config the configuration of each DMA channel. *enable_chan_mask* selects the DMA channels to use: a channel can receive once buffers have been prepared, and send once packets are queued. Most often only channel 0 exists, then set it to 1. *rxmaxlen* is the largest packet received on the channel, 1 to 0xffffff for an enabled channel; every RX buffer of the channel must hold it.

```
/* node_en of struct grspw_nodecfg. CONFIG_SET leaves the node address of
 * a channel another kernel driver uses (not in chan_avail_mask) alone,
 * whatever node_en is.
 */
#define GRSPW_NODE_ADDR_DISABLE 0 /* Use the default node address */
#define GRSPW_NODE_ADDR_ENABLE 1 /* Use node_addr and node_mask */

/* Node address of a DMA channel */
struct grspw_nodecfg {
 __u8 node_en;  /* GRSPW_NODE_ADDR_* */
 __u8 node_addr;
 __u8 node_mask;
 __u8 reserved;
};

struct grspw_addrcfg {
 __u32 promiscuous; /* 1: all packets to DMA channel 0 */
 __u8 def_addr;  /* Default node address and mask */
 __u8 def_mask;
 __u8 reserved[2];
 struct grspw_nodecfg chan[4];
};

/* rmap_cfg */
#define GRSPW_RMAPOPTS_EN_RMAP 0x0001 /* Enable the RMAP target */
#define GRSPW_RMAPOPTS_EN_BUF 0x0002 /* Enable RMAP buffering */

/* tc_cfg */
#define GRSPW_TCOPTS_EN_RXIRQ 0x0001 /* Time-code RX interrupt: read only,
      * CONFIG_SET ignores it
      */
#define GRSPW_TCOPTS_EN_TX 0x0004
#define GRSPW_TCOPTS_EN_RX 0x0008

/* flags of struct grspw_dmacfg */
#define GRSPW_DMAFLAG_NO_SPILL 0x00000001 /* DMA-CTRL NS bit */
#define GRSPW_DMAFLAG_STRIP_ADR 0x00000004 /* DMA-CTRL SA bit */
#define GRSPW_DMAFLAG_STRIP_PID 0x00000008 /* DMA-CTRL SP bit */
#define GRSPW_DMAFLAG_TXIE 0x00100000 /* TX interrupt, tx_irq_en_cnt 0 */
#define GRSPW_DMAFLAG_RXIE 0x00200000 /* RX interrupt, rx_irq_en_cnt 0 */
#define GRSPW_DMAFLAG_MASK (GRSPW_DMAFLAG_NO_SPILL | GRSPW_DMAFLAG_STRIP_ADR | \
     GRSPW_DMAFLAG_STRIP_PID | GRSPW_DMAFLAG_TXIE | \
     GRSPW_DMAFLAG_RXIE)

/* Configuration of a DMA channel */
struct grspw_dmacfg {
 __u32 flags;  /* GRSPW_DMAFLAG_* */
 __u32 rxmaxlen;  /* Largest packet received, 1 to 0xffffff
     * for an enabled channel
     */
 __u32 rx_irq_en_cnt; /* RX interrupt every cnt descriptors */
 __u32 tx_irq_en_cnt; /* TX interrupt every cnt descriptors */
};

struct grspw_config {
 struct grspw_addrcfg adrcfg;
 __u32 rmap_cfg;  /* GRSPW_RMAPOPTS_* */
 __u8 rmap_dstkey;
 __u8 reserved0[3];
 __u32 tc_cfg;  /* GRSPW_TCOPTS_* */
 __u32 enable_chan_mask; /* DMA channels to use */
 struct grspw_dmacfg chan[4];
 __u32 reserved[6];
};
```

2.2.2.4. Statistics (STATS_READ)

All counters are 64-bit. The counters of DMA channels that are not enabled are zero.

```
struct grspw_stats_core {
 __u64 irq_cnt;
 __u64 err_credit;
 __u64 err_eeop;
 __u64 err_addr;
 __u64 err_parity;
 __u64 err_disconnect;
 __u64 err_escape;
 __u64 err_wsync; /* GRSPW1 only */
 __u64 reserved[4];
};

struct grspw_stats_chan {
 __u64 irq_cnt;  /* DMA interrupts of the channel */
 __u64 tx_pkts;  /* Packets sent */
 __u64 tx_err_link; /* Packets sent with a link error */
 __u64 rx_pkts;  /* Packets received */
 __u64 rx_err_trunk; /* Packets received truncated */
 __u64 rx_err_endpkt; /* Packets received with a bad end */
 /* Queue sizes seen, to help size the number of buffers */
 __u64 send_cnt_min; /* TX SEND queue */
 __u64 send_cnt_max;
 __u64 tx_sched_cnt_min; /* TX SCHED queue */
 __u64 tx_sched_cnt_max;
 __u64 sent_cnt_max; /* TX SENT queue */
 __u64 tx_work_cnt; /* Times the work thread processed TX */
 __u64 tx_work_enabled; /* TX descriptors enabled by the work thread */
 __u64 ready_cnt_min; /* RX READY queue */
 __u64 ready_cnt_max;
 __u64 rx_sched_cnt_min; /* RX SCHED queue */
 __u64 rx_sched_cnt_max;
 __u64 recv_cnt_max; /* RX RECV queue */
 __u64 rx_work_cnt; /* Times the work thread processed RX */
 __u64 rx_work_enabled; /* RX descriptors enabled by the work thread */
 __u64 reserved[4];
};

struct grspw_stats {
 struct grspw_stats_core core;
 struct grspw_stats_chan chan[4]; /* Zero for unused channels */
};
```

2.2.2.5. Link (LINKCTRL, PORTCTRL, LINKSTATE)

`ctrl` of struct grspw_link_ctrl takes the GRSPW_LINKOPTS_* options and *`stscfg`* the GRSPW_LINKSTS_* bits. *`link_ctrl`* of struct grspw_link_state reports the GRSPW_LINKOPTS_* options, *`link_state`* one of GRSPW_LS_*, *`port_cfg`* one of the PORTCTRL values and *`port_active`* the active port, 0 or 1.

```
/* ctrl of struct grspw_link_ctrl, link_ctrl of struct grspw_link_state */
#define GRSPW_LINKOPTS_ENABLE  0x0000
#define GRSPW_LINKOPTS_DISABLE  0x0001
#define GRSPW_LINKOPTS_START  0x0002
#define GRSPW_LINKOPTS_AUTOSTART 0x0004
#define GRSPW_LINKOPTS_DIS_ONERR 0x0008 /* Stop DMA TX on a link error */
#define GRSPW_LINKOPTS_EIRQ  0x0200 /* Link error interrupt */
#define GRSPW_LINKOPTS_DIS_ON_CE 0x0020000 /* Disable link on: credit error */
#define GRSPW_LINKOPTS_DIS_ON_ER 0x0040000 /* escape error */
#define GRSPW_LINKOPTS_DIS_ON_DE 0x0080000 /* disconnect */
#define GRSPW_LINKOPTS_DIS_ON_PE 0x0100000 /* parity error */
#define GRSPW_LINKOPTS_DIS_ON_WE 0x0400000 /* write sync error (GRSPW1) */
#define GRSPW_LINKOPTS_DIS_ON_EE 0x1000000 /* early EOP/EEP */
#define GRSPW_LINKOPTS_MASK  0x15e020f

/* stscfg of struct grspw_link_ctrl: the status register bits the driver
 * clears when it handles them; the others are left for STATUS_READ and
 * STATUS_CLR
 */
#define GRSPW_LINKSTS_CE 0x002 /* Credit error */
#define GRSPW_LINKSTS_ER 0x004 /* Escape error */
#define GRSPW_LINKSTS_DE 0x008 /* Disconnect error */
#define GRSPW_LINKSTS_PE 0x010 /* Parity error */
#define GRSPW_LINKSTS_WE 0x040 /* Write sync error (GRSPW1) */
#define GRSPW_LINKSTS_IA 0x080 /* Invalid address */
#define GRSPW_LINKSTS_EE 0x100 /* Early EOP/EEP */
#define GRSPW_LINKSTS_MASK 0x1de

struct grspw_link_ctrl {
```

```
 __u32 ctrl;  /* GRSPW_LINKOPTS_* */
 __u8 clkdiv_start; /* Clock divisor during startup */
 __u8 clkdiv_run; /* Clock divisor in run state */
 __u8 reserved0[2];
 __u32 stscfg;  /* GRSPW_LINKSTS_* */
 __u32 reserved;
};

/* PORTCTRL: the argument is the port, not a pointer */
#define GRSPW_PORT0  0
#define GRSPW_PORT1  1
#define GRSPW_PORT_AUTO  3 /* Selected by the hardware */

/* link_state of struct grspw_link_state */
#define GRSPW_LS_ERRRST  0
#define GRSPW_LS_ERRWAIT 1
#define GRSPW_LS_READY  2
#define GRSPW_LS_CONNECTING 3
#define GRSPW_LS_STARTED 4
#define GRSPW_LS_RUN  5

struct grspw_link_state {
 __u32 link_ctrl; /* GRSPW_LINKOPTS_* */
 __u8 clkdiv_start;
 __u8 clkdiv_run;
 __u8 reserved0[2];
 __u32 link_state; /* GRSPW_LS_* */
 __u32 port_cfg;  /* GRSPW_PORT* */
 __u32 port_active; /* 0 or 1 */
 __u32 reserved;
};
```

2.2.2.6. Time-codes (TC_SEND, TC_READ)

```
/* TC_SEND: the argument is a value, not a pointer */
#define GRSPW_TC_SEND_OPTION_TCTRL 0x0ff /* TCTRL (7..6) and TIMECNT (5..0) */
#define GRSPW_TC_SEND_OPTION_SET_TCTRL 0x100 /* Write TCTRL|TIMECNT */
#define GRSPW_TC_SEND_OPTION_TCTX 0x200 /* Then send a time-code (Tick-In) */
```

2.2.2.7. Queues (QPKTCNT)

The number of packets in the driver's queues of each DMA channel, see Figure 2.1 and Figure 2.2. The *_hw* fields count the packets in the hardware descriptor table.

```
struct grspw_qpktcnt_chan {
 __u32 rx_ready;
 __u32 rx_sched;
 __u32 rx_recv;
 __u32 rx_hw;
 __u32 tx_send;
 __u32 tx_sched;
 __u32 tx_sent;
 __u32 tx_hw;
};

struct grspw_qpktcnt {
 struct grspw_qpktcnt_chan chan[4]; /* Zero for unused channels */
};
```

2.2.2.8. Waits (RX_WAIT, TX_WAIT)

RX_WAIT waits until *recv_cnt* or more received packets are in the RECV queue, *op*, at most *ready_cnt* buffers are left in the READY and SCHED queues to receive into. TX_WAIT waits until at most *send_cnt* packets are left in the SEND and SCHED queues, *op*, *sent_cnt* or more packets are in the SENT queue. *op* combines the two conditions with AND or OR.

```
/* op of the wait structs */
#define GRSPW_OP_AND 0
#define GRSPW_OP_OR 1

struct grspw_rx_wait_chan {
 __u32 chan;  /* DMA channel */
 __u32 flags;  /* None defined, must be 0 */
 __u32 timeout_ms; /* 0: no timeout */
 __u32 recv_cnt;
 __u32 op;  /* GRSPW_OP_* */
 __u32 ready_cnt;
};
```

```
struct grspw_tx_wait_chan {
 __u32 chan;  /* DMA channel */
 __u32 flags;  /* None defined, must be 0 */
 __u32 timeout_ms; /* 0: no timeout */
 __u32 send_cnt;
 __u32 op;  /* GRSPW_OP_* */
 __u32 sent_cnt;
};
```

2.3. Packet Transfer Interface

The packet transfer interface is used to send and receive SpaceWire packets on the DMA channels. The GRSPW core is configurable how many DMA channels it has, a core may have from one up to four DMA channels. From the control interface one can read how many DMA channels are present on the GRSPW device. This interface is open to the user when DMA operation has been started from the control interface (START). Trying to access the interface when it is not started will result in an error and errno will be set to EBUSY. After STOP the RX_RECV and TX_RECLAIM commands remain available, to read out the remaining packet buffers, until BUFCFG is issued, whereas RX_PREPARE and TX_SEND fail with EBUSY. They also fail with EBUSY when the driver has stopped the DMA channel itself after a link error (see `ctrl` of LINKCTRL) or an AHB error.

Similar to the control interface this driver provides an interface to the GRSPW Kernel Library. The GRSPW Kernel Library documentation in Chapter 3 describes the buffering, packet queues, DMA operations, interrupts etc. in more detail.

Since the GRSPW driver does not manage packet buffer memory itself, but relies on the CMA dma-heap pool configured via BUFCFG and the user for that, the user must prepare the driver with ready RX buffers to be able to receive packets in the future. The user is also responsible to reuse sent packet buffers, in order for the user to know when a packet buffer has been sent and is ready to be reused the driver let the user read back/reclaim TX buffers.

The interface supports four basic operations that can be performed independently per DMA channel, see list below. All packet operations are completed in the order they are given to the driver, for example if multiple packet buffers are requested to be sent the order in which the buffers are sent and also reclaimed is the same as the order they where given to the driver using the TX_SEND command.

- RX(PREPARE), prepare the driver with free RX packet buffers.
- RX(RECEIVE), read out received SpaceWire packets, the packet data are placed in previously prepared packet buffers.
- TX(SEND), queue one or multiple packets for transmission by handing over initialized packet buffers.
- TX(RECLAIM), read out snet packet buffers from the driver (previously sent)

The above operations are implemented by the `ioctl` commands below, with the GRSPW_IOCTL_ prefix. They all take a pointer to a struct grspw_pkts, which describes an array of packet entries in user space. The driver reads the entries from the array (RX_PREPARE, TX_SEND) or writes them to it (RX_RECV, TX_RECLAIM), and sets `count` to the number of entries it used. The format of the entries is described in the sections below.

Table 2.4. Packet transfer `ioctl` commands

Command (number)	Data Direction	Argument Type	Entries
RX_PREPARE (0x38)	Input/Output	struct grspw_pkts *	struct grspw_wrxpkt, read by the driver
RX_RECV (0x39)	Input/Output	struct grspw_pkts *	struct grspw_rrxpkt, written by the driver
TX_SEND (0x3a)	Input/Output	struct grspw_pkts *	struct grspw_wtxpkt, read by the driver
TX_RECLAIM (0x3b)	Input/Output	struct grspw_pkts *	struct grspw_rtxpkt, written by the driver

```
struct grspw_pkts {
 __u64 pkts;  /* User space address of the array */
 __u32 count;  /* In: entries in the array, out: entries used */
 __u32 chan;  /* PREPARE, SEND: the DMA channel */
 __u32 chan_mask; /* RECV, RECLAIM: the DMA channels */
 __u32 flags;  /* None defined, must be 0 */
```

```
};
```

- *pkts* is the address of the array as a 64-bit integer, for example (uintptr_t)array, also for 32-bit user space. *count* must be 1 to INT_MAX and *flags* must be zero, otherwise EINVAL is returned.
- RX_PREPARE and TX_SEND: *chan* is a DMA channel number and *chan_mask* must be zero. EINVAL is returned if the DMA channel is not present in hardware or *chan_mask* is not zero, and EBUSY if the DMA channel is not enabled.
- RX_RECV and TX_RECLAIM: *chan_mask* is a nonzero mask of DMA channels to read from, and *chan* must be zero. EINVAL is returned if the mask is zero, has a bit for a DMA channel the hardware lacks, or *chan* is not zero. DMA channels that are not enabled are skipped.

Offsets are byte offsets into the pool dma-buf given by BUFCFG. Each offset and length must resolve to one physically contiguous range of the pool.

RX_PREPARE and TX_SEND queue as many entries as the driver has free packet structures for (see BUFCFG), from the start of the array, and return that number, which may be zero, in *count*. Fewer free packet structures than entries is not an error. They check each of those entries before they queue any. An invalid entry, with an unknown flag, a nonzero reserved field, a range outside the pool or (RX_PREPARE) a buffer smaller than *rxmaxlen*, fails the call with EINVAL and nothing is queued. EFAULT is returned, and nothing is queued, if the array cannot be read. Entries beyond the free packet structures are not read.

RX_RECV and TX_RECLAIM return up to *count* packets in the array, and *count* 0 when there are none; this is not an error. The DMA channels of the mask are read in order, from channel 0. If the array cannot be written the call fails with EFAULT, and the packets stay in the driver: the next call returns them first. If entries of an earlier DMA channel in the mask were already written when the fault occurs, the call succeeds with *count* set to the entries written.

Packets still in a DMA channel that CONFIG_SET disables, or in the driver at BUFCFG or at the next START, are dropped: read them out with RX_RECV and TX_RECLAIM after STOP first.

The flags of the packet entries are defined as follows. An input flag not listed gives EINVAL.

```
/* TX, input (struct grspw_wtxpkt) */
#define GRSPW_TXPKT_NOCRC_MASK 0x000f /* Header bytes left out of the CRC */
#define GRSPW_TXPKT_NOCRC_LEN(n) ((n) & GRSPW_TXPKT_NOCRC_MASK)
#define GRSPW_TXPKT_IE  0x0040 /* Interrupt when sent */
#define GRSPW_TXPKT_HCRC 0x0100 /* Append a header CRC */
#define GRSPW_TXPKT_DCRC 0x0200 /* Append a data CRC */
#define GRSPW_TXPKT_INPUT_MASK (GRSPW_TXPKT_NOCRC_MASK | GRSPW_TXPKT_IE | \
     GRSPW_TXPKT_HCRC | GRSPW_TXPKT_DCRC)
/* TX, output (struct grspw_rtxpkt) */
#define GRSPW_TXPKT_TX  0x4000 /* Transmitted */
#define GRSPW_TXPKT_LINKERR 0x8000 /* Link error */

/* RX, input (struct grspw_wrxpkt) */
#define GRSPW_RXPKT_IE  0x0010 /* Interrupt when received */
#define GRSPW_RXPKT_INPUT_MASK GRSPW_RXPKT_IE
/* RX, output (struct grspw_rrxpkt) */
#define GRSPW_RXPKT_EEOP 0x0100 /* Error in End-of-Packet */
#define GRSPW_RXPKT_HCRC 0x0200 /* Header CRC error (RMAP CRC enabled) */
#define GRSPW_RXPKT_DCRC 0x0400 /* Data CRC error (RMAP CRC enabled) */
#define GRSPW_RXPKT_TRUNK 0x0800 /* Truncated */
#define GRSPW_RXPKT_RX  0x8000 /* Received */
```

2.3.1. Packet Reception

When the SpaceWire link is in run state and DMA operation has been started from the control interface, packets buffers can be scheduled for future reception. There are two different states of a DMA channel, when descriptors has been prepared and enabled for transmission and when there are no enabled descriptors (out of buffers). In the latter case the core can be programmed to discard incoming packets or to wait for new enabled descriptors (packet buffers), that is controlled through the control interface (see NO-SPILL option in GRSPW hardware documentation).

Packet reception basically comes down to enabling descriptors with new empty buffers. The driver must process the core's descriptor table to handle received SpaceWire packets and enable unused descriptors with new packet buffers. That process might be triggered in two different ways:

- DMA receive interrupt, the driver will schedule work to process the descriptor table later on in non-interrupt context.

- The user calls RX(PREPARE) or RX(RECEIVE).

The user can configure the behavior of the first case by controlling how interrupts are generated. The driver can generate interrupt after every N number of packets have been received. The user can also control it completely custom by setting N=0 and enabling interrupts on a packet basis, see RX(PREPARE). If the driver is not able to process the RX descriptor table in time the transfer rate will drop (or packets will be discarded). Since the user might not be able to call RX(PREPARE) and RX(RECEIVE) often enough on high bit rates (or small packets) the DMA receiver interrupts can be used to start processing of descriptors. On DMA receive interrupt the driver will schedule a work queue that will process the descriptor table, in order to enable new packet buffers the user must have prepared buffers on beforehand. Prepared packets will be buffered temporarily in the READY queue until unused descriptors are available. Received packets will be buffered in the same order as the SpaceWire packets was received in the RECV queue. See Figure 2.1. Note that if N is set to a higher number than the number of RX descriptors (128) or when it is disabled, the descriptor table may not contain any enabled descriptors until RX(PREPARE) or RX(RECEIVE) is called by the user.

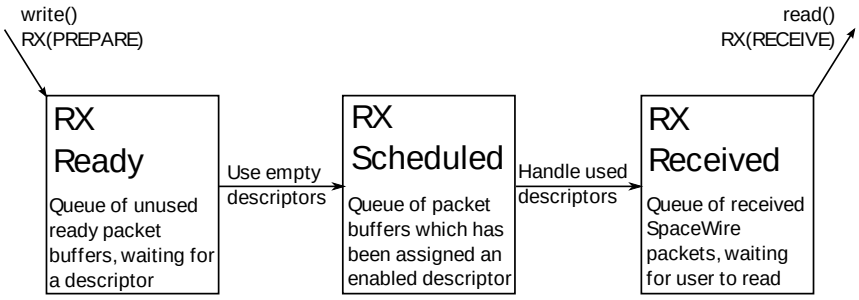

Figure 2.1. GRSPW Driver internal RX queues

The driver internal RX queues are all link lists of FIFO type. The RX-schedule queue can hold a maximum of 128 (number of descriptors supported by GRSPW at time of writing) packets, the other queues does not have any limitation except from the number of packet structures that the driver use internally to describe the packets. The number of packet structures can be configured through the control interface.

2.3.1.1. RX(PREPARE)

The process of preparing the GRSPW driver with new packet buffers is called RX(PREPARE) in this document. It is done with the RX_PREPARE command and one or an array of struct grspw_wrxpkt entries. Each entry describes one packet buffer, see below programlisting and table. If the driver is out of packet structures (used internally in driver) all packet buffers will not be prepared, instead the returned `count` determines how many packets was added to the ready queue.

```
/* RX_PREPARE: a buffer to receive into */
struct grspw_wrxpkt {
 __u64 user_data; /* Returned unchanged by RX_RECV */
 __u64 data_off;  /* Buffer offset into the pool */
 __u32 len;  /* Buffer size, 0 for the channel's rxmaxlen.
     * Must be 0 or at least rxmaxlen.
     */
 __u32 flags;  /* GRSPW_RXPKT_* input flags */
};
```

Table 2.5. GRSPW prepare RX buffers write format (struct grspw_wrxpkt)

Field	Description
`user_data`	A user defined 64-bit value, for example a pointer, which can be used to identify the packet buffer upon RX(RECEIVE). It is returned unchanged and does not affect the operation of the driver.
`data_off`	Byte offset into the pool dma-buf (set via BUFCFG) of the packet buffer that the driver will store one received SpaceWire Packet to. The `rxmaxlen` bytes from the offset (see CONFIG_SET) must be within the pool and physically contiguous, otherwise EINVAL is returned.
`len`	Size of the buffer. 0 means the `rxmaxlen` of the DMA channel. Any other value must be at least `rxmaxlen`, otherwise EINVAL is returned.

Field	Description
flags	Set to GRSPW_RXPKT_IE if this packet should generate a interrupt when a SpaceWire packet was received to this packet buffer, otherwise 0. Interrupts can be controlled using the control interface.

2.3.1.2. RX(RECEIVE)

After packet buffers have been prepared, assigned a descriptor, a SpaceWire packet received, the packet taken from the descriptor and put into the receive queue of the driver, the packet can be read with the RX_RECV command. This process is called RX(RECEIVE) in this document. The driver will fill the user provided array with packet buffer information according to the struct grspw_rrxpkt memory layout. See below programlisting and table. Each entry describes one packet which may have a valid SpaceWire packet in the packet buffer pointed to be *data_off*.

```
/* RX_RECV: a received packet, or an unused buffer after STOP */
struct grspw_rrxpkt {
 __u64 user_data;
 __u64 data_off;  /* As given to RX_PREPARE */
 __u32 dlen;  /* Received length */
 __u32 flags;  /* GRSPW_RXPKT_* output flags */
 __u8 dma_chan;  /* DMA channel it was received on */
 __u8 reserved[7];
};
```

Table 2.6. GRSPW receive RX packet buffers read format (struct grspw_rrxpkt)

Field	Description
user_data	The user defined value that was given to the driver together with the packet buffer in RX(PREPARE).
data_off	Byte offset into the pool dma-buf of the packet buffer, as given in RX(PREPARE). The *flags* field bit GRSPW_RXPKT_RX is set if the buffer contains a SpaceWire packet.
dlen	The length of SpaceWire packet that was received into the packet buffer pointed to by *data_off*.
flags	This field indicates if the data buffer contains a SpaceWire packet (GRSPW_RXPKT_RX), and if transfer errors where encountered during the reception: truncated (GRSPW_RXPKT_TRUNK), error in End-of-Packet (GRSPW_RXPKT_EEOP), header CRC error (GRSPW_RXPKT_HCRC) and data CRC error (GRSPW_RXPKT_DCRC).
dma_chan	Indicates which DMA channel (0..3) received this packet.
reserved	Zero.

2.3.2. Packet Transmission

The packet transmission interface works basically the same as the packet reception interface, with the TX_SEND and TX_RECLAIM commands instead of RX_PREPARE and RX_RECV. See the previous RX section introduction.

The packet queues are named differently as indicated in Figure 2.2, the TX scheduled queue also fits as many packets as there are descriptors, however the TX descriptors are 64 in number instead of 128 for RX.

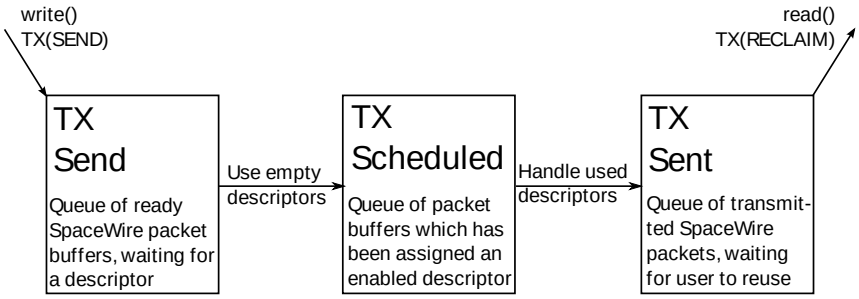

Figure 2.2. GRSPW Driver internal TX queues

2.3.2.1. TX(SEND)

The process of sending a SpaceWire packet (data and header) is called TX(SEND) in this document. A packet is sent with the TX_SEND command and one or an array of struct grspw_wtxpkt entries. Each entry describes one packet buffer, see below programlisting and table. If the driver is out of packet structures (used internally in driver) all packets will not be sent, instead the returned `count` determines how many packets was added to the send queue.

```
/* TX_SEND: a packet to send */
struct grspw_wtxpkt {
 __u64 user_data; /* Returned unchanged by TX_RECLAIM */
 __u64 hdr_off;  /* Header offset into the pool */
 __u64 data_off;  /* Data offset into the pool */
 __u32 dlen;  /* Data length, 0 for none */
 __u32 flags;  /* GRSPW_TXPKT_* input flags */
 __u8 hlen;  /* Header length, 0 for none */
 __u8 reserved[7];
};
```

Table 2.7. GRSPW send TX packet buffers write format (struct grspw_wtxpkt)

Field	Description
`user_data`	A user defined 64-bit value, for example a pointer, which can be used to identify the packet buffer upon TX(RECLAIM). It is returned unchanged and does not affect the operation of the driver.
`hdr_off`	Byte offset into the pool dma-buf of the packet header buffer. This is only used if `hlen` is larger than zero. The first `hlen` bytes are transmitted.
`data_off`	Byte offset into the pool dma-buf of the packet buffer that contains the data of one SpaceWire packet. This is only used if `dlen` is larger than zero. The first `dlen` bytes are transmitted.
`dlen`	Determines the length of the data that will be transmitted, zero if none. The maximum length is limited by the 24-bit length field of the hardware, and the data buffer must be physically contiguous within the pool dma-buf.
`flags`	This field hold the transmission options for one SpaceWire packet: interrupt when sent (`GRSPW_TXPKT_IE`), append a header CRC (`GRSPW_TXPKT_HCRC`) or a data CRC (`GRSPW_TXPKT_DCRC`), and the number of header bytes left out of the header CRC (`GRSPW_TXPKT_NOCRC_LEN(n)`, 0 to 15).
`hlen`	Determines the length of the header, set to zero if no header should be transmitted. At most 255 bytes.
`reserved`	Must be zero.

A header or data range that is not within the pool or not physically contiguous gives EINVAL, as does an unknown flag or a nonzero reserved field.

2.3.2.2. TX(RECLAIM)

After packet buffers have been request to be sent, assigned a descriptor, a SpaceWire packet generated and transmitted, the packet buffer taken from the descriptor and put into the sent queue of the driver, the packet buffer can be read with the TX_RECLAIM command. This process is called TX(RECLAIM) in this document. The driver will fill the user provided array with packet buffer information according to the struct grspw_rtxpkt memory layout. See below programlisting and table. Each entry describes one packet which may have been successfully sent.

```
/* TX_RECLAIM: a packet that was sent, or not sent after STOP */
struct grspw_rtxpkt {
 __u64 user_data;
 __u32 flags;  /* GRSPW_TXPKT_* output flags */
 __u8 dma_chan;  /* DMA channel it was queued on */
 __u8 reserved[3];
};
```

Table 2.8. GRSPW reclaim TX packet buffers read format (struct grspw_rtxpkt)

Field	Description
`user_data`	The user defined value that was given to the driver together with the packet in TX(SEND).

Field	Description
flags	This field hold the transmission parameters for one SpaceWire packet. If the the packet was sent (a descriptor with the data/header was enabled) the GRSPW_TXPKT_TX bit is set, if a link error occurred GRSPW_TXPKT_LINKERR bit is set.
dma_chan	Indicates which DMA channel (0..3) this packet was sent on.
reserved	Zero.

2.4. User-space access routines

In order to access the GRSPW SpaceWire driver the user application must call it using the standard UNIX system calls (open, ioctl, close). To simplify that task an API is provided part of the examples in the GRLIB driver package. The API provides means to access the driver by an easy to use API rather than letting the user application making the UNIX calls directly. The API also tries to simplify SpaceWire packet buffer handling and buffer management by use of buffer pools. At the same time the API provides an example how the driver can be called.

The API is declared in `grspwlib.h` and `spwlib.h`.

The API is undocumented since it is by itself considered as documentation/example.

---

3. GRSPW Kernel Library driver

3.1. Introduction

This section describes the GRSPW Kernel Library driver for Linux. Its interface is only accessible from kernel space. Most of the functionality is exported to user space as described in Chapter 2.

It is an advantage to understand the SpaceWire bus/protocols, GRSPW hardware and software driver design when developing using the user interface in Section 3.3 and Section 3.4. The Section 3.2.1 describes the overall software design of the driver.

3.1.1. Hardware Support

The GRSPW cores user interface are documented in the GRIP Core User's manual. Below is a list of the major hardware features it supports:

- GRSPW, GRSPW2 and GRSPW2_DMA (router AMBA port)
- Multiple DMA channels
- Time Code
- Link Control
- Port Control
- RMAP Control
- SpaceWire Interrupt codes
- Interrupt handling
- Multi-processor SMP support

3.1.2. Driver sources

The driver sources and header files are listed in Section 2.1.1.

3.1.3. Examples

The GRSPW SpaceWire driver and its samples are examples of how the GRSPW Kernel Library driver can be used. See Section 2.1.4.

3.1.4. Known driver limitations

The known limitations in the GRSPW Packet driver exists listed below:

- The statistics counters are not atomic, clearing at the same the interrupt handler is called could cause invalid statistics, one must disable interrupt when reading/clearing (a SMP problem).
- The SpaceWire Interrupt code support is not available yet.

3.2. Software design overview

3.2.1. Overview

The driver has been implemented using the platform device driver model. The driver provides a kernel function interface, an API, rather than implementing a IO system device. The API is intended for kernel tasks but has been designed so that a custom interface for processes can be implemented on top of the kernel space API, see Chapter 2. The driver can be compiled as a kernel module and loaded into the kernel at run-time or linked with the kernel at compile-time. The installation steps required for linking with kernel are described in the `kernel/drivers/grlib/README`.

The driver API has been split up in two major parts listed below:

- Device interface, see Section 3.3.
- DMA channel interface, see Section 3.4.

GRSPW device parameters that affects the GRSPW core and all DMA channels are accessed over the device API whereas DMA specific settings and buffer handling are accessed over the per DMA channel API. A GRSPW2 device may implement up to four DMA channels.

In order to access the driver the first thing is to open a GRSPW device using the device interface.

For controlling the device one must open a GRSPW device using `'id = grspw_open(dev_index)'` and call appropriate device control functions. Device operations naturally affects all DMA channels, for example when the link is disabled all DMA activity pause. However there is no connection software wise between the device functions and DMA function, except from that the last `grspw_close` requires that all of its DMA channels have been closed. Closing a device fails if DMA channels are still open.

Several kernel users, for example the user space driver described in Chapter 2 and other kernel modules, may use the same GRSPW device at the same time. The device is opened by each user and is reference counted, whereas a DMA channel can only be opened by one user at a time. A kernel driver that takes one DMA channel of a device opens it when it binds and keeps it open; the user space driver then reports the channel as not available and leaves it alone, see `grspw_dma_opened`. Kernel users register to be notified about GRSPW devices, see Section 3.3.2.

GRSPW devices can not be unbound from the driver through sysfs.

Packets are transferred using DMA channels. To open a DMA channel one calls `'dma_id = grspw_dma_open(id, dmachan_index)'` and use the appropriate transmission function with the *`dma_id`* to identify which DMA channel used.

3.2.2. Initialization

During early initialization when the operating system boots the driver performs some basic GRSPW device and software initialization. The following steps are performed or not performed:

- GRSPW device and DMA channels I/O registers are initialized to a state where most are zero.
- DMA is stopped on all channels
- Link state and settings are not changed (RMAP may be active).
- RMAP settings untouched (RMAP may be active).
- Port select untouched (RMAP may be active).
- Time Codes are disabled and TC register cleared.
- IRQ generation disabled.
- Status Register cleared.
- Node address / DMA channels node address is untouched (RMAP may be active).
- Hardware capabilities are read.
- Device index determined.

3.2.3. Link control

The GRSPW link interface handles the communication on the SpaceWire network. It consists of a transmitter, receiver, a FSM and FIFO interfaces. The current link state, status indicating past failures, parameters that affect the link interface such as transmitter frequency for example is controlled using the GRSPW register interface.

The SpaceWire link is controlled using the software device interface. The driver initialization sequence during boot does not affect the link parameters or state. The link is controlled separately from the DMA channels, even though the link goes out from run-mode this does not affect the DMA interface. The DMA activity of all channels are of course paused. It is possible to configure the driver to disable the link on certain error interrupts.

The link can be disabled when a link error is detected by the GRSPW interrupt handler. There are two options which can be combined, either the DMA transmitter is disabled on error (disabled by hardware) or the software interrupt handler disables the link on link error events selected by the user. When software disables the link the work queue is informed and stops all DMA channels, thus `grspw_dma_stop()` is called for each DMA channel by the work queue. The GRSPW interrupt handler will disable the link by writing "Link Disable" bit and clearing "Link Start" bit on link errors. The user is responsible to restart the link interface again. The status register (`grspw_link_status()`) and statistics interface can be used to determine which error(s) happened. The two options are configured by the link control interface of the device API using function `grspw_link_ctrl()`.

To make hardware disable the DMA transmitter automatically on error the option (*`LINKOPTS_DIS_ONERR`*) is used.

To activate the GRSWP interrupt routine when any link error occurs, the bitmask option *Enable Error Link IRQ* (*`LINKOPTS_EIRQ`*) shall be set. The bitmask options described as *Disable Link on XX Error*

(*LINKOPTS_DIS_ON_**) are used to select which events shall actually cause link disable in the interrupt routine and inform the work queue of a shutdown stop.

The options *LINKOPTS_DIS_ON** are in effect even when the option *LINKOPTS_EIRQ* is disabled. Thus, an interrupt routine invocation caused by a DMA channel interrupt event may disable the link in case any of the conditions in *LINKOPTS_DIS_ON_** are satisfied.

Statistics about the link errors can be read from the driver, see Section 3.3.9.

It is possible to circumvent the drivers action of clearing link status events in the GRSPW status register from the interrupt routine. This can be used for example when the user wants to detect and handle all occurrences of a specific link event. The function `grspw_link_ctrl()` is used to configure this via the *stscfg* parameter with values *LINKSTS_**. If a bit is set in this configuration parameter, the corresponding bit in the GRSPW status register is cleared by the interrupt routine. If the bit is not set, the interrupt routine will never clear the status flag and the user has full control of it. The status event can then be manually read and cleared with functions `grspw_link_status()` and `grspw_link_status_clr()`.

Statistics counters for events which are configured to be circumvented by the driver, as described above, shall not be relied upon.

Function names prefix: `grspw_link_*()`.

3.2.4. Time Code support

The GRSPW supports sending and receiving SpaceWire Time Codes. An interrupt can optionally be generated on Time Code reception and the last Time Code can be read out from a GRSPW register.

The GRSPW core's Time Code interface can be controlled from the device API. One can generate Time Codes and read out the last received or generated Time Code. An user assignable interrupt handler can be used to detect and handle Time Code reception, the callback is called from the GRSPW interrupt routine thus from interrupt context.

Function names prefix: `grspw_tc_*()`

3.2.5. RMAP support

The GRSPW device has optional support for an RMAP target implemented in hardware. The target interface is able to interpret RMAP protocol (*protid*=1) requests, take the necessary actions on the AMBA bus and generate a RMAP response without the software's knowledge or interaction. The RMAP target can be disabled in order to implement the RMAP protocol in software instead using the DMA operations. The RMAP CRC algorithm optionally present in hardware can also be used for checksumming the data payload.

The device interface is used to get the RMAP features supported by the hardware and configuring the below RMAP parameters:

- Probe if RMAP and RMAP CRC is supported by hardware
- RMAP enable/disable
- SpaceWire DESTKEY of RMAP packets

The SpaceWire node address, which also affects the RMAP target, is controlled from the address configuration routines, see Section 3.2.7.

Function names prefix: `grspw_rmap_*()`

3.2.6. Port support

The GRSPW device has optional support for two ports (two connectors), where only one port can be active at a time. The active SpaceWire port is either forced by the user or auto selected by the hardware depending on the link state of the SpaceWire ports at a certain condition.

The device interface is used to get information about the GRSPW hardware port support, current set up and to control how the active port is selected.

Function names prefix: `grspw_port_*()`

3.2.7. SpaceWire node address configuration

The GRSPW core supports assigning a SpaceWire node address or a range of addresses. The address affects the received SpaceWire Packets, both to the RMAP target and to the DMA receiver. If a received packet does not match the node address it is dropped and the GRSPW status indicates that one or more packets with invalid address was received.

The GRSPW2 and GRSPW2_DMA cores that implements multiple DMA channels use the node address as a way to determine which DMA channel a received packet shall appear at. A unique node address or range of node addresses per DMA channel must be configured in this case.

It is also possible to enable promiscuous mode to enable all node addresses to be accepted into the first DMA channel, this option does not affect the RMAP target node address decoding.

The GRSPW SpaceWire node address configuration is controlled using the device interface. A single DMA channel's node address can also be configured with `grspw_dma_addr_ctrl`, without affecting other DMA channels.

If supported by hardware the node address can be removed before DMA writes the packet to memory. This is a configuration option per DMA channel using the DMA channel API.

Function names prefix: `grspw_addr_*()`

3.2.8. User DMA buffer handling

The driver is designed with zero-copy in mind. The user is responsible for setting up data buffers on its own . The driver uses linked lists of packet buffers as input and output from/to the user. It makes it possible to handle multiple packets on a single driver entry, which typically has a positive impact when transmitting small sized packets.

The API supports header and data buffers for every packet, and other packet specific transmission parameters such as generate RMAP CRC and reception indicators such as if packet was truncated.

Since the driver never reads or writes to the header or data buffers the driver does not affect the CPU cache of the DMA buffers, it is the user's responsibility to handle potential cache effects.

Note that the UT699 does not have D-cache snooping, this means that when reading received buffers D-cache should either be invalidated or the load instructions should force cache miss when accessing DMA buffers (LEON LDA instruction) or map the packet buffer DMA pages non-cacheable using the MMU .

Function names prefix: `grspw_dma_*()`

3.2.8.1. Buffer List help routines

The GRSPW packet driver internally uses linked lists routines. The linked list operations are found in the header file and can be used by the user as well. The user application typically defines its own packet structures having the same layout as struct grspw_pkt in the top and adding custom fields for the application buffer handling as needed. For small implementations however the *`pkt_id`* field may be enough to implement application buffer handling. The *`pkt_id`* field is never accessed by the driver, instead is an optional application 32-bit data storage intended for identifying a specific packet, which packet pool the packet buffer belongs to, or a higher level protocol id information for example.

Function names prefix: `grspw_list_*()`

3.2.9. Driver DMA buffer handling

The driver allocates memory for DMA descriptor tables using Linux cohoerent memory allocation services `dma_alloc_coherent()` to map physical address space non-cachable for the DMA tables.

The driver represents packets with the struct grspw_pkt packet structure, see Table 3.36. They are arranged in linked lists that are called queues by the driver. The order of the linked lists are always maintained to ensure that the packet transmission order is represented correctly.

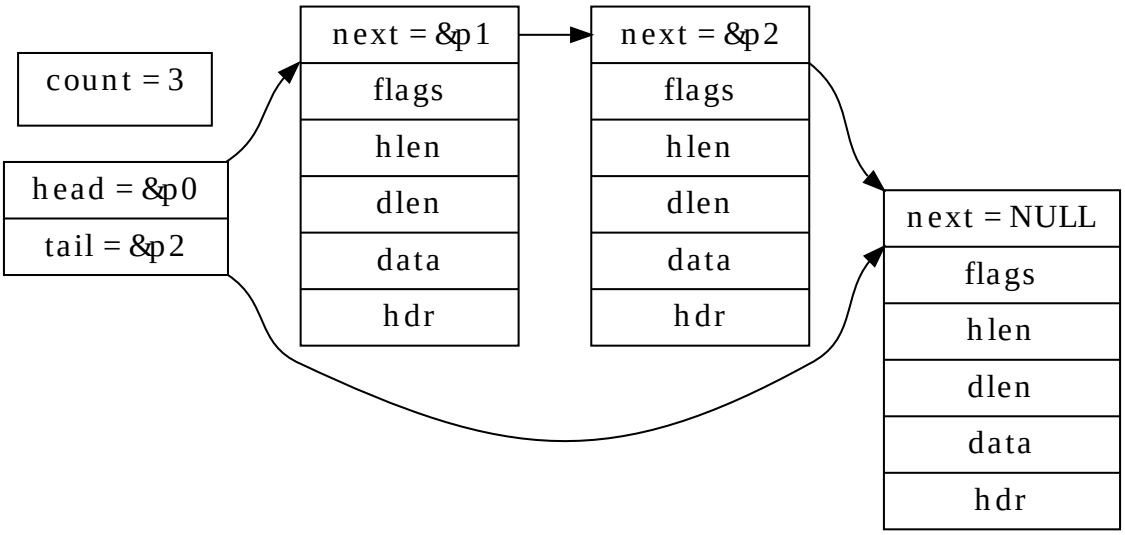

Figure 3.1. Queue example - linked list of three grspw_pkt packets

3.2.9.1. DMA Queues

The driver uses three queues per DMA channel transfer direction, thus six queues per DMA channel. The number of packets within a queue is maintained to optimize moving packets internally between queues and to the user which also needs this information. The different queues are listed below.

- RX READY queue - free packet buffers provided by the user.
- RX SCHED queue - packets that have been assigned a DMA descriptor.
- RX RECV queue - packets containing a received packet.
- TX SEND queue - user provided packets ready to be sent.
- TX SCHED queue - packets that have been assigned a DMA descriptor.
- TX SENT queue - packets sent

Packet in the SCHED queues always are assigned to a DMA descriptor waiting for hardware to perform RX or TX DMA operations. There is a limited number of DMA descriptor table, 64 TX or 128 RX descriptors. Naturally this also limits the number of packets that the SCHED queues contain simultaneously. The other queues does not have any maximum number of packets, instead it is up to the user to handle the sizing of the RX READY, RX RECV, TX SEND and TX SENT packet queues by controlling the input and output to them. Thereby it is possible to queue packets within the driver. Since the driver can move queued packets itself it can makes sense to queue up free buffers in the RX READY queue and TX SEND queue for future transmission.

The current number of packets in respective queue can be read by doing function calls using the DMA API, see Section 3.4.7. The user can for example use this to determine to wait or continue with packet processing.

3.2.9.2. DMA Queue operations

The user can control how the RX READY and TX SEND queue is populated, by providing packet buffers. The user can control how and when packets are moved from RX READY and TX SEND queues into the RX SCHED or TX SCHED by enabling the work queue and interrupt driven DMA or by manually trigger the moving calling reception and transmission routines as described in Section 3.4.6 and Section 3.4.5.

The packets always flow in one direction from RX READY -> RX SCHED -> RX RECV. Likewise the TX packets flow TX SEND -> TX SCHED -> TX SENT. The procedures triggering queue packet moves are listed below and in Figure 3.2 and Figure 3.3. The interface of theses procedures are described in the DMA channel API.

- USER -> RX READY queue - rx_prepare, Section 3.4.6.

- RX RECV -> USER - rx_recv, Section 3.4.6.
- USER -> TX SEND - tx_send, Section 3.4.5.
- TX SEND -> USER - tx_reclaim, Section 3.4.5.

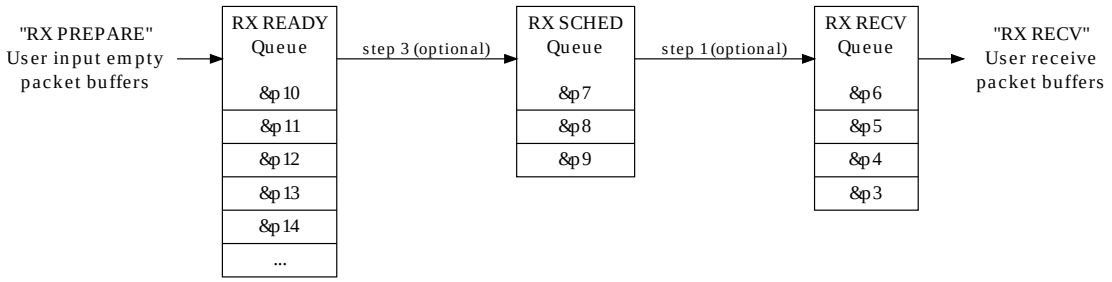

Figure 3.2. RX queue packet flow and operations

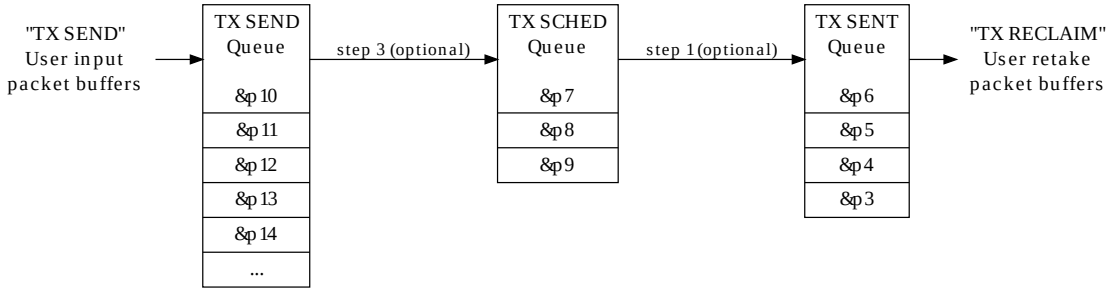

Figure 3.3. TX queue packet flow and operations

3.2.10. Polling and blocking mode

Both polling and blocking transfer modes are supported. Blocking mode is implemented using DMA interrupt and a work queue for processing the descriptor tables to avoid loading the CPU in interrupt context. The DMA interrupt queues DMA jobs by using work queues. In polling mode the user is responsible for processing the DMA descriptor tables at a user defined interval by calling reception and transmit routines of the driver.

DMA interrupt is generated every N received/transmitted packets or controlled individually per packet. The latter is configured in the packet data structures and the former using the DMA channel configuration. See Section 3.4.3 and Section 3.4.9 for more information.

Blocking mode is implemented by letting the user setting up a condition on the RX or TX DMA queues packet counters. The condition can optionally be timed out protected in a number of ticks, implemented by the semaphore service provided by the operating system. Each time after the work queue has completed processing the DMA descriptor table the condition is evaluated. If considered true then the blocked task is woken up by signaling on the semaphore the task is waiting for. There is only one RX and one TX condition per channel, thus only two tasks can block at a time per channel.

Blocking function names: `grspw_dma_{tx,rx}_wait()`

3.2.11. Interrupt and work queue

The driver can optionally spawn jobs on a work queue that is used service the GRSPW devices. The work queue execution is triggered from the GRSPW ISR at certain user configurable events, at link errors or DMA transmissions completed. When the a job has been scheduled on the work queue for a specific device or DMA channel the ISR has turned off the specific interrupt source that the job will handle, once the job has been completed the job re-enables interrupt source again. This is to lower the number of interrupts.

If a DMA event callback has been registered for a DMA channel the ISR calls the callback instead of scheduling the DMA job of that channel. The user is then responsible for the DMA descriptor processing, see Section 3.4.10.

The work queue can also be used to automatically stop DMA operation on certain link errors. This feature is enabled by activating the different *Disable Link on XX Error* (`LINKOPTS_DIS_ON_*`) options from the device API link control interface. See Section 3.2.3. For the configured link errors the GRSPW interrupt handler will trigger the shutdown work to start which will stop all DMA channels by calling `grspw_dma_stop()`.

3.2.12. Starting and stopping DMA

The driver has been designed to make it clear which functionality belongs to the device and DMA channel APIs. The DMA API is affected by started and stopped mode, where in stopped mode means that DMA is not possible and used to configure the DMA part of the driver. During started mode a DMA channel can accept incoming and send packets. Each DMA channel controls its own state. Parts of the DMA API is not available in during stopped mode and some during stopped mode to simplify the design. The device API is not affected by this.

Typically the DMA configuration is set and user buffers are initialized before DMA is started. The user can control the link interface separately from the DMA channel before and during DMA starts.

When the DMA channel is stopped by calling `grspw_dma_stop()` the driver will:

- Stop DMA transfers and DMA interrupts.
- Stop accepting new packets for transmission and reception. However the DMA functions will still be open for the user to retrieve sent and unsent TX packet buffers and to retrieve received and unused RX packet buffers.
- Wake up blocked DMA threads and return to the caller. Tasks can be blocked waiting for TX/RX event by using the TX/RX DMA wait functions.

The DMA close routines requires that the DMA channel is stopped. Similarly, the device close routine makes sure that all DMA channels are closed to be successful. This is to make sure that all user tasks has return and hardware is in a good state. It is the user's responsibility to stop the DMA channel before closing.

DMA operational function names: `grspw_dma_{start,stop}()`

3.2.13. Thread concurrency

The driver has been designed to allow multi-threading. There are five parts that can be operated simulaneously by different or the same thread(s):

- Device (link control) interface.
- DMA RX channel.
- DMA TX channel.
- work queue is a separate thread of execution.
- Interrupt Service Routine.

There may be multiple DMA channels in a GRSPW device. DMA channels are operated independently of each other. Each DMA channel has two spin-locks, RX and TX, to allow operations on different DMA channels simultaneously as well as simultaneous RX and TX operations on the same DMA channel. Multiple tasks using the same RX or TX interface of the same DMA channel is possible but will temporarily lock each other out during register and DMA descriptor table processing. The same spin-locks are taken by the work queue during DMA processing. The spin-locks are taken with interrupts disabled. There is a global device semaphore that serializes `grspw_open`, `grspw_close` and `grspw_dma_open`. The DMA channels and device interface share the same GRSPW I/O registers, they are protected from each other and from the ISR by a per-device spin-lock taken with interrupts disabled.

Each DMA channel also has two semaphores, RX-WAIT and TX-WAIT, to implement blocking on RX/TX operations. The DMA RX/TX interrupt wakes a worker which processes the DMA RX/TX descriptor tables and signals via the RX-WAIT and TX-WAIT that incoming/outgoing packets processing has finished.

Since the DMA channel locks are spin-locks, the DMA functions do not sleep and may be called from atomic context, for example from a softirq or NAPI poll function. The exceptions are `grspw_dma_open`, `grspw_dma_rx_wait` and `grspw_dma_tx_wait` which may sleep. The device functions `grspw_open` and `grspw_close` may also sleep. No GRSPW function may be called from a DMA event callback, see Section 3.4.10.

The table below summarises the lock operations of a DMA channel that the driver makes.

Table 3.1. DMA channel lock operations.

Function	Operation	Lock	Description
`dma_open`	Init locks	RX TX RX-WAIT TX-WAIT	RX and TX spin-locks are initialized. RX-WAIT and TX-WAIT are initialized to 0 (locked).
`dma_close`	Close channel	RX TX	Both RX and TX spin-locks are taken. The close fails if the channel is started or a task is waiting, otherwise the channel is marked closed. The locks are released in both cases. After a successful close the user can not enter other DMA functions than `dma_open`.
`dma_start`	Init semaphores	RX TX RX-WAIT TX-WAIT	With both RX and TX spin-locks taken the wait semaphores are initialized to 0 (locked) state. Fails with -EAGAIN if a task is still waiting. From this point onwards the RX/TX wait interface is available.
`dma_stop`	Shutdown DMA	RX TX RX-WAIT TX-WAIT	The RX and TX spin-locks are both held during the whole stop operation. The RX-WAIT and TX-WAIT semaphores are signalled in order for potential blocked tasks to be woken up and return to caller indicating DMA stopped (1).

`dma_rx_recv`
`dma_rx_prepare`
`dma_rx_count`
	RX DMA operations	RX	Holds the RX spin-lock while performing RX operations.

`dma_tx_send`
`dma_tx_reclaim`
`dma_tx_count`
	TX DMA operations	TX	Holds the TX spin-lock while performing TX operations.

`dma_tx_wait`	Wait for TX DMA.	TX TX-WAIT	Takes the TX spin-lock to initialize the wait structures and releases it before sleeping. TX-WAIT is taken to block the calling thread until the worker, DMA shutdown or timeout awakens the thread again.
`dma_rx_wait`	Wait for RX DMA.	RX RX-WAIT	Takes the RX spin-lock to initialize the wait structures and releases it before sleeping. RX-WAIT is taken to block the calling thread until the worker, DMA shutdown or timeout awakens the thread again.
DMA work	Normal DMA descriptor list processing.	RX TX RX-WAIT TX-WAIT	RX and TX locks taken in sequence. RX-WAIT and TX-WAIT given on matching conditions after the locks are released.
DMA work error	DMA AHB error handling.	RX TX	DMA RX/TX AHB errors leads to calling `grspw_dma_stop()` for one DMA channel. The work queue does not hold any locks itself.
Link work error	Link error handling.	RX TX	SpaceWire link errors configured to generate interrupt may be handled by worker to call `grspw_dma_stop()` for all DMA channels.

3.2.14. SMP Support

The driver has been designed with SMP in mind. Data structures, interrupt handling routine and GRSPW control register accesses are always protected by spin-locks taken with interrupts disabled.

The design using a worker task off-loads the interrupt handler and makes it possible to control which CPU (with CPU affinity in the scheduler) that should handle the descriptor table processing.

As described in Section 3.1.4 the SMP support requires testing.

3.2.15. User space support

The driver has been designed for kernel space where pointers and memory addresses are being exchanged with the API user and trusted. In Chapter 2 it is described how this driver can be used indirectly from user space. Packet buffers are allocated by the application as a dma-buf from a CMA dma-heap and mapped into user space with `mmap()`. The driver imports the dma-buf for DMA, which allows an efficient zero-copy implementation.

3.3. Device Interface

This section covers how the driver can be interfaced to an application to control the GRSPW hardware.

3.3.1. Opening and closing device

A GRSPW device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `grspw_dev_count`. A particular device can be opened using `grspw_open` and closed `grspw_close`. The functions are described below.

A device may be opened several times, for example by different kernel users. Each open increments a reference count and returns the same device handle, and must be matched by a close. The device is released by the last close, which fails if any DMA channel of the device is still open. This procedure is thread-safe by protecting from other threads by using the GRSPW driver's semaphore lock. The semaphore is used by all GRSPW devices on device opening, closing and DMA channel opening.

During the first opening of a GRSPW device the following steps are taken, later openings only increment the reference count:

- GRSPW device I/O registers are initialized to a state where most are zero.
- Descriptor tables memory for all DMA channels are allocated from the coherent DMA allocation service of Linux which provides non-cacheable linear memory address space. The descriptor table length is always the maximum 0x400 Bytes for RX and TX.
- Internal resources like spin-locks and data structures are initialized.
- The driver is configured to clear all link status events from the ISR, the Time Code ISR is removed and the device statistics are cleared.

The example below prints the number of GRSPW devices to screen then opens, prints the current link settings and closes the first GRSPW device present in the system.

```
int print_spw_link_properties()
{
  void *device;
  int count, options, clkdiv;

  count = grspw_dev_count();
  printf("%d GRSPW device present\n", count);

  device = grspw_open(0);
  if (!device)
   return -1; /* Failure */

  options = clkdiv = -1;
  grspw_link_ctrl(device, &options, NULL, &clkdiv);
  if (options & LINKOPTS_AUTOSTART) {
   printf("GRSPW0: Link is in auto-start after start-up\n");
  }
  printf("GRSPW0: Clock divisor reset value is %d\n", clkdiv);

  grspw_close(device);
  return 0; /* success */
}
```

Table 3.2. `grspw_dev_count` function declaration

Proto	`int grspw_dev_count(void)`
About	Retrieve number of GRSPW devices registered to the driver.
Return	int. Number of GRSPW devices registered in system, zero if none.

Table 3.3. `grspw_open` function declaration

Proto	`void *grspw_open(int dev_no)`	
About	Opens a GRSPW device. The GRSPW device is identified by index. The returned value is used as input argument to all functions operating on the device. Several kernel users may open the same device. Each call increments a reference count and returns the same handle, and each successful call must be matched by a call to `grspw_close`. Only the first open initializes the device, see Section 3.3.1.	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the registration order to the driver, normally dictated by the Plug & Play order. Must be equal or greater than zero, and smaller than that returned by `grspw_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if *`dev_no`* is invalid or if the descriptor table memory could not be allocated.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which GRSPW device.
Notes	May blocking until other GRSPW device operations complete.	

Table 3.4. `grspw_close` function declaration

Proto	`int grspw_close(void *d)`	
About	Closes a previously opened device. The reference count taken by `grspw_open` is decremented. The device is only released by the last close, which requires that all DMA channels have been stopped and closed. See the documentation for `grspw_dma_stop` and `grspw_dma_close`.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Return	Value. Description	
	0	Success. The reference count was decremented, and the device was released if this was the last close.
	1	Failure. The device is not open, or this is the last close and a DMA channel of the device is still open. In the latter case the device remains open.

3.3.2. Kernel users

A kernel module using the GRSPW Kernel Library registers itself with `grspw_initialize_user` to be notified when GRSPW devices are found and removed, and unregisters with `grspw_uninitialize_user`, typically when the module is unloaded. Several kernel users can be registered at the same time, for example the user space driver described in Chapter 2 and a custom kernel module. They may open the same GRSPW device, but must use different DMA channels.

The Linux device of a GRSPW core, needed to map packet buffers for DMA, is returned by `grspw_get_device` and `grspw_get_device_by_index`.

Table 3.5. `grspw_initialize_user` function declaration

Proto	`int grspw_initialize_user(void *(*devfound)(int), void (*devremove)(int, void *))`
About	Registers a kernel user of the GRSPW driver, for example a driver that implements an interface on top of the GRSPW Kernel Library. Several users may be registered at the same time. *`devfound`* is called once for every GRSPW device already present and then for every GRSPW device probed later. *`devremove`* is called when a device is removed, or when the user is unregistered with `grspw_uninitialize_user`, for every device where *`devfound`* returned a non-NULL value.

Param	*devfound* [IN] pointer to function Called with the device index as argument. The returned value is passed to *devremove*. The function pointer also identifies the registration in grspw_uninitialize_user.
Param	*devremove* [IN] pointer to function Called with the device index and the value returned by *devfound* for that device. May be NULL.
Return	int. Return code as indicated below.

Value	Description
0	Success.
-ENOMEM	The registration could not be allocated.

Notes	May block. The callbacks are called with an internal mutex held and must not call grspw_initialize_user or grspw_uninitialize_user.

Table 3.6. grspw_uninitialize_user function declaration

Proto	`void grspw_uninitialize_user(void *(*devfound)(int))`
About	Unregisters a kernel user previously registered with grspw_initialize_user. The registered *devremove* function is called for every device where *devfound* returned a non-NULL value, before the registration is removed.
Param	*devfound* [IN] pointer to function The *devfound* function pointer given to grspw_initialize_user. Identifies the registration.
Return	None.
Notes	May block.

Table 3.7. grspw_get_device function declaration

Proto	`struct device *grspw_get_device(void *d)`
About	Returns the Linux platform device of a GRSPW device. It is the device to use when mapping packet buffers for DMA, for example with the DMA mapping API or when attaching a dma-buf.
Param	*d* [IN] pointer Device identifier. Returned from grspw_open.
Return	Pointer. The Linux device of the GRSPW core.

Table 3.8. grspw_get_device_by_index function declaration

Proto	`struct device *grspw_get_device_by_index(int dev_no)`
About	Same as grspw_get_device, but the GRSPW device is identified by index and does not need to be opened.
Param	*dev_no* [IN] Integer Device index. Must be equal or greater than zero, and smaller than that returned by grspw_dev_count.
Return	Pointer. The Linux device of the GRSPW core, or NULL if the index is invalid.

3.3.3. Hardware capabilities

The features and capabilities present in hardware might not be symmetric in a system with several GRSPW devices. For example the two first GRSPW devices on the GR712RC implements RMAP whereas the others does not. The driver can read out the hardware capabilities and present it to the user. The set of functionality are determined at design time. In some system where two or more systems are connected together it is likely to have different capabilities.

The capabilities are read out from the GRSPW I/O registers and written to the user in an easier accessible way. See below function declarations for details.

Depending on the capabilities parts of the API may be inactivated due to missing hardware support. See respective section for details.

The function `grspw_rmap_support` and `grspw_port_count` retrieves a subset of the hardware capabilities. They are described in respective section.

Table 3.9. `grspw_hw_support` function declaration

Proto	`void grspw_hw_support(void *d, struct grspw_hw_sup *hw)`
About	Read hardware capabilities of GRSPW device and write them in an easy to use format described by the grspw_hw_sup data structure. The data structure is described by Table 3.10.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*hw* [OUT] pointer Address to where the driver will write the hardware capabilities. Pointer must point to memory and be valid.
Return	None. Always success, input is not range checked.

The grspw_hw_sup data structure is described by the declaration and table below. It is used to describe the GRSPW hardware capabilities.

```
/* Hardware Support in GRSPW Core */
struct grspw_hw_sup {
 char rmap;  /* If RMAP in HW is available */
 char rmap_crc; /* If RMAP CRC is available */
 char rx_unalign; /* RX unaligned (byte boundary) access allowed*/
 char nports;  /* Number of Ports (1 or 2) */
 char ndma_chans; /* Number of DMA Channels (1..4) */
 char strip_adr; /* Hardware can strip ADR from packet data */
 char strip_pid; /* Hardware can strip PID from packet data */
 int hw_version; /* GRSPW Hardware Version */
 char reserved[2];
};
```

Table 3.10. grspw_hw_sup data structure declaration

Member	Description	
rmap	0	RMAP target functionality is not implemented in hardware.
	1	RMAP target functionality is implemented by hardware.
rmap_crc	Non-zero if RMAP CRC is available in hardware.	
rx_unalign	Non-zero if hardware can perform RX unaligned (byte boundary) DMA accesses.	
nports	Number of SpaceWire ports in hardware. Values: 1 or 2.	
ndma_chans	Number of DMA Channels in hardware. Values: 1,2,3 or 4.	
strip_adr	non-zero if GRSPW can strip ADR from packet data.	
strip_pid	non-zero if device can strip PID from packet data.	
hw_version	27..16	The 12-bits indicates GRLIB AMBA Plug & Play device ID of APB device. Indicates if GRSPW, GRSPW2 or GRSPW2_DMA.
	4..0	The 5 LSB bits indicates GRLIB AMBA Plug & Play device version of APB device. Indicates subversion of GRSPW or GRSPW2.
reserved	Not used. Reserved for future use.	

3.3.4. Link Control

The SpaceWire link is controlled and configured using the device API functions described below. The link control functionality is described in Section 3.2.3.

Table 3.11. `grspw_link_ctrl` function declaration

Proto	`void grspw_link_ctrl(void *d, int *options, int *stscfg, int *clkdiv)`

About	Read and configure link interface settings, such as clock divisor, link start and error options.
Param	*d* [IN] pointer Device identifier. Returned from grspw_open.
Param	*options* [IO] pointer to bitmask If *options* points to -1, the link options are only read from the I/O registers, otherwise they are updated according to the value in memory pointed to by *options*. Use LINKOPTS_* defines for *option* bit declarations. The masks for *LINKOPTS_DIS_ON** are in effect even when the option *LINKOPTS_EIRQ* is not enabled.

Bitmask	Description (prefixed LINKOPTS_)
DISABLE	Read/Set enable/disable link option.
START	Read/Set start link option.
AUTOSTART	Read/Set enable/disable link auto-start option.
DIS_ONERR	Read/Set *disable DMA transmitters when a link error occurs* option.
EIRQ	Read/Set interrupt generation on link error option.
DIS_ON_CE	Read/Set disable link on credit error option.
DIS_ON_ER	Read/Set disable link on escape error option.
DIS_ON_DE	Read/Set disable link on disconnect error option.
DIS_ON_PE	Read/Set disable link on parity error option.
DIS_ON_WE	Read/Set disable link on write synchronization error option (GRSPW1 only).
DIS_ON_EE	Read/Set disable link on early EOP/EEP error option.

Param	*stscfg* [IO] pointer to bitmask If *stscfg* points to -1, the link status configuration is only read, otherwise it is updated according to the value in memory pointer to by *stscfg*. Use LINKSTS_* defines for *stscfg* bit declarations. The status configuration selects which link status bits to clear by the driver ISR. Bits in the link status register are cleared by the driver interrupt service routine if and only if the corresponding bit is set in the *stscfg* parameter.

Bitmask	Description (prefixed LINKSTS_)
CE	Read/Set clear status from ISR for credit error
ER	Read/Set clear status from ISR for escape error
DE	Read/Set clear status from ISR for disconnect error
PE	Read/Set clear status from ISR for parity error
WE	Read/Set clear status from ISR for write synchronization error (GRSPW1 only)
IA	Read/Set clear status from ISR for invalid address
EE	Read/Set clear status from ISR for early EOP/EEP

Param	*clkdiv* [IO] pointer to integer If *clkdiv* points to -1, the clock divisor fields are only read from the I/O registers, otherwise it is updated according to the value in memory pointed to by *clkdiv*.
Return	None.

Table 3.12. grspw_link_state function declaration

Proto	spw_link_state_t grspw_link_state(void *d)
About	Read and return current SpaceWire link status.
Param	*d* [IN] pointer Device identifier. Returned from grspw_open.

Return	enum spw_link_state_t. SpaceWire link status according to SpaceWire standard FSM state machine numbering. The possible return values are listed below, all numbers must be prefixed with SPW_LS_ declared by enum spw_link_state_t.	
	Value	Description.
	ERRRST	Error reset.
	ERRWAIT	Error Wait state.
	READY	Error Wait state.
	CONNECTING	Connecting state.
	STARTED	Stated state.
	RUN	Run state - link and DMA is fully operational.

Table 3.13. `grspw_link_status` function declaration

Proto	`unsigned int grspw_link_status(void *d)`
About	Reads and returns the current value of the GRSPW status register. The status register bits can be cleared by calling `grspw_link_status_clr` with return value as parameter.
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.
Return	unsigned int. Current value of the GRSPW Status Register.

Table 3.14. `grspw_link_status_clr` function declaration

Proto	`void grspw_link_status_clr(void *d, unsigned int mask)`
About	Clear bits in the GRSPW status register. The *`mask`* can be the return value of function `grspw_link_status`
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*`mask`* [IN] Integer Status bits to clear
Return	None.

3.3.5. Node address configuration

This part for the device API controls the node address configuration of the RMAP target and DMA channels. The node address configuration functionality is described in Section 3.2.7. The data structures and functions involved in controlling the node address configuration are listed below.

```
struct grspw_addr_config {
 /* Ignore address field and put all received packets to first
  * DMA channel.
  */
 int promiscuous;

 /* Default Node Address and Mask */
 unsigned char def_addr;
 unsigned char def_mask;
 /* DMA Channel custom Node Address and Mask */
 struct {
  char node_en;                   /* Enable Separate Addr */
  unsigned char node_addr;        /* Node address */
  unsigned char node_mask;        /* Node address mask */
 } dma_nacfg[4];
};
```

Table 3.15. grspw_addr_config data structure declaration

promiscu-ous	Enable (1) or disable (0) promiscuous mode. The GRSPW will ignore the address field and put all received packets to first DMA channel. See hardware manual for details. Read back as 0 or 1.

	This field is also used to by the driver indicate if the settings should be written and read, or only read. See function description.
def_addr	GRSPW default node address.
def_mask	GRSPW default node address mask.
dma_nacfg	DMA channel node address array configuration, see below field description. DMA channel N is described by *dma_nacfg[N]*.

Field	Description
node_en	Separate node address configuration for DMA channel N (determined by array index). Read back as 0 or 1. • DISABLE_DMA_NODE_ADDR (0): disable the separate node address. • ENABLE_DMA_NODE_ADDR (1): enable the separate node address. • IGNORE_DMA_NODE_ADDR (2): leave the DMA channel's node address configuration unchanged. Useful when other users of the device own other DMA channels, see also `grspw_dma_addr_ctrl`.
node_addr	If separate node address is enabled this option sets the node address for DMA channel N (determined by array index).

node_mask	If separate node address is enabled this option sets the node address mask for DMA channel N (determined by array index).

Table 3.16. `grspw_addr_ctrl` function declaration

Proto	`void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)`
About	Always read and optionally set the node addresses configuration. The GRSPW device is either configured to have one single node address or a range of addresses by masking. The *cfg* input memory layout is described by the grspw_addr_config data structure in Table 3.15. When using multiple DMA channels one must assign each DMA channel a unique node address or a unique range by masking. Each DMA channel is represented by the input *dma_nacfg[N]*. A DMA channel whose *node_en* is set to IGNORE_DMA_NODE_ADDR is not modified. To configure a single DMA channel only, use `grspw_dma_addr_ctrl`.
Param	*d* [IN] pointer Device identifier. Returned from grspw_open.
Param	*cfg* [IO] pointer Address to where the driver will read or write the address configuration from. If the *promiscuous* field is set to -1 the hardware is not written, instead the current configuration is only read and memory content updated accordingly.
Return	None.

Table 3.17. `grspw_dma_addr_ctrl` function declaration

Proto	`void grspw_dma_addr_ctrl(void *c, int node_en, unsigned char node_addr, unsigned char node_mask)`
About	Configures the separate node address of one DMA channel. Other DMA channels, the default node address and promiscuous mode are not changed. This is useful when several users share a GRSPW device, each using its own DMA channel.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*node_en* [IN] Integer Non-zero enables the separate node address of the DMA channel, zero disables it.
Param	*node_addr* [IN] Integer Node address of the DMA channel. Only written if *node_en* is non-zero.
Param	*node_mask* [IN] Integer

	Node address mask of the DMA channel. A received packet is accepted when (address & mask) == (node_addr & mask). Only written if *node_en* is non-zero.
Return	None.

3.3.6. Time Code support

SpaceWire Time Code handling is controlled and configured using the device API functions described below. The Time Code functionality is described in Section 3.2.4.

Table 3.18. `grspw_tc_ctrl` function declaration

Proto	`void grspw_tc_ctrl(void *d, int *options)`
About	Always read and optionally set TimeCode settings of GRSPW device. It is possible to enable/disable reception/transmission and interrupt generation of TimeCodes. See *TCOPTS_** defines for available options.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*options* [IO] pointer to bit-mask If options points to -1, the TimeCode options is only read from the I/O registers, otherwise it is updated according to the value in memory pointed to by options. Use TCOPTS_* defines for option bit declarations.
Return	None.

Value	Description
EN_RXIRQ	When 1 enable, when zero disable TimeCode receive interrupt generation (affects TQ and IE bit in control register).
EN_TX	Enable/disable TimeCode transmission (affects TT bit in control register).
EN_RX	Enable/disable TimeCode reception (affects TR bit in control register).

Table 3.19. `grspw_tc_tx` function declaration

Proto	`void grspw_tc_tx(void *d)`
About	Generates a TimeCode Tick-In.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Return	None.

Table 3.20. `grspw_tc_isr` function declaration

Proto	`void grspw_tc_isr(void *d, void (*tcisr)(void *data, int tc), void *data)`
About	Assigns a Interrupt Service Routine (ISR) to handle TimeCode interrupt events. The ISR is called from the GRSPW device's interrupt handler, thus the isr is called in interrupt context and care needs to be taken. The ISR is called when a Tick-Out event has happened and an interrupt has been generated. The ISR is called with a custom argument *data* and the current value of the GRSPW TC register. The TC register contains TimeCode control flags and counter. The GRSPW interrupt handler always clears the GRSPW status field. It is performed after the ISR has been called. Note that even if the Tick-Out interrupt generation has not been enabled the ISR may still be called if other GRSPW interrupts are generated and the GRSPW status indicates that a Tick-Out has been received.

Param	*d* [IN] pointer Device identifier. Returned from grspw_open.
Param	*tcisr* [IN] pointer to function If argument is NULL the Tick-Out ISR call is disabled. Otherwise the pointer will be used in a function call from interrupt context when a Tick-Out event is detected.
Param	*data* [IN] pointer to custom data This value is given as the first argument to the ISR.
Return	None.

Table 3.21. grspw_tc_time function declaration

Proto	void grspw_tc_time(void *d, int *time)	
About	Optionally writes and always reads the current TimeCode control flags and counter from hardware registers. The values are written into the address pointed to by *time*.	
Param	*d* [IN] pointer Device identifier. Returned from grspw_open.	
Param	*time* [IO] pointer to bit-mask If time points to -1, the TimeCode options are only read from the I/O registers. Otherwise hardware is updated according to the value in memory pointed to by time before reading the hardware registers. Use TCOPTS_* defines for time bit declarations.	
	bits	Description
	5..0	The 6 LSB bits reads/writes the time control flags.
	7..6	The 2 bits reads/writes the time counter value.
Return	None.	

3.3.7. Port Control

The SpaceWire port selection configuration, hardware support and current hardware status can be accessed using the device API functions described below. The SpaceWire port support functionality is described in Section 3.2.3.

In cases where only one SpaceWire port is implemented this part of the API can safely be ignored. The functions still deliver consistent information and error code failures when forcing Port1, however provides no real functionality.

Table 3.22. grspw_port_ctrl function declaration

Proto	int grspw_port_ctrl(void *d, int *port)	
About	Always read and optionally set port control settings of GRSPW device. The configuration determines how the hardware selects which SpaceWire port that is used. This is an optional feature in hardware to support one or two SpaceWire ports. An error is returned if operation not supported by hardware.	
Param	*d* [IN] pointer Device identifier. Returned from grspw_open.	
Param	*port* [IO] pointer to bit-mask The port configuration is first written if *port* does not point to -1. The port configuration is always read from the I/O registers and stored in the *port* address.	
	Value	Description
	-1	The current port configuration is read and stored into the *port* address.
	0	Force to use Port0.
	1	Force to use Port1.
	> 1	Hardware auto select between Port0 or Port1.
Return	Value. Description	

	0	Request successful.
	-1	Request failed. Port1 is not implemented in hardware.

Table 3.23. `grspw_port_count` function declaration

Proto	`int grspw_port_count(void *d)`	
About	Reads and returns number of ports that hardware supports.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Return	int. Number of ports implemented in hardware.	
	Value	Description
	1	One SpaceWire port is implemented in hardware. In this case `grspw_port_ctrl` function has no effect and `grspw_port_active` always returns 0.
	2	Two SpaceWire ports are implemented in hardware.

Table 3.24. `grspw_port_active` function declaration

Proto	`int grspw_port_active(void *d)`	
About	Reads and returns the currently actively used SpaceWire port.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Return	int. Currently active SpaceWire port	
	Value	Description
	0	SpaceWire port0 is active.
	1	SpaceWire port1 is active.

3.3.8. RMAP Control

The device API described below is used to configure the hardware supported RMAP target. The RMAP support is described in Section 3.2.5.

When RMAP CRC is implemented in hardware it can be used to generate and append a CRC on a per packet basis. It is controlled by the DMA packet flags. Header and data CRC can be generated individually. See Table 3.36 for more information.

Table 3.25. `grspw_rmap_support` function declaration

Proto	`int grspw_rmap_support(void *d, char *rmap, char *rmap_crc)`	
About	Reads the RMAP hardware support of a GRSPW device. It is equivalent to use the `grspw_hw_support` function to get the RMAP functionality present in hardware.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Param	*`rmap`* [OUT] pointer If not NULL the RMAP configuration is stored into the address of *`rmap`*.	
	Value	Description
	0	RMAP target is not implemented in hardware.
	1	RMAP target is implemented in hardware.
Param	*`rmap_crc`* [OUT] pointer If not NULL the RMAP configuration is stored into the address of *`rmap`*.	
	Value	Description

	0	RMAP CRC algorithm is not implemented in hardware
	1	RMAP CRC algorithm is implemented in hardware
Return	None.	

Table 3.26. `grspw_rmap_ctrl` function declaration

Proto	`int grspw_rmap_ctrl(void *d, int *options, int *dstkey)`	
About	Read and optionally write RMAP configuration and SpaceWire destination key value. This function controls the GRSPW hardware implemented RMAP functionality. Set *`option`* to NULL not to read or write RMAP configuration. Set *`dstkey`* to NULL to not read or write RMAP destination key. Setting both to NULL results in no operation.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Param	*`options`* [IO] pointer to bit-mask The RMAP configuration is first written if *`options`* does not point to -1. The RMAP configuration is always read from the I/O registers and stored in the *`options`* address. See RMAPOPTS_* definitions for bit declarations.	
	Bit	Description
	EN_RMAP	Enable (1) or Disable (0) RMAP target handling in hardware.
	EN_BUF	**Enable (0) or Disable (1)** RMAP buffer. Disabling ensures that all RMAP requests are processed in the order they arrive.
Param	*`dstkey`* [IO] pointer The SpaceWire 8-bit destination key is first written if *`dstkey`* does not point to -1. The destination key configuration is always read from the I/O registers and stored in the *`dstkey`* address.	
Return	int. Status	
	0	Request successful.
	-1	Failed to enable RMAP handling in hardware. Not present in hardware.

3.3.9. Statistics

The driver counts statistics at certain events. The GRSPW device driver counters can be read out using the device API. The number of interrupts serviced and different kinds of link error can be obtained.

Statistics related to a specific DMA channel activity can be accessed using the DMA channel API.

> The read function is not protected by locks. A GRSPW interrupt could cause the statistics to be out of sync. For example the number of link parity errors may not match the number of interrupts, by one.

```
struct grspw_core_stats {
 int irq_cnt;
 int err_credit;
 int err_eeop;
 int err_addr;
 int err_parity;
 int err_disconnect;
 int err_escape;
 int err_wsync; /* only in GRSPW1 */
};
```

Table 3.27. grspw_core_stats data structure declaration

irq_cnt	Number of interrupts serviced for this GRSPW device.
err_credit	Number of credit errors experienced for this GRSPW device.
err_eeop	Number of Early EOP/EEP errors experienced for this GRSPW device.
err_addr	Number of invalid address errors experienced for this GRSPW device.

err_parity	Number of parity errors experienced for this GRSPW device.
err_disconnect	Number of disconnect errors experienced for this GRSPW device.
err_escape	Number of escape errors experienced for this GRSPW device.
err_wsync	Number of write synchronization errors experienced for this GRSPW device. This is only applicable for GRSPW cores.

Table 3.28. `grspw_stats_read` function declaration

Proto	`void grspw_stats_read(void *d, struct grspw_core_stats *sts)`
About	Reads the current driver statistics collected from earlier events by GRSPW device and driver usage. The statistics are stored to the address given by the second argument. The layout and content of the statistics are defined by the grspw_core_stats data structure described in Table 3.27. Note that the snapshot is taken without lock protection, as a consequence the statistics may not be synchronized with each other. This could be caused if the function is interrupted by a the GRSPW interrupt.
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*`sts`* [OUT] pointer If NULL no operating is performed. Otherwise a snapshot of the current driver statistics are copied to this user provided buffer. The layout and content of the statistics are defined by the grspw_core_stats data structure described in Table 3.27.
Return	None.

Table 3.29. `grspw_stats_clr` function declaration

Proto	`void grspw_stats_clr(void *d)`
About	Resets the driver GRSPW device statistical counters to zero.
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.
Return	None.

3.4. DMA interface

This section covers how the driver can be interfaced to an application to send and transmit SpaceWire packets using the GRSPW hardware.

GRSPW2 and GRSPW2_DMA devices supports more than one DMA channel. The device channel zero is always present.

3.4.1. Opening and closing DMA channels

The first step before any SpaceWire packets can be transferred is to open a DMA channel to be used for transmission. As described in the device API Section 3.3.1 the GRSPW device the DMA channel belongs to must be opened and passed onto the DMA channel open routines.

The number of DMA channels of a GRSPW device can obtained by calling `grspw_hw_support`.

An opened DMA channel can not be reopened unless the channel is closed first. When opening a channel the channel is marked opened by the driver. This procedure is thread-safe by protecting from other threads by using the GRSPW driver's semaphore lock. The semaphore is used by all GRSPW devices on device opening, closing and DMA channel opening.

During opening of a GRSPW DMA channel the following steps are taken:

- DMA channel I/O registers are initialized to a state where most are zero.
- Resources used for the DMA channel implementation itself, such as the descriptor ring memory, spin-locks and wait semaphores, are allocated and initialized. A previously registered DMA event callback is removed.
- The channel is marked opened to protect the caller from other users of the DMA channel.

Below is a partial example of how the first GRSPW device's first DMA channel is opened, link is started and a packet can be received.

```
int spw_receive_one_packet()
{
  void *device;
  void *dma0;
  int count, options, clkdiv;
  spw_link_state_t state;
  struct grspw_list lst;

  device = grspw_open(0);
  if (!device)
   return -1; /* Failure */

  /* Start Link */
  options = LINKOPTS_ENABLE | LINKOPTS_START; /* Start Link */
  clkdiv = (9 << 8) | 9; /* Clock Divisor factor of 10 (100MHz input) */
  grspw_link_ctrl(device, &options, NULL, &clkdiv);

  /* wait until link is in run-state */
  do {
   state = grspw_link_state(device);
  } while (state != SPW_LS_RUN);

  /* Open DMA channel */
  dma0 = grspw_dma_open(device, 0);
  if (!dma0) {
   grspw_close(device);
   return -2;
  }

  /* Initialize and activate DMA */
  if (grspw_dma_start(dma0)) {
   grspw_dma_close(dma0);
   grspw_close(device);
   return -3;
  }

  /* ... */

  /* Prepare driver with RX buffers */
  grspw_dma_rx_prepare(dma0, 1, &preinited_rx_unused_buf_list0,
                       preinited_rx_unused_buf_count0);

  /* Start sending a number of SpaceWire packets */
  grspw_dma_tx_send(dma0, 1, &preinited_tx_send_buf_list,
                    preinited_tx_send_buf_count);

  /* Receive at least one packet */
  do {
   /* Try to receive as many packets as possible */
   count = -1;
   grspw_dma_rx_recv(dma0, 0, &lst, &count);
  } while (count <= 0);

  printf("GRSPW0.DMA0: Received %d packets\n", count);

  /* ... */

  grspw_dma_close(dma0);
  grspw_close(device);
  return 0; /* success */
}
```

Table 3.30. `grspw_dma_open` function declaration

Proto	`void *grspw_dma_open(void *d, int chan_no)`
About	Opens a DMA channel of a previously opened GRSPW device. The GRSPW device is identified by its device handle *`d`* and the DMA channel is identified by index *`chan_no`*. The returned value is used as input argument to all functions operating on the DMA channel.
Param	*`d`* [IN] pointer

	Device identifier. Returned from `grspw_open`.
Param	*`chan_no`* [IN] Integer DMA channel identification number. DMA channels are indexed by 0, 1, 2 or 3. Other input values cause NULL to be returned. The index must be equal or greater than zero, and smaller than the number of DMA channels reported by `grspw_hw_support`.
Return	Pointer. Status and driver's internal device identification.

Value	Description
NULL	Indicates failure to open the DMA channel. Fails if the DMA channel does not exist, the DMA channel already has been opened or the DMA channel resource allocation fails.
Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all DMA channel API functions, identifies which DMA channel.

Notes	May blocking until other GRSPW device operations complete.

Table 3.31. `grspw_dma_close` function declaration

Proto	`int grspw_dma_close(void *c)`
About	Closes a previously opened DMA channel. The specified DMA channel must be in stopped state before calling this function. Prior to closing the user is responsible for calling `grspw_dma_stop` to stop on-going DMA transfers and interrupts, and to unblock tasks waiting for RX/TX events on this DMA channel. Blocked tasks must have exited the driver, otherwise an error code is returned. The function does not wait for them. A DMA event callback registered with `grspw_dma_set_event_cb` is removed.
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Return	int. Return code as indicated below.

Value	Description
0	Success, or the DMA channel was not open.
1	Failure due to DMA channel is active (started) or a task is still within the RX/TX wait interface of this DMA channel.

Table 3.32. `grspw_dma_opened` function declaration

Proto	`unsigned int grspw_dma_opened(void *d)`
About	Returns which DMA channels of the device are open, by any user. A kernel driver that takes one DMA channel of a device opens it when it binds and keeps it open; the user space driver uses this function to leave such channels alone, see Chapter 2.
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.
Return	unsigned int. Bit N is set if DMA channel N is open.

3.4.2. Starting and stopping DMA operation

The start and stop operational modes are described in Section 3.2.12. The functions described below are used to change the operational mode of a DMA channels. A summary of which DMA API functions are affected are listed in Table 3.33, see function description for details on limitations.

Table 3.33. functions available in the two operational modes

Function	Stopped	Started
`grspw_dma_open`	N/A	N/A

Function	Stopped	Started
grspw_dma_close	Yes	Yes
grspw_dma_start	Yes	No
grspw_dma_stop	No	Yes
grspw_dma_rx_recv	Yes, with limitations, see Section 3.4.6	Yes
grspw_dma_rx_prepare	Yes, with limitations, see Section 3.4.6	Yes
grspw_dma_rx_count	Yes, with limitations, see Section 3.4.7	Yes
grspw_dma_rx_wait	No	Yes
grspw_dma_tx_send	Yes, with limitations, see Section 3.4.5	Yes
grspw_dma_tx_reclaim	Yes, with limitations, see Section 3.4.5	Yes
grspw_dma_tx_count	Yes with limitations, see Section 3.4.7	Yes
grspw_dma_tx_wait	No	Yes
grspw_dma_config	Yes	No
grspw_dma_config_read	Yes	Yes
grspw_dma_stats_read	Yes	Yes
grspw_dma_stats_clr	Yes	Yes
grspw_dma_set_event_cb	Yes	No
grspw_dma_irq_enable	Yes	Yes
grspw_dma_rx_pending	Yes	Yes

Table 3.34. `grspw_dma_start` function declaration

Proto	`int grspw_dma_start(void *c)`
About	Starts DMA operational mode for the DMA channel indicated by the argument. After this step it is possible to send and receive SpaceWire packets. If the DMA channel already is in started mode, no action will be taken. The start routine clears and initializes the following: • DMA descriptor rings. • DMA queues. • Statistic counters. • Interrupt counters • I/O registers to match DMA configuration • Interrupt • DMA Status • Enables the receiver Even though the receiver is enabled the user is required to prepare empty receive buffers after this point, see `grspw_dma_rx_prepare`. The transmitter is enabled when the user provides send buffers that ends up in the TX SCHED queue, see `grspw_dma_tx_send`.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Return	int. Return code as indicated below.

Value	Description
0	Success, or the DMA channel was already started.

	-EAGAIN	A task woken by a previous `grspw_dma_stop` has not yet returned from `grspw_dma_rx_wait` or `grspw_dma_tx_wait`. Nothing was changed, the caller may retry.

Table 3.35. `grspw_dma_stop` function declaration

Proto	`void grspw_dma_stop(void *c)`
About	Stops DMA operational mode for the DMA channel indicated by the argument. The transmitter will abort ongoing transfers and the receiver disabled. Tasks blocked in the RX/TX wait functions of the DMA channel are woken up and return to the caller indicating that DMA has stopped. This function does not wait for them to exit the driver and never blocks. Packets in the RX READY, RX SCHED queues will be moved to the RX RECV queue. The RXPKT_FLAG_RX packet flag is used to signal if the packet was received or just moved. Similarly, the packets in the TX SEND and TX SCHED queues are moved to the TX SENT queue and the TXPKT_FLAG_TX marks if the packet actually was transferred or not.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Return	None.

3.4.3. Packet buffer description

The GRSPW packet driver describes packets for both RX and TX using a common memory layout defined by the data structure grspw_pkt. It is described in detail below.

There are differences in which fields and bits are used between RX and TX operations. The bits used in the *flags* field are defined different. When sending packets the user can optionally provide two different buffers, the header and data. The header can maximally be 256Bytes due to hardware limitations and the data supports 24-bit length fields. For RX operations *hdr* and *hlen* are not used. Instead all data received is put into the data area.

```
struct grspw_pkt {
 struct grspw_pkt *next; /* Next packet in list. NULL if last packet */
 unsigned int pkt_id;    /* User assigned ID (not touched by driver) */
 unsigned short flags;   /* RX/TX Options and status */
 unsigned char reserved; /* Reserved, must be zero */
 unsigned char hlen;     /* Length of Header Buffer (only TX) */
 unsigned int dlen;      /* Length of Data Buffer */
 u32 data;               /* 4-byte or byte aligned address depends on HW */
 u32 hdr;                /* 4-byte or byte aligned address depends on HW (only TX) */
};
```

Table 3.36. grspw_pkt data structure declaration

next	The packet structure can be part of a linked list. This field is used to point out the next packet in the list. Set to NULL if this packet is the last in the list or a single packet.	
pkt_id	User assigned ID. This field is never touched by the driver. It can be used to store a pointer or other data to help implement the user buffer handling.	
flags	RX/TX transmission options and flags indicating resulting status. The bits described below is to be prefixed with TXPKT_FLAG_ or RXPKT_FLAG_ in order to match the TX or RX options definitions declared by the driver's header file.	
	Bits	TX Description (prefixed TXPKT_FLAG_)
	NOCRC_MASK	Indicates to driver how many bytes should not be part of the header CRC calculation. 0 to 15 bytes can be omitted. Use NOCRC_LENN to select a specific length.
	IE	Enable (1) or Disable (0) IRQ generation on packet transmission completed.
	HCRC	Enable (1) or disable (0) Header CRC generation (if CRC is available in hardware). Header CRC will be appended (one byte at end of header).
	DCRC	Enable (1) or disable (0) Data CRC generation (if CRC is available in hardware). Data CRC will be appended (one byte at end of packet).

	TX	Is set by the driver to indicate that the packet was transmitted. This does no signal a successful transmission, but that transmission was attempted, the LINKERR bit needs to be checked for error indication.
	LINKERR	Set if a link error was exhibited during transmission of this packet.
	Bits	RX Description (prefixed RXPKT_FLAG_)
	IE	Enable (1) or Disable (0) IRQ generation on packet reception completed.
	TRUNK	Set if packet was truncated.
	DCRC	Set if data CRC error detected (only valid if RMAP CRC is enabled).
	HCRC	Set if header CRC error detected (only valid if RMAP CRC is enabled).
	EEOP	Set if an End-of-Packet error occurred during reception of this packet.
	RX	Marks if packet was received or not.
hlen	Header length. The number of bytes hardware will transfer using DMA from the address indicated by the hdr pointer. This field is not used by RX operation.	
dlen	Data payload length. The number of bytes hardware DMA read or written from/to the address indicated by the data pointer. On RX this is the complete packet data received.	
data	Header Buffer Address. DMA will read from this. The address can be 4-byte or byte aligned depending on hardware.	
hdr	Header Buffer Address. DMA will read *`hlen`* bytes from this. The address can be 4-byte or byte aligned depending on hardware. This field is not used by RX operation.	

3.4.4. Blocking/Waiting on DMA activity

Blocking and polling mode are described in the Section 3.2.10. The functions described below are used to set up RX or TX wait conditions and blocks the calling thread until condition evaluates true.

Table 3.37. `grspw_dma_tx_wait` function declaration

Proto	`int grspw_dma_tx_wait(void *c, int send_cnt, int op, int sent_cnt, long timeout)`
About	Block until *`send_cnt`* or fewer packets are queued in TX "Send and Scheduled" queue, *`op`* (AND or OR), *`sent_cnt`* or more packet "have been sent" (Sent Q) condition is met. If a link error occurs and the "Disable on Link error" is defined, this function will also return to caller. The timeout argument is used to return after *`timeout`* ticks, regardless of the other conditions. If timeout is zero, the function will wait forever until the condition is satisfied. --- If IRQ of TX descriptors are not enabled conditions are never checked, this may hang infinitely unless a timeout has been specified. The waiting task is not woken by DMA interrupts when a DMA event callback has been registered with `grspw_dma_set_event_cb`. ---
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*`send_cnt`* [IN] int Sets the condition's number of packets in TX SEND queue.
Param	*`op`* [IN] boolean Condition operation. Set to zero for AND or one for OR.
Param	*`sent_cnt`* [IN] int Sets the condition's number of packets in TX SENT queue.
Param	*`timeout`* [IN] long Sets the timeout in jiffies. Set to zero or a negative value to disable the timeout and wait until the condition is met or DMA is stopped.
Return	Int. See return code below.

	Value	Description
	-1	Error. Also returned if a wait without timeout is interrupted by a signal.
	0	Returning to caller because specified conditions are now fullfilled.
	1	DMA stopped.
	2	Timeout, conditions are not met.
	3	Another task is already waiting. Service is Busy.
Notes	May block. Must not be called from atomic context.	

Table 3.38. `grspw_dma_rx_wait` function declaration

Proto	`int grspw_dma_rx_wait(void *c, int recv_cnt, int op, int ready_cnt, long timeout)`
About	Block until *`recv_cnt`* or more packets are queued in RX RECV queue, *`op`* (AND or OR), *`ready_cnt`* or fewer packet buffers are available in the RX "READY and Scheduled" queues, condition is met. If a link error occurs and the "Disable on Link error" is defined, this function will also return to caller, however with an error. The *`timeout`* argument is used to return after *`timeout`* number of ticks, regardless of the other conditions. If timeout is zero, the function will wait forever until the condition is satisfied. --- If IRQ of RX descriptors are not enabled conditions are never checked, this may hang infinitely unless a timeout has been specified. The waiting task is not woken by DMA interrupts when a DMA event callback has been registered with `grspw_dma_set_event_cb`. ---
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*`recv_cnt`* [IN] int Sets the condition's number of packets in RX RECV queue.
Param	*`op`* [IN] boolean Condition operation. Set to zero for AND or one for OR.
Param	*`ready_cnt`* [IN] int Sets the condition's number of packets in RX READY queue.
Param	*`timeout`* [IN] long Sets the timeout in jiffies. Set to zero or a negative value to disable the timeout and wait until the condition is met or DMA is stopped.
Return	Int. See return code below.

Value	Description
-1	Error. Also returned if a wait without timeout is interrupted by a signal.
0	Returning to caller because specified conditions are now fullfilled.
1	DMA stopped.
2	Timeout, conditions are not met.
3	Another task is already waiting. Service is Busy.

Notes	May block. Must not be called from atomic context.

3.4.5. Sending packets

Packets are sent by adding packets to the SEND queue. Depending on the driver configuration and usage the packets eventually are put into SCHED queue where they will be assigned a DMA descriptor and scheduled for transmission. After transmission has completed the packet buffers can be retrieved to view the transmission status

and to be able to reuse the packet buffers for new transfers. During the time the packet is in the driver it must not be accessed by the user.

Transmission of SpaceWire packets are described in Section 3.2.1.

In the below example Figure 3.4 three SpaceWire packets are scheduled for transmission. The `count` should be set to three. The second packet is programmed to generate an interrupt when transmission finished, GRSPW hardware will also generate a header CRC using the RMAP CRC algorithm resulting in a 16 bytes long SpaceWire packet.

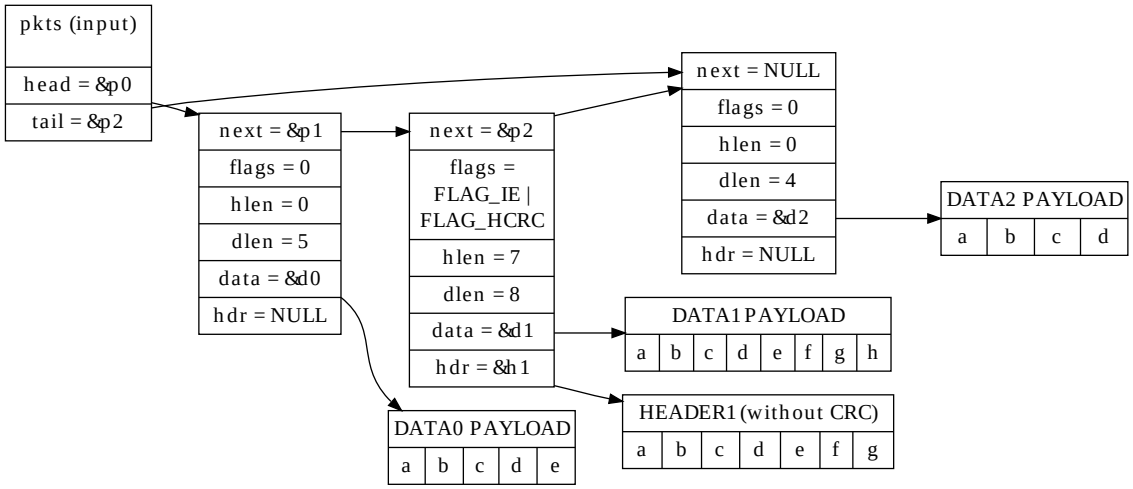

Figure 3.4. TX packet description `pkts` input to `grspw_dma_tx_send`

The below tables describe the functions involved in initiating and completing transmissions.

Table 3.39. `grspw_dma_tx_send` function declaration

Proto	`int grspw_dma_tx_send(void *c, int opts, struct grspw_list *pkts, int count)`
About	Schedules a list of packets for transmission at some point in future. The packets are put to the SEND queue of the driver. Depending on the input arguments a selection of the below steps are performed: 1. Move transmitted packets to SENT List (SCHED->SENT). 2. Add the requested packets to the SEND List (USER->SEND) 3. Schedule as many packets as possible for transmission (SEND->SCHED) Skipping both step 1 and 3 may be useful when IRQ is enabled, then the worker thread will be responsible for handling descriptors. The GRSPW transmitter is enabled when packets are added to the TX SCHED queue. The fastest solution in retrieving sent TX packets and sending new frames is to call: 1. grspw_dma_tx_reclaim(opts=0) 2. grspw_dma_tx_send(opts=1) NOTE: the TXPKT_FLAG_TX flag must not be set in the packet structure.
Param	`c` [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	`opts` [IN] Integer bit-mask The above steps 1 and/or 3 may be skipped by setting `opts` argument according the description below.
	Bit: Description 0: Set to 1 to skip Step 1. 1: Set to 1 to skip Step 3.
Param	`pkts` [IN] pointer

	A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that *head* points to the first packet and *tail* points to the last. Call this function with *pkts* set to NULL to avoid step 2. Just doing step 1 and 3 as determined by *opts* is normally performed in polling-mode. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 3.36. Note that TXPKT_FLAG_TX of the *flags* field must not be set.
Param	*count* [IN] integer Number of packets in the packet list.
Return	Status. See return codes below

Value	Description
-1	Error occurred, DMA channel may not be valid.
0	Successfully added pkts to TX SEND/SCHED list.
1	DMA stopped. No operation.

Notes	This function performs no operation when the DMA channel is stopped.

Table 3.40. `grspw_dma_tx_reclaim` function declaration

Proto	`int grspw_dma_tx_reclaim(void *c, int opts, struct grspw_list *pkts, int *count)`
About	Reclaim TX packet buffers that has previously been scheduled for transmission with `grspw_dma_tx_send`. The packets in the SENT queue are moved to the *pkts* packet list. When the move has been completed the packet can safely be reused again by the user. The packet structures have been updated with transmission status to indicate transfer failures of individual packets. Depending on the input arguments a selection of the below steps are performed: 1. Move transmitted packets to SENT List (SCHED->SENT). 2. Move all SENT List to pkts list (SENT->USER). 3. Schedule as many packets as possible for transmission (SEND->SCHED) Skipping both step 1 and 3 may be useful when IRQ is enabled, then the worker thread will be responsible for descriptor processing. Skipping only step 2 can be useful in polling mode. The fastest solution in retrieving sent TX packets and sending new frames is to call: 1. grspw_dma_tx_reclaim(opts=0) 2. grspw_dma_tx_send(opts=1) NOTE: the TXPKT_FLAG_TX flag indicates if the packet was transmitted.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*opts* [IN] Integer bit-mask The above steps 1 and/or 3 may be skipped by setting *opts* argument according the description below.

Bit	Description
0	Set to 1 to skip Step 1.
1	Set to 1 to skip Step 3.

Param	*pkts* [OUT] pointer The list will be initialized to contain the SpaceWire packets moved from the SENT queue to the packet list. The grspw_list structure will be initialized so that *head* points to the first packet, *tail* points to the last and the last packet (tail) next pointer is NULL. Call this function with *pkts* set to NULL to avoid step 2. Just doing step 1 and 3 as determined by *opts* is normally performed in polling-mode.

	The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 3.36. Note that TXPKT_FLAG_TX of the *flags* field indicates if the packet was sent of not. In case of DMA being stopped one can use this flag to see if the packet was transmitted or not. The TXPKT_FLAG_LINKERR indicates if a link error occurred during transmission of the packet, regardless the TXPKT_FLAG_TX is set to indicate packet transmission attempt.	
Param	*count* [IO] pointer Number of packets in the packet list.	
	Value	Input Description
	NULL	Move all packets from the SENT queue to the packet list.
	-1	Move all packets from the SENT queue to the packet list.
	0	No packets are moved. Same as if pkts is NULL.
	>0	Move a maximum of '*count' packets to the packet list.
	Value	Output Description
	NULL	Nothing performed.
	others	'*count' is updated to reflect number of packets in packet list.
Return	Status. See return codes below	
	Value	Description
	-1	Error occurred, DMA channel may not be valid.
	0	Successful. pkts list filled with all packets from sent list.
	1	Indicates that DMA is stopped. Same as 0 but step 1 and 3 were never done.
Notes	This function can only do step 1 and 2 to allow read out sent packets when in stopped mode. This is useful when a link goes down and the DMA activity is stopped by user of by driver automatically.	

3.4.6. Receiving packets

Packets are received by adding empty/free packets to the RX READY queue. Depending on the driver configuration and usage the packets eventually are put into RX SCHED queue where they will be assigned a DMA descriptor and scheduled for reception. After a packet is received into the buffer(s) the packet buffer(s) can be retrieved to view the reception status and to be able to reuse the packet buffers for new transfers. During the time the packet is in the driver it must not be accessed by the user.

Reception of SpaceWire packets are described in Section 3.2.1.

In the Figure 3.5 example three SpaceWire packets are received. The *count* parameters is set to three by the driver to reflect the number of packets. The first packet exhibited an early end-of-packet during reception which also resulted in header and data CRC error. All header points and header lengths have been set to zero by the user since they are no used, however the values in those fields does not affect the RX operations. The RX flag is set to indicate that DMA transfer was performed.

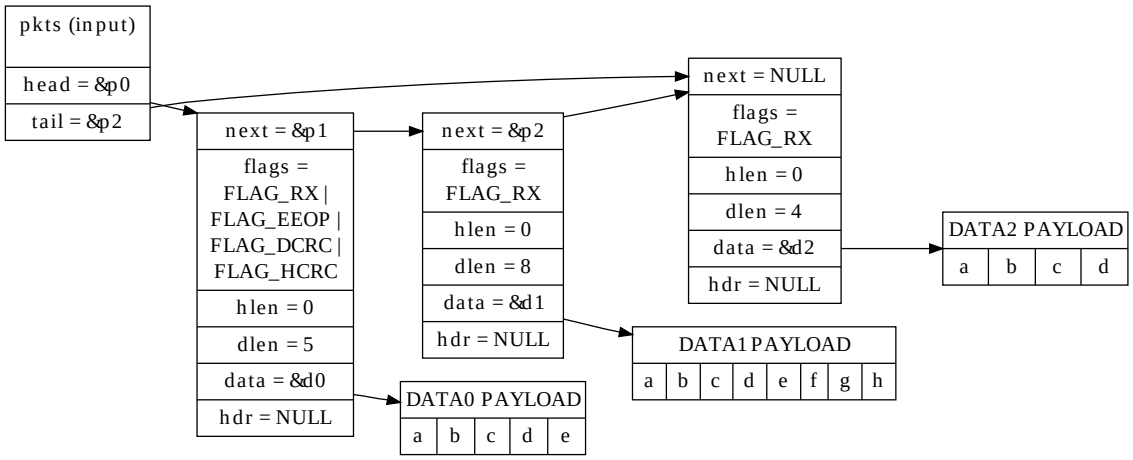

Figure 3.5. RX packet output from `grspw_dma_rx_recv`

The below tables describe the functions involved in initiating and completing transmissions.

Table 3.41. `grspw_dma_rx_prepare` function declaration

Proto	`int grspw_dma_rx_prepare(void *c, int opts, struct grspw_list *pkts, int count)`
About	Add more RX packet buffers for future for reception. The received packets can later be read out with `grspw_dma_rx_recv`. The packets are put to the READY queue of the driver. Depending on the input arguments a selection of the below steps are performed: 1. Move Received packets to RECV List (SCHED->RECV). 2. Add the *`pkt`* packet buffers to the READY List (USER->READY). 3. Schedule as many packets as possible (READY->SCHED). Skipping both step 1 and 3 may be useful when IRQ is enabled, then the worker thread will be responsible for handling descriptors. Skipping only step 2 can be useful in polling mode. The fastest solution in retrieving received RX packets and preparing new packet buffers for future receive, is to call: 1. grspw_dma_rx_recv(opts=2, &recvlist) (Skip step 3) 2. grspw_dma_rx_prepare(opts=1, &freelist) (Skip step 1) NOTE: the RXPKT_FLAG_RX flag must not be set in the packet structure.
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*`opts`* [IN] Integer bit-mask The above steps 1 and/or 3 may be skipped by setting *`opts`* argument according the description below.

Bit	Description
0	Set to 1 to skip Step 1.
1	Set to 1 to skip Step 3.

Param	*`pkts`* [IN] pointer A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that *`head`* points to the first packet and *`tail`* points to the last. Call this function with *`pkts`* set to NULL to avoid step 2. Just doing step 1 and 3 as determined by *`opts`* is normally performed in polling-mode. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 3.36. Note that RXPKT_FLAG_RX of the *`flags`* field must not be set.
Param	*`count`* [IN] integer

	Number of packets in the packet list.
Return	Status. See return codes below

Value	Description
-1	Error occurred, DMA channel may not be valid.
0	Successfully added pkts to RX READY/SCHED list.
1	DMA stopped. No operation.

Notes	This function performs no operation when the DMA channel is stopped.

Table 3.42. `grspw_dma_rx_recv` function declaration

Proto	`int grspw_dma_rx_recv(void *c, int opts, struct grspw_list *pkts, int *count)`
About	Get received RX packet buffers that has previously been scheduled for reception with `grspw_dma_rx_prepare`. The packets in the RX RECV queue are moved to the `pkts` packet list. When the move has been completed the packet(s) can safely be reused again by the user. The packet structures have been updated with reception status to indicate transfer failures of individual packets, received packet length. The header pointer and length fields are not touched by the driver, all data ends up in the data area. Depending on the input arguments a selection of the below steps are performed: 1. Move scheduled packets to RECV List (SCHED->RECV). 2. Move all RECV packet to the callers list (RECV->USER). 3. Schedule as many free packet buffers as possible (READY->SCHED). Skipping both step 1 and 3 may be useful when IRQ is enabled, then the worker thread will be responsible for descriptor processing. Skipping only step 2 can be useful in polling mode. The fastest solution in retrieving received RX packets and preparing new packet buffers for future receive, is to call: 1. grspw_dma_rx_recv(opts=2, &recvlist) (Skip step 3) 2. grspw_dma_rx_prepare(opts=1, &freelist) (Skip step 1) NOTE: the RXPKT_FLAG_RX flag indicates if a packet was received, thus if the data field contains new valid data or not.
Param	`c` [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	`opts` [IN] Integer bit-mask The above steps 1 and/or 3 may be skipped by setting `opts` argument according the description below.

Bit	Description
0	Set to 1 to skip Step 1.
1	Set to 1 to skip Step 3.

Param	`pkts` [OUT] pointer The list will be initialized to contain the SpaceWire packets moved from the RECV queue to the packet list. The grspw_list structure will be initialized so that `head` points to the first packet, `tail` points to the last and the last packet (tail) next pointer is NULL. Call this function with `pkts` set to NULL to avoid step 2. Just doing step 1 and 3 as determined by `opts` is normally performed in polling-mode. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 3.36. Note that RXPKT_FLAG_RX of the `flags` field indicates if the packet was received or not. In case of DMA being stopped one can use this flag to see if the packet was received or not. The TRUNK, DCRC, HCRC and EEOP flags indicates if an error occurred during reception of the packet, regardless the RXPKT_FLAG_RX is set to indicate packet reception attempt.
Param	`count` [IO] pointer

	Number of packets in the packet list.	
	Value	Input Description
	NULL	Move all packets from the RECV queue to the packet list.
	-1	Move all packets from the RECV queue to the packet list.
	0	No packets are moved. Same as if pkts is NULL.
	>0	Move a maximum of '*count' packets to the packet list.
	Value	Output Description
	NULL	Nothing performed.
	others	'*count' is updated to reflect number of packets in packet list.
Return	Status. See return codes below	
	Value	Description
	-1	Error occurred, DMA channel may not be valid.
	0	Successful. pkts list filled with all packets from recv list.
	1	Indicates that DMA is stopped. Same as 0 but step 1 and 3 were never done.
Notes	This function can only do step 1 and 2 to allow read out received packets when in stopped mode. This is useful when a link goes down and the DMA activity is stopped by user or by driver automatically.	

3.4.7. Transmission queue status

The current status of send and receive transmissions can be obtained by looking at the DMA channel's packet queues. Note that the queues content does not change unless the user calls the driver to perform work or if the work thread triggered via DMA interrupts is enabled. The current number of packets actually processed by hardware can also be read using the functions described below.

Table 3.43. `grspw_dma_tx_count` function declaration

Proto	`void grspw_dma_tx_count(void *c, int *send, int *sched, int *sent, int *hw)`
About	Get current number of packets in respective TX queue and current number of unhandled packets that hardware processed (from descriptor table).
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*send* [OUT] pointer If not NULL the TX SEND Queue count is stored into the address of *send*.
Param	*sched* [OUT] pointer If not NULL the TX SCHED Queue count is stored into the address of *sched*.
Param	*sent* [OUT] pointer If not NULL the TX SENT Queue count is stored into the address of *sent*.
Param	*hw* [OUT] pointer If not NULL the number of packets completed transmitted by hardware. This is determined by looking at the TX descriptor pointer register. The number represents how many of the SCHED queue that actually have been transmitted by hardware but not handled by the driver yet. The number is stored into the address of *hw*.
Return	None.

Table 3.44. `grspw_dma_rx_count` function declaration

Proto	`void grspw_dma_rx_count(void *c, int *ready, int *sched, int *recv, int *hw)`
About	Get current number of packets in respective RX queue and current number of unhandled packets that hardware processed (from descriptor table).

Param	*c* [IN] pointer DMA channel identifier. Returned from grspw_dma_open.
Param	*ready* [OUT] pointer If not NULL the RX READY Queue count is stored into the address of *ready*.
Param	*sched* [OUT] pointer If not NULL the RX SCHED Queue count is stored into the address of *sched*.
Param	*recv* [OUT] pointer If not NULL the RX RECV Queue count is stored into the address of *recv*.
Param	*hw* [OUT] pointer If not NULL the number of packets completed received by hardware. This is determined by looking at the RX descriptor pointer register. The number represents how many of the SCHED queue that actually have been received by hardware but not handled by the driver yet. The number is stored into the address of *hw*.
Return	None.

3.4.8. Statistics

The driver counts statistics at certain events. The driver's DMA channel counters can be read out using the DMA API. The number of interrupts serviced by the worker task, packet transmission statistics, packet transmission errors and packet queue statistics can be obtained.

> The read function is not protected by locks. A GRSPW interrupt or other tasks performing driver operations on the same device could cause the statistics to be out of sync. Similar to the statistic functionality of the device API.

```
struct grspw_dma_stats {
 /* IRQ Statistics */
 int irq_cnt;            /* Number of DMA IRQs generated by channel */

 /* Descriptor Statistics */
 int tx_pkts;            /* Number of Transmitted packets */
 int tx_err_link;        /* Number of Transmitted packets with Link Error*/
 int rx_pkts;            /* Number of Received packets */
 int rx_err_trunk;       /* Number of Received Truncated packets */
 int rx_err_endpkt;      /* Number of Received packets with bad ending */

 /* Diagnostics to help developers sizing their number buffers to avoid
  * out-of-buffers or other phenomenons.
  */
 int send_cnt_min;       /* Minimum number of packets in TX SEND queue */
 int send_cnt_max;       /* Maximum number of packets in TX SEND queue */
 int tx_sched_cnt_min;   /* Minimum number of packets in TX SCHED queue */
 int tx_sched_cnt_max;   /* Maximum number of packets in TX SCHED queue */
 int sent_cnt_max;       /* Maximum number of packets in TX SENT queue */
 int tx_work_cnt;        /* Times the work thread processed TX BDs */
 int tx_work_enabled;    /* No. TX BDs enabled by work thread */

 int ready_cnt_min;      /* Minimum number of packets in RX READY queue */
 int ready_cnt_max;      /* Maximum number of packets in RX READY queue */
 int rx_sched_cnt_min;   /* Minimum number of packets in RX SCHED queue */
 int rx_sched_cnt_max;   /* Maximum number of packets in RX SCHED queue */
 int recv_cnt_max;       /* Maximum number of packets in RX RECV queue */
 int rx_work_cnt;        /* Times the work thread processed RX BDs */
 int rx_work_enabled;    /* No. RX BDs enabled by work thread */
};
```

Table 3.45. grspw_dma_stats data structure declaration

irq_cnt	Number of interrupts serviced for this DMA channel.
tx_pkts	Number of transmitted packets with link errors.
tx_err_link	Number of transmitted packets with link errors.
rx_pkts	Number of received packets.
rx_err_trunk	Number of received Truncated packets.

rx_err_endpkt	Number of received packets with bad ending.
send_cnt_min	Minimum number of packets in TX SEND queue.
send_cnt_max	Maximum number of packets in TX SEND queue.
tx_sched_cnt_min	Minimum number of packets in TX SCHED queue.
tx_sched_cnt_max	Maximum number of packets in TX SCHED queue.
sent_cnt_max	Maximum number of packets in TX SENT queue.
tx_work_cnt	Times the work thread processed TX BDs.
tx_work_enabled	Number of TX BDs enabled by work thread.
ready_cnt_min	Minimum number of packets in RX READY queue.
ready_cnt_max	Maximum number of packets in RX READY queue.
rx_sched_cnt_min	Minimum number of packets in RX SCHED queue.
rx_sched_cnt_max	Maximum number of packets in RX SCHED queue.
recv_cnt_max	Maximum number of packets in RX RECV queue.
rx_work_cnt	Times the work thread processed RX BDs.
rx_work_enabled	Number of RX BDs enabled by work thread.

Table 3.46. `grspw_dma_stats_read` function declaration

Proto	`void grspw_dma_stats_read(void *d, struct grspw_dma_stats *sts)`
About	Reads the current driver statistics collected from earlier events by a DMA channel and DMA channel usage. The statistics are stored to the address given by the second argument. The layout and content of the statistics are defined by the grspw_dma_stats data structure is described in Table 3.45. Note that the snapshot is taken without lock protection, as a consequence the statistics may not be synchronized with each other. This could be caused if the function is interrupted by a the GRSPW interrupt or other tasks performing driver operations on the same DMA channel.
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*`sts`* [OUT] pointer A snapshot of the current driver statistics are copied to this user provided buffer. The layout and content of the statistics are defined by the grspw_dma_stats data structure is described in Table 3.45.
Return	None.

Table 3.47. `grspw_dma_stats_clr` function declaration

Proto	`void grspw_dma_stats_clr(void *c)`
About	Resets one DMA channel's statistical counters. Most of the driver's counters are set to zero, however some have other initial values, for example the *`send_cnt_min`*.
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Return	None.

3.4.9. DMA channel configuration

Various aspects of DMA transfers can be configured using the functions described in this section. The configuration affects:

- DMA transfer options, no-spill, strip address/PID.
- Receive max packet length.
- RX/TX Interrupt generation options.

```
struct grspw_dma_config {
 int flags;              /* DMA config flags, see DMAFLAG_* options */
 int rxmaxlen;           /* RX Max Packet Length */
 int rx_irq_en_cnt;      /* Enable RX IRQ every cnt descriptors */
 int tx_irq_en_cnt;      /* Enable TX IRQ every cnt descriptors */
};
```

Table 3.48. grspw_dma_config data structure declaration

flags	RX/TX DMA transmission options See below.	
	Bits	Description (prefixed DMAFLAG_ or DMAFLAG2_)
	NO_SPILL	Enable (1) or Disable (0) packet spilling, flow control.
	STRIP_ADR	Enable (1) or Disable (0) stripping node address byte from DMA write transfers (packet reception). See hardware support to determine if present in hardware. See hardware documentation about DMA CTRL SA bit.
	STRIP_PID	Enable (1) or disable (0) stripping PID byte from DMA write transfers (packet reception).(if CRC is available in hardware). See hardware support to determine if present in hardware. See hardware documentation about DMA CTRL SP bit.
	TXIE	Enable (1) or disable (0) DMA TX interrupts on DMA transmission. This affects the DMA-CTRL TI register bit. This can be used in combination with packet flags to allow the user to control precisely which TX SpW buffers will generate interrupt(s) on send completed.
	RXIE	Enable (1) or disable (0) DMA RX interrupts on DMA reception. This affects the DMA-CTRL RI register bit. This can be used in combination with packet flags to allow the user to control precisely which RX SpW buffers will generate interrupt(s) on receive completed.
rxmaxlen	Max packet reception length. Longer packets with will be truncated see RXPKT_FLAG_TRUNK flag in packet structure.	
rx_irq_en_cnt	Controls RX interrupt generation. This integer number enable RX DMA IRQ every 'cnt' descriptors.	
tx_irq_en_cnt	Controls TX interrupt generation. This integer number enable TX DMA IRQ every 'cnt' descriptors.	

Table 3.49. `grspw_dma_config` function declaration

Proto	`int grspw_dma_config(void *c, struct grspw_dma_config *cfg)`
About	Set the DMA channel configuration options as described by the input arguments. It is only possible the change the configuration on stopped DMA channels, otherwise an error code is returned. The hardware registers are not written directly. The settings take effect when the DMA channel is started calling `grspw_dma_start`.
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*`cfg`* [IN] pointer Address to where the driver will read or write the DMA channel configuration from. The configuration options are described in Table 3.48.
Return	int. Return code as indicated below.

Value	Description
0	Success.
-1	Failure due to invalid input arguments or DMA has already been started.

Table 3.50. `grspw_dma_config_read` function declaration

Proto	`void grspw_dma_config_read(void *c, struct grspw_dma_config *cfg)`

About	Copies the DMA channel configuration to user defined memory area.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*sts* [OUT] pointer The driver DMA channel configuration options are copied to this user provided buffer. The layout and content of the statistics are defined by the grpsw_dma_config data structure is described in Table 3.48.
Return	None.

3.4.10. DMA event callback

By default DMA interrupts are handled by the work queue, which processes the descriptor tables and wakes tasks blocked in the RX/TX wait functions, see Section 3.2.11. A kernel user can instead register a DMA event callback per DMA channel, for example to process packets from a NAPI poll function. The callback is called by the GRSPW interrupt handler, in hard interrupt context with the device lock held, after the RX/TX DMA interrupts of the channel have been disabled.

While a callback is registered the driver does not process the descriptor tables of the channel, does not wake tasks in `grspw_dma_rx_wait` or `grspw_dma_tx_wait` and does not stop the DMA channel on DMA AHB errors. The callback must not call any GRSPW Kernel Library function. It typically schedules work or NAPI, which calls `grspw_dma_rx_recv`, `grspw_dma_tx_reclaim` and so on, then re-enables the interrupts with `grspw_dma_irq_enable` and finally checks `grspw_dma_rx_pending` for packets received before the interrupts were enabled. On a DMA AHB error, indicated by GRSPW_DMA_STATUS_ERROR in the status given to the callback, the user should call `grspw_dma_stop` from the scheduled work.

Table 3.51. `grspw_dma_set_event_cb` function declaration

Proto	`int grspw_dma_set_event_cb(void *c, event_cb_t cb, void *cb_data)`
About	Registers a DMA event callback for the DMA channel, or removes it when *cb* is NULL. The callback type is `typedef void (*event_cb_t)(void *cb_data, unsigned int dma_stat)`. When a callback is registered the GRSPW interrupt handler calls it instead of scheduling DMA work on the work queue. The interrupt handler has then disabled RX/TX DMA interrupts of the channel. The driver does not process the descriptor tables, does not wake tasks in `grspw_dma_rx_wait` or `grspw_dma_tx_wait`, and does not stop the channel on DMA AHB errors. The user is responsible for this, typically by scheduling work or NAPI from the callback that calls `grspw_dma_rx_recv`, `grspw_dma_tx_reclaim` and so on, and then `grspw_dma_irq_enable`. The callback is called in hard interrupt context with the device lock held. It must not call any GRSPW Kernel Library function.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*cb* [IN] pointer to function Callback, or NULL to remove the callback. Called with *cb_data* and the DMA channel control/status register value *dma_stat*. The bits GRSPW_DMA_STATUS_RA and GRSPW_DMA_STATUS_TA (GRSPW_DMA_STATUS_ERROR) indicate an RX or TX AHB error.
Param	*cb_data* [IN] pointer to custom data Given as the first argument to the callback.
Return	int. Return code as indicated below.

Value	Description
0	Success.
-1	The DMA channel is started. The callback can only be changed in stopped mode.

Notes	The callback is removed by grspw_dma_open and grspw_dma_close.

Table 3.52. `grspw_dma_irq_enable` function declaration

Proto	`void grspw_dma_irq_enable(void *c)`
About	Re-enables the RX and TX DMA interrupts of the DMA channel according to the DMA channel configuration, after they have been disabled by the GRSPW interrupt handler. Intended to be used together with a DMA event callback, see `grspw_dma_set_event_cb`. The interrupts are not enabled if a DMA AHB error is pending.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Return	None.

Table 3.53. `grspw_dma_rx_pending` function declaration

Proto	`int grspw_dma_rx_pending(void *c)`	
About	Checks if the oldest scheduled RX descriptor has been completed by hardware. Intended to be called after `grspw_dma_irq_enable` to detect packets received before the interrupt was enabled, which would otherwise not generate an interrupt.	
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.	
Return	int. Return code as indicated below.	
	Value	Description
	0	No received packet is pending.
	1	At least one received packet is waiting to be processed.

3.5. API reference

This section lists all functions and data structures part of the GRSPW driver API, and in which section(s) they are described. The API is also documented in the source header file of the driver, see Section 3.1.2.

3.5.1. Data structures

The data structures used together with the Device and/or DMA API are summarized in the table below.

Table 3.54. Data structures reference

Data structure name	Section
struct grspw_pkt	3.4.3
struct grspw_list	3.2.9
struct grspw_addr_config	3.3.5
struct grspw_hw_sup	3.3.3
struct grspw_core_stats	3.3.9
struct grspw_dma_config	3.4.9
struct grspw_dma_stats	3.4.8

3.5.2. Device functions

The GRSPW device API. The functions listed in the table below operates on the GRSPW common registers and driver set up. Changes here typically affects all DMA channels and link properties.

Table 3.55. Device function reference

Prototype	Section
`int grspw_dev_count(void)`	3.3.1

Prototype	Section
`void *grspw_open(int dev_no)`	3.3.1
`int grspw_close(void *d)`	3.3.1
`int grspw_initialize_user(void *(*devfound)(int), void (*devremove)(int, void *))`	3.3.2
`void grspw_uninitialize_user(void *(*devfound)(int))`	3.3.2
`struct device *grspw_get_device(void *d)`	3.3.2
`struct device *grspw_get_device_by_index(int dev_no)`	3.3.2
`void grspw_hw_support(void *d, struct grspw_hw_sup *hw)`	3.3.3
`void grspw_stats_read(void *d, struct grspw_core_stats *sts)`	3.3.9
`void grspw_stats_clr(void *d)`	3.3.9
`void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)`	3.3.5, 3.2.7
`spw_link_state_t grspw_link_state(void *d)`	3.3.4, 3.2.3
`void grspw_link_ctrl(void *d, int *options, int *stscfg, int *clkdiv)`	3.3.4, 3.2.3
`unsigned int grspw_link_status(void *d)`	3.3.4, 3.2.3
`void grspw_link_status_clr(void *d, unsigned int mask)`	3.3.4, 3.2.3
`void grspw_tc_ctrl(void *d, int *options)`	3.3.6, 3.2.4
`void grspw_tc_tx(void *d)`	3.3.6, 3.2.4
`void grspw_tc_isr(void *d, void (*tcisr)(void *data, int tc), void *data)`	3.3.6, 3.2.4
`void grspw_tc_time(void *d, int *time)`	3.3.6, 3.2.4
`int grspw_rmap_ctrl(void *d, int *options, int *dstkey)`	3.3.8, 3.2.5
`void grspw_rmap_support(void *d, char *rmap, char *rmap_crc)`	3.3.8, 3.2.5, 3.3.3
`int grspw_port_ctrl(void *d, int *port)`	3.3.7, 3.2.6
`int grspw_port_count(void *d)`	3.3.7, 3.2.6, 3.3.3
`int grspw_port_active(void *d)`	3.3.7, 3.2.6

3.5.3. DMA functions

The GRSPW DMA channel API. The functions listed in the table below operates on one GRSPW DMA channel and its driver set up. This interface is used to send and receive SpaceWire packets.

GRSPW2 and GRSPW2_DMA devices supports more than one DMA channel.

Table 3.56. DMA channel function reference

Prototype	Section
`void *grspw_dma_open(void *d, int chan_no)`	3.2.1, 3.4.1, 3.3.1
`int grspw_dma_close(void *c)`	3.2.1, 3.4.1, 3.3.1
`int grspw_dma_start(void *c)`	3.4.2, 3.2.12
`void grspw_dma_stop(void *c)`	3.4.2, 3.2.12
`int grspw_dma_rx_recv(void *c, int opts, struct grspw_list *pkts, int *count)`	3.4.6, 3.2.1
`int grspw_dma_rx_prepare(void *c, int opts, struct grspw_list *pkts, int count)`	3.4.6, 3.2.1
`void grspw_dma_rx_count(void *c, int *ready, int *sched, int *recv, int *hw)`	3.4.7, 3.2.9.1
`int grspw_dma_rx_wait(void *c, int recv_cnt, int op, int ready_cnt, long timeout)`	3.4.4, 3.2.10
`int grspw_dma_tx_send(void *c, int opts, struct grspw_list *pkts, int count)`	3.4.5, 3.2.1
`int grspw_dma_tx_reclaim(void *c, int opts, struct grspw_list *pkts, int *count)`	3.4.5, 3.2.1
`void grspw_dma_tx_count(void *c, int *send, int *sched, int *sent, int *hw)`	3.4.7, 3.2.9.1
`int grspw_dma_tx_wait(void *c, int send_cnt, int op, int sent_cnt, long timeout)`	3.4.4, 3.2.10
`int grspw_dma_config(void *c, struct grspw_dma_config *cfg)`	3.4.9
`void grspw_dma_config_read(void *c, struct grspw_dma_config *cfg)`	3.4.9
`void grspw_dma_stats_read(void *c, struct grspw_dma_stats *sts)`	3.4.8
`void grspw_dma_stats_clr(void *c)`	3.4.8
`void grspw_dma_addr_ctrl(void *c, int node_en, unsigned char node_addr, unsigned char node_mask)`	3.3.5, 3.2.7
`int grspw_dma_set_event_cb(void *c, event_cb_t cb, void *cb_data)`	3.4.10
`void grspw_dma_irq_enable(void *c)`	3.4.10
`int grspw_dma_rx_pending(void *c)`	3.4.10

4. SpaceWire Router APB Register Driver

4.1. Introduction

This section describes the Linux Frontgrade Gaisler SpaceWire Router APB registers kernel driver. It provides user space applications with a SpaceWire Router configuration interface. The driver allows the user to configure the router and control the SpaceWire links.

The SpaceWire router is accessed using the standard UNIX `ioctl` routine.

4.1.1. Sources

The GRSPW driver sources are provided under the GPL license, they are available in the GRLIB driver package as described in the table below. Applications include only the user space interface header. All files are relative to `kernel/drivers/grlib/` in the driver package.

Table 4.1. SpaceWire Router driver sources

Location	Description
`spw/grspw_router.c`	SpaceWire Router APB Registers Driver
`include/uapi/linux/grlib/grspw_router.h`	SpaceWire Router APB Registers user space interface header
`include/linux/grlib/grspw_router.h`	SpaceWire Router APB Registers header, kernel only (includes the user space interface header)

4.1.2. Using the driver

Applications wanting to access SpW Router registers from user-space include `<linux/grlib/grspw_router.h>`. The header is self-contained: it needs only `<linux/types.h>` and `<linux/ioctl.h>`. In the driver package it is in `kernel/drivers/grlib/include/uapi`, so a build adds `-I`*`drvpkg`*`/kernel/drivers/grlib/include/uapi` to the compiler flags. A distribution may install it in `/usr/include/linux/grlib`. The headers in `include/linux/grlib` are for the kernel only, and stop a user space build with `#error`.

The user space interface header is licensed `GPL-2.0-or-later WITH Linux-syscall-note`, as the kernel's own user space headers are, so applications built with it are not covered by the GPL.

Each SpW Router core is accessed through a single device node, `/dev/spwrouter`*`N`* (at most four routers). The Major/Minor numbers are assigned dynamically, see Section 1.5.

4.1.3. Examples

Within the GRLIB driver package there is a user space example of how this driver can be used, the example file is named `user/grspw_router/spwrouter_custom_config.c`. It uses the small helper library `user/grspw_router/grspw_router_lib.c`, whose `router_set_treload()` checks the port number against the hardware information (HWINFO). The command line tool **spwrtr**, described in Section 4.2, can do the same from the command line or from a file.

4.2. The spwrtr command line tool

spwrtr (`user/grspw_router/spwrtr.c`) shows and changes the routing table, the ports and the router configuration through the control interface described in the next section. It changes only what it is told to, and prints each change it makes:

```
spwrtr [-d DEVICE] [-n] [-q] [-x] COMMAND [ARGS]
```

- `-d DEVICE`: the router device, by default `/dev/spwrouter0`.
- `-n`: dry run, print the changes without writing them.
- `-q`: do not print the changes.
- `-x`: also show the raw register values.

spwrtr help lists the commands. The tool exits with status 1 and a message, with the file name and line number when the command comes from a file, on the first error.

4.2.1. Showing the router

- **info**: the hardware (ports, timers, Plug-and-Play) and the router configuration.
- **show [cfg|ports|routes]**: the current state, all three parts by default.
- **port PORTS [show]**: link state, clock divisor, timer reload, control flags and errors of the ports. Port 0, the configuration port, shows its own error bits and the latest RMAP and Plug-and-Play error codes.
- **route [show [ADDRS]]**: the routing table, as ranges of addresses with the same entry. Physical addresses routed only to their own port are shown as `own port`.
- **tc**: the current time-code.

4.2.2. Changing the router

PORTS is a list of port numbers and ranges such as `1,3-5`, or `all`, `spw`, `amba` and `fifo` for the ports of a type. ADDRS is a list of addresses and ranges such as `0x20,0x28-0x2f`; 1 to 31 are physical addresses and 32 to 255 logical addresses. Numbers are read as in C: decimal, hexadecimal with `0x`, and octal with a leading `0`.

- **set OPTION VALUE...**: the router configuration. `selfaddr on|off` sets self addressing, `iid N` the instance ID, `idiv N` the initialization clock divisor, `tpresc N` the timer prescaler, `config RAW` the configuration register (without the reset and memory error bits, see CFG_SET), and `cfgwe on|off` whether RMAP may write through the configuration port.
- **port PORTS OPTION...**: port control. `start` and `stop` start and stop the link, `clear-errors` clears the error bits (and on port 0 the error codes), `clkdiv N` sets the run state clock divisor, `reload N` the timer reload value, `prio high|low` the priority, and `ctrl RAW` the whole port control register. `autostart`, `autodisconnect`, `linkreq`, `timer`, `disable`, `timecode`, `cfgaccess` and `spill` take `on` or `off`. The link options exist only on the SpaceWire ports: with a port type such as `all` they are skipped for the other ports, with a port number they are an error. Port 0 accepts only `timer`, `reload`, `ctrl` and `clear-errors`. The options are checked for all ports before any port is written.
- **route set ADDRS OPTION...**: routing table entries. `port LIST` sets the output ports and enables the entry, `dist on|off` packet distribution, `hdel on|off` header deletion, `prio high|low` the priority, `spill on|off` spill if not ready and `enable on|off` the entry. A physical address is always routed to its own port, and only `port` and `dist` can be set for it.
- **route del ADDRS**: makes logical addresses invalid and routes physical addresses only to their own port.
- **route defaults**: the routing table of `spwrouter_custom_config`.
- **route clear**: **route del** on all addresses.

The routing table is read once, changed by the commands, and written before the next port change or when all commands have run, so the routes are in place before a link is started. Only the changed entries are printed.

4.2.3. Configuration files

load FILE runs the commands in a file, one per line, without the **spwrtr** name. Text after # is a comment. A file can run other files with **include FILE**, relative to the including file, up to eight levels deep. **load** first checks every command in the file, and its includes, without writing anything, so a mistake in the file leaves the router unchanged.

save FILE writes the router configuration, the routing table and the port control and timer reload of every port as commands that **load** reads back. `-` writes to standard output. The configuration write enable cannot be read back and is not saved.

`user/grspw_router/base.cfg` sets what `spwrouter_custom_config` sets:

```
# No RMAP writes through the configuration port
set cfgwe off

# Self addressing on, everything else in the configuration register off
set config 0x10
set iid 1

# 0x20-0x2f to port 1, then eight logical addresses per port, all with
# header deletion. Physical addresses go to their own port.
route defaults
```

```
# SpaceWire ports: clkdiv 1, timecode, cfgaccess, autostart and start on
port spw ctrl 0x0100002e
# AMBA and FIFO ports: timecode and cfgaccess on
port amba,fifo ctrl 0x00000028
```

4.2.4. Examples

```
# Configure the router as spwrouter_custom_config does
spwrtr load base.cfg

# Route logical address 0x40 to ports 2 and 3 with packet distribution
spwrtr route set 0x40 port 2,3 dist on hdel on

# Run the SpaceWire links at a lower rate, and show the result
spwrtr port spw clkdiv 4
spwrtr port spw

# See what a file would change, without changing anything
spwrtr -n load my.cfg

# Save the current configuration and restore it later
spwrtr save saved.cfg
spwrtr load saved.cfg
```

4.3. Control Interface

4.3.1. Overview

The SpaceWire router can be configured using the control interface described in this section. The interface is router hardware specific and a good knowledge of the hardware is necessary. See hardware documentation. The data structures are described in Section 4.3.2.

The control interface is accessed using the standard UNIX `ioctl` routine.

All GRLIB drivers use the ioctl magic `'J'` (`GRLIB_IOCTL_MAGIC`), with a range of command numbers each: 0x00-0x3f grspw_user, 0x40-0x5f grspw_router (this driver), 0x80-0xbf grspfi; 0x60-0x7f and 0xc0-0xff are for other GRLIB drivers. A command sent to another driver's device, and an unknown command, fail with ENOTTY. The following rules are part of the interface:

- All structs have a fixed layout, the same for every ABI: every field has a fixed-size type and every hole is a reserved field. The same structs therefore work for 32-bit user space on a 64-bit kernel.
- A struct never changes size. A larger version gets a new ioctl number.
- User space sets input reserved fields to zero and ignores output reserved fields. The driver rejects nonzero input reserved fields and unknown flags with EINVAL.
- HWINFO reports the interface version in *`api_version`*, which is `GRSPWR_API_VERSION` (2) for the interface described here, and optional driver features in *`features`*, none defined yet (0).

In the table below all currently supported `ioctl` commands, their numbers and their argument are listed. All router commands starts with GRSPWR_IOCTL_ which has to be added to the command name given in the table below. The data direction below indicates in which direction data is transferred to the kernel:

- Input: Argument is an address. The driver reads data from the given address.
- Output: Argument is an address. The driver writes data to the given address.
- Input/Output: both above cases.
- Argument: an integer passed in place of the address, no data transferred between kernel/user.
- None: Argument ignored.

Table 4.2. `ioctl` commands supported by the SpaceWire Router driver.

Command (number)	Data Direction	Argument Type	Description
HWINFO (0x41)	Output	struct grspwr_hw_info *	Copy the interface version and the hardware configuration of the router core, such as number of SpaceWire ports, number of AMBA ports, number of FIFO ports, etc.
CFG_SET (0x42)	Input	struct grspwr_config *	Configure the router by writing the configuration bits of the Control/Status register (bit 2, ME, and bit 7, RE, are never written: use CFGSTS_SET to reset the router), setting the Instance ID, Start up Clock Divisor, Timer

Command (number)	Data Direction	Argument Type	Description
			prescaler and the timer reload registers. The *flags* field selects which of them are written, see GRSPWR_FLG_*. *timer_reload* is indexed by port number, starting with port 0 (the configuration port). Only entries 0 to *N* are written, where *N* is the number of SpaceWire, AMBA and FIFO ports reported by HWINFO. EINVAL is returned, and nothing is written, if *flags* has unknown bits or a reserved field is not zero.
CFG_GET (0x43)	Output	struct grspwr_config *	Reads the current router configuration into the user specified memory area. Only *timer_reload* entries 0 to *N* are read; the other entries, *flags* and the reserved fields are returned as zero.
ROUTES_SET (0x44)	Input	struct grspwr_routes *	Configure the 224 words long router table.
ROUTES_GET (0x45)	Output	struct grspwr_routes *	Copy the current 224 words long router table to user provided buffer.
PS_SET (0x46)	Input	struct grspwr_ps *	Configure the port setup registers according to user buffer.
PS_GET (0x47)	Output	struct grspwr_ps *	Copy the current port setup registers to user buffer.
WE_SET (0x48)	Argument	integer, 0 or 1	The argument is the value, not a pointer. If it is 1 the WE bit in the configuration write enable register is set, if it is 0 it is cleared. Any other value gives EINVAL. The WE bit only controls RMAP writes through the configuration port (port 0); the registers can always be written through this driver.
PORT (0x49)	Input/Output	struct grspwr_port *	Read and/or Write (in that order) the port control and port status registers of one port of the SpaceWire router. The *flag* field determines which operations should be performed. See GRSPWR_PORTFLG_*. The *port* field selects which port is to be read/written. The values returned are the ones read before the write. Valid ports are 0 to *N* (see CFG_SET), other ports and unknown *flag* bits give `-EINVAL`.
CFGSTS_SET (0x4a)	Argument	integer	Writes the argument to the Config/Status register as it is. The argument is the register value, not a pointer.
CFGSTS_GET (0x4b)	Output	__u32 *	Copies the current value of the Config/Status register to the user provided buffer.
TC_GET (0x4c)	Output	__u32 *	Copies the current value of the Time-code register to the user provided buffer.

The `ioctl` commands return 0 on success, or one of the following errors:

- `-EFAULT`: the user buffer is invalid.
- `-EINVAL`: CFG_SET was given unknown *flags* or a nonzero reserved field, PORT was given unknown *flag* bits or a port number outside 0 to *N*, or WE_SET a value other than 0 and 1.
- `-ENODEV`: the router device has been removed.
- `-ENOTTY`: unknown command, or a command of another driver.

Opening a device node without a router returns `-ENODEV`.

4.3.2. Data structures and constants

The structures and constants are listed below as declared in `grspw_router.h`. Fields named *reserved* must be zero on input and are to be ignored on output.

```
#define GRSPWR_API_VERSION 2
```

```
/* Hardware information (HWINFO) */
struct grspwr_hw_info {
 __u32 api_version; /* GRSPWR_API_VERSION of the driver */
 __u32 features;  /* Optional driver features, none yet */
 __u8 nports_spw; /* SpaceWire ports */
 __u8 nports_amba; /* AMBA ports */
 __u8 nports_fifo; /* FIFO ports */
 __u8 timers_avail; /* Port timers available */
 __u8 pnp_avail;  /* Plug and play available */
 __u8 ver_major;
 __u8 ver_minor;
 __u8 ver_patch;
 __u8 iid;  /* Instance ID */
 __u8 reserved[3];
};

/* flags of struct grspwr_config: what CFG_SET writes */
#define GRSPWR_FLG_CFG  0x01 /* config */
#define GRSPWR_FLG_IID  0x02 /* iid */
#define GRSPWR_FLG_IDIV  0x04 /* idiv */
#define GRSPWR_FLG_TPRES 0x08 /* timer_prescaler */
#define GRSPWR_FLG_TRLD  0x10 /* timer_reload */
#define GRSPWR_FLG_ALL  0x1f

/* CFG_SET, CFG_GET */
struct grspwr_config {
 __u32 flags;  /* GRSPWR_FLG_*, CFG_SET only */
 __u32 config;  /* Configuration bits of the config/status
     * register; CFG_SET never writes ME (bit 2)
     * or RE (bit 7), use CFGSTS_SET for those
     */
 __u8 iid;  /* Instance ID */
 __u8 idiv;  /* Link initialization clock divisor */
 __u8 reserved0[2];
 __u32 timer_prescaler;
 __u32 timer_reload[32]; /* Indexed by port number */
 __u32 reserved[4];
};

/* ROUTES_SET, ROUTES_GET: logical routing table, addresses 32 to 255 */
struct grspwr_routes {
 __u32 route[224];
};

/* PS_SET, PS_GET: port setup registers */
struct grspwr_ps {
 __u32 ps[31];  /* Ports 1 to 31 */
 __u32 ps_logical[224]; /* Logical addresses 32 to 255 */
};

/* flag of struct grspwr_port */
#define GRSPWR_PORTFLG_SET_CTRL 0x01 /* Write ctrl */
#define GRSPWR_PORTFLG_GET_CTRL 0x02 /* Read ctrl (before the write) */
#define GRSPWR_PORTFLG_SET_STS 0x04 /* Write sts */
#define GRSPWR_PORTFLG_GET_STS 0x08 /* Read sts (before the write) */
#define GRSPWR_PORTFLG_ALL 0x0f

/* PORT */
struct grspwr_port {
 __u32 flag;  /* GRSPWR_PORTFLG_* */
 __u32 port;  /* 0 to the number of ports */
 __u32 ctrl;  /* Port control register */
 __u32 sts;  /* Port status register */
};
```

5. GRSPFI SpaceFibre Driver

5.1. Introduction

This section describes the Linux GRSPFI driver. It provides user-space applications with a SpaceFibre link control and packet transfer interface. The driver is implemented using the GRSPFI Kernel library (described in Chapter 6) for GRSPFI device control and DMA transfer, and it imports a dma-buf allocated from the kernel's CMA dma-heap (a device under `/dev/dma_heap/`, allocated via `DMA_HEAP_IOCTL_ALLOC`) for physically continuous device memory (DMA memory) shared with user-space.

The `grspfi` module registers with the `grspfi_drv` kernel module and creates a hierarchy of character device nodes in the `/dev` filesystem:

- One *control device* per physical GRSPFI core: `/dev/grspfi`*`N`* (e.g. `/dev/grspfi0`).
- One *Virtual Channel device* per VC discovered at open time: `/dev/grspfi`*`N`*`_`*`V`* (e.g. `/dev/grspfi0_0`, `/dev/grspfi0_1`). VC devices are created when the control device is opened the first time and removed with the GRSPFI device. A VC device can only be opened while the control device is open (`-ENODEV` otherwise), and once the control device is closed the ioctls of a VC device that is still open fail with `-ENODEV`. The control device cannot be opened again until all its VC devices are closed (`-EBUSY`). Closing a VC device disables the VC if it is still enabled, and first disables (with a warning) the DMA channels that serve it.

Closing the control device disables any DMA channel still enabled (with a warning) and removes the interrupt callbacks, closes the Broadcast Channel, detaches the pool dma-buf, and closes the GRSPFI device, which resets the link and disables the core.

The control device and each VC device provide the read-only sysfs attributes `tx_packets` and `rx_packets`: the packet counts of the VC, or the sum over all VCs for the control device.

All user interaction is performed via `ioctl`. Each device node can only be open once at a time (`-EBUSY` is returned on further opens).

5.1.1. Sources

The GRSPFI driver sources are provided under the GPL license, they are available in the GRLIB driver package as described in the table below. Applications include only the user space interface header. All files are relative to `kernel/drivers/grlib/` in the driver package.

Table 5.1. GRSPFI driver sources

Location	Description
`spf/grspfi_drv.c`	GRSPFI kernel library
`spf/grspfi.c`	GRSPFI character device (user-space interface)
`include/linux/grlib/grspfi_drv.h`	GRSPFI kernel library header, kernel only
`include/uapi/linux/grlib/grspfi.h`	GRSPFI character driver user space interface header
`include/linux/grlib/grspfi.h`	GRSPFI character driver header, kernel only (includes the user space interface header)

5.1.2. Using the driver

Applications wanting to access GRSPFI devices from user-space include `<linux/grlib/grspfi.h>`. The header is self-contained: it needs only `<linux/types.h>` and `<linux/ioctl.h>`, not the GRSPFI kernel library header. In the driver package it is in `kernel/drivers/grlib/include/uapi`, so a build adds `-I`*`drvpkg`*`/kernel/drivers/grlib/include/uapi` to the compiler flags. A distribution may install it in `/usr/include/linux/grlib`. The headers in `include/linux/grlib` are for the kernel only, and stop a user space build with `#error`.

The user space interface header is licensed `GPL-2.0 WITH Linux-syscall-note`, as the kernel's own user space headers are, so applications built with it are not covered by the GPL.

Since the driver uses a CMA dma-heap for DMA memory management, the application is also responsible for allocating a memory pool from a dma-heap under `/dev/dma_heap/` (via `DMA_HEAP_IOCTL_ALLOC`) and passing the resulting dma-buf file descriptor to the driver.

Data buffers passed to the packet ioctls must reside in the pool dma-buf configured via the GRSPFI_IOCTL_CONFIG ioctl's *pool_dmabuf_fd* field, and are given as byte offsets into that dma-buf rather than user-space pointers. The driver resolves each offset to a physical/bus address for DMA. The driver holds its own reference to the dma-buf from GRSPFI_IOCTL_CONFIG until the next GRSPFI_IOCTL_CONFIG or until the control device is closed, so closing the pool_dmabuf_fd in user-space does not affect an in-progress DMA operation.

On systems without cache-coherent DMA (NOEL-V), RX buffers should start and end on a cache line boundary: data in the same cache line as an RX buffer may be lost while the driver owns the buffer.

5.1.3. Examples

Within the GRLIB driver package there is a user-space example of how this driver can be used, `user/grspfi_user/grspfi_user.c`. The example uses the user-space API to call the driver's ioctl interface, and allocates its pool from a CMA dma-heap with the helper in `user/common/dmaheap.c`.

5.2. ioctl Reference

5.2.1. Overview

The control and packet transfer interfaces are accessed using the standard UNIX `ioctl` routine. All ioctls are defined in `<linux/grlib/grspfi.h>`. Unless otherwise stated, each ioctl returns 0 on success or a negative `errno` value on error. Every ioctl takes a pointer argument. The driver serves 32-bit user space on a 64-bit kernel with the same structs.

All GRLIB drivers use the ioctl magic `'J'` (GRLIB_IOCTL_MAGIC, also defined as GRSPFI_IOCTL_MAGIC), with a range of command numbers each: 0x00-0x3f grspw_user, 0x40-0x5f grspw_router, 0x80-0xbf grspfi (this driver); 0x60-0x7f and 0xc0-0xff are for other GRLIB drivers. A command sent to another driver's device fails with `-ENOTTY`.

The following rules are part of the interface:

- All structs have a fixed layout, the same for every ABI: every field has a fixed-size type and every hole is a reserved field.
- A struct never changes size. A larger version gets a new ioctl number.
- User space sets input reserved fields to zero and ignores output reserved fields. A nonzero input reserved field, an unknown flag, and a boolean field (*enable*, *fifo_en*, ...) other than 0 or 1 give `-EINVAL`. Fields named *flags* with no flags defined must be 0.
- GRSPFI_IOCTL_GET_CAPABILITIES reports the interface version in *api_version*, which is GRSPFI_API_VERSION (2) for the interface described here, and optional driver features in *features*, none defined yet (0).

Some ioctls are valid on the control device (`/dev/grspfi`*N*), others on a VC device (`/dev/grspfi`*N*`_`*V*). An ioctl sent to the wrong kind of device fails with `-EINVAL`.

Table 5.2. GRSPFI `ioctl` commands

Command	Number	Argument	Device
GRSPFI_IOCTL_GET_CAPABILITIES	0x81	struct grspfi_hwcap	Control or VC
GRSPFI_IOCTL_CONFIG	0x82	struct grspfi_config	Control
GRSPFI_IOCTL_VC_CONFIG	0x83	struct grspfi_vc_config	VC
GRSPFI_IOCTL_BC_CONFIG	0x84	struct grspfi_bc_config	Control
GRSPFI_IOCTL_TIME_CONFIG	0x85	struct grspfi_time_config	Control or VC
GRSPFI_IOCTL_TIME_SET	0x86	__u32	Control or VC
GRSPFI_IOCTL_TIME_GET	0x87	__u32	Control or VC
GRSPFI_IOCTL_DMA_CONFIG	0x88	struct grspfi_dma_config	Control
GRSPFI_IOCTL_RX_PREPARE	0x89	struct grspfi_pkts	VC (open)
GRSPFI_IOCTL_RX_RECEIVE	0x8a	struct grspfi_pkts	VC (open)

Command	Number	Argument	Device
GRSPFI_IOCTL_TX_SEND	0x8b	struct grspfi_pkts	VC (open)
GRSPFI_IOCTL_TX_RECLAIM	0x8c	struct grspfi_pkts	VC (open)
GRSPFI_IOCTL_BC_SEND	0x8d	struct grspfi_bc_msg	Control (BC open)
GRSPFI_IOCTL_BC_RECEIVE	0x8e	struct grspfi_bc_msg	Control (BC open)
GRSPFI_IOCTL_TX_WAIT	0x8f	struct grspfi_wait	VC (open)
GRSPFI_IOCTL_RX_WAIT	0x90	struct grspfi_wait	VC (open)
GRSPFI_IOCTL_BCRX_WAIT	0x91	struct grspfi_wait	Control (BC open)

The following general error codes may be returned by any ioctl:

`-EFAULT`

: The user-space buffer pointer is invalid.

`-EINVAL`

: A nonzero reserved field, an unknown flag or a boolean field other than 0 or 1. The ioctl was sent to the wrong kind of device, a VC ioctl to a VC that is not open (enabled with `GRSPFI_IOCTL_VC_CONFIG`), or a broadcast ioctl while the Broadcast Channel is not open.

`-ENODEV`

: On a VC device: the control device has been closed. Otherwise no private data is associated with the file descriptor (internal error).

`-ENOTTY`

: Unrecognised ioctl command number, or a command of another driver.

A typical usage sequence is:

1. Open `/dev/grspfi`*N*.
2. Call `GRSPFI_IOCTL_GET_CAPABILITIES` to discover hardware features.
3. Call `GRSPFI_IOCTL_CONFIG` to select the pool dma-buf and start the link.
4. Open `/dev/grspfi`*N*`_`*V* for each VC to use.
5. Call `GRSPFI_IOCTL_VC_CONFIG` on each VC device.
6. Optionally call `GRSPFI_IOCTL_BC_CONFIG` on the control device.
7. Call `GRSPFI_IOCTL_DMA_CONFIG` on the control device to open a DMA channel.
8. Transfer data using the RX/TX ioctls described below.
9. Tear down in reverse order.

5.2.2. GRSPFI_IOCTL_GET_CAPABILITIES

Direction	Read (kernel → user)
Argument type	struct grspfi_hwcap
Valid on	Control device or VC device

Copies the interface version and the hardware capabilities into the user-space buffer pointed to by the ioctl argument. The caller should issue this ioctl immediately after opening the control device to learn the number of Virtual Channels, DMA channels, descriptors per VC, and optional hardware features (RMAP, CCSDS CRC, etc.).

```
#define GRSPFI_API_VERSION 2

struct grspfi_hwcap_gen {
 __u8 num_ebc;  /* External broadcast channels */
 __u8 num_evc;  /* External virtual channels */
 __u8 width_bwdcc; /* Width of the bandwidth credit counter */
 __u8 width_rfctc; /* Width of the remote FCT counter */
 __u8 fct_mult;  /* FCT multiplier */
 __u8 num_vc;  /* Virtual channels, 1 to 32 */
 __u8 prbs_init;  /* INIT1 embedded in random numbers */
 __u8 bitmode_slct; /* 16/20 bit mode */
 __u8 int_encdec; /* Internal 8b10b encoder/decoder */
```

```
 __u8 tx_clock;  /* Separate TX clock */
 __u8 reserved[2];
};

struct grspfi_hwcap_buf {
 __u8 depth_tx;  /* Depth of the VC TX buffers */
 __u8 depth_rx;  /* Depth of the VC RX buffers */
 __u8 depth_bc_retr; /* Depth of the broadcast retry buffer */
 __u8 depth_fct_retr; /* Depth of the FCT retry buffer */
 __u8 depth_data_retr; /* Depth of the data retry buffers */
 __u8 reserved[3];
};

struct grspfi_hwcap_dma {
 __u8 spfi;  /* SpaceFibre protocol */
 __u8 ccsds_crc;  /* CCSDS CRC logic */
 __u8 rmap_crc;  /* RMAP CRC logic */
 __u8 rmap_tar;  /* RMAP target */
 __u8 num_dma;  /* DMA channels, 1 to 8 */
 __u8 reserved0;
 __u16 num_tx_desc; /* TX descriptors per VC */
 __u16 num_rx_desc; /* RX descriptors per VC */
 __u16 reserved1;
};

struct grspfi_hwcap {
 __u32 api_version; /* GRSPFI_API_VERSION of the driver */
 __u32 features;  /* Optional driver features, none yet */
 struct grspfi_hwcap_gen gensup;
 struct grspfi_hwcap_buf bufsup;
 struct grspfi_hwcap_dma dmasup;
 __u32 hw_version;
 __u32 reserved[4];
};
```

Relevant fields in struct grspfi_hwcap:

Table 5.3. Fields in struct grspfi_hwcap

Field	Description
api_version	GRSPFI_API_VERSION of the driver.
features	Optional driver features, none defined yet (0).
gensup.num_vc	Number of Virtual Channels present in hardware.
dmasup.num_dma	Number of DMA channels.
dmasup.num_tx_desc *dmasup.num_rx_desc*	Number of TX and RX descriptors per VC: the most packets a VC holds in each direction, and the largest *count* of GRSPFI_IOCTL_TX_WAIT and GRSPFI_IOCTL_RX_WAIT.
dmasup.rmap_tar	Non-zero if hardware RMAP target is supported.

Errors:

-EFAULT

The user-space buffer pointer is invalid.

5.2.3. GRSPFI_IOCTL_CONFIG

Direction	Write (user → kernel)
Argument type	struct grspfi_config
Valid on	Control device only

Configures the SpaceFibre link, selects the pool dma-buf, and (re)starts the link. Must be called on the control device before any data transfer can occur, and before any VC is enabled with GRSPFI_IOCTL_VC_CONFIG.

```
struct grspfi_config {
 __u8 address;
 __u8 address_mask;
 __u8 reserved0[2];
 __s32 pool_dmabuf_fd;
 __u32 reserved[2];
};
```

Fields in struct grspfi_config:

Table 5.4. Fields in struct grspfi_config

Field	Description
address	8-bit default node address.
address_mask	8-bit address mask.
pool_dmabuf_fd	File descriptor of the pool dma-buf to attach for DMA address translation, allocated from a CMA dma-heap under /dev/dma_heap/ via DMA_HEAP_IOCTL_ALLOC. The driver keeps its own reference, so the file descriptor may be closed after this call.

Errors:

-EFAULT

The user-space buffer pointer is invalid.

-EINVAL

Called on a VC device, or a reserved field is not zero.

-EBUSY

A DMA channel is enabled or a VC is enabled (GRSPFI_IOCTL_VC_CONFIG with *enable* set). The pool can only be replaced while all VCs are disabled and DMA is off.

-EBADF, -EINVAL

pool_dmabuf_fd is not a dma-buf. Other errors come from attaching and mapping it. The previous pool is kept.

5.2.4. GRSPFI_IOCTL_VC_CONFIG

Direction	Write (user → kernel)
Argument type	struct grspfi_vc_config
Valid on	VC device only

Enables or disables a Virtual Channel and sets its operating parameters. Must be issued on a VC device node (/dev/grspfi*N_V*).

```
struct grspfi_vc_config {
 __u64 timeslots; /* Bit per time-slot the VC may send in */
 __u32 max_rx_len; /* Largest packet received, not 0 */
 __u8 enable;  /* 1: open the VC, 0: close it */
 __u8 prio;  /* Priority, 0 to 15 */
 __u8 bandwidth;  /* Percent, up to 95 */
 __u8 fifo_en;
 __u8 fifo_nr;
 __u8 rmap_en;
 __u8 rmap_key;
 __u8 address;  /* Address filter, if address_filter_en */
 __u8 address_mask;
 __u8 address_filter_en;
 __u8 reserved[10];
};
```

Fields in struct grspfi_vc_config:

Table 5.5. Fields in struct grspfi_vc_config

Field	Description
timeslots	64-bit bitmask of transmit time-slots.
max_rx_len	Maximum incoming packet length in bytes, not 0. Every RX buffer of the VC must hold it.
enable	1 to open the VC with the parameters below; 0 to close it. Closing the VC drops the packets still in it.
prio	Transmit priority (0–15, 0 = highest).

Field	Description
bandwidth	Bandwidth allocation in percent (1–95).
fifo_en	1 to enable FIFO routing for received packets, otherwise 0.
fifo_nr	External FIFO index (used when *fifo_en* is set).
rmap_en	1 to enable hardware RMAP target processing, otherwise 0. Ignored if the core has no RMAP target (*dmasup.rmap_tar* is zero).
rmap_key	RMAP destination key.
address	VC address for filtering (8-bit).
address_mask	VC address mask (8-bit).
address_filter_en	1 to enable VC-level address acceptance filtering, otherwise 0.

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

Called on the control device, the VC index embedded in the device node is out of range, disabling a VC that is not enabled, enabling a VC that is already enabled, a reserved field is not zero, *enable*, *fifo_en*, *rmap_en* or *address_filter_en* is not 0 or 1, or invalid parameters when enabling (*max_rx_len* 0, *prio* > 15, *bandwidth* 0 or > 95, *max_rx_len* too large, or *fifo_nr* out of range when *fifo_en* is set). The VC configuration is unchanged on error.

`-EBUSY`

Disabling a VC that a DMA channel still serves. Disable the DMA channel with `GRSPFI_IOCTL_DMA_CONFIG` first.

`-ENOMEM`

The descriptor rings could not be allocated.

5.2.5. GRSPFI_IOCTL_BC_CONFIG

Direction	Write (user → kernel)
Argument type	struct grspfi_bc_config
Valid on	Control device only

Opens or closes the Broadcast Channel.

```
struct grspfi_bc_config {
 __u8 channel;
 __u8 enable;  /* 1: open, 0: close */
 __u8 bandwidth;
 __u8 fifo_en;
 __u32 reserved[3];
};
```

Fields in struct grspfi_bc_config:

Table 5.6. Fields in struct grspfi_bc_config

Field	Description
channel	Broadcast channel number.
enable	1 to open; 0 to close.
bandwidth	Bandwidth allocation in percent (1–95).
fifo_en	1 to enable FIFO broadcast delivery, otherwise 0.

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

Called on a VC device, a reserved field is not zero, *`enable`* or *`fifo_en`* is not 0 or 1, invalid parameters (*`bandwidth`* 0 or > 95, or *`fifo_en`* without external broadcast support), or disabling a channel that is not open.

`-EBUSY`

The Broadcast Channel is already open.

`-ENOMEM`

The broadcast buffers could not be allocated.

5.2.6. GRSPFI_IOCTL_TIME_CONFIG

Direction	Write (user → kernel)
Argument type	struct grspfi_time_config
Valid on	Control device or VC device

Configures time-slot generation.

```
struct grspfi_time_config {
 __u32 int_gen;  /* 1: the core increments the time-slot */
 __u32 nom_len;  /* Time-slot length in clock cycles */
 __u32 reserved[2];
};
```

Fields in struct grspfi_time_config:

Table 5.7. Fields in struct grspfi_time_config

Field	Description
`int_gen`	1 to enable internal time-slot generation; 0 to accept external time-slot input.
`nom_len`	Nominal time-slot length in clock cycles (non-zero).

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

`int_gen` is not 0 or 1, a reserved field is not zero, or *`nom_len`* is zero or out of hardware range.

5.2.7. GRSPFI_IOCTL_TIME_SET

Direction	Write (user → kernel)
Argument type	__u32
Valid on	Control device or VC device

Sets the current time-slot, 0 to 63, in the TS hardware register (bits 5:0). The argument is a pointer to a __u32 holding the value.

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

The value is above 63.

5.2.8. GRSPFI_IOCTL_TIME_GET

Direction	Read (kernel → user)
Argument type	__u32
Valid on	Control device or VC device

Reads the current time-slot, 0 to 63, from the TS hardware register into the __u32 the argument points to.

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

5.2.9. GRSPFI_IOCTL_DMA_CONFIG

Direction	Write (user → kernel)
Argument type	struct grspfi_dma_config
Valid on	Control device only

Opens or closes a DMA channel. When a channel is opened, interrupt handlers for DMA errors, VC TX/RX events, and BC events are registered automatically. They are removed when the last enabled channel is closed.

```
/* txrx of struct grspfi_dma_config */
#define GRSPFI_DMACFG_RX 0x1 /* txrx: receive */
#define GRSPFI_DMACFG_TX 0x2 /* txrx: transmit */

struct grspfi_dma_config {
 __u8 channel;
 __u8 enable;  /* 1: enable, 0: disable */
 __u8 bc_enable;  /* Serve the broadcast channel */
 __u8 txrx;  /* GRSPFI_DMACFG_RX | GRSPFI_DMACFG_TX */
 __u32 vc_mask;  /* VCs served, bit per VC */
 __u32 reserved[2];
};
```

Fields in struct grspfi_dma_config:

Table 5.8. Fields in struct grspfi_dma_config

Field	Description
channel	DMA channel index (0 to *dmasup.num_dma* _ 1).
enable	1 to open the DMA channel; 0 to close it.
bc_enable	1 to also route Broadcast Channel traffic through this DMA channel, otherwise 0. The BC must be open, and only one DMA channel at a time can carry it.
txrx	Direction bitmask: combine `GRSPFI_DMACFG_TX` (TX enable) and/or `GRSPFI_DMACFG_RX` (RX enable).
vc_mask	Bitmask of Virtual Channels to bind to this DMA channel. Each referenced VC must have been opened with `GRSPFI_IOCTL_VC_CONFIG` first.

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

Called on a VC device, a reserved field is not zero, *enable* or *bc_enable* is not 0 or 1, *txrx* has unknown bits, invalid channel index, disabling a channel that is not enabled, a referenced VC is not enabled or is already bound to another DMA channel, or *bc_enable* is set while the BC is not open or already mapped to another channel.

`-EBUSY`

Enabling a channel that is already enabled.

5.2.10. Packet transfer

The four packet ioctls, GRSPFI_IOCTL_RX_PREPARE, GRSPFI_IOCTL_RX_RECEIVE, GRSPFI_IOCTL_TX_SEND and GRSPFI_IOCTL_TX_RECLAIM, are valid on a VC device whose VC is open, and move a batch of packets per call. Each takes a struct grspfi_pkts, which describes an array of struct grspfi_upkt entries in user space.

```
struct grspfi_upkt {
 __u64 user_data; /* Returned unchanged */
 __u64 data_off;
 __u64 hdr_off;  /* TX only */
 __u32 data_len;
 __u32 flags;  /* GRSPFI_TXPKT_* or GRSPFI_RXPKT_* */
 __u8 hdr_len;  /* TX only */
 __u8 reserved[7];
};

struct grspfi_pkts {
 __u64 pkts;  /* User space address of the array */
 __u32 count;  /* In: entries in the array, out: entries used */
 __u32 flags;  /* None defined, must be 0 */
};

/* TX flags, input */
#define GRSPFI_TXPKT_IE   (0x1u << 10) /* Interrupt when sent */
#define GRSPFI_TXPKT_NONCRCLEN(n) (((n) & 0xfu) << 12) /* Header bytes
         * left out of
         * the CRC
         */
#define GRSPFI_TXPKT_HC   (0x1u << 16) /* Append a header CRC */
#define GRSPFI_TXPKT_DC   (0x1u << 17) /* Append a data CRC */
#define GRSPFI_TXPKT_CRCT_RMAPCRC (0x0u << 18) /* CRC type */
#define GRSPFI_TXPKT_CRCT_CRC16  (0x1u << 18)
#define GRSPFI_TXPKT_CRCT_ISOCHECKSUM (0x2u << 18)
#define GRSPFI_TXPKT_INPUT_MASK  (GRSPFI_TXPKT_IE | GRSPFI_TXPKT_NONCRCLEN(0xf) | \
      GRSPFI_TXPKT_HC | GRSPFI_TXPKT_DC | (0x3u << 18))

/* RX flags: IE is input, the others output */
#define GRSPFI_RXPKT_IE  (0x1u << 27) /* Interrupt when received */
#define GRSPFI_RXPKT_EP  (0x1u << 28) /* Ended with an error */
#define GRSPFI_RXPKT_TR  (0x1u << 29) /* Truncated */
#define GRSPFI_RXPKT_HC  (0x1u << 30) /* Header CRC error */
#define GRSPFI_RXPKT_DC  (0x1u << 31) /* Data CRC error */
#define GRSPFI_RXPKT_INPUT_MASK GRSPFI_RXPKT_IE
```

pkts is the address of the array as a 64-bit integer, for example (uintptr_t)array, also for 32-bit user space. *count* is the number of entries in the array on input, and the number of entries used on output. A *count* of 0 or a nonzero *flags* gives -EINVAL. Offsets are byte offsets into the pool dma-buf given by GRSPFI_IOCTL_CONFIG; each offset and length must resolve to one physically contiguous range of the pool. *user_data* is any 64-bit value, for example a pointer, and is returned unchanged by GRSPFI_IOCTL_RX_RECEIVE and GRSPFI_IOCTL_TX_RECLAIM.

GRSPFI_IOCTL_RX_PREPARE and GRSPFI_IOCTL_TX_SEND queue as many entries as the VC has free descriptors for, from the start of the array, and return that number in *count*; fewer free descriptors than entries is not an error, and *count* is 0 when the ring is full. They check each of those entries before they queue any. An invalid entry (unknown flag, CRC type 3, nonzero reserved field, range outside the pool) fails the call with -EINVAL, and an array that cannot be read with -EFAULT; nothing is then queued. Entries beyond the free descriptors are not read. Each entry is checked again as it is queued, as user space may change the array meanwhile; if it then fails, the entries queued before it stay queued and *count* says how many.

GRSPFI_IOCTL_RX_RECEIVE and GRSPFI_IOCTL_TX_RECLAIM return up to *count* packets in the array, in the order they were queued, and *count* 0 when none is ready; this is not an error. If the array cannot be written the call fails with -EFAULT, and the packets stay in the driver: the next call returns them first. Closing the VC with GRSPFI_IOCTL_VC_CONFIG drops the packets still in it.

The use of the entry fields per ioctl:

Table 5.9. Fields of struct grspfi_upkt per ioctl

ioctl	Fields
GRSPFI_IOCTL_RX_PREPARE	Submits empty receive buffers. *data_off* is the buffer; the *max_rx_len* bytes from it (from GRSPFI_IOCTL_VC_CONFIG) must be within the pool. *data_len*

ioctl	Fields
	is the buffer size, 0 for *max_rx_len*, otherwise at least *max_rx_len*. *flags* is 0 or GRSPFI_RXPKT_IE (interrupt when the packet has been received; needed for GRSPFI_IOCTL_RX_WAIT to be woken). *hdr_off* and *hdr_len* must be 0.
GRSPFI_IOCTL_RX_RECEIVE	Retrieve completed receive buffers, as prepared, with *data_len* the number of bytes received and *flags* the status flags of the RX descriptor: GRSPFI_RXPKT_DC / GRSPFI_RXPKT_HC (data/header RMAP CRC error), GRSPFI_RXPKT_TR (truncated, longer than *max_rx_len*), GRSPFI_RXPKT_EP (ended with an EEP), and GRSPFI_RXPKT_IE as given to GRSPFI_IOCTL_RX_PREPARE. The core always checks the RMAP CRCs, so DC and HC are also set for packets that are not RMAP packets; for those only TR and EP indicate a transfer error.
GRSPFI_IOCTL_TX_SEND	Submits packets for transmission: *data_len* bytes at *data_off*, after a header of *hdr_len* bytes at *hdr_off* if *hdr_len* is not 0. *flags* holds the TX descriptor control flags: CRC type (GRSPFI_TXPKT_CRCT_*; the value 3 is invalid), header and data CRC (GRSPFI_TXPKT_HC, GRSPFI_TXPKT_DC), header bytes left out of the CRC (GRSPFI_TXPKT_NONCRCLEN(n)), and interrupt when sent (GRSPFI_TXPKT_IE, needed for GRSPFI_IOCTL_TX_WAIT to be woken).
GRSPFI_IOCTL_TX_RECLAIM	Reclaim transmitted packets, as sent, allowing the caller to free or reuse the buffers.

Errors:

-EINVAL

Called on the control device or the VC is not enabled, *count* is 0 or *flags* is not zero. RX_PREPARE and TX_SEND: an entry has an unknown flag, CRC type 3, a nonzero reserved field, a range that is not within the pool dma-buf or not physically contiguous, or no pool dma-buf has been configured. RX_PREPARE also: a header, or a *data_len* other than 0 that is smaller than *max_rx_len*.

-EFAULT

The struct grspfi_pkts or the array cannot be read or written.

5.2.11. GRSPFI_IOCTL_BC_SEND

Direction	Write (user → kernel)
Argument type	struct grspfi_bc_msg
Valid on	Control device (BC must be open)

Queues a Broadcast Channel message for transmission.

```
/* flags of struct grspfi_bc_msg, output */
#define GRSPFI_BC_LT  0x1 /* Late */
#define GRSPFI_BC_DL  0x2 /* Delayed */

struct grspfi_bc_msg {
 __u64 data;
 __u8 flags;  /* GRSPFI_BC_* */
 __u8 channel;  /* Sender's broadcast channel */
 __u8 type;  /* Broadcast type */
 __u8 reserved[5];
};
```

Fields in struct grspfi_bc_msg:

Table 5.10. Fields in struct grspfi_bc_msg

Field	Description
data	64-bit broadcast data payload.
flags	Output only: GRSPFI_BC_DL (delayed) and GRSPFI_BC_LT (late) on receive. Must be 0 on send.
channel	Ignored on send, the configured broadcast channel number is used. Filled in with the sender's channel number on receive.

Field	Description
type	Broadcast message type (8-bit).

Errors:

-EINVAL

Called on a VC device, the broadcast channel is not open, *flags* is not 0, or a reserved field is not zero.

-EFAULT

Copy from user failed.

-EAGAIN

The BC TX buffer is full.

5.2.12. GRSPFI_IOCTL_BC_RECEIVE

Direction	Read (kernel → user)
Argument type	struct grspfi_bc_msg
Valid on	Control device (BC must be open)

Dequeues the oldest received Broadcast Channel message. All fields of struct grspfi_bc_msg are populated from the hardware receive buffer; the reserved fields are zero.

Errors:

-EINVAL

Called on a VC device, or the broadcast channel is not open.

-EFAULT

Copy to user failed.

-EAGAIN

No received BC message is available.

5.2.13. GRSPFI_IOCTL_TX_WAIT, GRSPFI_IOCTL_RX_WAIT and GRSPFI_IOCTL_BCRX_WAIT

Direction	Write (user → kernel)
Argument type	struct grspfi_wait
Valid on	TX_WAIT, RX_WAIT: VC device (VC open). BCRX_WAIT: control device (BC open)

```
struct grspfi_wait {
 __u32 timeout_ms; /* 0: no timeout */
 __u32 count;  /* RX_WAIT, TX_WAIT: packets to wait for, 0 for
     * one; at most the VC's descriptors.
     * BCRX_WAIT: must be 0.
     */
 __u32 flags;  /* None defined, must be 0 */
 __u32 reserved;
};
```

GRSPFI_IOCTL_TX_WAIT blocks the calling thread until at least *count* TX packets (1 if *count* is 0) can be reclaimed with GRSPFI_IOCTL_TX_RECLAIM. GRSPFI_IOCTL_RX_WAIT blocks until at least *count* packets (1 if 0) have been received and can be read with GRSPFI_IOCTL_RX_RECEIVE. Packets the driver kept after an -EFAULT count as ready. The driver only checks these waits on the interrupts of packets sent with GRSPFI_TXPKT_IE or buffers prepared with GRSPFI_RXPKT_IE; without them only the timeout ends the wait. If another thread takes the packets meanwhile, the wait goes on for the rest of its timeout. *count* may be at most *dmasup.num_tx_desc* (TX_WAIT) or *dmasup.num_rx_desc* (RX_WAIT).

GRSPFI_IOCTL_BCRX_WAIT blocks until at least one Broadcast Channel message has been received and is available via GRSPFI_IOCTL_BC_RECEIVE. It is woken by the device-level BC-RX interrupt handler, so only while a DMA channel is enabled. *count* must be 0.

timeout_ms is the timeout in milliseconds, 0 for none. The waits take no lock other ioctls need, and any number of threads may wait.

Errors:

-EINVAL

TX_WAIT, RX_WAIT: called on the control device or the VC is not enabled. BCRX_WAIT: called on a VC device or the broadcast channel is not open. Also *flags* or *reserved* is not zero, or *count* is above the number of descriptors (TX_WAIT, RX_WAIT) or not 0 (BCRX_WAIT).

-ETIMEDOUT

The timeout expired first.

-EINTR

A signal arrived, with or without a timeout.

-EIO

TX_WAIT, RX_WAIT: the VC is disabled, no DMA channel serves it, or that DMA channel stopped on an AHB error, while fewer than *count* packets are ready. Enabling a DMA channel for the VC again clears the error. BCRX_WAIT: the broadcast channel was closed during the wait.

-EFAULT

The user-space buffer pointer is invalid.

6. GRSPFI Kernel Library Driver

6.1. Introduction

This section describes the GRSPFI Kernel Library driver for Linux. Its interface is only accessible from kernel space. Most of the functionality is exported to user space as described in Chapter 5.

The GRSPFI IP core implements the SpaceFibre high-speed serial link protocol. The kernel library probes the core via the Linux platform-device framework and exports a set of symbols that allow other kernel modules to open and control GRSPFI device instances.

6.1.1. Driver sources

The driver sources and header files are listed in Section 5.1.1.

6.1.2. Examples

The GRSPFI SpaceFibre character device driver is an example of how the GRSPFI Kernel Library driver can be used. See Section 5.1.3.

6.2. Software design overview

6.2.1. Overview

The GRSPFI kernel library has been implemented using the platform device driver model. The driver provides a kernel function interface, an API, rather than implementing an IO system device. The API is intended for kernel tasks but has been designed so that a custom interface for user-space processes can be implemented on top of the kernel space API, see Chapter 5. The driver can be compiled as a kernel module and loaded into the kernel at run-time or linked with the kernel at compile-time. The installation steps required for linking with kernel are described in the `kernel/drivers/grlib/README`.

The driver API has been split up in the following major parts:

- Module registration, see Section 6.2.3.
- Device lifecycle (open, close, capabilities), see Section 6.2.4.
- Link layer control, see Section 6.2.5.
- Time-slot management, see Section 6.2.6.
- Virtual Channel API, see Section 6.2.7.
- Broadcast Channel API, see Section 6.2.8.
- DMA Channel API, see Section 6.2.9.
- Status registers, see Section 6.2.10.
- Interrupt management, see Section 6.2.11.

A typical usage sequence is:

1. Call `grspfi_init_user()` to register device-found and device-removed callbacks.
2. In the device-found callback, call `grspfi_open()`.
3. Query capabilities with `grspfi_cap_get()`.
4. Open Virtual Channels with `grspfi_vc_open()`.
5. Optionally open the Broadcast Channel with `grspfi_bc_open()`.
6. Open one or more DMA channels with `grspfi_dma_open()`.
7. Register interrupt handlers and enable interrupts.
8. Start the link with `grspfi_link_start()`.
9. Transfer data using the VC and BC send/receive/prepare/reclaim API.
10. Tear down in reverse order.

6.2.2. Key Data Structures

The API is built around the following opaque handle and packet types, all declared in `include/linux/grlib/grspfi_drv.h`:

Table 6.1. GRSPFI kernel library data structures

Structure	Description
grspfi_dev	Opaque handle to an opened GRSPFI device instance. Returned by `grspfi_open()` and passed to most API functions.
grspfi_vch	Virtual Channel handle, initialised by `grspfi_vc_open()`. Carries descriptor ring state and per-VC statistics.
grspfi_bch	Broadcast Channel handle, initialised by `grspfi_bc_open()`.
grspfi_dmah	DMA Channel handle, initialised by `grspfi_dma_open()`.
grspfi_pkt	Describes a single TX or RX packet. Fields include *`data`* (DMA address), *`data_length`*, *`header`*, *`header_length`*, and *`flags`*.
grspfi_bc_pkt	Describes a single Broadcast Channel packet with fields *`channel`*, *`type`*, *`flags`*, and a 64-bit *`data`* payload.
grspfi_cap	Hardware capability structure returned by `grspfi_cap_get()`. Contains sub-structures *`gensup`* (general capabilities), *`bufsup`* (buffer depths), and *`dmasup`* (DMA capabilities including descriptor counts).
grspfi_dev_status	Snapshot of the Lane Layer, Retry Layer, and DMA Layer status registers.
grspfi_vc_status	Snapshot of a Virtual Channel Status register.
grspfi_dma_status	Snapshot of the DMA Channel Status and extended DMA Channel Status registers.
grspfi_dma_interrupt_info	Passed to the DMA ISR callback. Contains a *`dma_status`* snapshot and a *`vc_status`* snapshot for the offending VC.

6.2.3. Module Registration

6.2.3.1. grspfi_init_user

```
void grspfi_init_user(void *(*devfound)(int, struct device *),
                      void (*devremove)(int, void *));
```

Registers callbacks that the `grspfi_drv` module invokes when a GRSPFI hardware instance is discovered or removed by the platform driver.

Table 6.2. Parameters of `grspfi_init_user()`

Parameter	Description
`devfound`	Called when a new device is probed. The first argument is the device index (0-based), the second is the underlying struct device. The callback must return a pointer to caller-owned private data (or `NULL`); this pointer is passed back to *`devremove`*.
`devremove`	Called when the platform device is removed. Receives the device index and the pointer previously returned by *`devfound`*.

Pass `NULL` for both arguments to deregister the callbacks. If `grspfi_init_user()` is called before the platform driver has probed the hardware, the *`devfound`* callback will be invoked from the probe function when the hardware is later discovered.

6.2.4. Device Lifecycle

6.2.4.1. grspfi_open

```
struct grspfi_dev * grspfi_open(u32 id);
```

Opens the GRSPFI device with the given index and returns an opaque device handle. The hardware is enabled and all DMA channels, Virtual Channels, and status registers are reset to a clean state.

Table 6.3. Parameters of `grspfi_open()`

Parameter	Description
`id`	Zero-based index of the GRSPFI device instance.

Returns a pointer to grspfi_dev on success, or `NULL` if the device is already open or the index is out of range.

6.2.4.2. grspfi_close

```
void grspfi_close(struct grspfi_dev *dev);
```

Closes a previously opened device, disabling SpaceFibre output and resetting the link. All Virtual Channels, DMA channels and the Broadcast Channel should be closed before calling this function, since `grspfi_open()` forgets their state.

Table 6.4. Parameters of `grspfi_close()`

Parameter	Description
dev	Device handle returned by `grspfi_open()`.

6.2.4.3. grspfi_cap_get

```
void grspfi_cap_get(struct grspfi_dev *dev,
                    struct grspfi_cap *capabilities);
```

Reads hardware capability registers and populates the grspfi_cap structure. Capabilities include:

- *gensup.num_vc* — number of Virtual Channels.
- *gensup.num_evc* — number of external VCs.
- *gensup.num_ebc* — number of external BCs.
- *dmasup.num_dma* — number of DMA channels.
- *dmasup.num_tx_desc* — number of TX descriptors.
- *dmasup.num_rx_desc* — number of RX descriptors.
- *dmasup.rmap_tar* — RMAP target support.
- *dmasup.ccsds_crc* — CCSDS CRC support.

Table 6.5. Parameters of `grspfi_cap_get()`

Parameter	Description
dev	Device handle.
capabilities	Caller-allocated structure to be filled in.

6.2.5. Link Layer Control

6.2.5.1. grspfi_link_start

```
void grspfi_link_start(struct grspfi_dev *dev, bool start, u8 address,
                       u8 address_mask);
```

Programs the default (link-layer) address and mask, then starts or gracefully stops the SpaceFibre link.

Table 6.6. Parameters of `grspfi_link_start()`

Parameter	Description
dev	Device handle.
start	`true` to start the link (LS bit in CCTRL), `false` to abort (AS bit in CCTRL).
address	8-bit default node address written to DEFADDR.
address_mask	8-bit address mask written to DEFADDR.

6.2.5.2. grspfi_link_close

```
void grspfi_link_close(struct grspfi_dev *dev, bool reset);
```

Stops the SpaceFibre link.

Table 6.7. Parameters of `grspfi_link_close()`

Parameter	Description
dev	Device handle.

Parameter	Description
reset	`true` to perform a link reset (LIR), `false` for a normal close (LNR).

6.2.5.3. grspfi_link_status

```
u8 grspfi_link_status(struct grspfi_dev *dev);
```

Returns the 4-bit Lane Layer Status (LSTS field) from the LLSTAT register, indicating the current link state machine state.

6.2.5.4. grspfi_address_get

```
u16 grspfi_address_get(struct grspfi_dev *dev);
```

Returns the current value of the DEFADDR register (address and mask packed into a 16-bit value).

6.2.6. Time-slot Management

6.2.6.1. grspfi_time_conf

```
int grspfi_time_conf(struct grspfi_dev *dev, bool internal_generation,
                     u32 nom_len);
```

Configures the time-slot mechanism.

Table 6.8. Parameters of `grspfi_time_conf()`

Parameter	Description
dev	Device handle.
internal_generation	`true` to enable internal time-slot generation (EG bit in TSCTRL), `false` to accept external time-slot signals.
nom_len	Nominal time-slot length in link characters (written as *nom_len* _ 1 to the TSLEN register). Must be non-zero.

Returns 0 on success, `-EINVAL` if *nom_len* is zero or exceeds the hardware field width.

6.2.6.2. grspfi_time_set

```
int grspfi_time_set(struct grspfi_dev *dev, u8 time);
```

Writes a 6-bit time value to the TS register. Returns 0 on success or `-EINVAL` if *time* exceeds 6 bits.

6.2.6.3. grspfi_time_get

```
u32 grspfi_time_get(struct grspfi_dev *dev);
```

Reads and returns the current 6-bit time value from the TS register.

6.2.7. Virtual Channel API

6.2.7.1. grspfi_vc_open

```
int grspfi_vc_open(struct grspfi_dev *dev, u8 virtual_channel_number,
                   u32 max_receive_length, u8 priority, u8 bandwidth,
                   u64 timeslots, bool fifo_enable, u8 fifo_number,
                   bool rmap_enable, u8 rmap_key, struct grspfi_vch *vch);
```

Configures and opens a Virtual Channel. Allocates DMA-coherent TX and RX descriptor rings sized to the hardware maximum.

Table 6.9. Parameters of `grspfi_vc_open()`

Parameter	Description
dev	Device handle.
virtual_channel_number	VC index (0 to *gensup.num_vc* _ 1).

Parameter	Description
max_receive_length	Maximum incoming packet length in bytes, written to VCMAXLEN.
priority	Transmit priority (0 - 15, 0 = highest), written to VCCTRL.PR.
bandwidth	Bandwidth allocation as a percentage of link capacity (1–95), converted to the hardware representation and written to VCCTRL.VBW.
timeslots	64-bit bitmask of time-slots during which this VC may transmit. Written to VCTS1 (bits 0–31) and VCTS2 (bits 32–63).
fifo_enable	Enable FIFO routing for received packets.
fifo_number	External FIFO index to use when *fifo_enable* is true.
rmap_enable	Enable hardware RMAP target processing (requires hardware support).
rmap_key	RMAP destination key written to VCKEY.
vch	Caller-allocated grspfi_vch handle to initialise. Must remain valid until `grspfi_vc_close()`.

Returns 0 on success, `-EINVAL` for invalid parameters or a VC that is already open, or a negative error code from descriptor allocation failure.

6.2.7.2. grspfi_vc_close

```
void grspfi_vc_close(struct grspfi_vch *vch);
```

Closes a Virtual Channel: disables the VC and its descriptor control (CTD and CRD in VCDCTRL) before freeing the descriptor rings, and removes it from the device's VC list. Must only be called on an open VC; afterwards its rings are empty.

6.2.7.3. VC Address Filtering

```
void grspfi_vc_address_set(struct grspfi_vch *vch, u8 conf,
                           u8 vc_address, u8 vc_address_mask);
```

Programs the VC-level address filter. The *conf* argument is a bitmask of flags: `GRSPFI_CONF_FILTER` to write the address and mask, and/or `GRSPFI_EN_FILTER` to enable address-based acceptance filtering (AE bit in VCCTRL).

```
void grspfi_vc_address_clear(struct grspfi_vch *vch);
```

Disables the VC address filter (clears AE bit in VCCTRL).

```
u16 grspfi_vc_address_get(struct grspfi_vch *vch);
```

Returns the current contents of the VCADDR register (address in bits 7:0, mask in bits 15:8).

6.2.7.4. VC Configuration Getters

```
u32 grspfi_length_get(struct grspfi_vch *vch);
```

Returns the maximum RX packet length from VCMAXLEN.

```
u8 grspfi_priority_get(struct grspfi_vch *vch);
```

Returns the TX priority from VCCTRL.PR.

```
u32 grspfi_bandwidth_get(struct grspfi_vch *vch);
```

Returns the bandwidth allocation percentage stored in the VC handle.

```
u64 grspfi_timeslots_get(struct grspfi_vch *vch);
```

Returns the 64-bit time-slot bitmask from VCTS1 and VCTS2.

```
u8 grspfi_fifo_get(struct grspfi_vch *vch);
```

Returns the FIFO number from VCCTRL.EFC if FIFO routing is enabled, or 0xFF if FIFO routing is disabled.

```
u16 grspfi_rmap_get(struct grspfi_vch *vch);
```

Returns the RMAP key (bits 7:0) and the RMAP enable flag (bit 15) combined into a 16-bit value.

6.2.7.5. VC Data Transfer

6.2.7.5.1. grspfi_vc_prepare

```
int grspfi_vc_prepare(struct grspfi_vch *vch, struct grspfi_pkt *packet);
```

Enqueues an RX descriptor so the hardware can DMA incoming packet data into *packet->data*. The descriptor is written to the ring at the current prepare index and the hardware is notified. Returns 0 on success or -EAGAIN if the descriptor ring is full.

6.2.7.5.2. grspfi_vc_receive

```
int grspfi_vc_receive(struct grspfi_vch *vch, struct grspfi_pkt **packet);
```

Attempts to dequeue the oldest completed RX descriptor. On success, **packet* points to the packet with *data_length* and *flags* populated from the descriptor. The flags are GRSPFI_PKT_RX_DC / GRSPFI_PKT_RX_HC (data/header RMAP CRC error), GRSPFI_PKT_RX_TR (truncated), GRSPFI_PKT_RX_EP (EEP) and GRSPFI_PKT_RX_IE. The core always checks the RMAP CRCs, so DC and HC are also set for packets that are not RMAP packets; for those only TR and EP indicate a transfer error. Returns -EAGAIN if no completed descriptor is available (EN bit still set, or ring empty).

6.2.7.5.3. grspfi_vc_send

```
int grspfi_vc_send(struct grspfi_vch *vch, struct grspfi_pkt *packet);
```

Enqueues a TX descriptor for transmission. Packet fields *data*, *data_length*, *header*, *header_length*, and *flags* must be set by the caller. The hardware is notified after the descriptor is written. Returns 0 on success or -EAGAIN if the TX ring is full.

6.2.7.5.4. grspfi_vc_reclaim

```
int grspfi_vc_reclaim(struct grspfi_vch *vch, struct grspfi_pkt **packet);
```

Reclaims a completed TX descriptor. On success, returns the packet originally passed to grspfi_vc_send() via **packet*, allowing the caller to free or reuse the buffer. Returns -EAGAIN if no sent descriptor is ready to reclaim, or -EINVAL if the ring contains no in-flight packets.

6.2.7.6. VC Descriptor Index Helpers

The following functions return ring indices for use when mapping kernel-space descriptor slots to caller-owned packet arrays. All are thread-safe.

```
int grspfi_vc_get_prep_index(struct grspfi_vch *vch);
```

Returns the index into the RX descriptor ring where the next grspfi_vc_prepare() call will write, or -EAGAIN if the RX ring is full.

```
int grspfi_vc_get_rx_index(struct grspfi_vch *vch);
```

Returns the ring index of the oldest outstanding RX descriptor.

```
int grspfi_vc_get_tx_index(struct grspfi_vch *vch);
```

Returns the TX ring index where the next grspfi_vc_send() will write, or -EAGAIN if the TX ring is full.

Check the result of grspfi_vc_get_prep_index() and grspfi_vc_get_tx_index() before using it as an index, and store the packet in that slot only after the call succeeded.

```
int grspfi_vc_get_reclaim_index(struct grspfi_vch *vch);
```

Returns the TX ring index of the oldest packet awaiting reclaim.

6.2.7.7. VC Packet Count Helpers

```
u32 grspfi_vc_scheduled(struct grspfi_vch *vch);
```

Returns the number of TX descriptors currently enabled in the hardware (submitted but not yet transmitted).

```
u32 grspfi_vc_reclaimable(struct grspfi_vch *vch);
```

Returns the number of TX packets that have been transmitted and can be reclaimed (in-flight minus scheduled).

```
u32 grspfi_vc_received(struct grspfi_vch *vch);
```

Returns the number of RX descriptors that the hardware has completed (EN bit cleared) and are ready to be retrieved with `grspfi_vc_receive()`.

6.2.8. Broadcast Channel API

6.2.8.1. grspfi_bc_open

```
int grspfi_bc_open(struct grspfi_dev *dev, u8 channel, u8 bandwidth,
                   bool fifo_enable, struct grspfi_bch *bch);
```

Opens and configures the Broadcast Channel. Allocates DMA-coherent TX and RX message buffers (32 slots × 16 bytes each).

Table 6.10. Parameters of `grspfi_bc_open()`

Parameter	Description
`dev`	Device handle.
`channel`	Broadcast channel number written to BCCONF.CHN.
`bandwidth`	Bandwidth allocation percentage (1–95), converted and written to BCCONF.BW.
`fifo_enable`	Enable FIFO broadcast delivery (BD bit in BCCONF).
`bch`	Caller-allocated grspfi_bch handle to initialise.

Returns 0 on success, `-EBUSY` if the BC is already open, `-EINVAL` for invalid parameters, or `-ENOMEM` on allocation failure.

6.2.8.2. grspfi_bc_close

```
void grspfi_bc_close(struct grspfi_bch *bch);
```

Closes the Broadcast Channel, clears hardware registers, and frees DMA buffer memory.

6.2.8.3. BC Data Transfer

```
int grspfi_bc_send(struct grspfi_bch *bch, struct grspfi_bc_pkt *packet);
```

Enqueues a Broadcast packet for transmission by advancing the TX write pointer. Returns 0 on success, `-EAGAIN` if the TX buffer is full.

```
int grspfi_bc_receive(struct grspfi_bch *bch,
                      struct grspfi_bc_pkt *packet);
```

Dequeues the oldest received Broadcast packet by advancing the RX read pointer. Returns 0 on success, `-EAGAIN` if no packet is available.

```
u32 grspfi_bc_scheduled(struct grspfi_bch *bch);
```

Returns the number of BC packets queued for transmission.

```
u32 grspfi_bc_received(struct grspfi_bch *bch);
```

Returns the number of received BC packets available for consumption.

6.2.8.4. BC Configuration Getters

```
u8 grspfi_bc_channel_number_get(struct grspfi_bch *bch);
```

Returns the broadcast channel number from BCCONF.CHN.

```
u32 grspfi_bc_bandwidth_get(struct grspfi_bch *bch);
```

Returns the BC bandwidth allocation percentage stored in the BC handle.

```
bool grspfi_bc_fifo_get(struct grspfi_bch *bch);
```

Returns `true` if FIFO broadcast delivery is enabled.

6.2.9. DMA Channel API

6.2.9.1. grspfi_dma_open

```
int grspfi_dma_open(struct grspfi_dev *dev, u8 dma_number,
                    u32 virtual_channel_mask, bool broadcast_enable,
                    u8 txrx, struct grspfi_dmah *dmah);
```

Opens a DMA channel and binds it to a set of Virtual Channels and optionally the Broadcast Channel.

Table 6.11. Parameters of `grspfi_dma_open()`

Parameter	Description
dev	Device handle.
dma_number	DMA channel index (0 to *dmasup.num_dma* _ 1).
virtual_channel_mask	Bitmask of Virtual Channels to bind (bit *n* = VC *n*). All referenced VCs must already be open.
broadcast_enable	If `true`, this DMA channel handles Broadcast Channel traffic. The BC must be open, and only one DMA channel may be mapped to the BC at a time.
txrx	Direction bitmask: `GRSPFI_DMA_TE` to enable TX, `GRSPFI_DMA_RE` to enable RX (may be OR'd).
dmah	Caller-allocated grspfi_dmah handle to initialise.

Returns 0 on success, `-EBUSY` if the DMA channel is already open, or `-EINVAL` for an invalid channel, a VC that is not open or already bound to another DMA channel, or a Broadcast Channel that is not open or already mapped to another channel.

6.2.9.2. grspfi_dma_close

```
void grspfi_dma_close(struct grspfi_dmah *dmah);
```

Stops a DMA channel, unbinds its Virtual Channels, and removes it from the device's DMA list. If the channel carried the Broadcast Channel, that mapping is released so that a later `grspfi_dma_open()` can map broadcast again.

6.2.9.3. DMA Configuration Getters

```
u32 grspfi_dma_vc_map_get(struct grspfi_dmah *dmah);
```

Returns the VC bitmask bound to this DMA channel.

```
bool grspfi_dma_bc_map_get(struct grspfi_dmah *dmah);
```

Returns `true` if the Broadcast Channel is mapped to this DMA channel, by `grspfi_dma_open()` with broadcast enabled. The mapping ends when the DMA channel or the Broadcast Channel is closed.

6.2.10. Status Registers

6.2.10.1. Device Status

```
void grspfi_device_status_read(struct grspfi_dev *dev,
                               struct grspfi_dev_status *status);
```

Reads the Lane Layer (LLSTAT), Retry Layer (RLSTAT), and DMA Layer (DLSTAT) status registers into *status*.

```
void grspfi_device_status_clear(struct grspfi_dev *dev,
                                struct grspfi_dev_status *status_mask);
```

Clears (write-1-to-clear) device status bits indicated by *status_mask*.

6.2.10.2. Virtual Channel Status

```
void grspfi_vc_status_read(struct grspfi_vch *vch,
```

```
                            struct grspfi_vc_status *status);
```

Reads the VCSTAT register for this Virtual Channel.

```
void grspfi_vc_status_clear(struct grspfi_vch *vch,
                            struct grspfi_vc_status *status_mask);
```

Clears VC status bits indicated by *status_mask*.

6.2.10.3. DMA Status

```
void grspfi_dma_status_read(struct grspfi_dmah *dmah,
                            struct grspfi_dma_status *status);
```

Reads the DMA Channel Status (DCSTAT) and extended DMA Channel Status (DCSTAT2) registers.

```
void grspfi_dma_status_clear(struct grspfi_dmah *dmah,
                             struct grspfi_dma_status *status_mask);
```

Clears DMA status bits indicated by *status_mask*.

6.2.11. Interrupt Management

Interrupts are routed through the single per-device ISR installed by the platform driver. Upper layers register callbacks which are called from that ISR. There is one callback of each kind per device, not per DMA channel. Passing `NULL` as *isr* deregisters it. The function and its argument are stored together under the device spinlock.

The callbacks run in interrupt context with the device spinlock held. They must not sleep, and must not call `grspfi_dev_isr_register()`, `grspfi_dma_isr_register()`, `grspfi_vc_isr_register()`, `grspfi_bc_send()`, `grspfi_bc_receive()`, `grspfi_bc_scheduled()`, `grspfi_bc_received()`, `grspfi_open()` or `grspfi_close()`, which take the same lock and would deadlock.

6.2.11.1. Device-level ISR

```
void grspfi_dev_isr_register(struct grspfi_dev *dev,
                             grspfi_dev_isr_func isr, void *arg);
```

Registers a callback for device-level interrupts (link error, broadcast TX/RX). The callback prototype is: `void isr(struct grspfi_dev *dev, u32 events, struct grspfi_dev_status *status, void *arg)`. Pass `NULL` for *isr* to deregister.

```
void grspfi_dev_isr_mask(struct grspfi_dev *dev, u32 interrupt_mask);
```

Enables selected device interrupts by writing to the DMA Layer Control register (DLCTRL). Valid bits are: `GRSPFI_IE_DEV_BTE` (broadcast TX error), `GRSPFI_IE_DEV_BRE` (broadcast RX error), and `GRSPFI_IE_DEV_LE` (link error).

6.2.11.2. DMA-level ISR

```
void grspfi_dma_isr_register(struct grspfi_dev *dev,
                             grspfi_dma_isr_func isr, void *arg);
```

Registers a callback for DMA channel interrupts. The callback prototype is: `void isr(struct grspfi_dmah *dmah, u32 events, struct grspfi_dma_interrupt_info *info, void *arg)`.

```
void grspfi_dma_isr_mask(struct grspfi_dmah *dmah, u32 interrupt_mask);
```

Enables selected DMA interrupts via the DMA Channel IRQ Control register (DCICTRL). Valid bits are: `GRSPFI_IE_DMA_RI` (RX done), `GRSPFI_IE_DMA_TI` (TX done), `GRSPFI_IE_DMA_BEI` (bus error), and `GRSPFI_IE_DMA_RMI` (RX max-length exceeded).

6.2.11.3. Virtual Channel ISR

```
void grspfi_vc_isr_register(struct grspfi_dev *dev,
                            grspfi_vc_isr_func isr, void *arg);
```

Registers a callback for per-VC packet events (TX done or RX done). The callback prototype is: `void isr(int vc, u32 events, void *arg)`. The *events* bitmask uses `GRSPFI_EVENT_VC_TX` and

GRSPFI_EVENT_VC_RX. Events are detected by comparing the per-VC packet statistics counter in hardware with a software shadow.

6.3. Device tree

6.3.1. Bindings

In order to use the device drivers on NOEL-V the user needs to provide device tree bindings.

The grspfi_drv platform driver binds to device-tree nodes with the following compatible strings:

- gaisler,grspfi
- GAISLER_SPFI
- 01_0bc
- GAISLER_HSSL
- 01_0c8

Each node must provide a register memory region (index 0) and an interrupt (index 0).

The character device module (grspfi) is not a platform driver. When it is loaded it creates /dev/grspfi*N* for every probed GRSPFI core. The optional gaisler,grspfiu node only lets udev load the module automatically.

GRSPFI devices cannot be unbound through sysfs. They are only removed when grspfi_drv is unloaded, which is not possible while grspfi is loaded.

Below is an example device tree description of two GRSPFI devices and their gaisler,grspfi binding, plus the optional gaisler,grspfiu node.

```
grspfi0: grspfi@ff1c0000 {
  compatible = "gaisler,grspfi";
  reg = <0xff1c0000 0x1000>;
  interrupt-parent = <&plic0>;
  interrupts = <23>;
};

grspfi1: grspfi@ff1c1000 {
  compatible = "gaisler,grspfi";
  reg = <0xff1c1000 0x1000>;
  interrupt-parent = <&plic0>;
  interrupts = <23>;
};

grspfi0user: grspfiuser@0 {
  compatible = "gaisler,grspfiu";
};
```

Frontgrade Gaisler AB
Kungsgatan 12
411 19 Göteborg
Sweden
frontgrade.com/gaisler
sales@gaisler.com
T: +46 31 7758650
F: +46 31 421407

Frontgrade Gaisler AB, reserves the right to make changes to any products and services described herein at any time without notice. Consult the company or an authorized sales representative to verify that the information in this document is current before using this product. The company does not assume any responsibility or liability arising out of the application or use of any product or service described herein, except as expressly agreed to in writing by the company; nor does the purchase, lease, or use of a product or service from the company convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual rights of the company or of third parties. All information is provided as is. There is no warranty that it is correct or suitable for any purpose, neither implicit nor explicit.

Copyright © 2026 Frontgrade Gaisler AB