FRONTGRADE Gaisler

USER MANUAL

RELEASED JUNE 2026

GRLIB Linux device drivers

LINDRV

GRLIB Linux Drivers User's Manual

LINDRV-UM
Jun 2026, Version 1.4.0

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
  1.3. Installing 6
  1.4. Device tree bindings 6
  1.5. Device node numbering 6
2. GRSPW SpaceWire Driver 7
  2.1. Introduction 7
    2.1.1. Sources 7
    2.1.2. Using the driver 7
    2.1.3. Examples 8
  2.2. Control Interface 8
    2.2.1. Overview 8
  2.3. Packet Transfer Interface 10
    2.3.1. Packet Reception 11
    2.3.2. Packet Transmission 13
  2.4. User-space access routines 14
3. GRSPW Kernel Library driver 15
  3.1. Introduction 15
    3.1.1. Hardware Support 15
    3.1.2. Driver sources 15
    3.1.3. Examples 15
    3.1.4. Known driver limitations 15
  3.2. Software design overview 15
    3.2.1. Overview 15
    3.2.2. Initialization 16
    3.2.3. Link control 16
    3.2.4. Time Code support 17
    3.2.5. RMAP support 17
    3.2.6. Port support 17
    3.2.7. SpaceWire node address configuration 17
    3.2.8. SpaceWire Interrupt Code support 18
    3.2.9. User DMA buffer handling 18
    3.2.10. Driver DMA buffer handling 18
    3.2.11. Polling and blocking mode 20
    3.2.12. Interrupt and work queue 20
    3.2.13. Starting and stopping DMA 21
    3.2.14. Thread concurrency 21
    3.2.15. SMP Support 22
    3.2.16. User space support 22
  3.3. Device Interface 22
    3.3.1. Opening and closing device 23
    3.3.2. Hardware capabilities 24
    3.3.3. Link Control 25
    3.3.4. Node address configuration 27
    3.3.5. Time Code support 28
    3.3.6. Port Control 29
    3.3.7. RMAP Control 30
    3.3.8. Statistics 31
  3.4. DMA interface 32
    3.4.1. Opening and closing DMA channels 32
    3.4.2. Starting and stopping DMA operation 34
    3.4.3. Packet buffer description 36
    3.4.4. Blocking/Waiting on DMA activity 37
    3.4.5. Sending packets 38
    3.4.6. Receiving packets 41

3.4.7. Transmission queue status 43
3.4.8. Statistics 44
3.4.9. DMA channel configuration 46
3.5. API reference 47
3.5.1. Data structures 47
3.5.2. Device functions 47
3.5.3. DMA functions 48
4. SpaceWire Router APB Register Driver 50
4.1. Introduction 50
4.1.1. Sources 50
4.1.2. Using the driver 50
4.1.3. Examples 50
4.2. Control Interface 50
4.2.1. Overview 50
5. MAPLIB Device Memory Driver 52
5.1. Introduction 52
5.1.1. Sources 52
5.1.2. Using the driver 52
5.1.3. Examples 52
5.2. Control Interface 53
5.3. Mapping Interface 53
6. GRSPFI SpaceFibre Driver 55
6.1. Introduction 55
6.1.1. Sources 55
6.1.2. Using the driver 55
6.1.3. Examples 55
6.2. ioctl Reference 55
6.2.1. Overview 55
6.2.2. GRSPFI_IOCTL_GET_CAPABILITIES 56
6.2.3. GRSPFI_IOCTL_CONFIG 56
6.2.4. GRSPFI_IOCTL_VC_CONFIG 57
6.2.5. GRSPFI_IOCTL_BC_CONFIG 57
6.2.6. GRSPFI_IOCTL_TIME_CONFIG 58
6.2.7. GRSPFI_IOCTL_TIME_SET 58
6.2.8. GRSPFI_IOCTL_TIME_GET 59
6.2.9. GRSPFI_IOCTL_DMA_CONFIG 59
6.2.10. GRSPFI_IOCTL_RX_PREPARE 59
6.2.11. GRSPFI_IOCTL_RX_RECEIVE 60
6.2.12. GRSPFI_IOCTL_TX_SEND 61
6.2.13. GRSPFI_IOCTL_TX_RECLAIM 61
6.2.14. GRSPFI_IOCTL_BC_SEND 61
6.2.15. GRSPFI_IOCTL_BC_RECEIVE 62
6.2.16. GRSPFI_IOCTL_TX_WAIT 62
6.2.17. GRSPFI_IOCTL_RX_WAIT 62
6.2.18. GRSPFI_IOCTL_BCRX_WAIT 63
7. GRSPFI Kernel Library Driver 64
7.1. Introduction 64
7.1.1. Driver sources 64
7.1.2. Examples 64
7.2. Software design overview 64
7.2.1. Overview 64
7.2.2. Key Data Structures 64
7.2.3. Module Registration 65
7.2.4. Device Lifecycle 65
7.2.5. Link Layer Control 66
7.2.6. Time-slot Management 67
7.2.7. Virtual Channel API 67
7.2.8. Broadcast Channel API 70
7.2.9. DMA Channel API 70

7.2.10. Status Registers 71
7.2.11. Interrupt Management 72
7.3. Device tree 72
7.3.1. Bindings 72

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. Currently the drivers has not been tested as modules, so for the time being please install the driver sources into the kernel and link them into the kernel.

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):

```
[grlib_drivers/]$ make KERNELDIR=/path/to/kernel/linux/sources 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
- MAPLIB, Device memory handling. Enables a user to memory map blocks of linear memory that can be used by device drivers for DMA access. GRLIB Drivers that implement zero-copy to user-space and between device nodes though user-space require the MAPLIB char 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

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 MK-LINUXIMG, 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* maplib* spwrouter*
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      248,   0 Apr  29  2025 maplib0
crw-------    1 root     root      249,   0 Apr  29  2025 spwrouter0
```

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 uses the memory map driver (MAPLIB described in Chapter 5) for allocating physically continuous device memory (DMA memory) for 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 three parts it is possible to reuse specific parts of the driver source. For example the GRSPW kernel library does not depend on MAPLIB or 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 MAPLIB does not either depend on the other parts, hence it can be used solely in other drivers or together with other drivers. This makes it for example possible to receive a SpaceWire packet and transmitting it using a driver for another interface also supporting the MAPLIB driver.

The driver provides two different types of interfaces through the standard UNIX access routines (`open, close, ioctl, read, write`), one GRSPW device control interface and one packet transfer interface. The control interface is accessed using `ioctl`, whereas the packet transfer interface is accessed using `read` and `write`. The actual packet data transferred on SpaceWire is not read or written using the `read` and `write` routines, instead pointers 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. There are other issues as well that must be solved, they are taken care of in the MAPLIB driver.

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 should include the "GRSPW Kernel Driver header" file. All files are relative the base of the driver package.

Table 2.1. GRSPW driver sources

Location	Description
`spw/grspw.c`	GRSPW Kernel library
`spw/grspw_user.c`	GRSPW Kernel Driver
`misc/maplib.c`	Device memory library
`include/linux/grlib/grspw.h`	GRSPW Kernel library header
`include/linux/grlib/grspw_user.h`	GRSPW Kernel Driver header
`include/linux/grlib/maplib.h`	Device memory library header

2.1.2. Using the driver

Applications wanting to access GRSPW devices from user-space should include the GRSPW kernel driver header file, if the include path is set correct it will include the kernel library header as well. As mentioned above the user is also responsible to setup device memory using the MAPLIB driver, so the application should also include the MAPLIB header file.

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.

2.1.3. 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, read and write 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/stopping 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 and their argument type is listed. The data structures referenced are declared in the `grspw_user.h` header file. 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: 32-bit simple Argument, no memory transferred between kernel/user.
- None: Argument ignored.

Table 2.2. `ioctl` commands supported by the GRSPW Kernel driver.

Command	Data Direction	Argument Type	Description
HWSUP	Output	struct grspw_hw_sup *	Copy hardware configuration for the GRSPW core, such as number of DMA Channels, if RMAP/RMAP-CRC is supported by core, number of SpW ports, etc.
BUFCFG	Input	struct grspw_bufcfg *	Set up packet buffers. Even though the user is responsible for allocating memory for packet data/header, the driver must allocate structures for packet handling. The packet structures stores the packet state, data/header pointers, packet number etc. This command specifies how many packets maximally can simultaneously be buffer internally by the driver. 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	Input	struct grspw_config *	Configure driver according to input. One can configure promiscuous mode, which DMA channels will be used, DMA channel configuration, register a custom time code ISR handler (note that it must be an address to a function in kernel, typically to a custom user-written module), time code RX/TX enable and RMAP options (destination key, RMAP enable, RMAP buffer).
CONFIG_READ	Output	struct grspw_config *	Copies the current configuration to the address given by the argument. DMA Channel configuration will only be

Command	Data Direction	Argument Type	Description
			copied for previously enabled channels, for other channels the data is undefined.
STATS_READ	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.
STATS_CLR	None	N/A	Clears the current gathered statistics. Resets all counters.
LINKCTRL	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 LINKOPTS_* options.
PORTCTRL	Argument	int	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. • 0: Port0 always selected. • 1: Port1 always selected. • Others: Both Port0 and Port1, core selects between them.
LINKSTATE	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	Argument	int	This command sets the TCTRL and TIMECNT bits of the GRSPW core if bit 8 is set to one. The TCTRL and TIMECNT values are taken from the low 8-bits of the argument. After (optionally) setting the TCTRL:TIMECNT a Tick-In is generated if bit 9 is set to one.
TC_READ	Output	int *	This command stores the current value of the GRSPW core TCTRL:TIMECNT bits to the address given by the argument.
QPKTCNT	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.
STATUS_READ	Output	unsigned int *	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	Input	unsigned int *	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 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.
START	None	N/A	Start all DMA activity on all DMA channels. The receiver is enabled however packet buffers must be prepared in order to actually receive anything. After starting the `read/write` interface of the driver is open. See the Packet Transfer Interface on how packets are sent/received. After

Command	Data Direction	Argument Type	Description
			start the BUFCFG and CONFIG_SET `ioctl` commands are not available until stopped again. If this command fails with the errno ENOMEM packet structures was not able to be allocated due to either not enough memory or too many requested. If errno is set to EPERM the driver indicates that the MAPLIB was not satisfied (for example not mapped to user space).
STOP	None	N/A	Stops DMA operation, this till disable the receiver and transmitter of the GRSPW core. 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.
RX_WAIT	Input	struct grspw_rx_wait_chan *	Blocks the caller until the RX queue packet counters conditions are fullfilled. The conditions and timeout are described by the input data structure, see struct grspw_rx_wait_chan for usage. If timeout expires before the conditions are fullfilled -ETIME will be returned. If -EIO is returned if the DMA channels is not started or is stopped during the waiting. There is only one RX wait object per DMA channels which means that only one thread can simultaneously wait on RX. If two threads tries to wait on the same DMA channel -EBUSY error code is returned.
TX_WAIT	Input	struct grspw_tx_wait_chan	Blocks the caller until the TX queue packet counters conditions are fullfilled. The conditions and timeout are described by the input data structure, see struct grspw_tx_wait_chan for usage. If timeout expires before the conditions are fullfilled -ETIME will be returned. If -EIO is returned if the DMA channels is not started or is stopped during the waiting. There is only one TX wait object per DMA channels which means that only one thread can simultaneously wait on TX. If two threads tries to wait on the same DMA channel -EBUSY error code is returned.

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.

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 buffers itself, but relies on MAPLIB 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 `write` function.

- 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 using the standard UNIX `read/write` file operation calls. Since both `read` and `write` takes different input depending on which of the two operation is requested, the MSB 16-bit of the length is used to determine operation and which DMA channels are involved in the request. See GRSPW_READ_* and GRSPW_WRITE_* definitions in header file.

> The way the driver uses the read/write lenght is not standard and the LIBC compile-time or run-time checks may complain or fail. If so the LIBC provides alternative functions that can be used where the checks are not performed.

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.

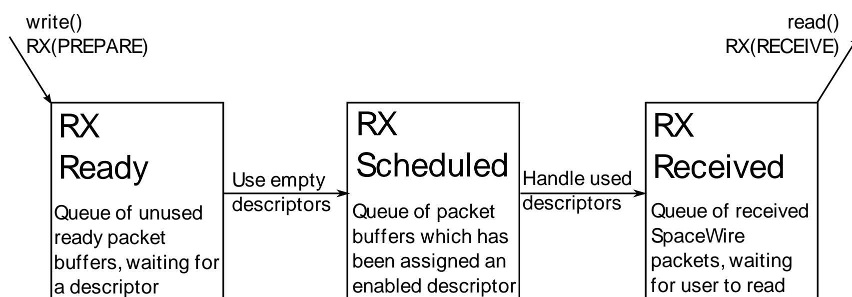

Figure 2.1. GRSPW Driver internal RX queues

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 by calling the standard UNIX `write` function with one or an array of struct grspw_wrxpkt entries. Each entry describes one packet buffer, see below programlisting and table. The length of the write buffer must be a multiple of the size of one entry, the MSB bits of the length determines which channel the packet buffers are for and selects between the RX(PREPARE) and the TX(SEND) operation. If the driver is out of packet structures (used internally in driver) all packet buffers will not be prepared, instead the length returned determines how many packets was added to the ready queue.

```
/* GRSPW Write RX-Packet Entry (PREPARE RX BUFFER) */
struct grspw_wrxpkt {
        int pkt_id;             /* Custom Packet ID */
        unsigned short flags;   /* See RXPKT_FLAG* above */
        unsigned short resv1;   /* Reserved, must be zero */
        void *data;             /* Data Pointer (Address from MMAP Lib). The
                                 * buffer must have room for max-packet */
} __attribute__((packed));
```

Table 2.3. GRSPW prepare RX buffers write format (struct grspw_wrxpkt)

Field	Description
pkt_id	A user defined packet ID which can be used to identify the packet buffer upon RX(RECEIVE). This is field is optional, and does not affect the operation of the driver.
flags	Set to RXPKT_FLAG_IE if this packet should generate a interrupt when a SpaceWire packet was received to this packet buffer. Interrupts can be controlled using the control interface.
data	Pointer to the packet buffer that the driver will store one received SpaceWire Packet to. The address must be within the range that was memory mapped with MAPLIB, a user space address is expected.

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 using the standard UNIX `read` function. This process is called RX(RECEIVE) in this document. The driver will fill the user provided buffer 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*. The length of the read buffer must be a multiple of the size of one entry, the MSB bit of the length determines which channels (bit mask of channels) to receive packets from and selects between the RX(RECEIVE) and TX(RECLAIM) operation.

```
/* GRSPW Read RX-Packet Entry (RECEIVE) */
struct grspw_rrxpkt {
        int pkt_id;              /* Custom Packet ID */
        unsigned short flags;    /* See RXPKT_FLAG* above */
        unsigned char dma_chan;  /* DMA Channel 0..3 */
        unsigned char resv1;     /* Reserved, must be zero */
        int dlen;                /* Data Length */
        void *data;              /* Data Pointer (Address from MMAP Lib) */
} __attribute__((packed));
```

Table 2.4. GRSPW receive RX packet buffers read format (struct grspw_rrxpkt)

Field	Description
pkt_id	A user defined packet ID that was given to the driver together with the packet buffer in RX(PREPARE).
flags	This field indicates if the data buffer contains a SpaceWire packet (RXPKT_FLAG_RX), and if transfer errors where encountered during the reception (Truncated, EEOP, Header CRC error, Data CRC error).
dma_chan	Indicates which DMA channel (0..3) received this packet.

Field	Description
dlen	The length of SpaceWire packet that was received into the packet buffer pointed to by `data`.
data	Pointer to the packet buffer that contains one SpaceWire packet. The `flags` field bit RXPKT_FLAG_RX is set if a the buffer contains a SpaceWire packet, other flags may also have been set to indicate some sort of SpaceWire transmission error.

2.3.2. Packet Transmission

The packet transmission interface works basically the same as the packet reception interface. The MSB bits of the length determine that TX(SEND) and TX(RECLAIM) should be used instead of the RX operations. 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.

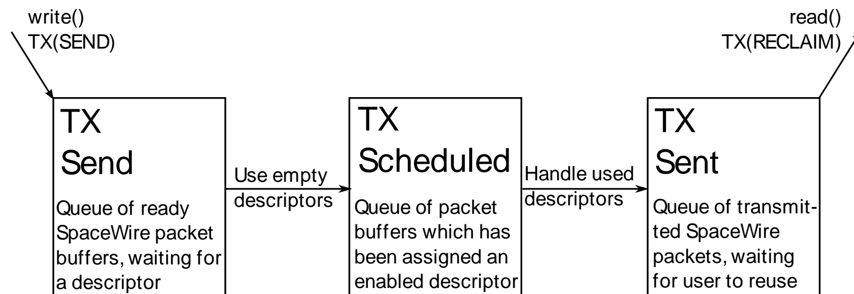

Figure 2.2. GRSPW Driver internal TX queues

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 by calling the standard UNIX `write` function with one or an array of struct grspw_wtxpkt entries. Each entry describes one packet buffer, see below programlisting and table. The length of the write buffer must be a multiple of the size of one entry, the MSB bits of the length determines which channel the packets will be sent upon and selects between the RX(PREPARE) and the TX(SEND) operation. If the driver is out of packet structures (used internally in driver) all packets will not be sent, instead the length returned determines how many packets was added to the send queue.

```
/* GRSPW Write TX-Packet Entry (SEND PACKET) */
struct grspw_wtxpkt {
        int pkt_id;             /* Custom Packet ID */
        unsigned short flags;   /* See TXPKT_FLAG* above */
        unsigned char resv;     /* Reserved */
        unsigned char hlen;     /* Header Length. Set to zero if none. */
        unsigned int dlen;      /* Data Length. Set to zero if none. */
        void *hdr;              /* Header Pointer (Address from MMAP Lib) */
        void *data;             /* Data Pointer (Address from MMAP Lib) */
} __attribute__((packed));
```

Table 2.5. GRSPW send TX packet buffers write format (struct grspw_wtxpkt)

Field	Description
pkt_id	A user defined packet ID which can be used to identify the packet buffer upon TX(RECLAIM). This is field is optional, and does not affect the operation of the driver.
flags	This field hold the transmission options for one SpaceWire packet. See TXPKT_FLAG_* for options. One can enable IRQ on DMA transmit operation, header and data CRC calculation.
hlen	Determines the length of the header, set to zero if no header should be transmitted. A length larger than 255 bytes is not allowed.
dlen	Determines the length of the data that will be transmitted. The maximum length is limited to 128KBytes due to the memory allocation.

Field	Description
hdr	Pointer to the packet header buffer. This is only used if `hlen` is larger than zero. The first `hlen` bytes are transmitted.
data	Pointer to the packet buffer that contains the data of one SpaceWire packet. The first `dlen` bytes are transmitted.

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 using the standard UNIX `read` function. This process is called TX(RECLAIM) in this document. The driver will fill the user provided read buffer 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.

The length of the read buffer must be a multiple of the size of one entry, the MSB bits of the length determines which channels (bit mask of channels) to reclaim packets from and selects between the RX(RECEIVE) and TX(RECLAIM) operation.

```
/* GRSPW Read TX-Packet Entry (RECLAIM TX BUFFER) */
struct grspw_rtxpkt {
        int pkt_id;                /* Custom Packet ID */
        unsigned short flags;      /* See TXPKT_FLAG* above */
        unsigned char dma_chan;    /* DMA Channel 0..3 */
        unsigned char resv1;       /* Reserved, must be zero */
} __attribute__((packed));
```

Table 2.6. GRSPW reclaim TX packet buffers read format (struct grspw_rtxpkt)

Field	Description
pkt_id	A user defined packet ID which can be used to identify the packet buffer upon TX(RECLAIM). This is field is optional, and does not affect the operation of the driver.
flags	This field hold the transmission parameters for one SpaceWire packet. See TXPKT_FLAG_*. If the the packet was sent (a descriptor with the data/header was enabled) the TXPKT_FLAG_TX bit is set, if a link error occurred TXPKT_FLAG_LINKERR bit is set.
dma_chan	Indicates which DMA channel (0..3) this packet was sent on.

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, read, etc.). 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 2.1.3.

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 `grspw_close` requires that all of its DMA channels have been closed. Closing a device fails if DMA channels are still open.

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 3.3.8.

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 specific DMA channel's node address is thus affected by the "global" device API and not controllable using the DMA channel interface.

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. SpaceWire Interrupt Code support

The GRSPW2 has optionally support for receiving and generating SpaceWire Interrupt codes. The Interrupt Codes implementation is based on the Time Code service but with a different Time Code Control content.

The SpaceWire Interrupt Code interface are controlled from the device interface.

Function names prefix: `grspw_ic_*()`

3.2.9. 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_*()`

3.2.9.1. 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_*()`

3.2.10. 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 3.30. 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.

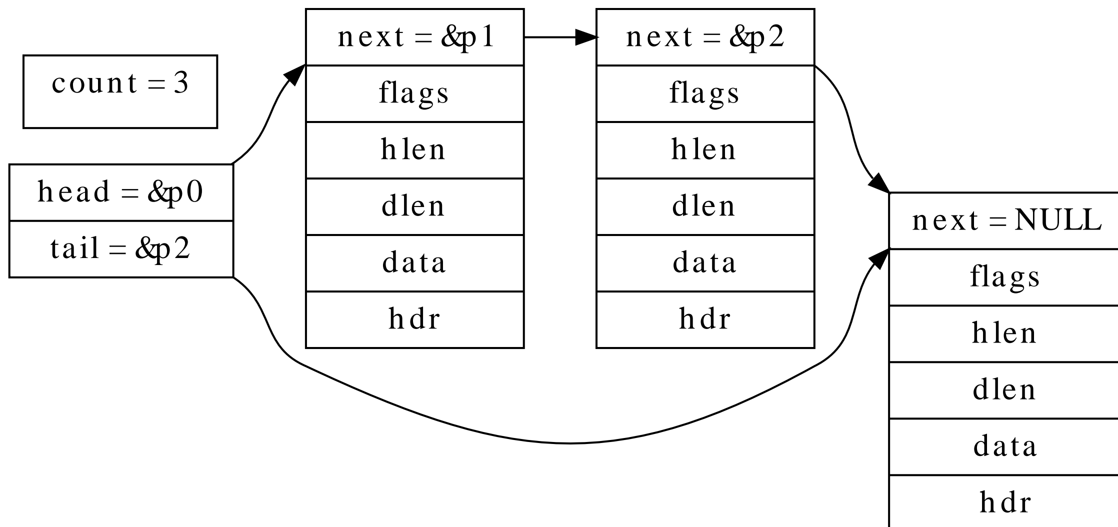

Figure 3.1. Queue example - linked list of three grspw_pkt packets

3.2.10.1. 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.

3.2.10.2. 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.

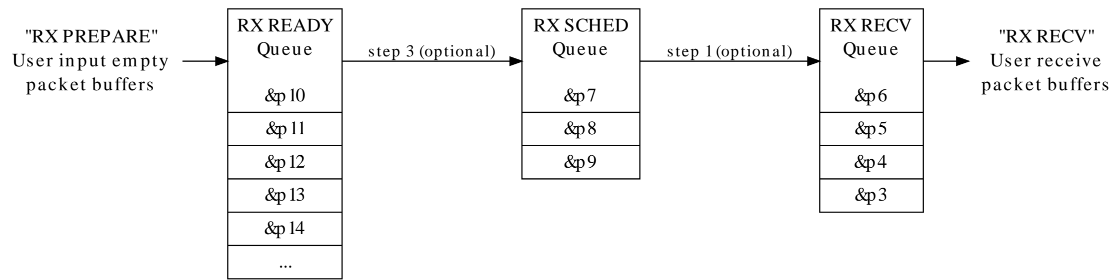

Figure 3.2. RX queue packet flow and operations

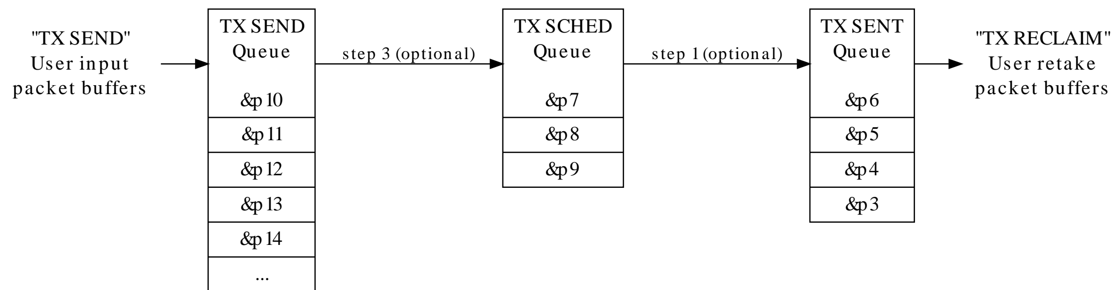

Figure 3.3. TX queue packet flow and operations

3.2.11. 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()`

3.2.12. 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.

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()`.

3.2.13. 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}()`

3.2.14. 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 semaphores to allow operations on different DMA channels simultaneously as well as simultaneous RX and TX operations on the same DMA channel. However multiple RX and TX tasks of 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 semaphores are taken by the work queue during DMA processing if the user has enabled it. There is a global device semaphore that manages device open/close operations that introduce dependencies between different GRSPW device and between DMA channels on those operations. The DMA channels and device interface share the same GRSPW I/O registers which needs in some cases to be protected, they are protected from each other by using interrupt disabling (or spin-locks on SMP).

Each DMA channel also has two semahpores 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 incomming/outgoing packets processing has finished.

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

Table 3.1. DMA channel semaphore operations.

Function	Operation	Semaphore	Description
`dma_open`	Init semaphores	RX TX	RX and TX semaphores are initialized to 1.
`dma_close`	Free semaphores	RX TX	Both RX and TX semaphores are taken and left in locked state or deleted on a successful close. From

Function	Operation	Semaphore	Description
			this point the user can not enter other DMA functions than `dma_open`.
`dma_start`	Init semaphores	RX-WAIT TX-WAIT	The wait semaphores are initialized to 0 (locked) state. 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 semaphores are taken and returned in sequence during stopping a DMA channel. The RX-WAIT and TX-WAIT semahpores are signalled in order for potential locked tasks to be worken up and return to caller with an error code or indicating DMA stopped (1) error code.
`dma_rx_recv` `dma_rx_prepare` `dma_rx_count`	RX DMA operations	RX	Holds the RX semahpore while performing RX operations.
`dma_tx_send` `dma_tx_reclaim` `dma_tx_count`	RX DMA operations	RX	Holds the RX semaphore while performing TX operations.
`dma_tx_wait`	Wait for TX DMA.	TX TX-WAIT	Takes the TX semaphore to initialize the wait structures. 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 semaphore to initialize the wait structures. 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.
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.

3.2.15. SMP Support

The driver has been designed with SMP in mind. Data structures, interrupt handling routine and GRSPW control register accesses are spin-lock protected when SMP is enabled.

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.

3.2.16. 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 buffer DMA memory is mapped into both kernel and user space using `mmap()` to allow an efficent 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.

An opened device can not be reopened unless the device is closed first. When opening a device the device 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 and closing.

During opening of a GRSPW device the following steps are taken:

- 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 GRSPW device Interrupt Service Routine (ISR) is installed and enabled. However hardware does not generate interrupt until the user configures the device or DMA channel to generate interrupts.
- The driver is configured to clear all link status events from the ISR.
- The device is marked opened to protect the caller from other users of the same device.

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, &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.
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 device semaphore fails or device already is open.
	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. All DMA channels must have been stopped and closed by the user prior to calling this function. 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	Device was successfully closed, or already previously closed.
	1	Failure due to a DMA channel is open for this device.
	-1	Failure due to invalid input arguments or unknown semaphore error.

3.3.2. 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 3.5. `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 3.6.
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 3.6. 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.	

3.3.3. 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 3.7. `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 3.8. `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 3.9. `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 3.10. `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.

3.3.4. 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 3.11. grspw_addr_config data structure declaration

<table>
<tr><td>promiscuous</td><td colspan="2">Enable (1) or disable (0) promiscous mode. The GRSPW will ignore the address field and put all received packets to first DMA channel. See hardware manual for. This field is also used to by the driver indicate if the settings should be written and read, or only read. See function description.</td></tr>
<tr><td>def_addr</td><td colspan="2">GRSPW default node address.</td></tr>
<tr><td>def_mask</td><td colspan="2">GRSPW default node address mask.</td></tr>
<tr><td rowspan="4">dma_nacfg</td><td colspan="2">DMA channel node address array configuration, see below field description. DMA channel N is described by <code>dma_nacfg[N]</code>.</td></tr>
<tr><td>Field</td><td>Description</td></tr>
<tr><td>node_en</td><td>Enable (1) or disable (1) separate node address for DMA channel N (determined by array index).</td></tr>
<tr><td>node_addr</td><td>If separate node address is enabled this option sets the node address for DMA channel N (determined by array index).</td></tr>
<tr><td>node_mask</td><td colspan="2">If separate node address is enabled this option sets the node address mask for DMA channel N (determined by array index).</td></tr>
</table>

Table 3.12. `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 3.11. 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]`*.
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 *promiscous* field is set to -1 the hardware is not written, instead the current configuration is only read and memory content updated accordingly.
Return	None.

3.3.5. 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 3.13. `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.	
	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).
Return	None.	

Table 3.14. `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 3.15. `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 3.16. `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.

3.3.6. 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 3.17. `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 3.18. `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 3.19. `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.

3.3.7. 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 3.30 for more information.

Table 3.20. `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 3.21. `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.

3.3.8. 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 3.22. 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 3.23. `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 3.22. 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 3.22.
Return	None.

Table 3.24. `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 and closing.

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 like semaphores used for the DMA channel implementation itself are allocated and initialized.
- 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, &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);

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

  /* 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 3.25. `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 DMA channel. Fails if device semaphore operation fails, DMA channel does not exists, DMA channel already has been opened or that DMA channel resource allocation or initialization 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 3.26. `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, free DMA channels resources and to unblock tasks waiting for RX/TX events on this DMA channel. Blocked tasks must have exited the device driver otherwise an error code is returned. If threads have been blocked within DMA operations they will be woken up and `grspw_dma_close` waits N ticks until they have returned to the caller with an error return value.
Param	`c` [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Return	int. Return code as indicated below. **Value** / **Description** 0 — Success. 1 — Failure due to DMA channel is active (started) or tasks may be blocked within the driver by the RX/TX wait interface of this specific device. -1 — Failure due to invalid input arguments or unknown semaphore error.

3.4.2. Starting and stopping DMA operation

The start and stop operational modes are described in Section 3.2.13. 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 3.27, see function description for details on limitations.

Table 3.27. functions available in the two operational modes

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

Function	Stopped	Started
`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

Table 3.28. `grspw_dma_start` function declaration

<table>
<tr><td>Proto</td><td><code>int grspw_dma_start(void *c)</code></td></tr>
<tr><td>About</td><td>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 <code>grspw_dma_rx_prepare</code>. The transmitter is enabled when the user provides send buffers that ends up in the TX SCHED queue, see <code>grspw_dma_tx_send</code>.</td></tr>
<tr><td>Param</td><td><code>d</code> [IN] pointer

Device identifier. Returned from <code>grspw_open</code>.</td></tr>
<tr><td>Return</td><td>int. Always returns zero.</td></tr>
</table>

Table 3.29. `grspw_dma_stop` function declaration

<table>
<tr><td>Proto</td><td><code>void grspw_dma_stop(void *c)</code></td></tr>
<tr><td>About</td><td>Stops DMA operational mode for the DMA channel indicated by the argument. The transmitter will abort ongoing transfers and the receiver disabled.

Blocked tasks within the DMA channel will be woken up and return to caller with an error indication. This will cause the stop function to wait in N ticks until the blovked tasks have exited the driver. When no tasks have previously been blocked this function is not blocking either.

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.</td></tr>
<tr><td>Param</td><td><code>d</code> [IN] pointer

Device identifier. Returned from <code>grspw_open</code>.</td></tr>
</table>

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 3.30. grspw_pkt data structure declaration

Field		
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 3.2.11. The functions described below are used to set up RX or TX wait conditions and blocks the calling thread until condition evaluates true.

Table 3.31. `grspw_dma_tx_wait` function declaration

Proto	`int grspw_dma_tx_wait(void *c, int send_cnt, int op, int sent_cnt, int 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.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_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] int Sets the timeout in number of system clock ticks. The operating system's semaphore service is used to implement the timeout functionality. Set to zero to disable timeout, negative value is invalid.
Return	Int. See return code below. Value | Description -1 | Error. 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.

Table 3.32. `grspw_dma_rx_wait` function declaration

Proto	`int grspw_dma_rx_wait(void *c, int recv_cnt, int op, int ready_cnt, int 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.
Param	`d` [IN] pointer Device identifier. Returned from `grspw_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] int Sets the timeout in number of system clock ticks. The operating system's semaphore service is used to implement the timeout functionality. Set to zero to disable timeout, negative value is invalid.
Return	Int. See return code below.

Value	Description
-1	Error.
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.

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.

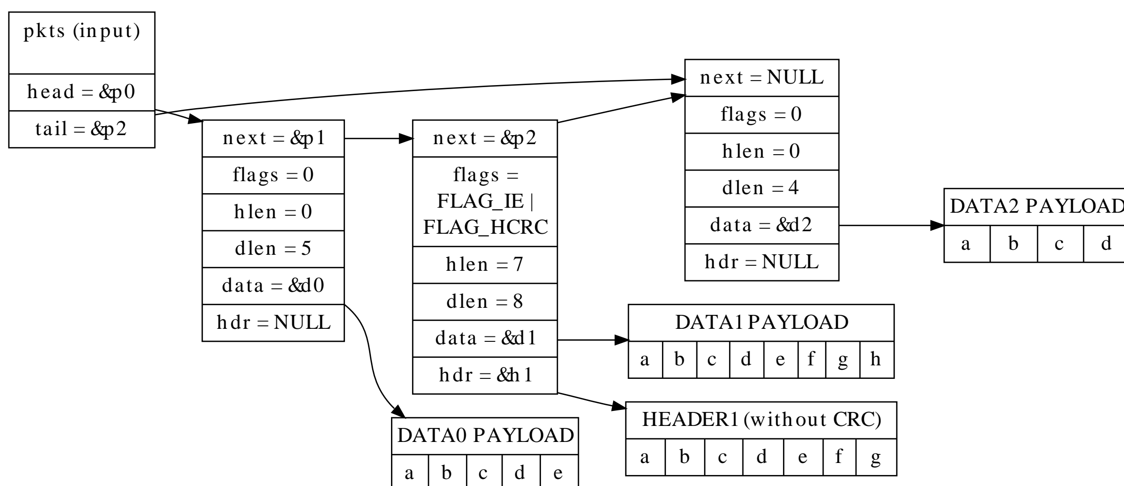

Figure 3.4. TX packet description `pkts` input to `grspw_tx_dma_send`

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

Table 3.33. `grspw_dma_tx_send` function declaration

<table>
<tr><td>Proto</td><td><code>int grspw_dma_tx_send(void *c, int opts, struct grspw_list *pkts, int count)</code></td></tr>
<tr><td>About</td><td>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.</td></tr>
<tr><td>Param</td><td><code>c</code> [IN] pointer
DMA channel identifier. Returned from <code>grspw_dma_open</code>.</td></tr>
<tr><td>Param</td><td><code>opts</code> [IN] Integer bit-mask
The above steps 1 and/or 3 may be skipped by setting <code>opts</code> argument according the description below.<table><tr><td>Bit</td><td>Description</td></tr><tr><td>0</td><td>Set to 1 to skip Step 1.</td></tr><tr><td>1</td><td>Set to 1 to skip Step 3.</td></tr></table></td></tr>
<tr><td>Param</td><td><code>pkts</code> [IN] pointer
A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that <code>head</code> points to the first packet and <code>tail</code> points to the last.

Call this function with <code>pkts</code> set to NULL to avoid step 2. Just doing step 1 and 3 as determined by <code>opts</code> 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 3.30. Note that TXPKT_FLAG_TX of the <code>flags</code> field must not be set.</td></tr>
<tr><td>Param</td><td><code>count</code> [IN] integer
Number of packets in the packet list.</td></tr>
<tr><td>Return</td><td>Status. See return codes below<table><tr><td>Value</td><td>Description</td></tr><tr><td>-1</td><td>Error occurred, DMA channel may not be valid.</td></tr><tr><td>0</td><td>Successfully added pkts to TX SEND/SCHED list.</td></tr><tr><td>1</td><td>DMA stopped. No operation.</td></tr></table></td></tr>
<tr><td>Notes</td><td>This function performs no operation when the DMA channel is stopped.</td></tr>
</table>

Table 3.34. `grspw_dma_tx_reclaim` function declaration

<table>
<tr><td>Proto</td><td><code>int grspw_dma_tx_reclaim(void *c, int opts, struct grspw_list *pkts, int *count)</code></td></tr>
<tr><td>About</td><td>Reclaim TX packet buffers that has previously been scheduled for transmission with <code>grspw_dma_tx_send</code>. The packets in the SENT queue are moved to the <code>pkts</code> 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:</td></tr>
</table>

<table>
<tr><td></td><td>
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.
</td></tr>
<tr><td>Param</td><td>
<code>c</code> [IN] pointer
DMA channel identifier. Returned from <code>grspw_dma_open</code>.
</td></tr>
<tr><td>Param</td><td>
<code>opts</code> [IN] Integer bit-mask
The above steps 1 and/or 3 may be skipped by setting <code>opts</code> argument according the description below.
<table>
<tr><td>Bit</td><td>Description</td></tr>
<tr><td>0</td><td>Set to 1 to skip Step 1.</td></tr>
<tr><td>1</td><td>Set to 1 to skip Step 3.</td></tr>
</table>
</td></tr>
<tr><td>Param</td><td>
<code>pkts</code> [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 <code>head</code> points to the first packet, <code>tail</code> points to the last and the last packet (tail) next pointer is NULL.

Call this function with <code>pkts</code> set to NULL to avoid step 2. Just doing step 1 and 3 as determined by <code>opts</code> 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 3.30. Note that TXPKT_FLAG_TX of the <code>flags</code> 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.
</td></tr>
<tr><td>Param</td><td>
<code>count</code> [IO] pointer
Number of packets in the packet list.
<table>
<tr><td>Value</td><td>Input Description</td></tr>
<tr><td>NULL</td><td>Move all packets from the SENT queue to the packet list.</td></tr>
<tr><td>-1</td><td>Move all packets from the SENT queue to the packet list.</td></tr>
<tr><td>0</td><td>No packets are moved. Same as if pkts is NULL.</td></tr>
<tr><td>>0</td><td>Move a maximum of '*count' packets to the packet list.</td></tr>
<tr><td>Value</td><td>Output Description</td></tr>
<tr><td>NULL</td><td>Nothing performed.</td></tr>
<tr><td>others</td><td>'*count' is updated to reflect number of packets in packet list.</td></tr>
</table>
</td></tr>
<tr><td>Return</td><td>
Status. See return codes below
<table>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>-1</td><td>Error occurred, DMA channel may not be valid.</td></tr>
<tr><td>0</td><td>Successful. pkts list filled with all packets from sent list.</td></tr>
<tr><td>1</td><td>Indicates that DMA is stopped. Same as 0 but step 1 and 3 were never done.</td></tr>
</table>
</td></tr>
<tr><td>Notes</td><td>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.</td></tr>
</table>

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.

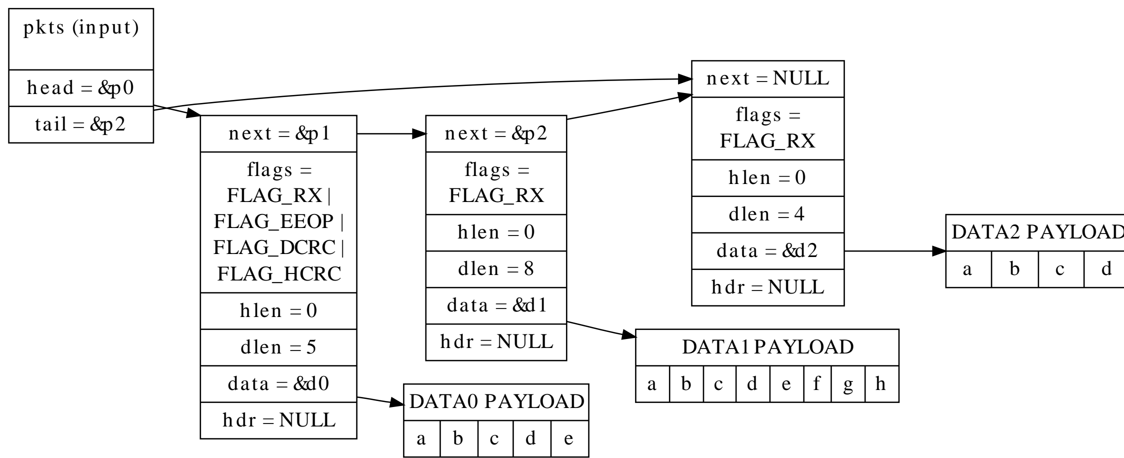

Figure 3.5. RX packet output from `grspw_rx_dma_recv`

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

Table 3.35. `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

<table>
<tr><td></td><td>The above steps 1 and/or 3 may be skipped by setting <code>opts</code> argument according the description below.
<table>
<tr><td>Bit</td><td>Description</td></tr>
<tr><td>0</td><td>Set to 1 to skip Step 1.</td></tr>
<tr><td>1</td><td>Set to 1 to skip Step 3.</td></tr>
</table></td></tr>
<tr><td>Param</td><td><code>pkts</code> [IN] pointer
A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that <code>head</code> points to the first packet and <code>tail</code> points to the last.

Call this function with <code>pkts</code> set to NULL to avoid step 2. Just doing step 1 and 3 as determined by <code>opts</code> 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 3.30. Note that RXPKT_FLAG_RX of the <code>flags</code> field must not be set.</td></tr>
<tr><td>Param</td><td><code>count</code> [IN] integer
Number of packets in the packet list.</td></tr>
<tr><td>Return</td><td>Status. See return codes below
<table>
<tr><td>Value</td><td>Description</td></tr>
<tr><td>-1</td><td>Error occurred, DMA channel may not be valid.</td></tr>
<tr><td>0</td><td>Successfully added pkts to RX READY/SCHED list.</td></tr>
<tr><td>1</td><td>DMA stopped. No operation.</td></tr>
</table></td></tr>
<tr><td>Notes</td><td>This function performs no operation when the DMA channel is stopped.</td></tr>
</table>

Table 3.36. `grspw_dma_rx_recv` function declaration

<table>
<tr><td>Proto</td><td><code>int grspw_dma_rx_recv(void *c, int opts, struct grspw_list *pkts, int *count)</code></td></tr>
<tr><td>About</td><td>Get received RX packet buffers that has previously been scheduled for reception with <code>grspw_dma_rx_prepare</code>. The packets in the RX RECV queue are moved to the <code>pkts</code> 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:

Move scheduled packets to RECV List (SCHED->RECV).
Move all RECV packet to the callers list (RECV->USER).
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:

grspw_dma_rx_recv(opts=2, &recvlist) (Skip step 3)
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.</td></tr>
<tr><td>Param</td><td><code>c</code> [IN] pointer
DMA channel identifier. Returned from <code>grspw_dma_open</code>.</td></tr>
<tr><td>Param</td><td><code>opts</code> [IN] Integer bit-mask
The above steps 1 and/or 3 may be skipped by setting <code>opts</code> argument according the description below.
<table>
<tr><td>Bit</td><td>Description</td></tr>
</table></td></tr>
</table>

	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 3.30. 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 3.37. `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 3.38. 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 3.39. 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 3.40. `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 3.39. 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 3.39.
Return	None.

Table 3.41. 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 3.42. grspw_dma_config data structure declaration

<table>
<tr><td rowspan="7">flags</td><td colspan="2">RX/TX DMA transmission options See below.</td></tr>
<tr><td>Bits</td><td>Description (prefixed DMAFLAG_ or DMAFLAG2_)</td></tr>
<tr><td>NO_SPILL</td><td>Enable (1) or Disable (0) packet spilling, flow control.</td></tr>
<tr><td>STRIP_ADR</td><td>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.</td></tr>
<tr><td>STRIP_PID</td><td>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.</td></tr>
<tr><td>TXIE</td><td>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.</td></tr>
<tr><td>RXIE</td><td>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.</td></tr>
<tr><td>rxmaxlen</td><td colspan="2">Max packet reception length. Longer packets with will be truncated see RXPKT_FLAG_TRUNK flag in packet structure.</td></tr>
<tr><td>rx_irq_en_cnt</td><td colspan="2">Controls RX interrupt generation. This integer number enable RX DMA IRQ every 'cnt' descriptors.</td></tr>
<tr><td>tx_irq_en_cnt</td><td colspan="2">Controls TX interrupt generation. This integer number enable TX DMA IRQ every 'cnt' descriptors.</td></tr>
</table>

Table 3.43. 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 3.42.
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 3.44. `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 3.42.
Return	None.

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 3.45. Data structures reference

Data structure name	Section
struct grspw_pkt	3.4.3
struct grspw_list	3.2.10
struct grspw_addr_config	3.3.4
struct grspw_hw_sup	3.3.2
struct grspw_core_stats	3.3.8
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 3.46. Device function reference

Prototype	Section
`int grspw_dev_count(void)`	3.3.1

Prototype	Section
void *grspw_open(int dev_no)	3.3.1
void grspw_close(void *d)	3.3.1
void grspw_hw_support(void *d, struct grspw_hw_sup *hw)	3.3.2
void grspw_stats_read(void *d, struct grspw_core_stats *sts)	3.3.8
void grspw_stats_clr(void *d)	3.3.8
void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)	3.3.4, 3.2.7
spw_link_state_t grspw_link_state(void *d)	3.3.3, 3.2.3
void grspw_link_ctrl(void *d, int *options, int *clkdiv)	3.3.3, 3.2.3
unsigned int grspw_link_status(void *d)	3.3.3, 3.2.3
void grspw_link_status_clr(void *d, unsigned int mask)	3.3.3, 3.2.3
void grspw_tc_ctrl(void *d, int *options)	3.3.5, 3.2.4
void grspw_tc_tx(void *d)	3.3.5, 3.2.4
void grspw_tc_isr(void *d, void (*tcisr)(void *data, int tc), void *data)	3.3.5, 3.2.4
void grspw_tc_time(void *d, int *time)	3.3.5, 3.2.4
int grspw_rmap_ctrl(void *d, int *options, int *dstkey)	3.3.7, 3.2.5
void grspw_rmap_support(void *d, char *rmap, char *rmap_crc)	3.3.7, 3.2.5, 3.3.2
int grspw_port_ctrl(void *d, int *port)	3.3.6, 3.2.6
int grspw_port_count(void *d)	3.3.6, 3.2.6, 3.3.2
int grspw_port_active(void *d)	3.3.6, 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 3.47. DMA channel function reference

Prototype	Section
void *grspw_dma_open(void *d, int chan_no)	3.2.1, 3.4.1, 3.3.1

Prototype	Section
void grspw_dma_close(void *c)	3.2.1, 3.4.1, 3.3.1
int grspw_dma_start(void *c)	3.4.2, 3.2.13
void grspw_dma_stop(void *c)	3.4.2, 3.2.13
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 *pk-ts, int count)	3.4.6, 3.2.1
void grspw_dma_rx_count(void *c, int *ready, int *sched, int *recv)	3.4.7, 3.2.10.1
int grspw_dma_rx_wait(void *c, int recv_cnt, int op, int ready_cnt, int timeout)	3.4.4, 3.2.11
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 *pk-ts, int *count)	3.4.5, 3.2.1
void grspw_dma_tx_count(void *c, int *send, int *sched, int *sent)	3.4.7, 3.2.10.1
int grspw_dma_tx_wait(void *c, int send_cnt, int op, int sent_cnt, int timeout)	3.4.4, 3.2.11
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

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 should include the "GRSPW Kernel Driver header" file. All files are relative the base of the driver package.

Table 4.1. SpaceWire Router driver sources

Location	Description
`spw/grspw_router.c`	SpaceWire Router APB Registers Driver
`include/linux/grlib/grspw_router.h`	SpaceWire Router APB Registers header

4.1.2. Using the driver

Applications wanting to access SpW Router registers from user-space should include the Router driver header file.

Each SpW Router core is accessed using a single major/minor number. The Major/Minor numbers are determined by the driver package configuration, 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 `spwrouter_custom_config.c`.

4.2. Control Interface

4.2.1. 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 the header file available in the GRLIB driver package.

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

In the table below all currently supported `ioctl` commands and their argument is 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: 32-bit simple Argument, no data transferred between kernel/user.
- None: Argument ignored.

Table 4.2. `ioctl` commands supported by the GRSPW Kernel driver.

Command	Data Direction	Argument Type	Description
HWINFO	Output	struct grspw_hw_info *	Copy hardware configuration of the router core, such as number of SpaceWire ports, number DMA port, number of FIFO port, etc.

Command	Data Direction	Argument Type	Description
CFG_SET	Input	struct router_config *	Configure the router by writing the configuration bit of the Control/Status register, setting the Instance ID, Start up Clock Divisor, Timer prescaler and the timer reload registers.
CFG_GET	Output	struct router_config *	Reads the current router configuration into the user specified memory area.
ROUTES_SET	Input	struct router_routes *	Configure the 224 words long router table.
ROUTES_GET	Output	struct router_routes *	Copy the current 224 words long router table to user provided buffer.
PS_SET	Input	struct router_ps *	Configure the port setup registers according to user buffer.
PS_GET	Output	struct router_ps *	Copy the current port setup registers to user buffer.
WE_SET	Argument	*int*	If the argument's bit zero is one then the WE bit in the configuration write enable register is set, otherwise it is cleared. This enabled the user to write protect the current configuration.
PORT	Input/Output	struct router_port *	Write and/or Read (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 ROUTER_PORTFLG_*. The `port` field selects which port is to be written/read.
CFGSTS_SET	Argument	*unsigned int*	Writes the Config/Status register.
CFGSTS_GET	Output	*unsigned int **	Copies the current value of the Config/Status register to the user provided buffer.
TC_GET	Output	*unsigned int **	Copies the current value of the Time-code register to the user provided buffer.

5. MAPLIB Device Memory Driver

5.1. Introduction

This section describes the Linux MAPLIB kernel driver. It provides user space applications with a possibility to memory map a configurable number 128 KBytes blocks of memory to user space. The memory is direct memory access (DMA) capable and can therefore be used in other GRLIB drivers which implements user provided device memory buffers. In order for memory to be DMA capable a number of things must be satisfied, for example that memory is linear with one DMA operation and that the cache is handled correctly. Currently the MAPLIB driver memory maps with the memory management unit (MMU) cacheable bit set, this means that the driver will not work for systems with lacks data cache snooping (unless flush is performed by the using driver).

Memory is mapped and unmapped to user space using the `mmap, mmap2` and `unmap` functions. The functions are described in the man-page of respective function.

The driver provides a secure way of mapping, calling the using drivers when the memory is unmapped or changed in any other way. The using driver should then stop all DMA operation to that memory area and report an error to the user.

The driver's main intention is to let other drivers more easily implement zero-copy between user space and kernel space, both between the the same device instance and between different device instance and even between device instances of different drivers. For example a SpaceWire packet received on GRSPW[0] may be sent on GRSPW[2] without copying the actual data, or for example parts of a SpaceWire packet received on GRSPW[1] may be sent to ground using the driver for GRTM[0] device.

Blocks of 128KBytes are allocated within the Linux Kernel in low memory. The amount of memory allocated is configurable through the standard UNIX `ioctl` interface of the MAPLIB driver.

5.1.1. Sources

The MAPLIB driver sources are provided under the GPL license, they are available in the GRLIB driver package as described in the table below. Applications should include the "MAPLIB Driver header" file. All files are relative the base of the driver package.

Table 5.1. MAPLIB driver sources

Location	Description
`misc/maplib.c`	Device memory library
`include/linux/grlib/maplib.h`	Device memory library header

5.1.2. Using the driver

Applications wanting to access DMA capable memory from user space using the MAPLIB device driver should include the MAPLIB driver header file. The amount of memory requested

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

Each MAPLIB driver is accessed using a major/minor number. The driver has a build-time configurable number of "memory pools" (device nodes). The Major/Minor numbers are determined by the driver package configuration, see Section 1.5.

One can list the current address space mappings of a process by concatenating the `/proc/PROCESS_NUMBER/maps`. Reading the file after the mapping processes is completed will reveal the mapping range and access permissions and so on.

5.1.3. Examples

Within the GRLIB driver package there are (at the time of writing) two examples, one example using the MAPLIB driver only `teset_maplib.c`, and one SpaceWire example which demonstrates how the MAPLIB can be used in a real application using the GRSPW driver.

5.2. Control Interface

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

In the table below all currently supported `ioctl` commands and their argument is listed. All MAPLIB commands starts with MAPLIB_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: 32-bit simple Argument, no data transferred between kernel/user.
- None: Argument ignored.

Table 5.2. `ioctl` commands supported by the MAPLIB Kernel driver.

Command	Data Direction	Argument Type	Description
SETUP	Input	struct maplib_setup *	Configure Memory MAP Library, and allocate all need memory, all previous (if any) memory mapped pages must be unmapped otherwise and error will occur and errno set to EINVAL.
MMAPINFO	Output	struct maplib_mmap_info *	Get Current MMAP Info from Driver, this tells the user how to memory map the memory into user space. It tells the user how many blocks, their size and the offset into the MAPLIB device memory `mmap()` should try to map from.

5.3. Mapping Interface

Once the driver has been configured using the control interface the memory must be mapped to the user space process address space before any other driver or the application itself can start using the DMA capable memory. Once the memory is used by a device driver the driver will be signaled if munmap() or close() is called upon the MAPLIB memory/device, it will also be signaled if a process is terminated.

The memory must be mapped in one `mmap()` call, creating one linear memory mapping in user space. However in physical address space the memory is linear in blocks of 128KBytes.

The MMAPINFO command reveals how large and at what offset the device memory is located within the MAPLIB device, after it has been configured using SETUP. Below is an example how to memory map.

```
struct maplib_mmap_info mapi;
unsigned int start, end;
int fd;

fd = open("/dev/maplib0", O_RDWR);
if ( fd < 0 ) {
        printf("Failed to open MMAPLib\n");
        return -1;
}

/* CONFIGURE MAPLIB HERE USING MAPLIB_IOCTL_SETUP */

/* Get MMAP information calculated by driver */
if ( ioctl(fd, MAPLIB_IOCTL_MMAPINFO, &mapi) ) {
        printf("Failed to get MMAPINFO, errno: %d\n", errno);
        return -1;
}

/* Map all SpaceWire Packet Buffers */
start = mapi->buf_offset;
end = mapi->buf_offset + mapi->buf_length;

/* Memory MAP driver's Buffers READ-and-WRITE  */
adr = mmap(NULL, mapi.buf_length, PROT_READ|PROT_WRITE, MAP_SHARED,
           fd, start);
if ( (unsigned int)adr == 0xffffffff ) {
        printf("MMAP Bufs Failed: %p, errno %d, %x\n", adr, errno, mapi->buf_length);
        return -1;
```

```
}
```

6. GRSPFI SpaceFibre Driver

6.1. 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 7) for GRSPFI device control and DMA transfer and it uses the memory map driver (MAPLIB described in Chapter 5) for allocating physically continuous device memory (DMA memory) for 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/grspfiN` (e.g. `/dev/grspfi0`).
- One *Virtual Channel device* per VC discovered at open time: `/dev/grspfiN_V` (e.g. `/dev/grspfi0_0`, `/dev/grspfi0_1`). VC devices are created dynamically when the control device is opened and destroyed when it is closed.

All user interaction is performed via `ioctl`. Only one process may hold any given device node open at a time (`-EBUSY` is returned on concurrent opens).

6.1.1. 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 should include the GRSPFI character driver header file. All files are relative the base of the driver package.

Table 6.1. GRSPFI driver sources

Location	Description
spf/grspfi_drv.c	GRSPFI kernel library
spf/grspfi.c	GRSPFI character device (user-space interface)
misc/maplib.c	Device memory library
include/linux/grlib/grspfi_drv.h	GRSPFI kernel library header
include/linux/grlib/grspfi.h	GRSPFI character driver header
include/linux/grlib/maplib.h	Device memory library header

6.1.2. Using the driver

Applications wanting to access GRSPFI devices from user-space should include the GRSPFI character driver header file (`include/linux/grlib/grspfi.h`). Since the driver uses the MAPLIB driver for DMA memory management, the application is also responsible for setting up a maplib memory pool and must include the MAPLIB header file.

Data buffers passed to the TX/RX ioctls must reside in a memory pool registered with the `maplib` subsystem. The `GRSPFI_IOCTL_CONFIG` ioctl selects which `maplib` pool to use. The driver automatically translates user-space virtual addresses to physical addresses. If user-space unmaps memory while a DMA operation is in progress, all DMA channels are closed immediately to prevent memory corruption.

6.1.3. 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 to call the driver's ioctl interface.

6.2. ioctl Reference

6.2.1. 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.

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

`-ENODEV`

No private data associated with the file descriptor (internal error).

`-ENOIOCTLCMD`

Unrecognised ioctl command number.

A typical usage sequence is:

1. Open `/dev/grspfiN`.
2. Call `GRSPFI_IOCTL_GET_CAPABILITIES` to discover hardware features.
3. Call `GRSPFI_IOCTL_CONFIG` to select the maplib pool and start the link.
4. Open `/dev/grspfiN_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.

6.2.2. GRSPFI_IOCTL_GET_CAPABILITIES

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

Copies the hardware capability structure 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, and optional hardware features (RMAP, CCSDS CRC, etc.).

Relevant fields in struct grspfi_cap:

Table 6.2. Fields in struct grspfi_cap

Field	Description
gensup.num_vc	Number of Virtual Channels present in hardware.
dmasup.num_dma	Number of DMA channels.
dmasup.rmap_tar	Non-zero if hardware RMAP target is supported.

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

6.2.3. GRSPFI_IOCTL_CONFIG

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

Configures the SpaceFibre link and selects the maplib memory pool. Must be called on the control device before any data transfer can occur.

Fields in struct grspfi_config_t:

Table 6.3. Fields in struct grspfi_config_t

Field	Description
maplib_pool_idx	Index of the maplib memory pool to use for DMA address translation. The pool must already be registered with the maplib subsystem.

Field	Description
address	8-bit default node address.
address_mask	8-bit address mask.

Errors:

-EFAULT

The user-space buffer pointer is invalid.

-EINVAL

maplib_pool_idx is not a valid pool.

-EPERM

Registration with the maplib driver failed.

6.2.4. GRSPFI_IOCTL_VC_CONFIG

Direction	Write (user → kernel)
Argument type	struct grspfi_vc_config_t
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*).

Fields in struct grspfi_vc_config_t:

Table 6.4. Fields in struct grspfi_vc_config_t

Field	Description
enable	1 to open the VC with the parameters below; 0 to close it.
max_rx_len	Maximum incoming packet length in bytes.
prio	Transmit priority (0 = highest).
bandwidth	Bandwidth allocation in percent (1–100).
timeslots	64-bit bitmask of transmit time-slots.
fifo_en	Enable FIFO routing for received packets.
fifo_nr	External FIFO index (used when *fifo_en* is set).
rmap_en	Enable hardware RMAP target processing.
rmap_key	RMAP destination key.
address_filter_en	Enable VC-level address acceptance filtering.
address	VC address for filtering (8-bit).
address_mask	VC address mask (8-bit).

Errors:

-EFAULT

The user-space buffer pointer is invalid.

-EINVAL

Called on the control device, or the VC index embedded in the device node is out of range.

6.2.5. GRSPFI_IOCTL_BC_CONFIG

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

Opens or closes the Broadcast Channel.

Fields in struct grspfi_bc_config_t:

Table 6.5. Fields in struct grspfi_bc_config_t

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

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

Called on a VC device, or invalid parameters.

6.2.6. GRSPFI_IOCTL_TIME_CONFIG

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

Configures time-slot generation.

Fields in struct grspfi_time_config_t:

Table 6.6. Fields in struct grspfi_time_config_t

Field	Description
`int_gen`	`true` to enable internal time-slot generation; `false` to accept external time-slot input.
`nom_len`	Nominal time-slot length in link characters (non-zero).

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

`nom_len` is zero or out of hardware range.

6.2.7. GRSPFI_IOCTL_TIME_SET

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

Sets the 6-bit time value in the TS hardware register.

Fields in struct grspfi_time_val_t:

Table 6.7. Fields in struct grspfi_time_val_t

Field	Description
`time`	6-bit time value to write (bits 5:0 of the TS register).

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

time exceeds 6 bits.

6.2.8. GRSPFI_IOCTL_TIME_GET

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

Reads the current 6-bit time value from the TS hardware register and returns it in *grspfi_time_val_t.time*.

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

6.2.9. GRSPFI_IOCTL_DMA_CONFIG

Direction	Write (user → kernel)
Argument type	struct grspfi_dma_config_t
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.

Fields in struct grspfi_dma_config_t:

Table 6.8. Fields in struct grspfi_dma_config_t

Field	Description
enable	1 to open the DMA channel; 0 to close it.
channel	DMA channel index (0 to *cap.dmasup.num_dma* – 1).
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.
bc_enable	Non-zero to also route Broadcast Channel traffic through this DMA channel. The BC must be open.
txrx	Direction bitmask: combine `GRSPFI_DMA_TE` (TX enable) and/or `GRSPFI_DMA_RE` (RX enable).

Errors:

`-EFAULT`

The user-space buffer pointer is invalid.

`-EINVAL`

Called on a VC device, invalid channel index, or referenced VCs not open.

6.2.10. GRSPFI_IOCTL_RX_PREPARE

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

Submits an empty receive buffer so the hardware can DMA incoming packet data into it. The user-space virtual address in *grspfi_pkt.data* is translated to a physical address using the maplib pool configured by `GRSPFI_IOCTL_CONFIG`.

Relevant fields in struct grspfi_pkt:

Table 6.9. Fields in struct grspfi_pkt (RX_PREPARE)

Field	Description
data	User-space virtual address of the receive buffer. Must lie within the registered maplib pool. The buffer must be at least *max_rx_len* bytes (from `GRSPFI_IOCTL_VC_CONFIG`) to avoid overflow.
flags	Optional flags written to the RX descriptor; see descriptor flag definitions for details (e.g. interrupt-enable bit).

Errors:

`-EINVAL`

Called on the control device or VC index out of range.

`-EFAULT`

Copy from user failed.

`-EPERM`

The supplied address is not in the maplib pool.

`-EAGAIN`

The RX descriptor ring is full.

6.2.11. GRSPFI_IOCTL_RX_RECEIVE

Direction	Read (kernel → user)
Argument type	struct grspfi_pkt
Valid on	VC device only

Dequeues the oldest completed receive packet. The returned structure reports how many bytes were received and status flags from the hardware descriptor. The *data* field is restored to the original user-space virtual address supplied to `GRSPFI_IOCTL_RX_PREPARE`.

Fields populated by the kernel in struct grspfi_pkt:

Table 6.10. Fields in struct grspfi_pkt (RX_RECEIVE)

Field	Description
data	Original user-space virtual address of the buffer.
data_length	Number of bytes received.
flags	Status flags from the RX descriptor (e.g. data-CRC present, header-CRC present, truncated, end-of-packet error).

Errors:

`-EINVAL`

Called on the control device or VC index out of range.

`-EAGAIN`

No completed packet is available.

`-EFAULT`

Copy to user failed.

6.2.12. GRSPFI_IOCTL_TX_SEND

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

Submits a packet for transmission. Both `data` and the optional `header` user-space virtual addresses are translated to physical addresses via the maplib pool. The TX descriptor is written to the ring and the hardware is notified.

Relevant fields in struct grspfi_pkt:

Table 6.11. Fields in struct grspfi_pkt (TX_SEND)

Field	Description
`data`	User-space virtual address of the packet payload.
`data_length`	Length of the payload in bytes.
`header`	User-space virtual address of an optional packet header (may be 0).
`header_length`	Length of the header in bytes (0 if no header).
`flags`	TX descriptor control flags, e.g. CRC type selection, data/header CRC enable, non-CRC length, and interrupt-enable.

Errors:

`-EINVAL`

Called on the control device or VC index out of range.

`-EFAULT`

Copy from user failed.

`-EPERM`

Data or header address not found in the maplib pool.

`-EAGAIN`

The TX descriptor ring is full.

6.2.13. GRSPFI_IOCTL_TX_RECLAIM

Direction	Read (kernel → user)
Argument type	struct grspfi_pkt
Valid on	VC device only

Reclaims a completed TX descriptor. The user-space virtual addresses for both the payload and header are restored into `data` and `header` respectively, allowing the caller to free or reuse those buffers.

Errors:

`-EINVAL`

Called on the control device, or no in-flight TX packets exist.

`-EAGAIN`

The oldest in-flight TX descriptor has not yet been transmitted.

`-EFAULT`

Copy to user failed.

6.2.14. GRSPFI_IOCTL_BC_SEND

Direction	Write (user → kernel)

Argument type	struct grspfi_bc_pkt
Valid on	Control device (BC must be open)

Queues a Broadcast Channel packet for transmission.

Fields in struct grspfi_bc_pkt:

Table 6.12. Fields in struct grspfi_bc_pkt

Field	Description
flags	Broadcast flags (e.g. data-link flag, last-timeslot flag).
type	Broadcast message type (8-bit).
data	64-bit broadcast data payload.

Errors:

`-EFAULT`

Copy from user failed.

`-EAGAIN`

The BC TX buffer is full.

6.2.15. GRSPFI_IOCTL_BC_RECEIVE

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

Dequeues the oldest received Broadcast Channel packet. All fields of struct grspfi_bc_pkt are populated from the hardware receive buffer.

Errors:

`-EFAULT`

Copy to user failed.

`-EAGAIN`

No received BC packet is available.

6.2.16. GRSPFI_IOCTL_TX_WAIT

Direction	None
Argument	Unused
Valid on	VC device only

Blocks the calling process until at least one TX packet becomes reclaimable (i.e., has been transmitted by the hardware). Implemented using an interruptible wait queue that is woken up by the VC TX interrupt handler.

Errors:

`-ERESTARTSYS`

The wait was interrupted by a signal.

6.2.17. GRSPFI_IOCTL_RX_WAIT

Direction	None
Argument	Unused
Valid on	VC device only

Blocks the calling process until at least one RX packet has been received and is available via `GRSPFI_IOCTL_RX_RECEIVE`. Woken by the VC RX interrupt handler.

Errors:

`-ERESTARTSYS`

The wait was interrupted by a signal.

6.2.18. GRSPFI_IOCTL_BCRX_WAIT

Direction	None
Argument	Unused
Valid on	Control device (BC must be open)

Blocks the calling process until at least one Broadcast Channel packet has been received and is available via `GRSPFI_IOCTL_BC_RECEIVE`. Woken by the device-level BC-RX interrupt handler.

Errors:

`-ERESTARTSYS`

The wait was interrupted by a signal.

7. GRSPFI Kernel Library Driver

7.1. 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 6.

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.

7.1.1. Driver sources

The driver sources and header files are listed in Section 6.1.1.

7.1.2. Examples

The GRSPFI SpaceFibre character device driver is an example of how the GRSPFI Kernel Library driver can be used. See Section 6.1.3.

7.2. Software design overview

7.2.1. 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 6. 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 7.2.3.
- Device lifecycle (open, close, capabilities), see Section 7.2.4.
- Link layer control, see Section 7.2.5.
- Time-slot management, see Section 7.2.6.
- Virtual Channel API, see Section 7.2.7.
- Broadcast Channel API, see Section 7.2.8.
- DMA Channel API, see Section 7.2.9.
- Status registers, see Section 7.2.10.
- Interrupt management, see Section 7.2.11.

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.

7.2.2. 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 7.1. 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.

7.2.3. Module Registration

7.2.3.1. 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 7.2. 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.

7.2.4. Device Lifecycle

7.2.4.1. 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 7.3. 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.

7.2.4.2. grspfi_close

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

Closes a previously opened device, disabling SpaceFibre output and resetting the link. All Virtual Channels and DMA channels should be closed before calling this function.

Table 7.4. Parameters of `grspfi_close()`

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

7.2.4.3. 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 7.5. Parameters of `grspfi_cap_get()`

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

7.2.5. Link Layer Control

7.2.5.1. 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 7.6. 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.

7.2.5.2. grspfi_link_close

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

Stops the SpaceFibre link.

Table 7.7. 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).

7.2.5.3. 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.

7.2.5.4. 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).

7.2.6. Time-slot Management

7.2.6.1. grspfi_time_conf

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

Configures the time-slot mechanism.

Table 7.8. 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.

7.2.6.2. 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.

7.2.6.3. grspfi_time_get

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

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

7.2.7. Virtual Channel API

7.2.7.1. 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 7.9. Parameters of `grspfi_vc_open( )`

Parameter	Description
dev	Device handle.
virtual_channel_number	VC index (0 to *max_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 negative error code from descriptor allocation failure.

7.2.7.2. grspfi_vc_close

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

Closes a Virtual Channel, clears its hardware registers, frees DMA descriptor memory, and removes it from the device's VC list. The associated DMA channel must be closed first.

7.2.7.3. 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).

7.2.7.4. 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.

7.2.7.5. VC Data Transfer

7.2.7.5.1. 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.

7.2.7.5.2. 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. Returns `-EAGAIN` if no completed descriptor is available (EN bit still set, or ring empty).

7.2.7.5.3. 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.

7.2.7.5.4. 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.

7.2.7.6. 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.

```
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.

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

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

7.2.7.7. 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()`.

7.2.8. Broadcast Channel API

7.2.8.1. 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 7.10. 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.

7.2.8.2. grspfi_bc_close

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

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

7.2.8.3. 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.

7.2.8.4. 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.

7.2.9. DMA Channel API

7.2.9.1. 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 7.11. Parameters of `grspfi_dma_open()`

Parameter	Description
dev	Device handle.
dma_number	DMA channel index (0 to *max_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, or a negative error code on failure (`-EINVAL`, `-EBUSY`).

7.2.9.2. 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.

7.2.9.3. 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 this DMA channel is mapped to the Broadcast Channel (i.e., its index matches BCMAP).

7.2.10. Status Registers

7.2.10.1. 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*.

7.2.10.2. 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*.

7.2.10.3. 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*.

7.2.11. 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.

7.2.11.1. 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).

7.2.11.2. 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).

7.2.11.3. 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.

7.3. Device tree

7.3.1. 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 user-space module (`grspfi`) binds to `gaisler,grspfiu`.

Below is an example device tree description of two GRSPFI devices and their `gaisler,grspfi` binding. The character device driver can be added using the `gaisler,grspfiu` binding.

```
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