GRLIB Linux Drivers User's Manual

GRLIB Linux drivers manual

Written by Daniel Hellstrom

LINDRV
Version 1.0.0
November 2010

Kungsgatan 12
413 11 Gothenburg
Sweden

tel +46 31 7758650
fax +46 31 421407
www.aeroflex.com/gaisler

GRLIB Linux Drivers User's Manual

Daniel Hellstrom

Table of Contents

1. Introduction

The purpose of the GRLIB Driver package is for Aeroflex Gaisler to provide Linux drivers for GRLIB cores that does not really benefit from being part of the official kernel tree or for other reasons no 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 andlink 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:

```
[linux-2.6/]$ 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:

```
[grlib_drivers/]$ make KERNELDIR=/path/to/kernel/linux-2.6/sources
```

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/.

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.

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 Aeroflex 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.

1.3. Installing

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

1.4. Device node numbering

The GRLIB drivers are assigned major numbers from the "LOCAL/EXPERIMENTAL USE" series defined in the `linux-2.6/Documentation/devices.txt`. The driver major number and device minor numbers assigned is determined by the `include/linux/grlib/devno.h` header file in the GRLIB driver package.

Device nodes are normally created somewhere in `/dev` in the local file system, nodes can be created with the `mknod` utility.

2. GRSPW2 SpaceWire Driver

2.1. Introduction

This section describes the Linux GRSPW kernel driver. It provides user space applications with a SpaceWire packet interface. The driver is implemented using the GRSPW Kernel library for GRSPW device control and DMA transfer and it uses the memory map driver (MAPLIB) for ensuring that device memory (DMA memory) has been setup properly. The driver supports the GRSPW2 and the DMA interface of the Aeroflex Gaisler SpaceWire Router.

By splitting the GRSPW driver 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 underlaying 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 1.4.

2.1.3. Examples

Within the GRLIB driver package there is a user space example of how this driver can be used.

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 and starting/stopping DMA channels. 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 has "flow-control" packets will buffer up when the link state goes from run-state to any other state. The user is expected to handle the link state.

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 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 data 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 *	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

Command	**Data Direction**	**Argument Type**	**Description**
			only be 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.
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 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).

Command	Data Direction	Argument Type	Description
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.

2.3. Packet Transfer Interface

The packet transfer interface is used to send and receive SpaceWire packets on DMA channels. The GRSPW core is configurable how many DMA channels it has, a core may have from one up to four DMA channels. 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 open will result in an error and errno will be set to EBUSY.

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 RX packet buffers.
- RX(RECEIVE), get received SpaceWire packets, the packets are placed in previously prepared packet buffers.
- TX(SEND), send one or multiple packets by giving
- TX(RECLAIM), get used/sent 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.

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.

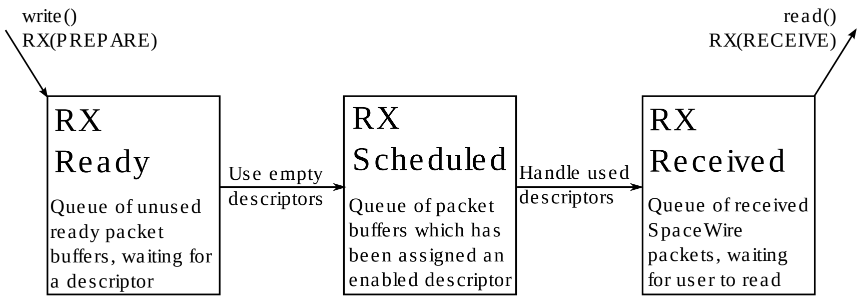
write()
RX(PREPARE)
read()
RX(RECEIVE)
RX Ready
Queue of unused ready packet buffers, waiting for a descriptor
Use empty descriptors
RX Scheduled
Queue of packet buffers which has been assigned an enabled descriptor
Handle used descriptors
RX Received
Queue of received SpaceWire packets, waiting for user to read

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.
`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.

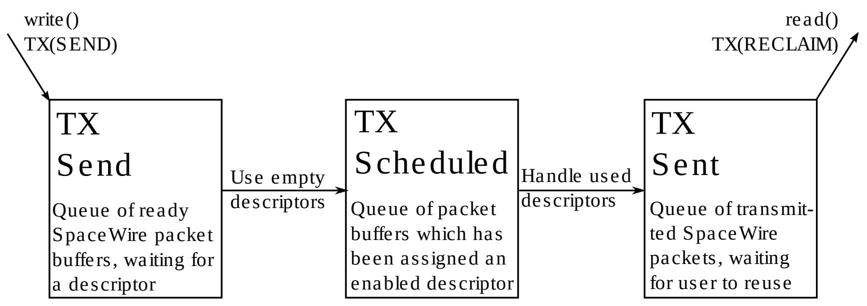
write()
TX(SEND)
TX
Send
Queue of ready SpaceWire packet buffers, waiting for a descriptor
Use empty descriptors
TX
Scheduled
Queue of packet buffers which has been assigned an enabled descriptor
Handle used descriptors
TX
Sent
Queue of transmitted SpaceWire packets, waiting for user to reuse
read()
TX(RECLAIM)

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.
`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.

3. SpaceWire Router APB Register Driver

3.1. Introduction

This section describes the Linux Aeroflex 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.

3.1.1. 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 3.1. 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

3.1.2. 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 1.4.

3.1.3. 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`.

3.2. Control Interface

3.2.1. 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 3.2. `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.

4. MAPLIB Device Memory Driver

4.1. 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.

4.1.1. 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 4.1. MAPLIB driver sources

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

4.1.2. 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 1.4.

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.

4.1.3. 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.

4.2. 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 4.2. `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.

4.3. 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;
}
```

5. Support

For Support, contact the Aeroflex Gaisler support team at support@gaisler.com.