Zephyr

Gaisler Zephyr distribution User's Manual

USER MANUAL

RELEASED DECEMBER 2023

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

Table of Contents

1. Introduction

Frontgrade Gaisler Zephyr distribution provides software support for processors and SoC systems in addition to what is publically available at the *Zephyr project* [https://www.zephyrproject.org].

In summary the package includes:

- Zephyr kernel improvements
  - Based on official zephyr-v3.5.0
  - Kernel patches (Section 2.1)
- GRLIB device drivers (Chapter 3)
- Examples
- Documentation

Specific support for the following SoC components are included:

- GR716A component
- GR716B component

The purpose of this software distribution is to provide a common Zephyr RTOS kernel environment adapted for SPARC LEON processors. This Zephyr support package is based on the Zephyr release zephyr-v3.5.0, it adds Frontgrade Gaisler kernel patches, provides device drivers which has not been upstreamed, and demonstrates usage on LEON with examples.

Currently only the LEON SPARC processor family is supported by this distribution. Please see the Frontgrade Gaisler website for updates on NOEL-V support in Zephyr.

1.1. Installing Zephyr

Zephyr development environment and documentation packages are freely available via the Zephyr Project web site and Git repositories. Application development information and the kernel reference is available online via the *Zephyr Project Documentation* [https://docs.zephyrproject.org/latest/index.html]. Examples and demos for Zephyr are also available.

To get started with Zephyr on LEON, the official *Getting started Guide* [https://docs.zephyrproject.org/latest/getting_started/index.html], is the recommended starting point. After following the guide, the Zephyr environment will reside in `${HOME}/zephyrproject` with the kernel source tree in `${HOME}/zephyrproject/zephyr`.

The official guide describes how to install the required host tools, the SDK (including compiler tools), and how to retrieve the Zephyr kernel and module source code using the **west** tool. By default, the guide will check out the Zephyr master branch. Section 2.1 below describes how to switch to the recommended base commit and apply the provided patches. Zephyr SDK version 0.16.4 has been tested together with zephyr-gaisler-1.0.0.

The examples in the Getting started Guide can be used with the GR716A-MINI board, by using the CMake argument `-DBOARD=gr716a_mini`. The output binary in `zephyr/zephyr.elf` can be loaded and run directly with **tsim-gr716** or **GRMON**. Further information about the Zephyr GR716A-MINI integration is available at: https://docs.zephyrproject.org/latest/boards/sparc/gr716a_mini/doc/index.html.

1.1.1. Extracting the archive

It is assumed that the steps in the Zephyr official *Getting started Guide* (above) have been performed before proceeding with the following.

After obtaining the compressed tar file for the Frontgrade Gaisler Zephyr distribution, uncompress and untar it to a suitable location. The distribution has been prepared to reside in the `/opt/zephyr-gaisler-1.0.0` directory, but can be installed in any location. It can be installed with the following commands:

```
$ mkdir -p /opt
$ cd /opt
$ tar -xf /path/to/zephyr-gaisler-1.0.0.tar.bz2
```

1.1.2. Installing kernel improvements

See Section 2.1.1 for instruction on how to apply the kernel patches.

1.1.3. Installing GRLIB drivers into Zephyr

See Section 3.2 for instruction on how enable the GRLIB driver in Zephyr.

1.2. Archive content

The extracted distribution archive contains the following directories and files:

`patch`	Kernel patches
`grlib-drivers`	Zephyr module with device drivers
`examples`	Example applications
`zephyr-gaisler-1.0.0.pdf`	This document

2. Zephyr kernel

Zephyr is an open-source Real-Time Operating System (RTOS) with device drivers and a cross-compilation toolchain that can be used with GRLIB System On Chip (SoC) processor designs. The *Zephyr Project* [https://www.zephyrproject.org] provides the software source code releases, documentation, forums and other resources available to the Zephyr Community.

An overview of the the Zephyr support for GRLIB processors can be found on the Frontgrade Gaisler website [https://gaisler.com/index.php/products/operating-systems/zephyr].

2.1. Kernel patches

Patches for the Zephyr kernel tree are provided in the directory named `patch`, and should be applied on top of the Zephyr tag `zephyr-v3.5.0`. These patches add improvements related to the SPARC architecture and the LEON3 SOC:s which reside in the kernel. For example device drivers and kernel improvements which were not part of the upstream Zephyr repository at the time of the release tag. Note that the patch set provided by this distribution may change between Zephyr release versions because they may become added to the official kernel tree between releases.

Summary of patches:

- SOC support for the GR716B component
- Board description for the GR716B-MINI board, compatible with TSIM3
- Extended device tree for GR716A
- Support for SPARC V8E single-vector trapping (SVT). Enabled by default on GR716A and GR716B
- Device driver for the GRLIB GRGPIO GPIO controller, using the Zephyr GPIO API. Enabled for GR716A and GR716B.
- Device driver for the GRLIB SPIMCTRL SPI master controller, using the Zephyr SPI API. Enabled for SPIMCTRL in GR716A and GR716B.
- Device driver for the GR716A ADC controllers, using the Zephyr ADC API. Allows using the 8 ADC controllers in GR716A.

2.1.1. Applying the patches

It is assumed that Zephyr is installed according to the Zephyr official documentation. To apply the patches, issue:

```
$ cd $HOME/zephyrproject/zephyr
$ git branch gaisler-3.5.0 zephyr-v3.5.0
$ git checkout gaisler-3.5.0
$ git am /path/to/the/patch/dir/*.patch
$ west update
```

The command **west update** is needed to synchronize third-party modules with any changes made to the file `zephyr/west.yml`.

The patches can be inspected for example with the Git log front end command **gitk**.

3. GRLIB device drivers

The drivers in the Zephyr GRLIB driver module are drivers which have either not been upstreamed (yet), or do not fit naturally in the Zephyr kernel, or share code with BCC bare-metal distribution from Frontgrade Gaisler. In general this is because there is no related API provided by Zephyr for the type interface class (SpaceWire, MIL-STD-1553B, etc.).

Users of the BCC LEON cross-compiler driver library (`libdrv`) will find these Zephyr drivers familiar: the user interface is the same.

3.1. Drivers included in the package

Below is a list of the drivers currently distributed in the GRLIB Zephyr driver module.

Table 3.1. Drivers included in the Zephyr GRLIB driver module

Driver	Kernel configuration (Kconfig) to include driver	Example provided
GR716A pin control	`CONFIG_GRLIB_GR716A_MISC`	Yes
GR716A PLL control	`CONFIG_GRLIB_GR716A_MISC`	Yes
AHBSTAT driver	`CONFIG_GRLIB_AHBSTAT`	Yes
GRLIB clock gating unit driver	`CONFIG_GRLIB_CLKGATE`	Yes
GR1553B driver	`CONFIG_GRLIB_GR1553B`	Yes
GRCAN and GRCANFD driver	`CONFIG_GRLIB_GRCAN`	No
GRLIB GRSPW2 packet driver	`CONFIG_GRLIB_GRSPW`	Yes
GRLIB SpaceWire router	`CONFIG_GRLIB_GRSPWROUTER`	Yes
I2C master driver	`CONFIG_GRLIB_I2CMST`	No
GR716A memory protection unit driver	`CONFIG_GRLIB_GR716A_MEMPROT`	Yes
Memory scrubber (MEMSCRUB) driver	`CONFIG_GRLIB_MEMSCRUB`	Yes
GRLIB SPI (SPICTRL) driver	`CONFIG_GRLIB_SPICTRL`	No

3.2. Enabling the drivers

The extracted `zephyr-gaisler-1.0.0.tar.bz2` contains a Zephyr module consisting of device driver source code and configuration files. To make the drivers available to an application, the application local `CMakeLists.txt` file will need a reference to the `grlib-drivers` path in its CMake variable `EXTRA_ZEPHYR_MODULES`. Assuming the module is installed in the default location, the line to add to `CMakeLists.txt` is:

```
set(EXTRA_ZEPHYR_MODULES /opt/zephyr-gaisler-1.0.0/grlib-drivers)
```

An example on setting `EXTRA_ZEPHYR_MODULES` can be found in `examples/ahb-stat/CMakeLists.txt`.

3.3. Application configuration

A general description on how to configure the Zephyr kernel and subsystems to adapt for a target application is is available in the Zephyr documentation: *Interactive Kconfig interfaces* [https://docs.zephyrproject.org/latest/build/kconfig/menuconfig.html]. That page describes the interactive *menuconfig* and *guiconfig* systems which are referenced in the following.

When the module has been added to `EXTRA_ZEPHYR_MODULES`, the configuration system will be aware of the GRLIB drivers and will make new options available in the Zephyr menuconfig. These new options are available under `"Modules" ---> "grlib-drivers"`. The menuconfig and guiconfig interfaces are useful for exploring the options described by the module Kconfig. However, changes made in the interactive configuration interfaces are stored in the build directory and will be lost if the build directory is manually removed, or when

using **west build --pristine**. If an appliction is designed to use one of the Zephyr device drivers permanently, the preferred route is to update the application `prj.conf` as described below.

A driver can be selected permanently in the application by adding the corresponding configuration option to the application local `prj.conf` file. For example by adding a line with the content `CONFIG_GRLIB_AHBSTAT=y` to enable building the AHBSTAT device driver. See also Table 3.1.

A useful method to determine kernel configuration parameters the application needs is to use the interactive menuconfig action `[D] Save minimal config (advanced)`. That will write the minimum set of application configuration parameters to a file, which can be merged with `prj.conf`.

3.3.1. Example

Below is an example of `prj.conf` and `CMakeLists.txt` for an application using the AHBSTAT device driver.

Example 3.1. A minimal `prj.conf`

```
CONFIG_GRLIB_AHBSTAT=y
```

Example 3.2. A minimal `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.20.0)
set(BOARD gr716a_mini)
set(EXTRA_ZEPHYR_MODULES /opt/zephyr-gaisler-1.0.0/grlib-drivers)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(hello_world)
target_sources(app PRIVATE src/main.c)
```

See also the directory `/opt/zephyr-gaisler-1.0.0/example/ahbstat`.

4. Support

For support contact the support team at support@gaisler.com.

When contacting support, please identify yourself in full, including company affiliation and site name and address. Please identify exactly what product that is used, specifying if it is an IP core (with full name of the library distribution archive file), component, software version, compiler version, operating system version, debug tool version, simulator tool version, board version, etc.

The support service is only for paying customers with a support contract.

Part I. Device drivers reference

The following sections describe the device drivers included in zephyr-gaisler-1.0.0. Each driver is described in a separate chapter.

Examples on how to use the drivers can be found in the `examples` directory.

Table of Contents

5. Driver registration

Device drivers in this library can operate on any number of peripherals (cores) of a specific type. Before operation starts, the drivers must have knowledge of the available peripheral devices. This knowledge is transferred at run-time in a process named *driver registration*.

Drivers in this library rely on static memory allocation and will never call `malloc()` and related functions. This means that memory required by the drivers need to be allocated by the user and communicated to the drivers. This is also performed in the *driver registration* step.

In the rest of this chapter, the APBUART driver will be used as an example on peripheral registration. The same procedures is used for the other drivers.

5.1. Manual registration

Manual registration does not require dynamic memory allocation or AMBA Plug&Play bus scanning. It can be useful for resource constrained systems.

Registration of a peripheral can be performed with the function

```
int apbuart_register(struct apbuart_devcfg *devcfg);
```

which takes a device configuration record as its parameter. For example:

```
#include <drv/apbuart.h>

struct apbuart_devcfg MYDEVCFG0 = {
  .regs = {
    .addr       = 0x80000100,
    .interrupt  = 2,
  },
};

int main(void) {
  struct apbuart_priv *dev;

  apbuart_register(&MYDEVCFG0);
  dev = apbuart_open(0);
  [...]
}
```

It is also possible to register multiple peripherals at once using the function

```
int apbuart_init(struct apbuart_devcfg *devcfgs[]);
```

which takes a `NULL` terminated array as parameter:

```
#include <drv/apbuart.h>

struct apbuart_devcfg MYDEVCFG[] = {
  { .regs = { .addr = 0x80000100, .interrupt = 2, }, },
  { .regs = { .addr = 0x80000200, .interrupt = 3, }, },
};

struct apbuart_devcfg *MYDEVCFGS[] = {
  &MYDEVCFG[0],
  &MYDEVCFG[1],
  NULL,
};

int main(void) {
  struct apbuart_priv *dev;

  apbuart_init(MYDEVCFGS);
  dev = apbuart_open(1);
  [...]
}
```

In addition to specifying register base addresses and interrupt numbers, the above examples also allocate (static) device private data. For more details, see the definition of the different `struct [driver]_devcfg` types.

5.2. System specific device registration tables

Device configuration tables have been prepared for the following systems:

Table 5.1. Device registration tables for manual registration

System	Header files
GR716	`gr716/`

6. GRSPW Packet driver

6.1. Introduction

This section describes the GRSPW packet driver for Zephyr.

It is an advantage to understand the SpaceWire bus/protocols, GRSPW hardware and software driver design when developing using the user interface in Section 6.3 and Section 6.4. The Section 6.2.1 describes the overall software design of the driver.

The driver uses linked lists of packet buffers to receive and transmit SpaceWire packets. The packet driver implements an API which allows efficient custom data buffer handling providing zero-copy ability and multiple DMA channel support. The link control handling has been separated from the DMA handling.

6.1.1. 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
- Link Control
- Port Control
- RMAP Control

6.1.2. Driver sources

The driver sources and definitions are listed in the table below, the path is given relative to the Zephyr source tree `src/libdrv/src/`.

Table 6.1. Source Location

Filename	Description
`include/drv/grspw_pkt.h`	GRSPW user interface definition
`src/grspw/*.c`	GRSPW driver implementation

6.1.3. Driver registration

This driver uses the driver registration mechanism described in Chapter 5.

Table 6.2. Driver registration functions

Registration method	Function
Register one device	`grspw_register()`
Register many devices	`grspw_init()`

6.1.4. Examples

Examples are available in the `src/libdrv/examples/` directory in the Zephyr distribution.

6.1.5. 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.

6.2. Software design overview

6.2.1. Overview

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

- Device interface, see Section 6.3.

- DMA channel interface, see Section 6.4.

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.

6.2.2. 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.

6.2.3. 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.

Function names prefix: `grspw_link_*()`.

6.2.4. 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.

Function names prefix: `grspw_tc_*()`

6.2.5. 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 check summing 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 6.2.7.

Function names prefix: `grspw_rmap_*()`

6.2.6. 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_*()`

6.2.7. 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_*()`

6.2.8. User DMA buffer handling

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

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

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

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

Function names prefix: `grspw_dma_*()`

6.2.8.1. Buffer List help routines

The GRSPW packet driver internally uses linked lists routines. The linked list operations are found in the header file and can be used by the user as well. The user application typically defines its own packet structures having the same layout as struct grspw_pkt in the top and adding custom fields for the application buffer handling as needed. For small implementations however the *pkt_id* field may be enough to implement application buffer handling. The *pkt_id* field is never accessed by the driver, instead is an optional application 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_*()`

6.2.9. Driver DMA buffer handling

The driver represents packets with the struct grspw_pkt packet structure, see Table 6.32. 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.

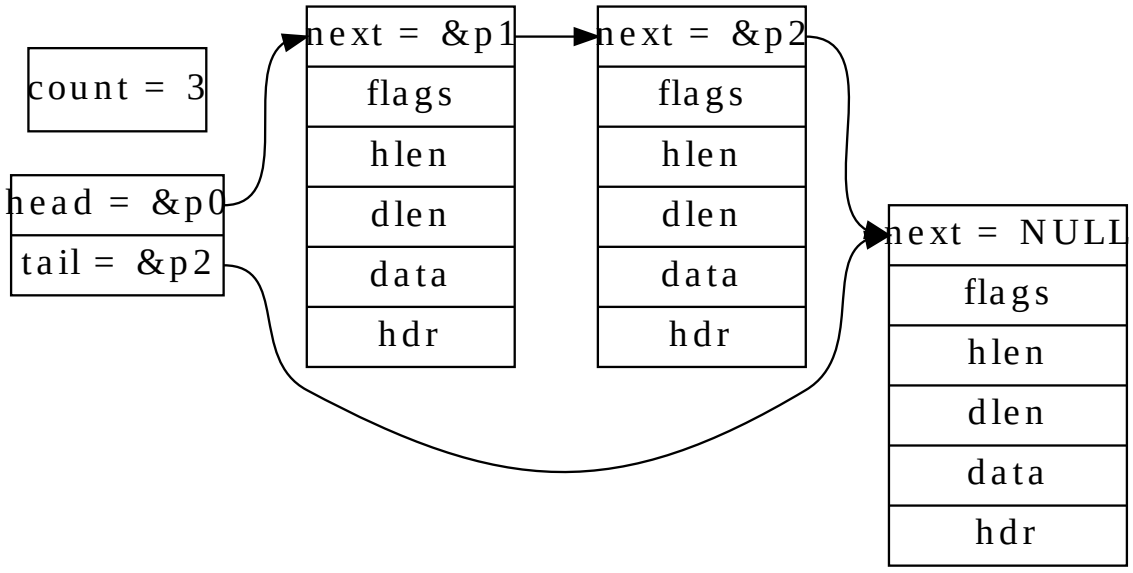

Figure 6.1. Queue example - linked list of three grspw_pkt packets

6.2.9.1. DMA Queues

The driver uses one queue per DMA channel transfer direction, thus two 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 SCHED queue - packets that have been assigned a RX DMA descriptor.
- TX SCHED queue - packets that have been assigned a TX DMA descriptor.

Packet in the SCHED queues always are assigned to a DMA descriptor waiting for hardware to perform RX or TX DMA operations.

The DMA descriptor table has a size limitation imposed by hardware. 64 TX or 128 RX descriptors can be defined for one hardware descriptor table in memory. Naturally this also limits the number of packets that the SCHED queues may contain at any single point in time. It is up to the user to control the input and output to them by queuing and dequeueing from and to private queues.

The current number of packets in respective queue can be read by doing function calls using the DMA API, see Section 6.4.7. The user can for example use this to determine to wait or continue with packet processing.

6.2.9.2. DMA Queue operations

The user can control how the RX SCHED and TX SCHED queues are populated, by providing and removing packet buffers. The user can control how and when packets are moved from RX SCHED and TX SCHED queues

into user provided queues by manually trigger the move by calling reception and transmission routines as described in Section 6.4.6 and Section 6.4.5.

For RX, the packets always flow in one direction from USER RX READY -> RX SCHED -> USER RX RECV. Likewise the TX packets flow USER TX SEND -> TX SCHED -> USER TX SENT. The procedures triggering queue packet moves are listed below and in Figure 6.2 and Figure 6.3. The interface of theses procedures are described in the DMA channel API.

- USER -> RX SCHED – `grspw_dma_rx_prepare`, Section 6.4.6.
- RX SCHED -> USER – `grspw_dma_rx_recv`, Section 6.4.6.
- USER -> TX SCHED queue – `grspw_dma_tx_send`, Section 6.4.5.
- TX SCHED -> USER – `grspw_dma_tx_reclaim`, Section 6.4.5.

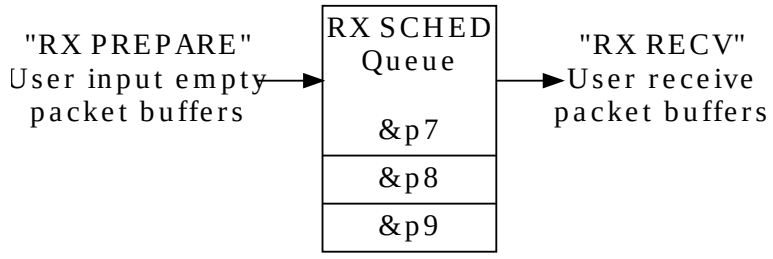

Figure 6.2. RX queue packet flow and operations

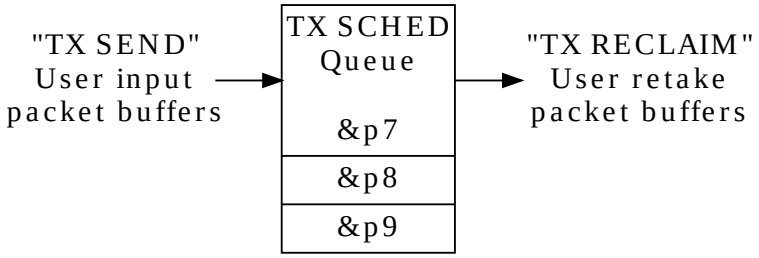

Figure 6.3. TX queue packet flow and operations

Packets which the user has provided to the driver shall be considered owned by the driver until the user takes the packets back again. In particular, the struct grspw_pkt fields should not be accessed by the user while the packet buffers are assigned to the driver.

6.2.10. Polling mode and interrupts

All user DMA operations are non-blocking and the user is thus responsible for processing the DMA descriptor tables at a user defined interval by calling reception and transmit routines of the driver. DMA interrupt generation is controlled individually per packet. It is configured in the packet data structure.

The driver does not contain an interrupt service routine. The user can install an ISR by using the operating system.

6.2.11. 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.

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}()`

6.3. Device Interface

This section covers how the driver can be interfaced to an application to control the GRSPW hardware on device level, such as link state and node addresses.

6.3.1. 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 heap or from a user assigned address and cleared. The descriptor table length is always the maximum 0x400 Bytes for RX and TX.
- Internal data structures are initialized.
- 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 standard output. It then opens, prints the current link settings and closes the first GRSPW device present in the system.

```
int print_spw_link_properties(void)
{
  void *device;
  int count;
  uint32_t linkcfg, clkdiv;

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

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

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

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

Table 6.3. `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 to driver, zero if none.
Notes	The number of GRSPW devices registered to the driver may or may not be equal to the number of devices in the system

Table 6.4. `grspw_open` function declaration

Proto	`void *grspw_open(int dev_no)`	
About	Open a GRSPW device The GRSPW device is identified by index. Index value (*`dev_no`*) must be equal to or greater than zero, and smaller than value returned by `grspw_dev_count`. The returned value is used as input argument to all functions operating on the device. It is not possible to open an already opened device index.	
Param	*`dev_no`* [IN] Integer Device identification number.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Failed to open device. Fails if device is already open, if *`dev_no`* is out of range, or if driver failed to install its ISR.
	Pointer	GRSPW device handle to use as input parameter to all device API functions for the opened device.

Table 6.5. `grspw_close` function declaration

Proto	`int grspw_close(void *d)`	
About	Close a GRSPW device All DMA channels are also stopped and closed automatically, similar to calling `grspw_dma_stop` and `grspw_dma_close` for all channels.	
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.	
Return	int.	
	Value	Description
	DRV_OK	Successfully closed device.
	others	Device closed, but failed to unregister interrupt handler.

6.3.2. 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 device capabilities, parts of the driver 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 6.6. `grspw_hw_support` function declaration

Proto	`void grspw_hw_support(void *d, struct grspw_hw_sup *hw)`
About	Get GRSPW hardware capabilities Write hardware capabilities of GRSPW device to user parameter *`hw`*.
Param	*`d`* [IN] pointer

	Device handle returned by `grspw_open`.
Param	*hw* [OUT] pointer Address to where the driver will write the hardware capabilities. Pointer must to memory and be valid.
Return	None.

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 {
        int8_t  rmap;           /* If RMAP in HW is available */
        int8_t  rmap_crc;       /* If RMAP CRC is available */
        int8_t  rx_unalign;     /* RX unaligned (byte boundary) access allowed*/
        int8_t  nports;         /* Number of Ports (1 or 2) */
        int8_t  ndma_chans;     /* Number of DMA Channels (1..4) */
        int     hw_version;     /* GRSPW Hardware Version */
        int8_t  irq;            /* SpW Distributed Interrupt available if 1 */
};
```

Table 6.7. grspw_hw_sup data structure declaration

rmap	0	RMAP target functionality is not implemented in hardware.
	1	RMAP target functionality is implemented in hardware.
rmap_crc	Non-zero if RMAP CRC is available in hardware.	
rx_unalign	Non-zero if hardware can perform RX unalibned (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.	
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.
irq	Non-zero if SpaceWire distributed interrupt functionality is implemented in hardware.	

6.3.3. Link Control

The SpaceWire link is controlled and configured using the device API functions described below. The link control functionality is described in Section 6.2.3.

In system where the GRSPW controller is connected directly to a GRSPW SpaceWire router, the link interface is configured in the corresponding router driver.

Table 6.8. `grspw_get_linkcfg` function declaration

Proto	`uint32_t grspw_get_linkcfg(void *d)`	
About	Get link configuration The function returns the link configuration, which can be masked with the `LINKOPTS_*` defines.	
Param	*d* [IN] pointer Device handle returned by `grspw_open`.	
Return	uint32_t. Link configuration read from I/O registers	
	Bits	Description
	0	Link is enabled. Mask: `LINKOPTS_ENABLE`/`LINKOPTS_DISABLE`
	1	Link is started. Mask: `LINKOPTS_START`
	2	Link is in autostart mode. Mask: `LINKOPTS_AUTOSTART`
	9	Interrupt generation on link error is enabled. Mask: `LINKOPTS_ERRIRQ`

Table 6.9. `grspw_set_linkcfg` function declaration

Proto	`int grspw_set_linkcfg(void *d, uint32_t cfg)`
About	Set link configuration The function sets the link configuration using the with the `LINKOPTS_*` defines.
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.
Param	*`cfg`* [IN] uint32_t Link configuration to set from I/O registers

Bits	Description
0	Link enable. Mask: `LINKOPTS_ENABLE`/`LINKOPTS_DISABLE`
1	Link started. Mask: `LINKOPTS_START`
2	Link in autostart mode. Mask: `LINKOPTS_AUTOSTART`
9	Enable interrupt generation on link error. Mask: `LINKOPTS_ERRIRQ`

Return	int. The function always returns DRV_OK.

Table 6.10. `grspw_get_clkdiv` function declaration

Proto	`uint32_t grspw_get_clkdiv(void *d)`
About	Get clock divisor The function reads and returns the clock divisor register, masked with `GRSPW_CLKDIV_MASK`. Start clock and run clock can be masked individually by using `GRSPW_CLKDIV_START` and `GRSPW_CLKDIV_RUN`. The referred defines are available in the file `include/regs/grspw-regs.h`.
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.
Return	uint32_t. Clock divisor read from I/O registers

Bits	Description
15..8	Clock divisor used during startup
7..0	Clock divisor used in RUN state

Table 6.11. `grspw_set_clkdiv` function declaration

Proto	`int grspw_set_clkdiv(void *d, uint32_t cfg)`
About	Set clock divisor The function sets the clock divisor register with value *`cfg`* masked with `GRSPW_CLKDIV_MASK` in `include/regs/grspw-regs.h`.
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.
Param	*`clkdiv`* [IN] uint32_t Clock devisor value to write to I/O registers.

Bits	Description
15..8	Clock divisor used during startup
7..0	Clock divisor used in RUN state

Return	int. The function always returns DRV_OK.

Table 6.12. `grspw_link_state` function declaration

Proto	`spw_link_state_t grspw_link_state(void *d)`

About	Get current SpaceWire link state.
Param	*d* [IN] pointer Device identifier returned by `grspw_open`.
Return	enum spw_link_state_t. SpaceWire link state according to SpaceWire standard FSM state machine numbering. The possible return values are listed below. The values are defined by enum `spw_link_state_t` and shall be prefixed with `SPW_LS_`.

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 6.13. `grspw_get_status` function declaration

Proto	`uint32_t grspw_get_status(void *d)`
About	Get status register value
Param	*d* [IN] pointer Device handle returned by `grspw_open`.
Return	uint32_t. Current value of the GRSPW Status Register. Register definitions for the GRSPW Status Register are available in the file `include/regs/grspw-regs.h`. The relevant defines are prefixed with *`GRSPW_STS_`*.

Table 6.14. `grspw_clear_status` function declaration

Proto	`void grspw_clear_status(void *d, uint32_t status)`
About	Clear bits in the status register
Param	*d* [IN] pointer Device handle returned by `grspw_open`.
Param	*`status`* [IN] uint32_t Mask of bits to clear in the GRSPW Status Register. Register definitions for the GRSPW Status Register are available in the file `include/regs/grspw-regs.h`. The relevant defines are prefixed with *`GRSPW_STS_`*.
Return	None.

6.3.4. 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 6.2.7. 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.
  */
 int8_t promiscuous;

 /* Default Node Address and Mask */
 uint8_t def_addr;
 uint8_t def_mask;
 /* DMA Channel custom Node Address and Mask */
 struct {
```

```
  int8_t node_en;    /* Enable Separate Addr */
  uint8_t node_addr; /* Node address */
  uint8_t node_mask; /* Node address mask */
 } dma_nacfg[4];
};
```

Table 6.15. grspw_addr_config data structure declaration

promiscu-ous	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.
def_addr	GRSPW default node address.
def_mask	GRSPW default node address mask.
dma_nacfg	DMA channel node address array configuration, see below field description. DMA channel N is described by *`dma_nacfg[N]`*.

Field	Description
node_en	Enable (1) or disable (1) separate node address for DMA channel N (determined by array index).
node_addr	Node address for DMA channel N (determined by array index).
node_mask	Node address mask for DMA channel N (determined by array index).

Table 6.16. `grspw_addr_ctrl` function declaration

Proto	`void grspw_addr_ctrl(void *d, const struct grspw_addr_config *cfg)`
About	Set node address 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 6.15. When using multiple DMA channels one must assign each DMA channel a unique node address or a unique range by masking. Each DMA channel is represented by the input *`dma_nacfg[N]`*.
Param	*`d`* [IN] pointer Device handle returned by grspw_open.
Param	*`cfg`* [IN] pointer Address configuration to set.
Return	None.

6.3.5. Time-control codes

SpaceWire Time Code handling is controlled and configured using the device API functions described below. The Time Code functionality is described in Section 6.2.4.

Table 6.17. `grspw_get_tccfg` function declaration

Proto	`uint32_t grspw_get_tccfg(void *d)`
About	Get time-code configuration The function reads and returns the time-code configration from GRSPW control register.
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.
Return	uint32_t. Time-code configuration read from I/O registers. The return value can be evaluated against the following masks:

Mask	Description
TCOPTS_EN_RX	Enable time-code receptions
TCOPTS_EN_TX	Enable time-code transmissions

	TCOPTS_EN_RXIRQ	Generate interrupt when a valid time-code is received.

Table 6.18. `grspw_set_tccfg` function declaration

Proto	`void grspw_set_tccfg(void *d, uint32_t cfg)`	
About	Set time-code configuration The function sets the time-code configuration in GRSPW control register.	
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.	
Param	*`cfg`* [IN] uint32_t Time-code configuration to write in I/O registers. The following masks can be used at configuration:	
	Mask	Description
	TCOPTS_EN_RX	Enable time-code receptions
	TCOPTS_EN_TX	Enable time-code transmissions
	TCOPTS_EN_RXIRQ	Generate interrupt when a valid time-code is received.
Return	None.	

Table 6.19. `grspw_get_tc` function declaration

Proto	`uint32_t grspw_get_tc(void *d)`	
About	Get time register value The function reads and returns the GRSPW time register value.	
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.	
Return	uint32_t. Time register read from I/O registers. The return value can be evaluated against the following masks:	
	Mask	Description
	TCTRL_MASK	Time control flags of time register
	TIMECNT_MASK	Time counter of time register

6.3.6. 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 6.2.3.

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 6.20. `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 6.21. `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 6.22. `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.

6.3.7. RMAP Control

The device API described below is used to configure the hardware supported RMAP target. The RMAP support is described in Section 6.2.5.

Availability of RMAP support can be determined by using the function `grspw_hw_support`.

> 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 6.32 for more information.

Table 6.23. `grspw_rmap_set_ctrl` function declaration

Proto	`int grspw_rmap_set_ctrl(void *d, uint32_t options)`
About	Set RMAP configuration
Param	*d* [IN] pointer Device handle returned by `grspw_open`.
Param	*options* [IN] uint32_t

	RMAP control options to set in I/O registers. The following bit masks, prefixed with `RMAPOPTS_` shall be used.	
	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.
Return	int. The function always returns DRV_OK.	

Table 6.24. `grspw_rmap_set_destkey` function declaration

Proto	`int grspw_rmap_set_destkey(void *d, uint32_t destkey)`
About	Set RMAP destination key
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.
Param	*`destkey`* [IN] uint32_t Destination key to set. The value shall be AND:ed with the define `GRSPW_DK_DESTKEY` available in the file `include/regs/grspw-regs.h`.
Return	int. The function always returns DRV_OK.

6.3.8. Interrupt handling

No interrupt service routine is installed by the GRSPW driver. The user can install and uninstall an ISR by using the Operating System Abstraction Layer functions `osal_isr_register` and `osal_isr_unregister`. At least one GRSPW interrupt source must be enabled in the driver for interrupts to be generated. Possible interrupt sources are time-code tick-out, link-error, and DMA interrupts.

The functions `grspw_dma_tx_count` and `grspw_dma_rx_count` can be used from interrupt context to determine how many TX/RX packets are (at least) available to the user. `grspw_get_status` can be used to determine whether a new time count value (Tick Out) is available. Section 6.6 lists the API functions allowed to be called from ISR context.

6.4. 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.

6.4.1. 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 6.3.1 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 operating system abstraction layer. Protection 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. The channel state is set to *stopped*.
- Resources 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)
{
  void *device;
  void *dma0;
  int count;
  uint32_t linkcfg, clkdiv;
  spw_link_state_t state;
  struct grspw_list lst;

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

  /* Start Link */
  linkcfg = LINKOPTS_ENABLE | LINKOPTS_START; /* Start Link */
  grspw_set_linkcfg(device, linkcfg);
  clkdiv = (9 << 8) | 9; /* Clock Divisor factor of 10 */
  grspw_set_clkdiv(device, 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 (DRV_OK != 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 = grspw_dma_rx_recv(dma0, &lst);
  } while (0 == count);

  if (-1 == count) {
   printf("GRSPW0.DMA0: Receive error\n");
  } else {
   printf("GRSPW0.DMA0: Received %d packets\n", count);
  }

  /* ... */

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

Table 6.25. `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 function allocates buffers as necessary using dynamic memory allocation (`malloc()`. The returned value is used as input argument to all functions operating on the DMA channel.
Param	*`d`* [IN] pointer Device handle returned by `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.
Return	Pointer. Status and driver's internal device identification.

Value	Description
NULL	Indicates failure to DMA channel. Fails if DMA channel does not exists, DMA channel already has been opened or that DMA channel resource allocation or initialization failes.
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 block until other GRSPW device operations complete.

Table 6.26. `grspw_dma_close` function declaration

Proto	`int grspw_dma_close(void *c)`
About	Closes a previously opened DMA channel. The specificed DMA channel is stopped and closed. This will result in the same functionality as calling `grspw_dma_stop` to stop on-going DMA transfers and then free DMA channel resources.
Param	*c* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Return	int. Return code as indicated below.

Value	Description
DRV_OK	Success.
DRV_NOTOPEN	DMA channel *c* was not open.

6.4.1.1. Static buffer allocation

The function `grspw_dma_open` uses dynamic memory for allocating DMA buffers. An alternative is to use `grspw_dma_open_userbuf`, which allows the user to provide the buffers instead. Note that the corresponding function for closing the DMA channel is `grspw_dma_close_userbuf` in this case.

Table 6.27. `grspw_dma_open_userbuf` function declaration

Proto	`void *grspw_dma_open_userbuf(void *d, int chan_no, struct grspw_ring *rx_ring, struct grspw_ring *tx_ring, struct grspw_rxbd *rx_bds, struct grspw_txbd *tx_bds)`
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 function requires the caller to provide buffers for the driver to use (*rx_ring* *tx_ring* *rx_bds* *tx_bds*). These memory areas shall not be referenced by the user as long as the DMA channel is opened. The areas can be reused when the channel has been closed with `grspw_dma_close_userbuf`. The returned value is used as input argument to all functions operating on the DMA channel.
Param	*d* [IN] pointer Device handle returned by `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.
Param	*rx_ring* [IN] Pointer RX buffer ring area. Size shall be `GRSPW_RXBD_NR * sizeof (struct grspw_ring)`, aligned to 32-bit word.
Param	*tx_ring* [IN] Pointer

	TX buffer ring area. Size shall be `GRSPW_TXBD_NR * sizeof (struct grspw_ring)`, aligned to 32-bit word.
Param	*`rx_bds`* [IN] Pointer RX DMA buffer descriptor table area. Must be 1 KiB, and aligned to 1 KiB address boundary.
Param	*`tx_bds`* [IN] Pointer TX DMA buffer descriptor table area. Must be 1 KiB, and aligned to 1 KiB address boundary.
Return	Pointer. Status and driver's internal device identification.
Notes	May block until other GRSPW device operations complete.

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

Table 6.28. `grspw_dma_close_userbuf` function declaration

Proto	`int grspw_dma_close_userbuf(void *c)`
About	Closes a previously opened DMA channel. The specificed DMA channel is stopped and closed. This will result in the same functionality as calling `grspw_dma_stop` to stop on-going DMA transfers and then free DMA channel resources.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open_userbuf`.
Return	int. Return code as indicated below.

Value	Description
DRV_OK	Success.
DRV_NOTOPEN	DMA channel *`c`* was not open.

6.4.2. Starting and stopping DMA operation

The start and stop operational modes are described in Section 6.2.11. 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 6.29, see function description for details on limitations.

Table 6.29. 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 6.4.6	Yes
`grspw_dma_rx_prepare`	Yes, with limitations, see Section 6.4.6	Yes
`grspw_dma_rx_flush`	Yes	No
`grspw_dma_tx_send`	Yes, with limitations, see Section 6.4.5	Yes
`grspw_dma_tx_reclaim`	Yes, with limitations, see Section 6.4.5	Yes
`grspw_dma_tx_flush`	Yes	No

Function	Stopped	Started
`grspw_dma_config`	Yes	No
`grspw_dma_config_read`	Yes	Yes
`grspw_dma_stats_read`	Yes	Yes
`grspw_dma_stats_clr`	Yes	Yes

Table 6.30. `grspw_dma_start` function declaration

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

Table 6.31. `grspw_dma_stop` function declaration

Proto	`void grspw_dma_stop(void *c)`
About	Stops DMA operational mode for the DMA channel indicated by the argument. The transmitter will abort ongoing transfers and the receiver disabled. Packets in the RX SCHED queue will remain in this queue. The RXPKT_FLAG_RX packet flag is used to signal if the packet contains received data or not. Similarly, the TXPKT_FLAG_TX packet flag marks if the packet was actually transferred or not.
Param	*d* [IN] pointer Device identifier returned by `grspw_open`.
Return	None.
Notes	The user may want to flush the RX/TX SCHED queues with functions `grspw_dma_rx_flush` and `grspw_dma_tx_flush` after stopping to get unprocessed packets back.

6.4.3. 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.

On some systems, the data buffer pointer must be 32-bit word aligned for reception.

```
struct grspw_pkt {
```

```
 struct grspw_pkt *next; /* Next packet in list. NULL if last packet */
 uintptr_t pkt_id;       /* User assigned ID (not touched by driver) */
 void *data;             /* 4-byte or byte aligned depends on HW */
 void *hdr;              /* 4-byte or byte aligned depends on HW (only TX) */
 uint32_t dlen;          /* Length of Data Buffer */
 uint16_t flags;         /* RX/TX Options and status */
 uint8_t hlen;           /* Length of Header Buffer (only TX) */
};
```

Table 6.32. grspw_pkt data structure declaration

<table>
<tr><td>next</td><td colspan="2">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.</td></tr>
<tr><td>pkt_id</td><td colspan="2">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.</td></tr>
<tr><td>data</td><td colspan="2">Data Buffer Address. DMA will read from this. The address must be 4-byte or byte aligned depending on hardware.</td></tr>
<tr><td>hdr</td><td colspan="2">Header Buffer Address. DMA will read hlen bytes from this. The address must be 4-byte or byte aligned depending on hardware. This field is not used by RX operation.</td></tr>
<tr><td>dlen</td><td colspan="2">Data payload lenght. 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.</td></tr>
<tr><td>flags</td><td colspan="2">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 definitionas declared by the driver's header file.</td></tr>
<tr><td></td><td>Bits</td><td>TX Description (prefixed TXPKT_FLAG_)</td></tr>
<tr><td></td><td>NOCRC_MASK</td><td>Indicates to driver how many bytes shuld not be part of the header CRC calculation. 0 to 15 bytes can be omitted. Use NOCRC_LENN to select a specific lenght.</td></tr>
<tr><td></td><td>IE</td><td>Enable (1) or Disable (0) IRQ generation on packet transmission completed.</td></tr>
<tr><td></td><td>HCRC</td><td>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).</td></tr>
<tr><td></td><td>DCRC</td><td>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).</td></tr>
<tr><td></td><td>TX</td><td>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.</td></tr>
<tr><td></td><td>LINKERR</td><td>Set if a link error was exibited during transmission of this packet.</td></tr>
<tr><td></td><td>Bits</td><td>RX Description (prefixed RXPKT_FLAG_)</td></tr>
<tr><td></td><td>IE</td><td>Enable (1) or Disable (0) IRQ generation on packet reception completed.</td></tr>
<tr><td></td><td>TRUNK</td><td>Set if packet was truncated.</td></tr>
<tr><td></td><td>DCRC</td><td>Set if data CRC error detected (only valid if RMAP CRC is enabled).</td></tr>
<tr><td></td><td>HCRC</td><td>Set if header CRC error detected (only valid if RMAP CRC is enabled).</td></tr>
<tr><td></td><td>EEOP</td><td>Set if an End-of-Packet error occured during reception of this packet.</td></tr>
<tr><td></td><td>RX</td><td>Marks if packet was recevied or not.</td></tr>
<tr><td>hlen</td><td colspan="2">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.</td></tr>
</table>

6.4.4. Packet buffer lists

The DMA transfer operations take packet lists as input parameters. A packet list is a linked list with elements of type struct grspw_pkt. The public driver interface header file includes functions for manipulating lists, prefixed with `grspw_list_*()`.

The following list is a summary of some of the available list manipulation functions.

- `grspw_list_clr` initializes a list.
- `grspw_list_is_empty` determines if a list is empty.

- `grspw_list_append` appends a packet to the end of a list.
- `grspw_list_append_list` appends packets from one list to the end of another list.

6.4.5. Sending packets

Packets are sent by adding packets to the TX 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 6.2.1.

In the below example Figure 6.4 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.

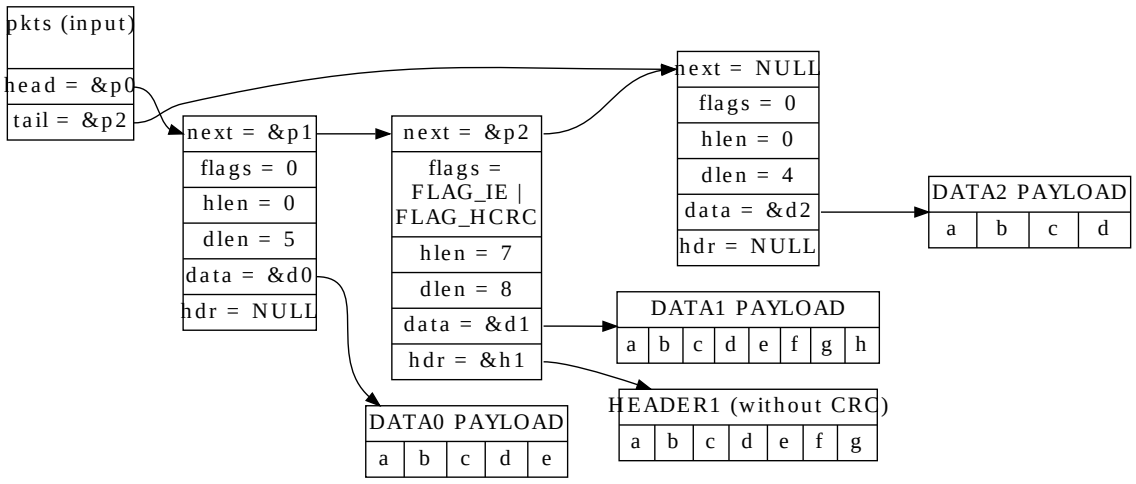

Figure 6.4. TX packet description `pkts` input to `grspw_tx_dma_send`

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

Table 6.33. `grspw_dma_tx_send` function declaration

Proto	`int grspw_dma_tx_send(void *c, struct grspw_list *pkts)`
About	Schedule list of packets for transmission at some point in future. The GRSPW transmitter is enabled when packets are added to the TX SCHED queue. (USER->SCHED) The fastest solution in retrieving sent TX packets and sending new frames is to call: 1. grspw_dma_tx_reclaim(opts=0) 2. grspw_dma_tx_send(opts=1) NOTE: the TXPKT_FLAG_TX flag must not be set in the packet structure.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*`pkts`* [IN] pointer A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that *`head`* points to the first packet and *`tail`* points to the last. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 6.32. Note that TXPKT_FLAG_TX of the *`flags`* field must not be set.

Return	int. See return codes below	
	Value	Description
	-1	Error: DMA channel is not in started mode.
	>=0	Successfully added pkts to TX SCHED list.
Notes	This function performs no operation when the DMA channel is stopped.	

Table 6.34. `grspw_dma_tx_reclaim` function declaration

Proto	`int grspw_dma_tx_reclaim(void *c, struct grspw_list *pkts)`
About	Reclaim TX packet buffers that has previously been scheduled for transmission with `grspw_dma_tx_send`. The packets in the SCHED queue which have been transmitted are moved to the `pkts` packet list. The user `pkts` list is not cleared by the function. 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. The typical solution for retrieving sent TX packets and sending new frames is to call: 1. grspw_dma_tx_reclaim() 2. grspw_dma_tx_send() NOTE: the TXPKT_FLAG_TX flag indicates if the packet was transmitted.
Param	`c` [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	`pkts` [OUT] pointer Sent TX packets will be taken from the SCHED queue and added to the `pkts` queue. The user queue `pkts` is not cleared. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 6.32. Note that TXPKT_FLAG_TX of the `flags` field indicates if the packet was sent of not. In case of DMA being stopped one can use this flag to see if the packet was transmitted or not. The TXPKT_FLAG_LINKERR indicates if a link error occurred during transmission of the packet, regardless the TXPKT_FLAG_TX is set to indicate packet transmission attempt.
Return	int. See return codes below

Value	Description
-1	Error: DMA channel is not in started mode.
0	No packet reclaimed (SCHED list contains no sent packets).
>0	Number of packets successfully reclaimed to user list.

Notes	This function can operate in stopped mode. This is useful when a link goes down and the DMA activity is stopped by user or by driver automatically.

6.4.6. Receiving packets

Packets are received by adding empty/free packets to the 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 6.2.1.

In the Figure 6.5 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 pointers 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.

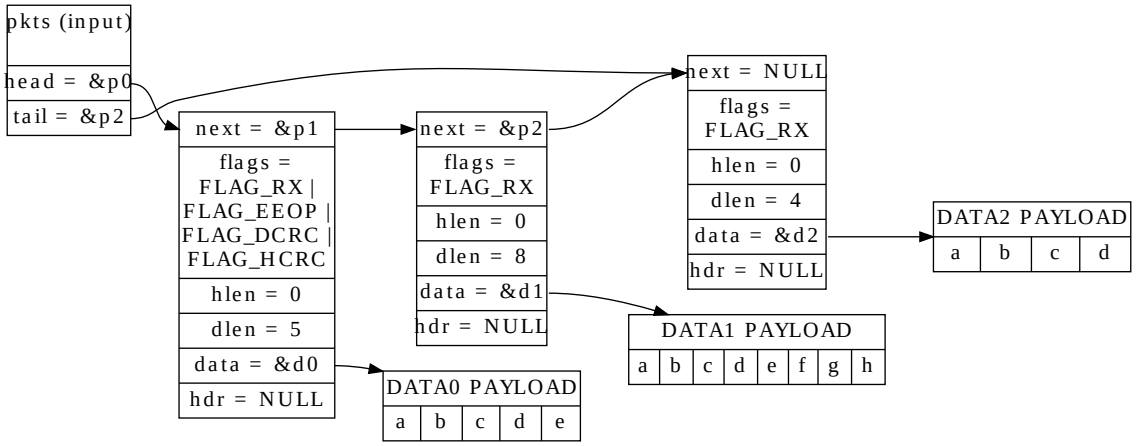

Figure 6.5. RX packet output from `grspw_dma_rx_recv`

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

Table 6.35. `grspw_dma_rx_prepare` function declaration

Proto	`int grspw_dma_rx_prepare(void *c, struct grspw_list *pkts)`
About	Add RX packet buffers for future reception. The received packets can later be read out with `grspw_dma_rx_recv`. The packets in *`pkts`* list are put to the SCHED queue of the driver (USER->SCHED). The typical solution for retreiving received RX packets and preparing new packet buffers for future receive, is to call: 1. grspw_dma_rx_recv(&recvlist) 2. grspw_dma_rx_prepare(&freelist) NOTE: the RXPKT_FLAG_RX flag must not be set in the packet structure.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*`pkts`* [IN] pointer A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that *`head`* points to the first packet and *`tail`* points to the last. The layout and content of the packet is defined by the grspw_pkt data structure described in Table 6.32. Note that RXPKT_FLAG_RX of the *`flags`* field must not be set.
Return	int. See return codes below

Value	Description
-1	Error: DMA channel is not in started mode.
0	No packets added (SCHED list is full).
>0	Number of packets successfully added to RX SCHED queue.

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

Table 6.36. `grspw_dma_rx_recv` function declaration

Proto	`int grspw_dma_rx_recv(void *c, struct grspw_list *pkts)`
About	Get received RX packet buffers which have previously been scheduled for reception with `grspw_dma_rx_prepare`. The packets in the RX SCHED queue which have been received are moved to the *`pkts`* packet list (SCHED->USER). 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 and received packet length. The header pointer and length fields are not touched by the driver, all data ends up in the data area. NOTE: the RXPKT_FLAG_RX flag indicates if a packet was received, thus if the data field contains new valid data or not.
Param	*c* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*pkts* [OUT] pointer Received RX packets will be taken from the SCHED queue and added to the *pkts* queue. The user queue *pkts* is not cleared. The layout and content of the packet is defined by the grspw_pkt data structure described in Table 6.32. 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 occured during transmission of the packet, regardless the RXPKT_FLAG_RX is set to indicate packet reception attempt.
Return	int. See return codes below

Value	Description
-1	Error: DMA channel is not in started mode.
0	No packet received (SCHED list contains no received packets).
>0	Number of received packets added to user list.

Notes	This function can be called when the DMA channel is in stopped mode. This is useful when a link goes down and the DMA activity is stopped by user or by driver automatically.

6.4.7. Transmission queue status

The current number of packets processed by hardware but not yet reclaimed/received by the driver can be queried using the functions described below. These numbers give a hint on how many packets will be reclaimed by a call to `grspw_dma_tx_reclaim` or received by `grspw_dma_rx_recv`.

Table 6.37. `grspw_dma_tx_count` function declaration

Proto	`int grspw_dma_tx_count(void *c)`
About	Get number of packets transmitted by hardware but not yet reclaimed by the driver. This is determined by looking at the TX descriptor pointer register. The number represents how many of the send packets that actually have been transmitted by hardware but not reclaimed by the driver yet.
Param	*c* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Return	int. The number of packets transmitted by hardware but not yet reclaimed by the driver.
Notes	This function can be called from interrupt context.

Table 6.38. `grspw_dma_rx_count` function declaration

Proto	`int grspw_dma_rx_count(void *c)`
About	Get number of packets received by hardware but not yet retrieved by the driver. This is determined by looking at the RX descriptor pointer register. The number represents how many of the prepared packets that actually have been received by hardware but not handled by the driver yet.
Param	*c* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Return	int. The number of packets received by hardware but not yet retrieved by the driver.
Notes	This function can be called from interrupt context.

6.4.8. Queue flushing

When a DMA channel is stopped after being in started state, it may contain scheduled unsent TX packets and scheduled unreceived RX packets. These packets can be given back to the user with the functions `grspw_dma_tx_flush` and `grspw_dma_rx_flush`.

Table 6.39. `grspw_dma_tx_flush` function declaration

Proto	`int grspw_dma_tx_flush(void *c, struct grspw_list *pkts)`
About	Flush TX packets from driver Like `grspw_dma_tx_reclaim`, but also move scheduled unsent packets to user list. This function can only be called when DMA channel is in stopped mode. Return value is the sum of sent packets and unsent packets. The `TXPKT_FLAG_TX` packet flag indicates, for each packet, if it was sent or not.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*`pkts`* [OUT] pointer The list will be initialized to contain the SpaceWire packets moved from the SCHED 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.
Return	Number of packets. See return codes below

Value	Description
-1	Error: DMA channel is in started mode.
others	Number of sent and unsent packets added to user list.

Notes	This function can only be called in DMA channel stopped mode.

Table 6.40. `grspw_dma_rx_flush` function declaration

Proto	`int grspw_dma_rx_flush(void *c, struct grspw_list *pkts)`
About	Flush RX packets from driver Like `grspw_dma_rx_recv`, but also move scheduled unreceived packets to user list. This function can only be called when DMA channel is in stopped mode. Returns sum of recevied packets and unreceived packets. The `RXPKT_FLAG_RX` packet flag indicates if the packet was received or not.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*`pkts`* [OUT] pointer The list will be initialized to contain the SpaceWire packets moved from the SCHED 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.
Return	Number of packts. See return codes below

Value	Description
-1	Error: DMA channel is in started mode.
others	Number of received and unreceived packets added to user list.

Notes	This function can only be called in DMA channel stopped mode.

6.4.9. Statistics

The driver counts statistics at certain events. The driver's DMA channel counters can be read out using the DMA API. Packet transmission statistics, packet transmission errors and packet queue statistics can be obtained.

```
struct grspw_dma_stats {
 /* 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 */
};
```

Table 6.41. grspw_dma_stats data structure declaration

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.

Table 6.42. `grspw_dma_stats_read` function declaration

Proto	`void grspw_dma_stats_read(void *c, 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 6.41. Note that the snapshot is taken without lock protection, as a consequence the statistics may not be synchonized 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 6.41.
Return	None.

Table 6.43. `grspw_dma_stats_clr` function declaration

Proto	`void grspw_dma_stats_clr(void *c)`
About	Resets a DMA channel's statistic counters. The channel counters are set to zero.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Return	None.

6.4.10. 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.

```
struct grspw_dma_config {
 int flags;              /* DMA config flags, see DMAFLAG_* options */
 int rxmaxlen;           /* RX Max Packet Length */
};
```

Table 6.44. grspw_dma_config data structure declaration

flags	RX/TX DMA transmission options See below.	
	Bits	Description (prefixed DMAFLAG_)

	NO_SPILL	Enable (1) or Disable (0) packet spilling, flow control.
	STRIP_ADR	Enable (1) or Disable (0) stripping node address byte from DMA write transfers (packet reception). See hardware support to determine if present in hardware. See hardware documenation about DMA CTRL SA bit.
	STRIP_PID	Enable (1) or disable (0) stripping PID byte from DMA write transfers (packet reception).(if CRC is available in hardware). See hardware support to determine if present in hardware. See hardware documenation about DMA CTRL SP bit.
rxmaxlen	Max packet reception length. Longer packets with will be truncated see RXPKT_FLAG_TRUNK flag in packet structure.	

If the function `grspw_dma_config` is not called after the user has opened the DMA channel with `grspw_dma_open`, then the configuration will have default values:

- Packet spilling is enabled (*`NO_SPILL=0`*).
- Node address byte stripping is disabled (*`STRIP_ADR=0`*).
- PID byte stripping is disabled (*`STRIP_PID=0`*).
- Maximum packet reception length is 4096 bytes (*`rxmaxlen=4096`*).

If the DMA channel is stopped the last configuration set with `grspw_dma_config` is used the next time the channel is started with `grspw_dma_start`.

Table 6.45. `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 handle returned by `grspw_dma_open`.
Param	*`cfg`* [IN] pointer Address to where the driver will read the DMA channel configuration from. The configuration options are described in Table 6.44.
Return	int. Return code as indicated below.

Value	Description
DRV_OK	Success.
DRV_FAIL	Failure due to invalid input arguments or DMA has already been started.

Table 6.46. `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 handle returned by `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 6.44.
Return	int. Return code as indicated below.

	Value	Description
	DRV_OK	Success.
	DRV_FAIL	Failure due to invalid input argument.

6.4.11. DMA channel status

Status information unique to a DMA channel is exported by the drivers DMA channel status interface. It reads and manipulates status bits available in the GRSPW DMA control register.

The following status information is available:

- Bus errors caused by the receive DMA channel (`GRSPW_DMA_STATUS_RA`).
- Bus errors caused by the transmit DMA channel (`GRSPW_DMA_STATUS_TA`).
- A packets has been received (`GRSPW_DMA_STATUS_PR`).
- A packets has been sent (`GRSPW_DMA_STATUS_PS`).

Table 6.47. `grspw_dma_get_status` function declaration

Proto	`uint32_t grspw_dma_get_status(void *c)`
About	Get DMA channel status The function reads and returns status from the GRSPW DMA control register. Status bits in the register are not cleared. Use function `grspw_dma_clear_status` to clear the status bits.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Return	uint32_t. Mask of DMA channel status bits read from GRSPW DMA control register. The return value shall be evaluated against the following bit masks:

Mask	Description
GRSPW_DMA_STATUS_RA	RX AHB Error
GRSPW_DMA_STATUS_TA	TX AHB Error
GRSPW_DMA_STATUS_PR	Packet received
GRSPW_DMA_STATUS_PS	Packet sent

Table 6.48. `grspw_dma_clear_status` function declaration

Proto	`void grspw_dma_clear_status(void *c, uint32_t status)`
About	Clear DMA channel status The function clears the status bits in GRSPW DMA control register corresponding to the bits set in the *`status`* parameter. Current status can be retrieved with the function `grspw_dma_get_status`.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*`status`* [IN] uint32_t Mask of DMA channel status bits to clear in GRSPW DMA control register. The bit masks are the same as the masks for `grspw_dma_get_status` return value.
Return	None.

6.5. API reference

This section lists all functions and data structures of the GRSPW driver API, and in which section(s) they are described.

6.5.1. Data structures

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

Table 6.49. Data structures reference

Data structure name	Section
struct grspw_pkt	6.4.3
struct grspw_addr_config	6.3.4
struct grspw_hw_sup	6.3.2
struct grspw_dma_stats	6.4.9
struct grspw_dma_config	6.4.10

6.5.2. 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 6.50. Device function reference

Prototype	Section
`int grspw_dev_count(void)`	6.3.1
`void *grspw_open(int dev_no)`	6.3.1
`int grspw_close(void *d)`	6.3.1
`void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)`	6.3.4,
`spw_link_state_t grspw_link_state(void *d)`	6.3.3,
`uint32_t grspw_get_linkcfg(void *d)`	6.3.3,
`int grspw_set_linkcfg(void *d, uint32_t cfg)`	6.3.3,
`uint32_t grspw_get_clkdiv(void *d)`	6.3.3,
`int grspw_set_clkdiv(void *d, uint32_t clkdiv)`	6.3.3,
`uint32_t grspw_get_status(void *d)`	6.3.3,
`void grspw_clear_status(void *d, uint32_t status)`	6.3.3,
`uint32_t grspw_get_tccfg(void *d)`	6.3.5,
`void grspw_set_tccfg(void *d, uint32_t cfg)`	6.3.5,
`uint32_t grspw_get_tc(void *d)`	6.3.5,

6.5.3. 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 6.51. DMA channel function reference

Prototype	Section
`void *grspw_dma_open(void *d, int chan_no)`	6.4.1, 6.3.1
`void grspw_dma_close(void *c)`	6.4.1, 6.3.1
`void *grspw_dma_open_userbuf(void *d, int chan_no, struct grspw_ring *rx_ring, struct grspw_ring *tx_ring, struct grspw_rxbd *rx_bds, struct grspw_txbd *tx_bds)`	6.4.1, 6.3.1

Prototype	Section
`void grspw_dma_close_userbuf(void *c)`	6.4.1, 6.3.1
`int grspw_dma_start(void *c)`	6.4.2,
`void grspw_dma_stop(void *c)`	6.4.2,
`int grspw_dma_rx_recv(void *c, struct grspw_list *pkts)`	6.4.6,
`int grspw_dma_rx_prepare(void *c, struct grspw_list *pkts)`	6.4.6,
`int grspw_dma_rx_flush(void *c, struct grspw_list *pkts)`	6.4.8,
`int grspw_dma_tx_send(void *c, struct grspw_list *pkts)`	6.4.5,
`int grspw_dma_tx_reclaim(void *c, struct grspw_list *pkts)`	6.4.5,
`int grspw_dma_tx_flush(void *c, struct grspw_list *pkts)`	6.4.8,
`void grspw_dma_stats_read(void *c, struct grspw_dma_stats *sts)`	6.4.9
`void grspw_dma_stats_clear(void *c)`	6.4.9
`int grspw_dma_config(void *c, struct grspw_dma_config *cfg)`	6.4.10
`int grspw_dma_config_read(void *c, struct grspw_dma_config *cfg)`	6.4.10
`uint32_t grspw_dma_get_status(void *c)`	6.4.11
`void grspw_dma_clear_status(void *c, uint32_t status)`	6.4.11

6.6. Restrictions

To process interrupt events, the user ISR should typically wake up a task which performs the driver API functions necessary. The following GRSPW Packet driver functions are allowed to be called from an ISR:

- `grspw_get_status`
- `grspw_link_state`
- `grspw_dma_rx_count`
- `grspw_dma_tx_count`
- `grspw_dev_count`
- `grspw_clear_status`
- `grspw_get_clkdiv`
- `grspw_get_linkcfg`
- `grspw_get_tc`
- `grspw_get_tccfg`
- `grspw_dma_get_status`
- `grspw_dma_clear_status`

7. GRCAN CAN driver

7.1. Introduction

This section describes the driver used to control the GRLIB GRCAN and GRCANFD devices for CAN DMA operation.

7.1.1. User Interface

This section covers how the driver can be interfaced to an application to control both the GRCAN and GRCANFD hardware.

Controlling the driver and device is done with functions provided by the driver prefixed with `grcan_`. GRCANFD specific functions are prefixed with `grcanfd_`. All driver functions take a device handle returned by `grcan_open` as the first parameter. All supported commands and their data structures are defined in the CAN driver's header file `drv/grcan.h`.

All driver functions are non-blocking.

7.1.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 5.

Table 7.1. Driver registration functions

Registration method	Function
Register one device	`grcan_register()`
Register many devices	`grcan_init()`

7.1.3. Examples

Examples are available in the `src/libdrv/examples/` directory in the Zephyr distribution.

7.1.4. Known driver limitations

- The DMA buffers must be CPU accessible and within the same address space. No address translation is performed by the driver.

7.2. Opening and closing device

A GRCAN 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 `grcan_dev_count`. A particular device can be opened using `grcan_open` and closed `grcan_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 `osal_ldstub` from the OSAL. Protection is used by all GRCAN devices on opening and closing.

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

- GRCAN device I/O registers are initialized, including masking all interrupts.
- The core is disabled (to allow configuration).
- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the same device.

The example below prints the number of GRCAN devices to screen then opens and closes the first GRCAN device present in the system.

```
int print_grcan_devices(void)
{
        struct grcan_priv *device;
        int count;
```

```
        count = grcan_dev_count();
        printf("%d GRCAN device(s) present\n", count);

        device = grcan_open(0);
        if (!device) {
                return -1; /* Failure */
        }
        if (grcan_canfd_capable(device)) {
                printf("Device is CANFD capable!\n");
        }
        grcan_close(device);
        return 0; /* success */
}
```

Table 7.2. `grcan_dev_count` function declaration

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

Table 7.3. `grcan_open` function declaration

Proto	`struct grcan_priv *grcan_open(int dev_no)`	
About	Opens a GRCAN device. The GRCAN device is identified by index. The returned value is used as input argument to all functions operating on the device. The function allocates DMA buffers as necessary using dynamic memory allocation (`malloc()`).	
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 `grcan_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 GRCAN device.

Table 7.4. `grcan_close` function declaration

Proto	`int grcan_close(struct grcan_priv *d)`
About	Closes a previously opened device.
Param	*`d`* [IN] pointer Device identifier. Returned from `grcan_open`.
Return	int. This function always returns 0 (success)

Table 7.5. `grcan_canfd_capable` function declaration

Proto	`int grcan_canfd_capable(struct grcan_priv *priv);`
About	Checks if the given device is CANFD capable.
Param	*`priv`* [IN] Pointer Device identifier. Returned by `grcan_open`.
Return	int. Non-zer is device is CANFD capable, zero if not.

7.2.1. Static buffer allocation

The function `grcan_open` uses dynamic memory for allocating DMA buffers. An alternative is to use `grcan_open_userbuf`, which allows the user to provide the buffers instead. Note that the corresponding function for closing the DMA channel is `grcan_close_userbuf` in this case.

Table 7.6. `grcan_open_userbuf` function declaration

Proto	`struct grcan_priv *grcan_open_userbuf(int dev_no, void *rxbuf, int rxbuf_size, void *txbuf, int txbuf_size)`
About	Opens a GRCAN device. The GRCAN device is identified by index. The returned value is used as input argument to all functions operating on the device. The function requires the caller to provide DMA buffers for the driver to use (*`rxbuf`* and *`txbuf`*). These memory areas shall not be referenced by the user as long as the driver channel is opened. The areas can be reused when the driver has been closed with `grcan_close_userbuf`.
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 `grcan_dev_count`.
Param	*`rxbuf`* [IN] Pointer RX DMA buffer address. Must be aligned to 1 KiB address boundary.
Param	*`rxbuf_size`* [IN] Integer RX DMA buffer size in bytes. Must be a multiple of 64.
Param	*`txbuf`* [IN] Pointer TX DMA buffer address. Must be aligned to 1 KiB address boundary.
Param	*`txbuf_size`* [IN] Integer TX DMA buffer size in bytes. Must be a multiple of 64.
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 GRCAN device.

Table 7.7. `grcan_close_userbuf` function declaration

Proto	`int grcan_close_userbuf(struct grcan_priv *d)`
About	Closes a previously opened device.
Param	*`d`* [IN] pointer Device identifier. Returned from `grcan_open_userbuf`.
Return	int. This function always returns 0 (success)

7.3. Operation mode

The driver always operates in one of four modes: STATE_STARTED, STATE_STOPPED, STATE_BUSOFF or STATE_AHBERR. In STATE_STOPPED mode, the DMA is disabled and the user is allowed to configure the device and driver. In STATE_STARTED mode, the receive and transmit DMA can be active and only a limited number of configuration operations are possible.

The driver enters STATE_BUSOFF mode if a bus-off condition is detected and STATE_AHBERR if an AHB error is caused by the GRCAN DMA. When any of these two modes are entered, the user should call `grcan_stop()` followed by `grcan_start()` to put the driver in STATE_STARTED again.

Transitions between started and stopped mode are normally caused by the users interaction with the driver API functions. In some situations, such CAN bus-off or DMA AHB error condition, the driver itself makes the transition from started to stopped.

7.3.1. Starting and stopping

The `grcan_start()` function places the CAN core in STATE_STARTED mode. Configuration set by previous driver function calls are committed to hardware before started mode enters. It is necessary to enter started mode to

be able to receive and transmit messages on the CAN bus. The `grcan_start()` function call will fail if receive or transmit buffers are not correctly allocated or if the CAN core is already is in started mode.

The function `grcan_stop()` makes the CAN core leave the previous mode and enter STATE_STOPPED mode. After calling this function, further calls to `grcan_read()`/ `grcanfd_read()` or `grcan_write()` /`grcanfd_write()` will fail. It is necessary to enter stopped mode to change operating parameters of the CAN core such as the baud rate and for the driver to safely change configuration such as FIFO buffer lengths. The function will fail if the CAN core already is in stopped mode.

Function `grcan_get_state()` is used to determine the driver operation mode.

Table 7.8. `grcan_get_state` function declaration

Proto	`int grcan_get_state(struct grcan_priv *d)`	
About	Get current GRCAN software state If *STATE_BUSOFF* or *STATE_AHBERR* is returned then the function `grcan_stop()` shall be called before continue using the driver.	
Param	*d* [IN] Pointer Device identifier. Returned by `grcan_open`.	
Return	int. Status	
	Value	Description
	STATE_STOPPED	Stopped
	STATE_STARTED	Started
	STATE_BUSOFF	Bus-off has been detected
	STATE_AHBERR	AHB error has been detected
	GRCAN_RET_AHBERR	Similar to BUSOFF, but was caused by AHB error.

7.4. Configuration

The CAN core and driver are configured using function calls. Return values for most functions are 0 for success and non-zero on failure.

The function `grcan_set_silent()` sets the `SILENT` bit in the configuration register of the CAN hardware the next time the driver is started. If the `SILENT` bit is set the CAN core operates in listen only mode where `grcan_write()`/ `grcanfd_write()` calls fail and `grcan_read()`/`grcanfd_read()` calls proceed. This function fails and returns nonzero if called in started mode.

`grcan_set_abort()` sets the `ABORT` bit in the configuration register of the CAN hardware. The `ABORT` bit is used to cause the hardware to stop the receiver and transmitter when an AMBA AHB error is detected by hardware. This function fails and returns nonzero if called in started mode.

7.4.1. Channel selection

`grcan_set_selection()` selects active channel used during communication. The function takes a second argument, a pointer to a grcan_selection data structure described below. This function fails and returns nonzero if called in started mode.

The grcan_selection data structure is used to select active channel. Each channel has one transceiver that can be activated or deactivated using this data structure. The hardware can however be configured active low or active high making it impossible for the driver to know how to set the configuration register in order to select a predefined channel.

```
struct grcan_selection {
        int selection;
        int enable0;
        int enable1;
};
```

Table 7.9. grcan_selection member description

Member	Description
selection	Select receiver input and transmitter output.
enable0	Set value of output 1 enable
enable1	Set value of output 1 enable

7.4.2. GRCAN Timing parameters

`grcan_set_btrs()` sets the timing registers manually. See the CAN hardware documentation for a detailed description of the timing parameters. The function takes a pointer to a grcan_timing data structure containing all available timing parameters. The grcan_timing data structure is described below. This function fails and returns nonzero if called in started mode.

The grcan_timing data structure is used when setting GRCAN timing configuration registers manually. The parameters are used when hardware generates the baud rate and sampling points.

```
struct grcan_timing {
 unsigned char scaler;
 unsigned char ps1;
 unsigned char ps2;
 unsigned int rsj;
 unsigned char bpr;
};
```

Table 7.10. grcan_timing member description

Member	Description
scaler	Prescaler
ps1	Phase segment 1
ps2	Phase segment 2
rsj	Resynchronization jumps, 1..4
bpr	**Value** / **Baud rate** 0 / system clock / (scaler+1) / 1 1 / system clock / (scaler+1) / 2 2 / system clock / (scaler+1) / 4 3 / system clock / (scaler+1) / 8

The function `grcan_set_speed()` can be used to set the CAN bus frequency. It takes a parameter in Hertz and calculates the appropriate timing register parameters. If the timing register values could not be calculated, then a non-zero value is returned.

7.4.3. GRCANFD Timing parameters

`grcanfd_set_btrs()` sets the timing registers manually. See the CAN hardware documentation for a detailed description of the timing parameters. The function takes a pointer to two grcanfd_timing data structure containing all available timing parameters. One for nominal bit-rate and one for fd bitrate. The grcanfd_timing data structure is described below. This function fails and returns nonzero if called in started mode.

The grcanfd_timing data structure is used when setting GRCAN timing configuration registers manually. The parameters are used when hardware generates the baud rate and sampling points.

```
struct grcanfd_timing {
  unsigned char scaler;
  unsigned char ps1;
  unsigned char ps2;
  unsigned char sjw;
  unsigned char resv_zero;
};
```

Table 7.11. grcanfd_timing member description

Member	Description
scaler	Prescaler
ps1	Phase segment 1
ps2	Phase segment 2
rsw	Synchronization Jump Width
resv_zero	Reserved.

The function `grcanfd_set_speed()` can be used to set the CAN bus frequency. It takes two parameters in Hertz, nominal and FD, and calculates the appropriate timing register parameters. If the timing register values could not be calculated, then a non-zero value is returned.

7.5. Receive filters

7.5.1. Data structures

The grcan_filter structure is used when changing acceptance filter of the CAN receiver and the SYNC Rx/Tx Filter using the functions `grcan_set_afilter` and `grcan_set_sfilter`. This data structure is used differently for different driver functions.

```
struct grcan_filter {
        unsigned long long mask;
        unsigned long long code;
};
```

Table 7.12. grcan_filter member description

Member	Description
mask	Selects what bits in *code* will be used or not. A set bit is interpreted as don't care.
code	Specifies the pattern to match, only the unmasked bits are used in the filter.

7.5.2. Acceptance filter

`grcan_set_afilter()` sets acceptance filter which is matched for each message received. Let the second argument point to a grcan_filter data structure or NULL to disable filtering and let all messages pass the filter. Messages matching the condition below are passed and possible to read from user space:

```
(id XOR code) AND mask = 0
```

`grcan_set_afilter()` can be called in any mode and never fails.

7.5.3. Sync filter

`grcan_set_sfilter()` sets Rx/Tx SYNC filter which is matched by receiver for each message received. Let the second argument point to a grcan_filter data structure or NULL to disable filtering and let all messages pass the filter. Messages matching the condition below are treated as SYNC messages:

```
(id XOR code) AND mask = 0
```

`grcan_set_sfilter()` can be called in any mode and never fails.

7.6. Driver statistics

`grcan_get_stats()` copies the driver's internal counters to a user provided data area. The format of the data written is described below (grcan_stats). The function will fail if the user pointer is NULL.

`grcan_clr_stats()` clears the driver's collected statistics. This function never fails.

The grcan_stats data structure contains various statistics gathered by the CAN hardware.

```
struct grcan_stats {
        unsigned int rxsync_cnt;
```

```
        unsigned int txsync_cnt;
        unsigned int ahberr_cnt;
        unsigned int ints;
        unsigned int busoff_cnt;
};
```

Table 7.13. grcan_stats member description

Member	Description
rxsync_cnt	Number of received SYNC messages (matching SYNC filter)
txsync_cnt	Number of transmitted SYNC messages (matching SYNC filter)
ahberr_cnt	Number of DMA AHB errors
ints	Number of times the interrupt handler has been invoked.
busoff_cnt	Number of bus-off conditions

7.7. Device status

`grcan_get_status()` stores the current status of the CAN core to the location pointed to by the second argument. This function is typically used to determine the error state of the CAN core. The 32-bit status word can be matched against the bit masks in the table below.

Table 7.14. Device status word bit masks

Mask	Description
GRCAN_STAT_PASS	Error-passive condition
GRCAN_STAT_OFF	Bus-off condition
GRCAN_STAT_OR	Overrun during reception
GRCAN_STAT_AHBERR	AMBA AHB error
GRCAN_STAT_ACTIVE	Transmission ongoing
GRCAN_STAT_RXERRCNT	Reception error counter, 8-bit
GRCAN_STAT_TXERRCNT	Transmission error counter, 8-bit

`grcan_get_status()` fails if the user pointer is NULL.

7.8. CAN bus transfers

7.8.1. Data structures

The struct grcan_canmsg type is used for GRCAN when transmitting and receiving CAN messages. For GRCANFD the struct grcan_canfdmsg type is used instead. The structure describes the drivers view of a CAN message. See the transmission and reception section for more information.

```
struct grcan_canmsg {
        char extended;
        char rtr;
        char unused;
        unsigned char len;
        unsigned char data[8];
        unsigned int id;
};
```

Table 7.15. struct grcan_canfdmsg member description

Member	Description
extended	Indicates whether the CAN message has 29 or 11 bits ID tag. Extended or Standard frame.
rtr	Remote Transmission Request bit.
len	Length of *`data`*.
data	CAN message data, *`data[0]`* is the most significant byte – the first byte.

Member	Description
id	The ID field of the CAN message. An extended frame has 29 bits whereas a standard frame has only 11-bits. The most significant bits are not used.

```
struct grcan_canfdmsg {
        uint8_t extended;
        uint8_t rtr;
        uint8_t fdopts;
        uint8_t len;
        uint32_t id;
        union {
                uint64_t dwords[8];
                uint8_t  bytes[64];
        } data;
};
```

Table 7.16. struct grcan_canmsg member description

Member	Description
extended	Indicates whether the CAN message has 29 or 11 bits ID tag. Extended or Standard frame.
rtr	Remote Transmission Request bit.
fdopts	FD options. Bit1: 1=Switch bit rate. bit2: 1=FD frame.
len	Length of *`data`*.
id	The ID field of the CAN message. An extended frame has 29 bits whereas a standard frame has only 11-bits. The most significant bits are not used.
data	CAN message data, *`data[0]`* is the most significant byte/word – the first byte

7.8.2. Transmission

Messages are transmitted using the `grcan_write()` function for GRCAN cores and `grcanfd_write()` for GRCANFD cores. It is possible to transmit multiple CAN messages in one call. An example transmission is shown below:

```
result = grcan_write(d, &tx_msgs[0], msgcnt);
```

On success the number of CAN messages transmitted is returned and on failure a *`GRCAN_RET_`* value is returned. The parameter *`tx_msgs`* points to the beginning of a struct grcan_canmsg structure which includes data, length and transmission parameters. The last function parameter specifies the total number of CAN messages to be transmitted. For `grcanfd_write()` the parameter *`tx_msgs`* points to the beginning of a struct grcan_canfdmsg instead.

The transmit operation is non-blocking: `grcan_write()`/`grcanfd_write()` will return immediately with a return value indicating the number CAN messages scheduled.

Each message has an individual set of parameters controlled by the struct grcan_canmsg or struct grcan_canfdmsg type.

> The user is responsible for checking the number of messages actually sent when in non-blocking mode. A 3 message transmission requests may end up in only 2 transmitted messages for example.

Table 7.17. `grcan_write` function declaration

Proto	`int grcan_write(struct grcan_priv *d, struct grcan_canmsg *msg, size_t count)`
About	Transmit CAN messages Multiple CAN messages can be transmitted in one call.
Param	*`d`* [IN] Pointer Device identifier. Returned by `grcan_open`.

Param	*msg* [IN] Pointer First CAN messages to transmit	
Param	*count* [IN] Integer Total number of CAN messages to transmit.	
Return	int. Status	
	Value	Description
	>=0	Number of CAN messages transmitted. This can be less than the *count* parameter.
	GRCAN_RET_INVARG	Invalid argument: *count* parameter is less than one or the *msg* parameter is NULL.
	GRCAN_RET_NOTSTARTED	Driver is not in started mode or device is configured as silent. Nothing done.
	GRCAN_RET_BUSOFF	A write was interrupted by a bus-off error. Device has left started mode.
	GRCAN_RET_AHBERR	Similar to BUSOFF, but was caused by AHB error.

Table 7.18. `grcanfd_write` function declaration

Proto	`int grcanfd_write(struct grcan_priv *d, struct grcan_canfdmsg *msg, size_t count)`	
About	Transmit CAN-FD messages Multiple CAN messages can be transmitted in one call.	
Param	*d* [IN] Pointer Device identifier. Returned by `grcan_open`.	
Param	*msg* [IN] Pointer First CAN messages to transmit	
Param	*count* [IN] Integer Total number of CAN messages to transmit.	
Return	int. Status	
	Value	Description
	>=0	Number of CAN messages transmitted. This can be less than the *count* parameter.
	GRCAN_RET_INVARG	Invalid argument: *count* parameter is less than one or the *msg* parameter is NULL.
	GRCAN_RET_NOTSTARTED	Driver is not in started mode or device is configured as silent. Nothing done.
	GRCAN_RET_BUSOFF	A write was interrupted by a bus-off error. Device has left started mode.
	GRCAN_RET_AHBERR	Similar to BUSOFF, but was caused by AHB error.

7.8.3. Reception

CAN messages are received using the `grcan_read()` function for GRCAN and `grcanfd_read()` for GRCANFD. An example is shown below:

```
enum { NUM_MSG = 5 };
struct grcan_canmsg rx_msgs[NUM_MSG];

len = grcan_read(d, &rx_msgs[0], NUM_MSG);
```

The requested number of CAN messages to be read is given in the third argument and messages are stored in *rx_msgs*.

The actual number of CAN messages received is returned by the function on success. The function will fail and return a *GRCAN_RET_* value if a NULL buffer pointer is passed, buffer length is invalid or if the CAN core is not started.

The receive operation is non-blocking: the function will return immediately with the number of messages received. If no message was available then 0 is returned.

Table 7.19. `grcan_read` function declaration

Proto	`int grcan_read(struct grcan_priv *d, struct grcan_canmsg *msg, size_t count)`	
About	Receive CAN messages Multiple CAN messages can be received in one call.	
Param	*`d`* [IN] Pointer Device identifier. Returned by `grcan_open`.	
Param	*`msg`* [IN] Pointer Buffer for received messages	
Param	*`count`* [IN] Integer Number of CAN messages to receive.	
Return	int. Status	
	Value	Description
	>=0	Number of CAN messages received. This can be less than the *`count`* parameter.
	GRCAN_RET_INVARG	Invalid argument: *`count`* parameter is less than one or the *`msg`* parameter is NULL.
	GRCAN_RET_NOTSTARTED	Driver is not in started mode. Nothing done.
	GRCAN_RET_BUSOFF	A read was interrupted by a bus-off error. Device has left started mode.
	GRCAN_RET_AHBERR	Similar to BUSOFF, but was caused by AHB Error.

Table 7.20. `grcanfd_read` function declaration

Proto	`int grcanfd_read(struct grcan_priv *d, struct grcan_canfdmsg *msg, size_t count)`	
About	Receive CAN-FD messages Multiple CAN messages can be received in one call.	
Param	*`d`* [IN] Pointer Device identifier. Returned by `grcan_open`.	
Param	*`msg`* [IN] Pointer Buffer for received messages	
Param	*`count`* [IN] Integer Number of CAN messages to receive.	
Return	int. Status	
	Value	Description
	>=0	Number of CAN messages received. This can be less than the *`count`* parameter.
	GRCAN_RET_INVARG	Invalid argument: *`count`* parameter is less than one or the *`msg`* parameter is NULL.
	GRCAN_RET_NOTSTARTED	Driver is not in started mode. Nothing done.

	GRCAN_RET_BUSOFF	A read was interrupted by a bus-off error. Device has left started mode.
	GRCAN_RET_AHBERR	Similar to BUSOFF, but was caused by AHB Error.

7.8.4. Bus-off recovery

If either `grcan_write()`/`grcanfd_write()` or `grcan_read()`/`grcanfd_read()` returns *`GRCAN_RET_BUSOFF`*, then a bus-off condition was detected and the driver has entered STATE_BUSOFF mode. To continue using the driver, the user shall call `grcan_stop()` followed by `grcan_start()` to re-enter started mode.

7.8.5. AHB error recovery

Similar to the bus-off condition, an AHB error condition can be caused by the GRCAN DMA. The driver will enter STATE_AHBERR and the recovery procedure is the same as for bus-off.

7.9. Interrupt API

The GRCAN driver has its own interrupt service routine which may be engaged when the driver is in the started state. The main purpose of this ISR is to perform error-handling and to make sure the driver has an up-to-date view of bus errors. It also handles error conditions, statistics and sometimes transitions the driver out from the started the state. Actual CAN message RX and TX is done with DMA and is not controlled by the ISR.

The function grcan_set_isr() can be used to install a custom function which is called from the GRCAN driver ISR. A call to the callback will be done from the ISR context. Note that GRCAN driver functions should not be called from this callback since it may conflict with concurrent calls in non-interrupt context.

Table 7.21. `grcan_set_isr` function declaration

Proto	`void grcan_set_isr(struct grcan_priv *d, int (*isr)(struct grcan_priv *priv, void *data), void *data)`
About	Set user Interrupt Service Routine (ISR) callback function The *`isr`* parameter is the user callback function to be called from the GRCAN ISR. Only one callback can be registered at a time. A second call to `grcan_set_isr` replaces the previously registered callback. If *`isr`* is NULL, then no user callback will be called from the driver ISR. Parameter *`priv`* of the callback is the driver device handle. The *`data`* parameter is passed to the user callback `isr`. It may be NULL.
Param	*`d`* [IN] pointer Device handle returned by `grcan_open`.
Param	*`isr`* [IN] pointer User callback function as described above. If *`isr`* is NULL then the callback is uninstalled, but the GRCAN ISR is still active.
Param	*`data`* [IN] pointer Data to pass to the user callback. It may be NULL.
Return	None.

The GRCAN driver functions are in general not re-entrant for the same device context (struct grcan_priv). That is a driver design choice to avoid extensive locking to protect driver software state.

7.9.1. Interrupt generation

CAN RX and TX interrupts are not generated by default. The user can control generation of RX and TX interrupts using the functions `grcan_txint` and `grcan_rxint`.

Table 7.22. `grcan_txint` function declaration

Proto	`int grcan_txint(struct grcan_priv *d, int n)`
About	Generate TX interrupt The parameter *n* specifies which events generate CAN TX interrupts: • 0: never (default) • 1: every CAN message transmitted • -1: When all messages have been transmitted
Param	*d* [IN] Pointer Device identifier. Returned by `grcan_open`.
Param	*n* [IN] Integer Specifies condition for generating TX interrupt.
Return	int. 0

Table 7.23. `grcan_rxint` function declaration

Proto	`int grcan_rxint(struct grcan_priv *d, int n)`
About	Generate RX interrupt The parameter *n* specifies which events generate CAN RX interrupts: • 0: never (default) • 1: every CAN message transmitted • -1: When RX buffer is full
Param	*d* [IN] Pointer Device identifier. Returned by `grcan_open`.
Param	*n* [IN] Integer Specifies condition for generating RX interrupt.
Return	int. 0

8. SPI driver

8.1. Introduction

This section describes the driver used to control the GRLIB SPICTRL device for SPI master operation.

8.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 5.

Table 8.1. Driver registration functions

Registration method	Function
Register one device	`spi_register()`
Register many devices	`spi_init()`

8.3. Opening and closing device

A SPICTRL 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 `spi_dev_count`. A particular device can be opened using `spi_open` and closed `spi_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 `osal_ldstub` from the OSAL. Protection is used by all SPICTRL devices on opening and closing.

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

- SPICTRL device I/O registers are initialized, including clearing the event register and masking all interrupts.
- The core is disabled (to allow configuration).
- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the same device.

Table 8.2. `spi_dev_count` function declaration

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

Table 8.3. `spi_open` function declaration

Proto	`struct spi_priv *spi_open(int dev_no)`	
About	Opens a SPICTRL device. The SPICTRL 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 order registered to the driver. Must be equal to or greater than zero, and smaller than that returned by `spi_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device is already open, or invalid *`dev_no`* parameter.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which SPICTRL device.

Table 8.4. `spi_close` function declaration

Proto	`int spi_close(struct spi_priv *priv)`
About	Closes a previously opened device.
Param	*`d`* [IN] pointer

	Device identifier. Returned from `spi_open`.	
Return	int.	
	Value	Description
	DRV_OK	Successfully closed device.

8.4. Status service

SPI controller event status can be read by calling the `spi_get_event` function. It returns a copy of the SPI controller event register which can be used for determining if a transfer has completed or if more data shall be written to or read. Bits in the event register can be cleared by calling `spi_clear_event`.

Table 8.5. `spi_get_event` function declaration

Proto	`uint32_t spi_get_event(struct spi_priv *priv)`
About	Get event register value Bits in the event register can be cleared by calling `spi_clear_event`.
Param	*`d`* [IN] pointer Device handle returned by `spi_open`.
Return	uint32_t. Current value of the SPI event register. Register definitions for the SPICTRL event register are available in the file `include/regs/spictrl-regs.h`. The relevant defines are prefixed with *`SPICTRL_EVENT_`*.

Table 8.6. `spi_clear_event` function declaration

Proto	`void spi_clear_event(struct spi_priv *priv, uint32_t event)`
About	Clear bits in the event register
Param	*`d`* [IN] pointer Device handle returned by `spi_open`.
Param	*`event`* [IN] uint32_t Mask of bits to clear in the SPI event register. Register definitions for the SPICTRL event register are available in the file `include/regs/spictrl-regs.h`. The relevant defines are prefixed with *`SPICTRL_EVENT_`*.
Return	None.

8.5. Transfer Configuration

The SPI driver allows for configuring the SPI controller settings between transfers. This is useful when multiple SPI slaves are attached to the same SPICTRL device, and the slaves have different timing and transfer requirements. In this case, one configuration record can be associated with each slave device.

Interrupts can be enabled for transfers by configuring the SPI controller event mask register via the configuration service. This allows for user notification of when the transmit queue is empty or when the receive queue is non-empty.

The driver supports reconfiguration of the SPI controller at any time between calls to `spi_stop` and `spi_start`.

```
struct spi_config {
        unsigned int freq;              /* SPI clock frequency, Hz */
        int mode;                       /* SPI mode */
        enum spi_wordlen wordlen;       /* SPI Word length */
        int intmask;                    /* SPI controller interrupt mask */
        int msb_first;                  /* If true then send MSb first, else LSb. */
        int sync;                       /* Synchronous TX/RX mode */
        uint32_t aslave;                /* Automatic slave select, active high mask */
```

```
        int clock_gap;                  /* MODE.CG */
        int tac;                        /* Toggle automatic slave select during clock gap */
        int aseldel;                    /* Automatic slave select delay */
        int igsel;                      /* Ignore SPISEL input */
};
```

Table 8.7. spi_config data structure declaration

freq	The SPI clock frequency in Hz. Used to calculate values for the hardware registers controlling SPICLK.	
mode	SPI mode 0, 1, 2, or 3	
wordlen	Word length. Must be one of the following values:	
	Value	Description
	SPI_WORDLEN_4	4 bit word length
	SPI_WORDLEN_5	5 bit word length
	SPI_WORDLEN_6	6 bit word length
	SPI_WORDLEN_7	7 bit word length
	SPI_WORDLEN_8	8 bit word length
	SPI_WORDLEN_9	9 bit word length
	SPI_WORDLEN_10	10 bit word length
	SPI_WORDLEN_11	11 bit word length
	SPI_WORDLEN_12	12 bit word length
	SPI_WORDLEN_13	13 bit word length
	SPI_WORDLEN_14	14 bit word length
	SPI_WORDLEN_15	15 bit word length
	SPI_WORDLEN_16	16 bit word length
	SPI_WORDLEN_32	32 bit word length
intmask	Interrupt mask. This field is written to the SPI controller Mask register when `spi_config` is called. Register definitions for the SPI controller Mask register are available in the file `include/regs/spictrl-regs.h`. The relevant defines are prefixed with *`SPICTRL_MASK_`*.	
msb_first	If true then send MSb first, else LSb. This controls the *SPI controller Mode register* bit named *Reverse data (REV)*.	
sync	Synchronous TX/RX mode.	
	Value	Description
	0	Allow RX to overrun.
	1	Prevent RX from overrunning.
aslave	Automatic slave select, active high mask	
	Value	Description
	0	Disable automatic slave select.
	mask	This value is written, inverted, to the SPI controller automatic slave select register. In addition, automatic slave select (`ASEL`) will be enabled in the SPI controller mode register.
clock_gap	Number of SCK clock cycles to insert between consecutive words. A value between 0 and 31.	
tac	Toggle automatic slave select during clock gap	
	Value	Description
	0	Set `MODE.TAC='0'`
	1	Set `MODE.TAC='1'`

aseldel	Automatic slave select delay. A value in the range 0..3 which is written to `MODE.ASELDEL`.	
igsel	Ignore SPISEL input	
	Value	Description
	0	Set `MODE.IGSEL='0'`
	1	Set `MODE.IGSEL='1'`

Table 8.8. `spi_config` function declaration

Proto	`int spi_config(struct spi_priv *priv, struct spi_config *cfg)`	
About	Set transfer configuration in hardware. The *`cfg`* input layout is described by the spi_config data structure in Table 8.7.	
Param	*`d`* [IN] pointer Device identifier. Returned from `spi_open`.	
Param	*`cfg`* [IN] pointer Address to where the driver will read the transfer configuration from. (See Table 8.7.)	
Return	int.	
	Value	Description
	DRV_OK	Successfully configured device.
	DRV_FAIL	Invalid word length or frequency field in *`cfg`*. Device not configured.
	DRV_STARTED	Device is in started mode. Device not configured.

A default configuration is available in the symbol `SPI_CONFIG_DEFAULT`:

```
extern const struct spi_config SPI_CONFIG_DEFAULT;
```

It can be used to derive default parameters.

8.6. Transfer Interface

Two functions are available for performing SPI transfers. The `spi_write32` function writes words to the hardware transmit queue, and `spi_read32` reads words from the hardware receive queue. These functions never block and may return before the requested number of words have been processed. The transfer parameters set by the last call to `spi_config` are used.

For the user to determine status of the transfer queues during transfers, the `spi_status` service can be used to read out the event register. Transmit queue status is obtained by observing the Not full (NF) and Last character (LT) flags. Likewise, existence of receive data is determined by testing bits Not empty (NE). In addition, the bit Transfer in progress (TIP) can be used to determine if a transfer has completed.

For high performance transfers, or large transfers, using a custom interrupt service routine can come in handy. It can be responsible for supplying the transmit queue with data and for reading out received data to a user receive buffer. When the transfer is considered complete, the user may be informed by for example unblocking it with a semaphore or an event. As the driver usage varies heavily with the application and the connected SPI slaves, no default interrupt service routine is provided by the SPI driver.

If the user has activated interrupts at configuration, the user must install an interrupt handler prior to calling `spi_write32` and `spi_read32`.

Before the transfer functions can be used, the core must be configured with `spi_config` and enabled with `spi_start`. At end of transfers, the `spi_stop` function can be called to disabled the SPI core. Disabling the core is only needed if it shall be reconfigured.

The example below opens, configures and enables the first SPICTRL device. Then 8 words are written and 8 words are read.

```
int spi_transfers(void)
{
        struct spi_priv *device;
        int i;
        int ret;
        struct spi_config cfg;
        uint32_t txbuf[8];
        uint32_t rxbuf[8];

        ret = spi_dev_count();
        printf("%d SPICTRL devices present\n", ret);

        device = spi_open(0);
        if (!device) {
                return -1; /* Failure */
        }

        /* Base config on sane default */
        cfg = SPI_CONFIG_DEFAULT;
        cfg.freq = 125 * 1000;
        cfg.mode = 1;
        cfg.wordlen = SPI_WORDLEN_8;
        ret = spi_config(device, &cfg);
        if (DRV_OK != ret) {
                return -1;
        }

        spi_start();
        i = 0;
        do {
                i += spi_write32(device, &txbuf[i], 8-i);
        } while (i<8);
        i = 0;
        do {
                i += spi_read32(device, &rxbuf[i], 8-i);
        } while (i<8);
        spi_stop();

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

Table 8.9. `spi_start` function declaration

Proto	`int spi_start(struct spi_priv *priv)`	
About	Start SPI device. The SPICTRL core is enabled.	
Param	*d* [IN] pointer Device handle returned by `spi_open`.	
Return	int.	
	Value	Description
	DRV_OK	Device was started by the function call.
	DRV_STARTED	Device already in started mode. Nothing performed.

Table 8.10. `spi_stop` function declaration

Proto	`int spi_stop(struct spi_priv *priv)`	
About	Stop SPI device. The SPICTRL core is disabled.	
Param	*d* [IN] pointer Device handle returned by `spi_open`.	
Return	int.	
	Value	Description
	DRV_OK	Success

Table 8.11. `spi_write32` function declaration

Proto	`int spi_write32(struct spi_priv *priv, const uint32_t *txbuf, int count)`
About	Write words to SPICTRL transmit queue.

	The function tries to write *count* words of the configured word length to the transmit queue. Transmission data is indicated by *txbuf*. Each word is represented by an uint32_t, regardless of configured word length. Words in *txbuf* shall be represented with its LSB at bit 0. If *txbuf* is NULL then zero valued bits will be shifted out on MOSI. The function returns as soon as the transmit queue is full or the requested number of words have been installed. This function never blocks. Transfer properties are set with the the function spi_config.
Param	*d* [IN] pointer Device handle returned by spi_open.
Param	*txbuf* [IN] pointer Transmit data. If *txbuf* is NULL then zero valued words are shifted out.
Param	*count* [IN] Integer Number of words to transmit
Return	int. Number of words written to transmit queue, zero if none.

Table 8.12. spi_read32 function declaration

Proto	`int spi_read32(struct spi_priv *priv, uint32_t *rxbuf, int count)`
About	Read words from SPICTRL receive queue. The function tries to read *count* words of the configured word length from the receive queue. Received data is written to the location *rxbuf*. Each word is represented by an uint32_t, regardless of configured word length. Words stored in *rxbuf* are represented with its LSB at bit 0. If *rxbuf* is NULL then the MISO bits are not stored. The function returns as soon as the receive queue is empty or the requested number of words have been read. This function never blocks. Transfer properties are set with the the function spi_config.
Param	*d* [IN] pointer Device handle returned by spi_open.
Param	*rxbuf* [OUT] pointer Received data. Can be NULL to ignore shifted in data.
Param	*count* [IN] Integer Number of words to receive
Return	int. Number of words read from receive queue, zero if none.

8.7. Synchronous TX/RX mode

The SPI configuration option *cfg.sync* is used to determine the behaviour when an spi_write32 operation would cause the SPI receive queue to become full. The *sync* option is set and remembered when the SPI driver is configured using spi_config.

When *cfg.sync=0*, calls to spi_write32 will write words to the SPI transmit queue as long as there is room in the SPI transmit queue. The receive queue may overrun. It is up to the driver user to empty the SPI receive queue. Typically this involves user knowledge of how many SPI words are outstanding and restricts calling spi_write32 to when it will not cause the receive queue to overrun. One scenario is when the SPI slave is an output device, only capable of receiving commands but never sends anything back to the SPI master.

If *cfg.sync=1*, then calls to spi_write32 will only write words to the SPI transmit queue when it is guaranteed that the receive queue will not overrun. This relaxes the restrictions on how calls to spi_write32 and

spi_read32 can be combined. It means that the user does not have to maintain the number of outstanding words and the receive queue will never overrun.

For both settings of the *cfg.sync* option, the spi_write32 function writes at most *count* words to the transmit queue and returns the number of words actually written. The difference is when spi_write32 is allowed to write to the queue.

8.8. Slave select

When performing SPI transfers, the user may want to select and deselect SPI slaves. This can be done with the the function spi_slave_select. Another option is to use a dedicated GPIO signal.

Table 8.13. spi_slave_select function declaration

Proto	int spi_slave_select(struct spi_priv *priv, uint32_t mask)	
About	Select SPI slave This function writes the inverted value of *slavemask* parameter to the SPICTRL SLVSEL register. This function shall not be called when a transfer is in progress.	
Param	*d* [IN] pointer Device identifier. Returned from spi_open.	
Param	*mask* [IN] uint32_t Slave mask	
Return	int.	
	Value	Description
	DRV_OK	Success
	DRV_NOIMPL	Slave select not available in SPICTRL or *mask* out of range.
	DRV_WOULDBLOCK	Transfer in progress

The driver functions spi_read32() and spi_write32() do not automatically perform slave select.

8.9. Restrictions

The SPI driver is designed to operate each opened device in one task only. One or more SPI devices can be opened and operated by one task, but multiple tasks can not operate on the same SPI device.

The following functions are always allowed to be called from any task:

- spi_dev_count
- spi_open

The following functions are allowed to be called from an ISR.

- spi_get_event
- spi_clear_event

9. AHB Status Register driver

9.1. Introduction

This section describes the driver used to control the AHBSTAT device, commonly known as the *AHB status register*.

9.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 5.

Table 9.1. Driver registration functions

Registration method	Function
Register one device	`ahbstat_register()`
Register many devices	`ahbstat_init()`

9.3. Opening and closing device

An AHBSTAT 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 `ahbstat_dev_count`. A particular device can be opened using `ahbstat_open` and closed `ahbstat_close`. The functions are described below.

When opened, the device can not be reopened unless the device is closed first. When opening the device it is marked opened by the driver. This procedure is thread-safe by protecting from other threads by using `osal_ldstub` from the OSAL. Protection is used by the AHBSTAT device on opening and closing.

During opening of an AHBSTAT device the following steps are taken:

- AHB status register is initialized to start monitoring AMBA AHB bus transactions and correctable errors.
- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the device.

Table 9.2. `ahbstat_dev_count` function declaration

Proto	`int ahbstat_dev_count(void)`
About	Retrieve number of AHBSTAT devices registered to the driver.
Return	int. Number of AHBSTAT devices registered to driver, zero if none.

Table 9.3. `ahbstat_open` function declaration

Proto	`struct ahbstat_priv *ahbstat_open(int dev_no)`	
About	Opens an AHBSTAT device. The AHBSTAT 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 order registered to the driver. Must be equal or greater than zero, and smaller than that returned by `ahbstat_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device is already open, or invalid *`dev_no`* parameter.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies the AHBSTAT device.
Notes	The AHBSTAT ISR is not installed by `ahbstat_open`.	

Table 9.4. `ahbstat_close` function declaration

Proto	`int ahbstat_close(struct ahbstat_priv *d)`
About	Closes a previously opened device.

	If the AHB statu register interrupt service routine has been installed, it will be uninstalled by the close operation.
Param	*d* [IN] pointer Device handle returned by `ahbstat_open`.
Return	int.

Value	Description
DRV_OK	Successfully closed device.
others	Device closed, but failed to unregister interrupt handler.

9.4. Register interface

The AHB status registers base address can be retrieved using the `ahbstat_get_regs` function. Registers and bit definitions are provided in the C header file `drv/regs/ahbstat.h`. Individual bits are described in the GRLIB IP Core User's Manual (GRIP).

Table 9.5. `ahbstat_get_regs` function declaration

Proto	`volatile struct ahbstat_regs *ahbstat_get_regs(struct ahbstat_priv *d)`
About	Get AHBSTAT registers base address Register definitions for AHBSTAT are provided by the header file `drv/regs/ahbstat.h`.
Param	*d* [IN] pointer Device handle returned by `ahbstat_open`.
Return	Pointer. Address of AHBSTAT register area.

9.5. Interrupt service routine

An interrupt service routine is provided by the driver which is installed by calling the driver function `ahbstat_set_user`. The user can provide a callback function which is called by the interrupt routine, using function. When a user callback is installed, the drivers interrupt routine will re-enable bus monitoring only if the user callback returns 0. If the user callback returns a value other than 0, then the callback itself should re-enable AHBSTAT monitoring by clearing the `NE` bit. The callback is called with a custom argument as selected by `ahbstat_set_user`.

The example below defines and enables an ISR callback which rewrites the failing location in case of correctable error.

```
#include <drv/ahbstat.h>
#include <drv/regs/ahbstat.h>

volatile int user_ncerr = 0;

int user(
  volatile struct ahbstat_regs *regs,
  uint32_t status,
  uint32_t failing_address,
  void *userdata
)
{
  if (!(status & AHBSTAT_STS_CE)) {
    /* Not correctable so this callback can't handle it. */
    return 0;
  }
  int *ncerr;
  ncerr = (int *) userdata;
  (*ncerr)++;

  volatile uint32_t *data = (volatile uint32_t *) failing_address;
  uint32_t tmp;

  /* Read and write back */
  tmp = *data;
  *data = tmp;
```

```
  /* Reenable AHBSTAT probing */
  regs->status = 0;

  /* Returns 1 to prevent driver ISR to reenable AHBSTAT probing */
  return 1;
}

int user_example(void)
{
    const int DEVNO = 0;
    struct ahbstat_priv *device;
    int ret;

    device = ahbstat_open(DEVNO);
    if (NULL == device) {
      return -1; /* Failure */
    }

    ret = ahbstat_set_user(device, user, (void *) &user_ncerr);
    if (DRV_OK != ret) {
      return -2; /* Failure */
    }

    /* Force correctable errors etc... */
    [...]

    printf("Number of correctable errors detected and corrected: %d\n", user_ncerr);

    ret = ahbstat_close(device);
    if (DRV_OK != ret) {
      return -3; /* Failure */
    }
    return 0; /* success */
}
```

Table 9.6. `ahbstat_set_user` function declaration

Proto	`int ahbstat_set_user(struct ahbstat_priv *d, int (*userhandler)(volatile struct ahbstat_regs *regs, uint32_t status, uint32_t failing_address, void *userarg), void *userarg)`
About	Install the AHBSTAT ISR and set ISR user callback function. The *`userhandler`* parameter is the user callback function to be called from the AHBSTAT ISR. The callback is called by the AHBSTAT ISR only if the has checked that the `NE` status bit is 1. Only one callback can be registered at a time. A second call to `ahbstat_set_user` replaces the previously registered callback. If *`userhandler`* is NULL, then the AHBSTAT ISR is uninstalled. Parameter *`regs`* of the callback is the register base address of the AHBSTAT core. Parameter *`status`* of the callback is an unmodified copy of the AHBSTAT status register at entry to drivers interrupt routine. The *`failing_address`* parameter of the callback is a copy of the AHBSTAT failing address register at entry to the interrupt routine. If the callback returns 0, then the driver interrupt routine will reenable AHBSTAT by clearing the status register. Otherwise the status register is not touched by the interrupt routine after callback returns. The *`userarg`* parameter is passed to the user callback `userhandler`. It may be NULL.
Param	*`d`* [IN] pointer Device handle returned by `ahbstat_open`.
Param	*`userhandler`* [IN] pointer User callback function as described above. If *`userhandler`* is NULL then the callback is uninstalled, but the AHBSTAT ISR is still active.
Param	*`userdata`* [IN] pointer Data to pass to the user callback. It may be NULL.

Return	int. DRV_OK on success, else != DRV_OK if ISR install failed.
Notes	The AHBSTAT ISR can not be uninstalled once installed. However, the user handler can be disabled by calling `ahbstat_set_user` with *`userhandler`* set to NULL.

10. Clock gating unit driver

10.1. Introduction

This section describes the driver used to control the GRLIB clock gating unit, also known as CLKGATE or GR-CLKGATE.

10.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 5.

Table 10.1. Driver registration functions

Registration method	Function
Register one device	`clkgate_register()`
Register many devices	`clkgate_init()`

10.3. Opening and closing device

An 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 `clkgate_dev_count`. A particular device can be opened using `clkgate_open` and closed `clkgate_close`. The functions are described below.

When opened, the device can not be reopened unless the device is closed first. When opening the device it is marked opened by the driver. This procedure is thread-safe by protecting from other threads by using `osal_ldstub` from the OSAL. Protection is used by the driver on opening and closing.

During opening of a clock gating unit, the following steps are performed:

- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the device.

Table 10.2. `clkgate_dev_count` function declaration

Proto	`int clkgate_dev_count(void)`
About	Retrieve number of clock gating devices registered to the driver.
Return	int. Number of devices registered to driver, zero if none.

Table 10.3. `clkgate_open` function declaration

Proto	`struct clkgate_priv *clkgate_open(int dev_no)`	
About	Opens an clock gating unit device, identified by index. The returned value is used as input argument to all functions operating on the device. This function does not change any device state.	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the order registered to the driver. Must be equal or greater than zero, and smaller than that returned by `clkgate_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device is already open, or invalid *`dev_no`* parameter.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies the clock gating unit.

Table 10.4. `clkgate_close` function declaration

Proto	`int clkgate_close(struct clkgate_priv *d)`
About	Closes a previously opened device.

	This function does not change any device state.
Param	*d* [IN] pointer Device handle returned by `clkgate_open`.
Return	int.

Value	Description
DRV_OK	Successfully closed device.
others	Device closed, but failed to unregister interrupt handler.

10.4. Operation

Each core supported by the clock gating unit can be individually clock gated or enabled by the function `clkgate_gate` and `clkgate_enable`. The sequences performed by these functions are identical to the gate and enable procedures described in component User's Manual, Clock Gating Unit section.

Core to bit mappings are defined in the C header file `drv/regs/clkgate_bits.h` with names prefixed by `CLKGATE_<component>_`. Any number of the defines can be use (OR:ed) together when calling the driver functions.

A core which is enabled with `clkgate_enable` will also be reset.

The driver does not arbitrate for the device. Protecting the driver from concurrent calls can be done on application level if needed.

The example below, applicable to GR740, gates all cores and then enables the SpaceWire subsystem and the second GRETH core.

```
#include <drv/clkgate.h>

int clkgate_example(struct clkgate_priv *d)
{
  int ret;

  /* Gate all cores. */
  ret = clkgate_gate(d, CLKGATE_GR740_ALL);
  if (DRV_OK != ret) {
    return ret;
  }

  /* Enable and reset SpaceWire, GRETH1 */
  ret = clkgate_enable(d, CLKGATE_GR740_GRSPW2 | CLKGATE_GR740_GRETH);
  if (DRV_OK != ret) {
    return ret;
  }

  return 0; /* success */
}
```

Table 10.5. `clkgate_gate` function declaration

Proto	`int clkgate_gate(struct clkgate_priv *d, uint32_t coremask)`
About	Gate the clock for selected cores. Cores to gate are selected with the *`coremask`* parameter with values `CLKGATE_*` as defined in the file `include/clkgate.h`. Multiple cores can be gated at the same time by OR:ing these values together. To gate all component cores supporting clock gating, the mask `CLKGATE_<component>_ALL` can be used. The cores identified as *`coremask`* will be held in reset with its input clock disabled.
Param	*d* [IN] pointer Device handle returned by `clkgate_open`.
Param	*`coremask`* [IN] uint32_t Bitmask representing the cores to operate on. (Values are `CLKGATE_*`.)

Return	int. DRV_OK

Table 10.6. `clkgate_enable` function declaration

Proto	`int clkgate_enable(struct clkgate_priv *d, uint32_t coremask)`
About	Enable the clock and reset selected cores. Cores to enable are selected with the *`coremask`* parameter with values `CLKGATE_*` as defined in the file `include/clkgate.h`. Multiple cores can be enabled at the same time by OR:ing these values together.
Param	*`d`* [IN] pointer Device handle returned by `clkgate_open`.
Param	*`coremask`* [IN] uint32_t Bitmask representing the cores to operate on. (Values are `CLKGATE_*`.)
Return	int. DRV_OK

10.5. Core reset

A core can be reset by calling `clkgate_gate()` followed by `clkgate_enable()` with the same *`coremask`* parameter. For example:

```
void clkgate_reset(struct clkgate_priv *priv, uint32_t coremask)
{
        clkgate_gate(priv, coremask);
        clkgate_enable(priv, coremask);
}
```

10.6. Probe clock gating status

A function is available to read the current state of the clock gating unit registers. It provides the caller with information on which cores are gated and which are enabled.

Table 10.7. `clkgate_status` function declaration

Proto	`int clkgate_status(struct clkgate_priv *d, uint32_t *enabled, uint32_t *disabled)`
About	Get enable status of cores The function determines enabled and disbled state by reading the clock gating unit registers.
Param	*`d`* [IN] pointer Device identifier. Returned from `clkgate_open`.
Param	*`enabled`* [IN] Pointer Output mask of cores which are enabled.
Param	*`disabled`* [IN] Pointer Output mask of cores which are disabled.
Return	uint32_t. The register content (before *`newval`* value is written).

10.7. CPU override

The driver provides an interface to control the clock gating unit *CPU/FPU override register*, available in some implementations.

Table 10.8. `clkgate_override` function declaration

Proto	`uint32_t clkgate_override(struct clkgate_priv *d, int set, uint32_t newval)`
About	Get/set CPU/FPU override register

	The function returns and optionally sets the value of the register. If *set* is 0 then nothing will be written to the register, else the register is set to the value of the *newval* parameter.
Param	*d* [IN] pointer Device identifier. Returned from clkgate_open.
Param	*set* [IN] Integer Determines if register shall be updated with *newval*.
	0 — Do not write register.
	1 — Write value of *newval* to register.
Param	*newval* [IN] Integer New value
Return	uint32_t. The register content (before *newval* value is written).
Notes	The CPU/FPU override functionality is not available in all implementations. See the component datasheet for more information.

11. GR1553B Driver

11.1. Introduction

This document describes the device drivers specific to the GRLIB GR1553B core. The Remote Terminal(RT), Bus Monitor (BM and Bus Controller (BC) functionality are supported by the driver. Device discovery and resource sharing are commonly controlled by the GR1553B driver described in this chapter. Each 1553 mode is supported by a separate driver, the drivers are documented in separate chapters.

This section gives an brief introduction to the GRLIB GR1553B device allocation driver used internally by the BC, BM and RT device drivers. This driver controls the GR1553B device regardless of interfaces supported (BC, RT and/or BM). The device can be located at an on-chip AMBA or an AMBA-over-PCI bus. The driver provides an interface for the BC, RT and BM drivers.

Since the different interfaces (BC, BM and RT) are accessed from the same register interface on one core, the APB device must be shared among the BC, BM and RT drivers. The GR1553B driver provides an easy function interface that allows the APB device to be shared safely between the BC, BM and RT device drivers.

Any combination of interface functionality is supported, but the RT and BC functionality cannot be used simultaneously (limited by hardware).

The interface towards to the BC, BM and RT drivers is used internally by the device drivers and is not documented here. See respective driver for an interface description.

11.1.1. Considerations and limitations

Note that the following items must be taken into consideration when using the GR1553B drivers:

- The driver uses only Physical addressing, i.e it does not do MMU translation or memory mapping for the user. The user is responsible for mapping DMA memory buffers provided to the 1553 drivers 1:1.
- Physical buffers addresses (assigned by user) must be located at non-cacheable areas or D-Cache snooping must be present in hardware. If D-cache snooping is not present the user must edit the GR1553*_READ_MEM() macros in respective driver.
- SMP locking (spin-locks) has not been implemented, it does however not mean that SMP mode can not be used. The CPU handling the IRQ (CPU0 unless configured otherwise) must be the CPU and only CPU using the driver API. Only one CPU can use respective driver API at a time.

The above restrictions should not cause any problems for the AT697 + GR-RASTA-IO (RASTA-101) systems or similar.

11.1.2. GR1553B Hardware

The GRLIB GR1553B core may support up to three modes depending on configuration, Bus Controller (BC), Remote Terminal (RT) or Bus Monitor (BM). The BC and RT functionality may not be used simultaneously, but the BM may be used together with BC or RT or separately. All three modes are supported by the driver.

Interrupts generated from BC, BM and RT result in the same system interrupt, interrupts are shared.

11.1.3. Software driver

The driver provides an interface used internally by the BC, BM and RT device drivers, see respective driver for an interface declaration. The driver sources and definitions are listed in the table below, the path is given relative to the toolchains root directory.

Table 11.1. Source Location

Filename	Description
src/libdrv/src/gr1553b/gr1553b.c	GR1553B Driver source
src/libdrv/src/include/gr1553b.h	GR1553B Driver interface declaration

11.1.4. Driver Registration

This driver uses the driver registration mechanism described in Chapter 5.

Table 11.2. Driver registration functions

Registration method	Function
Register one device	`gr1553b_register()`
Register many devices	`gr1553b_init()`

The registration of the driver is crucial for the user to be able to access the driver application programming interfaces. The drivers use a classic C-language API.

12. GR1553B Bus Controller Driver

12.1. Introduction

This section describes the GRLIB GR1553B Bus Controller (BC) device driver interface. The driver relies on the GR1553B driver. The reader is assumed to be well acquainted with MIL-STD-1553 and the GR1553B core.

12.1.1. GR1553B Bus Controller Hardware

The GR1553B core supports any combination of the Bus Controller (BC), Bus Monitor (BM) and Remote Terminal (RT) functionality. This driver supports the BC functionality of the hardware, it can be used simultaneously with the Bus Monitor (BM) functionality. When the BM is used together with the BC, interrupts are shared between the drivers.

The three functions (BC, BM, RT) are accessed using the same register interface, but through separate registers. In order to share hardware resources between the three GR1553B drivers, the three depends on a lower level GR1553B driver, see Chapter 11.

The driver supports the on-chip AMBA bus and the AMBA-over-PCI bus.

12.1.2. Software driver

The BC driver is split in two parts, one where the driver access the hardware device and one part where the descriptors are managed. The two parts are described in two separate sections below.

Transfer and conditional descriptors are collected into a descriptor list. A descriptor list consists of a set of Major Frames, which consist of a set of Minor Frames which in turn consists of up to 32 descriptors (also called Slots). The composition of Major/Minor Frames and slots is configured by the user, and is highly dependent of application.

The Major/Minor/Slot construction can be seen as a tree, the tree does not have to be symmetrically, i.e. Major frames may contain different numbers of Minor Frames and Minor Frames may contain different numbers of Slot.

GR1553B BC descriptor lists are generated by the list API available in `gr1553bc_list.h`.

The driver provides the following services:

- Start, Stop, Pause and Resume descriptor list execution
- Synchronous and asynchronous descriptor list management
- Interrupt handling
- BC status
- Major/Minor Frame and Slot (descriptor) model of communication
- Current Descriptor (Major/Minor/Slot) Execution Indication
- Software External Trigger generation, used mainly for debugging or custom time synchronization
- Major/Minor Frame and Slot/Message ID
- Minor Frame time slot management

The driver sources and definitions are listed in the table below, the path is given relative to the extracted distribution archive.

Table 12.1. BC driver Source location

Filename	Description
src/libdrv/src/gr1553b/gr1553bc.c	GR1553B BC Driver source
src/libdrv/src/include/gr1553bc.h	GR1553B BC Driver interface declaration
src/libdrv/src/include/gr1553bc_list.h	GR1553B BC List handling interface declaration

12.1.3. Driver registration

The driver registration is handled by the GR1553B driver, see Chapter 11.

12.2. BC Device Handling

The BC device driver's main purpose is to start, stop, pause and resume the execution of descriptor lists. Lists are described in the Descriptor List section. In this section services related to direct access of BC hardware registers and Interrupt are described. The function API is declared in `gr1553bc.h`.

12.2.1. Device API

The device API consists of the functions in the table below.

Table 12.2. Device API function prototypes

Prototype	Description
`void *gr1553bc_open(int minor)`	Open a BC device by minor number. Private handle returned used in all other device API functions.
`void gr1553bc_close(void *bc)`	Close a previous opened BC device.
`int gr1553bc_start(void *bc, struct gr1553bc_list *list, struct gr1553bc_list *list_async)`	Schedule a synchronous and/or a asynchronous BC descriptor Lists for execution. This will unmask BC interrupts and start executing the first descriptor in respective List. This function can be called multiple times.
`int gr1553bc_pause(void *bc)`	Pause the synchronous List execution.
`int gr1553bc_restart(void *bc)`	Restart the synchronous List execution.
`int gr1553bc_stop(void *bc, int options)`	Stop Synchronous and/or asynchronous list.
`int gr1553bc_indication(void *bc, int async, int *mid)`	Get the current BC hardware execution position (MID) of the synchronous or asynchronous list.
`void gr1553bc_status(void *bc, struct gr1553bc_status *status)`	Get the BC hardware status and time.
`void gr1553bc_ext_trig(void *bc, int trig)`	Trigger an external trigger by writing to the BC action register.
`int gr1553bc_irq_setup(void *bc, bcirq_func_t func, void *data)`	Generic interrupt handler configuration. Handler will be called in interrupt context on errors and interrupts generated by transfer descriptors.

12.2.1.1. Data Structures

The `gr1553bc_status` data structure contains the BC hardware status sampled by the function `gr1553bc_status()`.

```
struct gr1553bc_status {
    unsigned int status;
    unsigned int time;
};
```

Table 12.3. gr1553bc_status member descriptions

Member	Description
status	BC status register
time	BC Timer register

12.2.1.2. gr1553bc_open

Opens a GR1553B BC device by device instance index. The minor number relates to the order in which a GR1553B BC device is found in the Plug&Play information. A GR1553B core which lacks BC functionality does not affect the minor number.

If a BC device is successfully opened a pointer is returned. The pointer is used internally by the GR1553B BC driver, it is used as the input parameter *bc* to all other device API functions.

If the driver failed to open the device, NULL is returned.

12.2.1.3. gr1553bc_close

Closes a previously opened BC device. This action will stop the BC hardware from processing descriptors/lists, disable BC interrupts, and free dynamically memory allocated by the driver.

12.2.1.4. gr1553bc_start

Calling this function starts the BC execution of the synchronous list and/or the asynchronous list. At least one list pointer must be non-zero to affect BC operation. The BC communication is enabled depends on list, and Interrupts are enabled.

This function can be called multiple times. If a list (of the same type) is already executing it will be replaced with the new list.

12.2.1.5. gr1553bc_pause

Pause the synchronous list. It may be resumed by `gr1553bc_resume()`. See hardware documentation.

12.2.1.6. gr1553bc_resume

Resume the synchronous list, must have been previously paused by `gr1553bc_pause()`. See hardware documentation.

12.2.1.7. gr1553bc_stop

Stop synchronous and/or asynchronous list execution. The second argument is a 2-bit bit-mask which determines the lists to stop, see table below for a description.

Table 12.4. gr1553bc_stop second argument

Member	Description
Bit 0	Set to one to stop the synchronous list.
Bit 1	Set to one to stop the asynchronous list.

12.2.1.8. gr1553bc_indication

Retrieves the current Major/Minor/Slot (MID) position executing into the location indicated by *`mid`*. The async argument determines which type of list is queried, the Synchronous (*`async`*=0) list or the Asynchronous (*`async`*=1).

Note that since the List API internally adds descriptors the indication may seem to be out of bounds.

12.2.1.9. gr1553bc_status

This function retrieves the current BC hardware status. Second argument determine where the hardware status is stored, the layout of the data stored follows the `gr1553bc_status` data structure. The data structure is described in Table 12.3.

12.2.1.10. gr1553bc_ext_trig

The BC supports an external trigger signal input which can be used to synchronize 1553 transfers. If used, the external trigger is normally generated by some kind of Time Master. A message slot may be programmed to wait for an external trigger before being executed, this feature allows the user to accurate send time synchronize messages to RTs. However, during debugging or when software needs to control the time synchronization behaviour the external trigger pulse can be generated from the BC core itself by writing the BC Action register.

This function sets the external trigger memory to one by writing the BC action register.

12.2.1.11. gr1553bc_irq_setup

Install a generic handler for BC device interrupts. The handler will be called on Errors (DMA errors etc.) resulting in interrupts or transfer descriptors resulting in interrupts. The handler is not called when an IRQ is generated by a condition descriptor. Condition descriptors have their own custom handler.

Condition descriptors are inserted into the list by user, each condition may have a custom function and data assigned to it, see `gr1553bc_slot_irq_prepare()`. Interrupts generated by condition descriptors are not handled by this function.

The third argument is custom data which will be given to the handler on interrupt.

12.3. Descriptor List Handling

The BC device driver can schedule synchronous and asynchronous lists of descriptors. The list contains a descriptor table and a software description to make certain operations possible, for example translate descriptor address into descriptor number (MID).

The BC stops execution of a list when a END-OF-LIST (EOL) marker is found. Lists may be configured to jump to the start of the list (the first descriptor) by inserting an unconditional jump descriptor. Once a descriptor list is setup the hardware may process the list without the need of software intervention. Time distribution may also be handled completely in hardware, by setting the "Wait for External Trigger" flag in a transfer descriptor the BC will wait until the external trigger is received or proceed directly if already received. See hardware manual.

12.3.1. Overview

This section describes the Descriptor List Application Programming Interface (API). It provides functionality to create and manage BC descriptor lists.

A list is built up by the following building blocks:

- Major Frame (Consists of N Minor Frames)
- Minor Frame (Consists of up to 32 1553 Slots)
- Slot (Transfer/Condition BC descriptor), also called Message Slot

The user can configure lists with different number of Major Frames, Minor Frames and slots within a Minor Frame. The List manages a strait descriptor table and a Major/Minor/Slot tree in order to easily find it's way through all descriptor created.

Each Minor frame consist of up to 32 slot and two extra slots for time management and descriptor find operations, see figure below. In the figure there are three Minor frames with three different number of slots 32, 8 and 4. The List manage time slot allocation per Minor frame, for example a minor frame may be programmed to take 8ms and when the user allocate a message slot within that Minor frame the time specified will be subtracted from the 8ms, and when the message slot is freed the time will be returned to the Minor frame again.

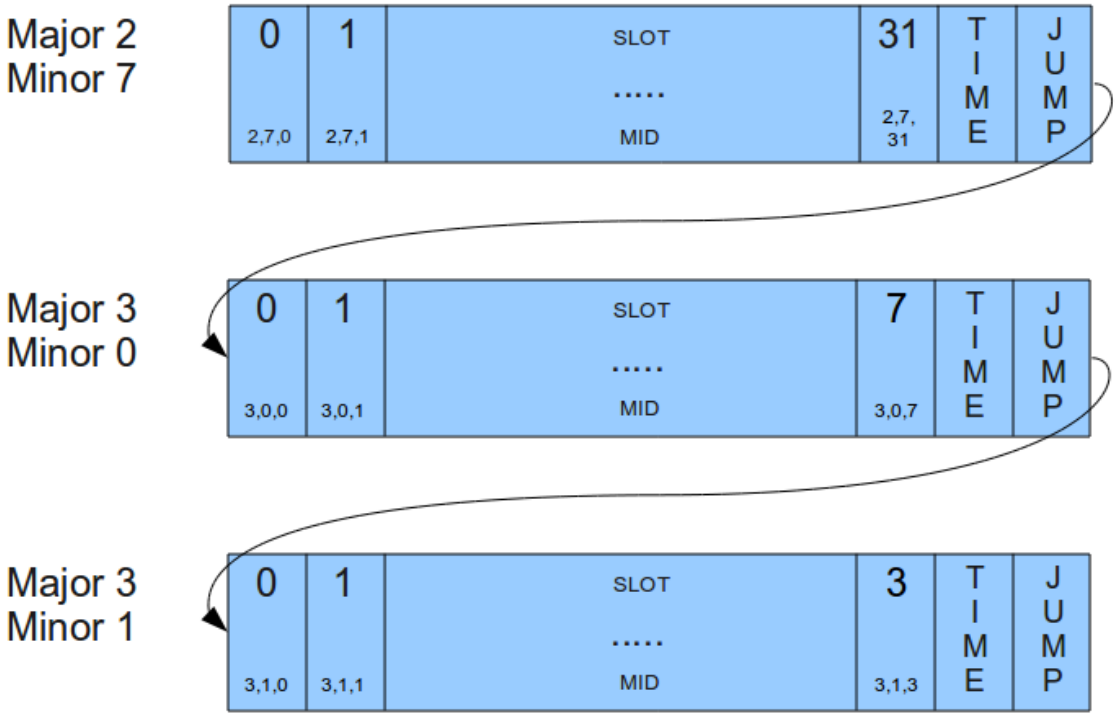

Figure 12.1. Three consecutive Minor Frames

A specific Slot [Major, Minor, Slot] is identified using a MID (Message-ID). The MID consist of three numbers Major Frame number, Minor Frame number and Slot Number. The MID is a way for the user to avoid using descriptor pointers to talk with the list API. For example a condition Slot that should jump to a message Slot can be created by knowing "MID and Jump-To-MID". When allocating a Slot (with or without time) in a List the user may specify a certain Slot or a Minor frame, when a Minor frame is given then the API will find the first free Slot as early in the Minor Frame as possible and return it to the user.

A MID can also be used to identify a certain Major Frame by setting the Minor Frame and Slot number to 0xff. A Minor Frame can be identified by setting Slot Number to 0xff.

A MID can be created using the macros in the table below.

Table 12.5. Macros for creating MID

MACRO Name	Description
GR1553BC_ID(major,minor,slot)	ID of a SLOT
GR1553BC_MINOR_ID(major,minor)	ID of a MINOR (Slot=0xff)
GR1553BC_MAJOR_ID(major)	ID of a Major (Minor=0xff,Slot=0xff)

12.3.2. Example: steps for creating a list

The typical approach when creating lists and executing it:

- gr1553bc_list_alloc(&list, MAJOR_CNT)
- gr1553bc_list_config(list, &listcfg)
- Create all Major Frames and Minor frame, for each major frame:
  1. gr1553bc_major_alloc_skel(&major, &major_minor_cfg)
  2. gr1553bc_list_set_major(list, &major, MAJOR_NUM)
- Link last and first Major Frames together:
  1. gr1553bc_list_set_major(&major7, &major0)
- gr1553bc_list_table_alloc() (Allocate Descriptor Table)
- gr1553bc_list_table_build() (Build Descriptor Table from Majors/Minors)
- Allocate and initialize Descriptors predefined before starting:

1. gr1553bc_slot_alloc(list, &MID, TIME_REQUIRED, ..)
2. gr1553bc_slot_transfer(MID, ..)

- START BC HARDWARE BY SCHEDULING ABOVE LIST
- Application operate on executing List

12.3.3. Major Frame

Consists of multiple Minor frames. A Major frame may be connected/linked with another Major frame, this will result in a Jump Slot from last Minor frame in the first Major to the first Minor in the second Major.

12.3.4. Minor Frame

Consists of up to 32 Message Slots. The services available for Minor Frames are Time-Management and Slot allocation.

Time-Management is optional and can be enabled per Minor frame. A Minor frame can be assigned a time in microseconds. The BC will not continue to the next Minor frame until the time specified has passed, the time includes the 1553 bus transfers. See the BC hardware documentation. Time is managed by adding an extra Dummy Message Slot with the time assigned to the Minor Frame. Every time a message Slot is allocated (with a certain time: Slot-Time) the Slot-Time will be subtracted from the assigned time of the Minor Frame's Dummy Message Slot. Thus, the sum of the Message Slots will always sum up to the assigned time of the Minor Frame, as configured by the user. When a Message Slot is freed, the Dummy Message Slot's Slot-Time is incremented with the freed Slot-Time. See figure below for an example where 6 Message Slots has been allocated Slot-Time in a 1 ms Time-Managed Minor Frame. Note that in the example the Slot-Time for Slot 2 is set to zero in order for Slot 3 to execute directly after Slot 2.

Figure 12.2. Time-Managed Minor Frame of 1ms

The total time of all Minor Frames in a Major Frame determines how long time the Major Frame is to be executed.

Slot allocation can be performed in two ways. A Message Slot can be allocated by identifying a specific free Slot (MID identifies a Slot) or by letting the API allocate the first free Slot in the Minor Frame (MID identifies a Minor Frame by setting Slot-ID to 0xff).

12.3.5. Slot (Descriptor)

The GR1553B BC core supports two Slot (Descriptor) Types:

- Transfer descriptor (also called Message Slot)
- Condition descriptor (Jump, unconditional-IRQ)

See the hardware manual for a detail description of a descriptor (Slot).

The BC Core is unaware of lists, it steps through executing each descriptor as the encountered, in a sequential order. Conditions resulting in jumps gives the user the ability to create more complex arrangements of buffer descriptors (BD) which is called lists here.

Transfer Descriptors (TBD) may have a time slot assigned, the BC core will wait until the time has expired before executing the next descriptor. Time slots are managed by Minor frames in the list. See Minor Frame section. A Message Slot generating a data transmission on the 1553 bus must have a valid data pointer, pointing to a location from which the BC will read or write data.

A Slot is allocated using the `gr1553bc_slot_alloc()` function, and configured by calling one of the function described in the table below. A Slot may be reconfigured later. Note that a conditional descriptor does not have a time slot, allocating a time for a conditional times slot will lead to an incorrect total time of the Minor Frame.

Table 12.6. Slot configuration

Function Name	Description
gr1553bc_slot_irq_prepare	Unconditional IRQ slot
gr1553bc_slot_jump	Unconditional jump
gr1553bc_slot_exttrig	Dummy transfer, wait for EXTERNAL-TRIGGER
gr1553bc_slot_transfer	Transfer descriptor
gr1553bc_slot_empty	Create Dummy Transfer descriptor
gr1553bc_slot_raw	Custom Descriptor handling

Existing configured Slots can be manipulated with the following functions.

Table 12.7. Slot manipulation

Function Name	Description
gr1553bc_slot_dummy	Set existing Transfer descriptor to Dummy. No 1553 bus transfer will be performed.
gr1553bc_slot_update	Update Data Pointer and/or Status of a TBD

12.3.6. Changing a scheduled BC list (during BC-runtime)

Changing a descriptor that is being executed by the BC may result in a race between hardware and software. One of the problems is that a descriptor contains multiple words, which can not be written simultaneously by the CPU. To avoid the problem one can use the INDICATION service to avoid modifying a descriptor currently in use by the BC core. The indication service tells the user which Major/Minor/ Slot is currently being executed by hardware, from that information an knowing the list layout and time slots the user may safely select which slot to modify or wait until hardware is finished.

In most cases one can do descriptor initialization in several steps to avoid race conditions. By initializing (allocating and configuring) a Slot before starting the execution of the list, one may change parts of the descriptor which are ignored by the hardware. Below is an example approach that will avoid potential races between software and hardware:

1. Initialize Descriptor as Dummy and allocated time (often done before starting/ scheduling list)
2. The list is started, as a result descriptors in the list are executed by the BC
3. Modify transfer options and data-pointers, but maintain the Dummy bit.
4. Clear the Dummy bit in one atomic data store.

12.3.7. Custom Memory Setup

For designs where dynamically memory is not an option, or the driver is used on an AMBA-over-PCI bus (where `malloc()` does not work), the API allows the user to provide custom addresses for the descriptor table and object descriptions (lists, major frames, minor frames).

Being able to configure a custom descriptor table may for example be used to save space or put the descriptor table in on-chip memory. The descriptor table is setup using the function `gr1553bc_list_table_alloc(list, CUSTOM_ADDRESS)`.

Object descriptions are normally allocated during initialization procedure by providing the API with an object configuration, for example a Major Frame configuration enables the API to dynamically allocate the software description of the Major Frame and with all it's Minor frames. Custom object allocation requires internal understanding of the List management parts of the driver, it is not described in this document.

12.3.8. Interrupt handling

There are different types of interrupts, Error IRQs, transfer IRQs and conditional IRQs. Error and transfer Interrupts are handled by the general callback function of the device driver. Conditional descriptors that cause Interrupts may be associated with a custom interrupt routine and argument.

Transfer Descriptors can be programmed to generate interrupt, and condition descriptors can be programmed to generate interrupt unconditionally (there exists other conditional types as well). When a Transfer descriptor causes

interrupt the general ISR callback of the BC driver is called to let the user handle the interrupt. Transfers descriptor IRQ is enabled by configuring the descriptor.

When a condition descriptor causes an interrupt a custom IRQ handler is called (if assigned) with a custom argument and the descriptor address. The descriptor address my be used to look up the MID of the descriptor. The API provides functions for placing unconditional IRQ points anywhere in the list. Below is an pseudo example of adding an unconditional IRQ point to a list:

```
void funcSetup()
{
    int MID;

    /* Allocate Slot for IRQ Point */
    gr1553bc_slot_alloc(&MID, TIME=0, ..);

    /* Prepare unconditional IRQ at allocated SLOT */
    gr1553bc_slot_irq_prepare(MID, funcISR, data);

    /* Enabling the IRQ may be done later during list
     * execution */
    gr1553bc_slot_irq_enable(MID);
}
void funcISR(*bd, *data)
{
    /* HANDLE ONE OR MULTIPLE DESCRIPTORS
     *(MULTIPLE IN THIS EXAMPLE): */
    int MID;

    /* Lookup MID from descriptor address */
    gr1553bc_mid_from_bd(bd, &MID, NULL);

    /* Print MID which caused the Interrupt */
    printk("IRQ ON %06x\n", MID);
}
```

12.3.9. List API

Table 12.8. List API function prototypes

Prototype	Description

`int gr1553bc_list_init(`
`  struct gr1553bc_list **list,`
`  int max_major)`
	Initialize a List description structure. First step in creating a descriptor list. This functions does not allocate any memory

`int gr1553bc_list_alloc(`
`  struct gr1553bc_list **list,`
`  int max_major)`
	Allocate and initialize a List description structure. First step in creating a descriptor list.

`void gr1553bc_list_free(` `  struct gr1553bc_list *list)`	Free a List previously allocated using `gr1553bc_list_alloc()`.

`int gr1553bc_list_config(`
`  struct gr1553bc_list *list,`
`  struct gr1553bc_list_cfg *cfg,`
`  void *bc)`
	Configure List parameters and associate it with a BC device that will execute the list later on. List parameters are used when generating descriptors.

`void gr1553bc_list_link_major(`
`  struct gr1553bc_major *major,`
`  struct gr1553bc_major *next)`
	Links two Major frames together, the Major frame indicated by next will be executed after the Major frame indicated by major. A unconditional jump is inserted to implement the linking.

`int gr1553bc_list_set_major(`
`  struct gr1553bc_list *list,`
`  struct gr1553bc_major *major,`
`  int no)`
	Assign a Major Frame a Major Frame number in a list. This will link Major (no-1) and Major (no+1) with the Major frame, the linking can be changed by calling `gr1553bc_list_link_major()` after all major frames have been assigned a number.

`int gr1553bc_minor_table_size(` `  struct gr1553bc_minor *minor)`	Calculate the size required in the descriptor table by one minor frame.
`int gr1553bc_list_table_size(` `  struct gr1553bc_list *list)`	Calculate the size required for the complete descriptor list.

`int gr1553bc_list_table_init(`
`  struct gr1553bc_list *list,`
`  void *bdtab_custom)`
	Initialize a descriptor list. The `bdtab_custom` argument can be used to assign a custom address of the descriptor list. This function does not allocate any memory.

`int gr1553bc_list_table_alloc(`
`  struct gr1553bc_list *list,`
`  void *bdtab_custom)`
	Allocate and initialize a descriptor list. The `bdtab_custom` argument can be used to assign a custom address of the descriptor list.

Prototype	Description
`void gr1553bc_list_table_free(` `  struct gr1553bc_list *list)`	Free descriptor list memory previously allocated by `gr1553bc_list_table_alloc()`.
`int gr1553bc_list_table_build(` `  struct gr1553bc_list *list)`	Build all descriptors in a descriptor list. Unused descriptors will be initialized as empty dummy descriptors. After this call descriptors can be initialized by user.

`int gr1553bc_major_init_skel(`
`  struct gr1553bc_major **major,`
`  struct gr1553bc_major_cfg *cfg)`
	Initialize a software description skeleton of a Major Frame and it's Minor Frames. This function does not allocate any memory.

`int gr1553bc_major_alloc_skel(`
`  struct gr1553bc_major **major,`
`  struct gr1553bc_major_cfg *cfg)`
	Allocate and initialize a software description skeleton of a Major Frame and it's Minor Frames.

`int gr1553bc_list_freetime(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Get total unused slot time of a Minor Frame. Only available if time management has been enabled for the Minor Frame.

`int gr1553bc_slot_alloc(`
`  struct gr1553bc_list *list,`
`  int *mid,`
`  int timeslot,`
`  union gr1553bc_bd **bd)`
	Allocate a Slot from a Minor Frame. The Slot location is identified by MID. If the MID identifies a Minor frame the first free slot is allocated within the minor frame.

`int gr1553bc_slot_free(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Return a previously allocated Slot to a Minor Frame. The slot-time is also returned.

`int gr1553bc_mid_from_bd(`
`  union gr1553bc_bd *bd,`
`  int *mid,`
`  int *async)`
	Get Slot/Message ID from descriptor address.

`union gr1553bc_bd *gr1553bc_slot_bd(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Get descriptor address from MID.

`int gr1553bc_slot_irq_prepare(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  bcirq_func_t func,`
`  void *data)`
	Prepare a condition Slot for generating interrupt. Interrupt is disabled. A custom callback function and data is assigned to Slot.

`int gr1553bc_slot_irq_enable(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Enable interrupt of a previously interrupt-prepared Slot.

`int gr1553bc_slot_irq_disable(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Disable interrupt of a previously interrupt-prepared Slot.

`int gr1553bc_slot_jump(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  uint32_t condition,`
`  int to_mid)`
	Initialize an allocated Slot, the descriptor is initialized as a conditional Jump Slot. The conditional is controlled by the third argument. The Slot jumped to is determined by the fourth argument.

`int gr1553bc_slot_exttrig(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Create a dummy transfer with the "Wait for external trigger" bit set.

`int gr1553bc_slot_transfer(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  int options,`
`  int tt,`
`  uint16_t *dptr)`
	Create a transfer descriptor.

`int gr1553bc_slot_dummy(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  unsigned int *dummy)`
	Manipulate the DUMMY bit of a transfer descriptor. Can be used to enable or disable a transfer descriptor.

`int gr1553bc_slot_empty(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Create an empty transfer descriptor, with the DUMMY bit set. The time- slot previously allocated is preserved.

`int gr1553bc_slot_update(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  uint16_t *dptr,`
`  unsigned int *stat)`
	Update a transfer descriptors data pointer and/or status field.

`int gr1553bc_slot_raw(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  unsigned int flags,`
`  uint32_t word0,`
	Custom descriptor initialization. Note that a bad initialization may break the BC driver.

Prototype	Description
uint32_t word1, uint32_t word2, uint32_t word3)	
void gr1553bc_show_list(struct gr1553bc_list *list, int options)	Print information about a descriptor list to standard out. Used for debugging.

12.3.9.1. Data structures

The `gr1553bc_major_cfg` data structure hold the configuration parameters of a Major frame and all it's Minor frames. The `gr1553bc_minor_cfg` data structure contain the configuration parameters of one Minor Frame.

```
struct gr1553bc_minor_cfg {
    int slot_cnt;
    int timeslot;
};

struct gr1553bc_major_cfg {
    int minor_cnt;
    struct gr1553bc_minor_cfg minor_cfgs[1];
};
```

Table 12.9. gr1553bc_minor_cfg member descriptions.

Member	Description
slot_cnt	Number of Slots in Minor Frame
timeslot	Total time-slot of Minor Frame [us]

Table 12.10. gr1553bc_major_cfg member descriptions.

Member	Description
minor_cnt	Number of Minor Frames in Major Frame.
minor_cfgs	Array of Minor Frame configurations. The length of the array is determined by minor_cnt.

The `gr1553bc_list_cfg` data structure hold the configuration parameters of a descriptor List. The Major and Minor Frames are configured separately. The configuration parameters are used when generating descriptor.

```
struct gr1553bc_list_cfg {
    unsigned char rt_timeout[31];
    unsigned char bc_timeout;
    int tropt_irq_on_err;
    int tropt_pause_on_err;
    int async_list;
};
```

Table 12.11. gr1553bc_list_cfg member descriptions.

Member	Description
rt_timeout	Number of us timeout tolerance per RT address. The BC has a resolution of 4us.
bc_timeout	Number of us timeout tolerance of broadcast transfers
tropt_irq_on_err	Determines if transfer descriptors should generate IRQ on transfer errors
tropt_pause_on_err	Determines if the list should be paused on transfer error
async_list	Set to non-zero if asynchronous list

12.3.9.2. gr1553bc_list_init

Initialize a List structure (no descriptors) with a maximum number of Major frames supported. The first argument is a pointer to where the newly allocated list pointer will be stored. The second argument determines the maximum number of major frames the List will be able to support.

The list is initialized according to the default configuration.

This function will not allocate any memory. Replace this function call with gr1553bc_list_alloc() if you want the driver to allocate the memory.

If a NULL pointer is provided, a negative result will be returned.

12.3.9.3. gr1553bc_list_alloc

Dynamically allocate and initialize a List structure (no descriptors) with a maximum number of Major frames supported. The first argument is a pointer to where the newly allocated list pointer will be stored. The second argument determines the maximum number of major frames the List will be able to support.

The list is initialized according to the default configuration.

If the list allocation fails, a negative result will be returned.

12.3.9.4. gr1553bc_list_free

Free a List that has been previously allocated with `gr1553bc_list_alloc()`.

12.3.9.5. gr1553bc_list_config

This function configures List parameters and associate the list with a BC device. The BC device may be used to translate addresses from CPU address to addresses the GR1553B core understand, therefore the list must not be scheduled on another BC device.

Some of the List parameters are used when generating descriptors, as global descriptor parameters. For example all transfer descriptors to a specific RT result in the same time out settings.

The first argument points to a list that is configure. The second argument points to the configuration description, the third argument identifies the BC device that the list will be scheduled on. The layout of the list configuration is described in Table 12.11.

12.3.9.6. gr1553bc_list_link_major

At the end of a Major Frame a unconditional jump to the next Major Frame is inserted by the List API. The List API assumes that a Major Frame should jump to the following Major Frame, however for the last Major Frame the user must tell the API which frame to jump to. The user may also connect Major frames in a more complex way, for example Major Frame 0 and 1 is executed only once so the last Major frame jumps to Major Frame 2.

The Major frame indicated by next will be executed after the Major frame indicated by major. A unconditional jump is inserted to implement the linking.

12.3.9.7. gr1553bc_list_set_major

Major Frames are associated with a number, a Major Frame Number. This function creates an association between a Frame and a Number, all Major Frames must be assigned a number within a List.

The function will link Major[no-1] and Major[no+1] with the Major frame, the linking can be changed by calling `gr1553bc_list_link_major()` after all major frames have been assigned a number.

12.3.9.8. gr1553bc_minor_table_size

This function is used internally by the List API, however it can also be used in an application to calculate the space required by descriptors of a Minor Frame.

The total size of all descriptors in one Minor Frame (in number of bytes) is returned. Descriptors added internally by the List API are also counted.

12.3.9.9. gr1553bc_list_table_size

This function is used internally by the List API, however it can also be used in an application to calculate the total space required by all descriptors of a List.

The total descriptor size of all Major/Minor Frames of the list (in number of bytes) is returned.

12.3.9.10. gr1553bc_list_table_init

The List is initialized with the new descriptor table, i.e. the software's internal representation is initialized. The descriptors themselves are not initialized.

The second argument *bdtab_custom* is the memory area. If NULL the function will fail, if non-zero the value will be taken as the base descriptor address. If bit zero is set the address is assumed to be readable by the GR1553B core, if bit zero is cleared the address is assumed to be readable by the CPU and translated for the GR1553B core. Bit zero makes sense to use on a GR1553B core located on a AMBA-over-PCI bus.

This function will not allocate any memory. Replace this function call with gr1553bc_list_table_alloc() if you want the driver to allocate the memory.

12.3.9.11. gr1553bc_list_table_alloc

This function allocates all descriptors needed by a List, either dynamically or by a user provided address. The List is initialized with the new descriptor table, i.e. the software's internal representation is initialized. The descriptors themselves are not initialized.

The second argument *bdtab_custom* determines the allocation method. If NULL the API will allocate memory using `malloc()`, if non-zero the value will be taken as the base descriptor address. If bit zero is set the address is assumed to be readable by the GR1553B core, if bit zero is cleared the address is assumed to be readable by the CPU and translated for the GR1553B core. Bit zero makes sense to use on a GR1553B core located on a AMBA-over-PCI bus.

12.3.9.12. gr1553bc_list_table_free

Free previously allocated descriptor table memory.

12.3.9.13. gr1553bc_list_table_build

This function builds all descriptors in a descriptor list. Unused descriptors will be initialized as empty dummy descriptors. Jumps between Minor and Major Frames will be created according to user configuration.

After this call descriptors can be initialized by user.

12.3.9.14. gr1553bc_major_init_skel

Initialize a Major Frame and it's Minor Frames according to the configuration pointed to by the second argument.

This function will not allocate any memory. Replace this function call with gr155bc_major_alloc_skel() if you want the driver to allocate the memory.

The configuration of the Major Frame is determined by the gr1553bc_major_cfg structure, described in Table 12.10.

On success zero is returned, on failure a negative value is returned.

12.3.9.15. gr1553bc_major_alloc_skel

Allocate and initialize a Major Frame and it's Minor Frames according to the configuration pointed to by the second argument.

The pointer to the allocated Major Frame is stored into the location pointed to by the major argument.

The configuration of the Major Frame is determined by the gr1553bc_major_cfg structure, described in Table 12.10.

On success zero is returned, on failure a negative value is returned.

12.3.9.16. gr1553bc_list_freetime

Minor Frames can be configured to handle time slot allocation. This function returns the number of microseconds that is left/unused. The second argument mid determines which Minor Frame.

12.3.9.17. gr1553bc_slot_alloc

Allocate a Slot from a Minor Frame. The Slot location is identified by *mid*. If the MID identifies a Minor frame the first free slot is allocated within the minor frame.

The resulting MID of the Slot is stored back to *mid*, the MID can be used in other function call when setting up the Slot. The *mid* argument is thus of in and out type.

The third argument, *timeslot*, determines the time slot that should be allocated to the Slot. If time management is not configured for the Minor Frame a time can still be assigned to the Slot. If the Slot should step to the next Slot directly when finished (no assigned time-slot), the argument must be set to zero. If time management is enabled for the Minor Frame and the requested time-slot is longer than the free time, the call will result in an error (negative result).

The fourth and last argument can optionally be used to get the address of the descriptor used.

12.3.9.18. gr1553bc_slot_free

Return Slot and timeslot allocated from the Minor Frame.

12.3.9.19. gr1553bc_mid_from_bd

Looks up the Slot/Message ID (MID) from a descriptor address. This function may be useful in the interrupt handler, where the address of the descriptor is given.

12.3.9.20. gr1553bc_slot_bd

Looks up descriptor address from MID.

12.3.9.21. gr1553bc_slot_irq_prepare

Prepares a condition descriptor to generate interrupt. Interrupt will not be enabled until `gr1553bc_slot_irq_enable()` is called. The descriptor will be initialized as an unconditional jump to the next descriptor. The Slot can be associated with a custom callback function and an argument. The callback function and argument is stored in the unused fields of the descriptor.

Once enabled and interrupt is generated by the Slot, the callback routine will be called from interrupt context.

The function returns a negative result if failure, otherwise zero is returned.

12.3.9.22. gr1553bc_slot_irq_enable

Enables interrupt of a previously prepared unconditional jump Slot. The Slot is expected to be initialized with `gr1553bc_slot_irq_prepare()`. The descriptor is changed to do a unconditional jump with interrupt.

The function returns a negative result if failure, otherwise zero is returned.

12.3.9.23. gr1553bc_slot_irq_disable

Disable unconditional IRQ point, the descriptor is changed to unconditional JUMP to the following descriptor, without generating interrupt. After disabling the Slot it can be enabled again, or freed.

The function returns a negative result if failure, otherwise zero is returned.

12.3.9.24. gr1553bc_slot_jump

Initialize a Slot with a custom jump condition. The arguments are declared in the table below.

Table 12.12. gr1553bc_list_cfg member descriptions.

Argument	Description
list	List that the Slot is located at.

Argument	Description
mid	Slot Identification.
condition	Custom condition written to descriptor. See hardware documentation for options.
to_mid	Slot Identification of the Slot that the descriptor will be jumping to.

Returns zero on success.

12.3.9.25. gr1553bc_slot_exttrig

The BC supports an external trigger signal input which can be used to synchronize 1553 transfers. If used, the external trigger is normally generated by some kind of Time Master. A message slot may be programmed to wait for an external trigger before being executed, this feature allows the user to accurate send time synchronize messages to RTs.

This function initializes a Slot to a dummy transfer with the "Wait for external trigger" bit set.

Returns zero on success.

12.3.9.26. gr1553bc_slot_transfer

Initializes a descriptor to a transfer descriptor. The descriptor is initialized according to the function arguments an the global List configuration parameters. The settings that are controlled on a global level (and not by this function):

- IRQ after transfer error
- IRQ after transfer (not supported, insert separate IRQ slot after this)
- Pause schedule after transfer error
- Pause schedule after transfer (not supported)
- Slot time optional (set when MID allocated), otherwise 0
- (OPTIONAL) Dummy Bit, set using slot_empty() or ..._TT_DUMMY
- RT time out tolerance (managed per RT)

The arguments are declared in the table below.

Table 12.13. gr1553bc_slot_transfer argument descriptions.

Argument	Description
list	List that the Slot is located at
mid	Slot Identification
options	Options: • Retry Mode • Number of retires • Bus selection (A or B) • Dummy bit
tt	Transfer options, see BC transfer type macros in header file: • transfer type • RT src/dst address • RT subaddress • word count • mode code
dptr	Descriptor Data Pointer. Used by Hardware to read or write data to the 1553 bus. If bit zero is set the address is translated by the driver into an address which the hardware can access(may be the case if BC device is located on an AMBA-over-PCI bus), if cleared it is assumed that no translation is required(typical case)

Returns zero on success.

12.3.9.27. gr1553bc_slot_dummy

Manipulate the DUMMY bit of a transfer descriptor. Can be used to enable or disable a transfer descriptor.

The *dummy* argument points to an area used as input and output, as input bit 31 is written to the dummy bit of the descriptor, as output the old value of the descriptors dummy bit is written.

Returns zero on success.

12.3.9.28. gr1553bc_slot_empty

Create an empty transfer descriptor, with the DUMMY bit set. The time-slot previously allocated is preserved.

Returns zero on success.

12.3.9.29. gr1553bc_slot_update

This function will update a transfer descriptors status and/or update the data pointer.

If the *dptr* pointer is non-zero the Data Pointer word of the descriptor will be updated with the value of *dptr*. If bit zero is set the driver will translate the data pointer address into an address accessible by the BC hardware. Translation is an option only for AMBA-over-PCI.

If the *stat* pointer is non-zero the Status word of the descriptor will be updated according to the content of *stat*. The old Status will be stored into *stat*. The lower 24-bits of the current Status word may be cleared, and the dummy bit may be set:

```
bd->status = *stat & (bd->status 0xffffff) | (*stat & 0x80000000);
```

Note that the status word is not written (only read) when value pointed to by *stat* is zero.

Returns zero on success.

12.3.9.30. gr1553bc_slot_raw

Custom descriptor initialization. Note that a bad initialization may break the BC driver.

The arguments are declared in the table below.

Table 12.14. gr1553bc_slot_transfer argument descriptions.

Argument	Description
list	List that the Slot is located at
mid	Slot Identification
flags	Determine which words are updated. If bit N is set wordN is written into descriptor wordN, if bit N is zero the descriptor wordN is not modified.
word0	32-bit Word written to descriptor address 0x00
word1	32-bit Word written to descriptor address 0x04
word2	32-bit Word written to descriptor address 0x08
word3	32-bit Word written to descriptor address 0x0C

Returns zero on success.

12.3.9.31. gr1553bc_show_list

Print information about a List to standard out. Each Major Frame's first descriptor for example is printed. This function is used for debugging only.

13. GR1553B Remote Terminal Driver

13.1. Introduction

This section describes the GRLIB GR1553B Remote Terminal (RT) device driver interface. The driver relies on the GR1553B driver. The reader is assumed to be well acquainted with MIL-STD-1553 and the GR1553B core.

13.1.1. GR1553B Remote Terminal Hardware

The GR1553B core supports any combination of the Bus Controller (BC), Bus Monitor (BM) and Remote Terminal (RT) functionality. This driver supports the RT functionality of the hardware, it can be used simultaneously with the Bus Monitor (BM) functionality. When the BM is used together with the RT interrupts are shared between the drivers.

The three functions (BC, BM, RT) are accessed using the same register interface, but through separate registers. In order to shared hardware resources between the three GR1553B drivers, the three depends on a lower level GR1553B driver, see GR1553B driver section.

The driver supports the on-chip AMBA bus and the AMBA-over-PCI bus.

13.1.2. Driver registration

The driver registration is handled by the GR1553B driver, see Chapter 11.

13.2. User Interface

13.2.1. Overview

The RT software driver provides access to the RT core and help with creating memory structures accessed by the RT core. The driver provides the services list below,

- Basic RT functionality (RT address, Bus and RT Status, Enabling core, etc.)
- Event logging support
- Interrupt support (Global Errors, Data Transfers, Mode Code Transfer)
- DMA-Memory configuration
- Sub Address configuration
- Support for Mode Codes
- Transfer Descriptor List Management per RT sub address and transfer type (RX/TX)

The driver sources and definitions are listed in the table below, the path is given relative to the extracted distribution archive.

Table 13.1. RT driver Source location

Filename	Description
src/libdrv/src/gr1553b/gr1553rt.c	GR1553B RT Driver source
src/libdrv/src/include/gr1553rt.h	GR1553B RT Driver interface declaration

13.2.1.1. Accessing an RT device

In order to access an RT core, a specific core must be identified (the driver support multiple devices). The core is opened by calling `gr1553rt_open()`, the open function allocates an RT device by calling the lower level GR1553B driver and initializes the RT by stopping all activity and disabling interrupts. After an RT has been opened it can be configured `gr1553rt_config_init()`, SA-table configured, descriptor lists assigned to SA, interrupt callbacks registered, and finally communication started by calling `gr1553rt_start()`. Once the RT is started interrupts may be generated, data may be transferred and the event log filled. The communication can be stopped by calling `gr1553rt_stop()`.

When the application no longer needs to access the RT core, the RT is closed by calling `gr1553rt_close()`.

13.2.1.2. Introduction to the RT Memory areas

For the RT there are four different types of memory areas. The access to the areas is much different and involve different latency requirements. The areas are:

- Sub Address (SA) Table
- Buffer Descriptors (BD)
- Data buffers referenced from descriptors (read or written)
- Event (EV) logging buffer

The memory types are described in separate sections below. Generally three of the areas (controlled by the driver) can be dynamically allocated by the driver or assigned to a custom location by the user. Assigning a custom address is typically useful when for example a low-latency memory is required, or the GR1553B core is located on an AMBA-over- PCI bus where memory accesses over the PCI bus will not satisfy the latency requirements by the 1553 bus, instead a memory local to the RT core can be used to shorten the access time. Note that when providing custom addresses the alignment requirement of the GR1553B core must be obeyed, which is different for different areas and sizes. The memory areas are configured using the `gr1553rt_config_init()` function.

13.2.1.3. Sub Address Table

The RT core provides the user to program different responses per sub address and transfer type through the sub address table (SA-table) located in memory. The RT core consult the SA-table for every 1553 data transfer command on the 1553 bus. The table includes options per sub address and transfer type and a pointer to the next descriptor that let the user control the location of the data buffer used in the transaction. See hardware manual for a complete description.

The SA-table is fixed size to 512 bytes.

Since the RT is required to respond to BC request within a certain time, it is vital that the RT has enough time to look up user configuration of a transfer, i.e. read SA-table and descriptor and possibly the data buffer as well. The driver provides a way to let the user give a custom address to the sub address table or dynamically allocate it for the user. The default action is to let the driver dynamically allocate the SA-table, the SA-table will then be located in the main memory of the CPU. For RT core's located on an AMBA-over- PCI bus, the default action is not acceptable due to the latency requirement mentioned above.

The SA-table can be configured per SA by calling the `gr1553rt_sa_setopts()` function. The mask argument makes it possible to change individual bit in the SA configuration. This function must be called to enable transfers from/to a sub address. See hardware manual for SA configuration options. Descriptor Lists are assigned to a SA by calling `gr1553rt_list_sa()`.

The indication service can be used to determine the descriptor used in the next transfer, see Section 13.2.1.8.

13.2.1.4. Descriptors

A GR1553B RT descriptor is located in memory and pointed to by the SA-table. The SA-table points out the next descriptor used for a specific sub address and transfer type. The descriptor contains three input fields: Control/Status Word determines options for a specific transfer ans status of a completed transfer; Data buffer pointer, 16-bit aligned; Pointer to next descriptor within sub address and transfer type, or end-of-list marker.

All descriptors are located in the same range of memory, which the driver refers to as the BD memory. The BD memory can by dynamically allocated (located in CPU main memory) by the driver or assigned to a custom location by the user. From the BD memory descriptors for all sub addresses are allocated by the driver. The driver works internally with 16-bit descriptor identifiers allowing 65k descriptor in total. A descriptor is allocated for a specific descriptor List. Each descriptor takes 32 bytes of memory.

The user can build and initialize descriptors using the API function `gr1553rt_bd_init()` and update the descriptor and/or view the status and time of a completed transfer.

Descriptors are managed by a data structure named `gr1553rt_list`. A List is the software representation of a chain of descriptors for a specific sub address and transfer type. Thus, 60 lists in total (two lists per SA, SA0 and SA31 are for mode codes) per RT. The List simplifies the descriptor handling for the user by introducing descriptor numbers (`entry_no`) used when referring to descriptors rather than the descriptor address. Up to 65k descriptors are supported per List by the driver. A descriptor list is assigned to a SA and transfer type by calling `gr1553rt_list_sa()`.

When a List is created and configured a maximal number of descriptors are given, giving the API a possibility to allocate the descriptors from the descriptor memory area configured.

Circular buffers can be created by a chain of descriptors where each descriptors data buffer is one element in the circular buffer.

13.2.1.5. Data Buffers

Data buffers are not accessed by the driver at all, the address is only written to descriptor upon user request. It is up to the user to provide the driver with valid addresses to data buffers of the required length.

Note that addresses given must be accessible by the hardware. If the RT core is located on a AMBA-over-PCI bus for example, the address of a data buffer from the RT core's point of view is most probably not the same as the address used by the CPU to access the buffer.

13.2.1.6. Event Logging

Transfer events (Transmission, Reception and Mode Codes) may be logged by the RT core into a memory area for (later) processing. The events logged can be controlled by the user at a SA transfer type level and per mode code through the Mode Code Control Register.

The driver API access the eventlog on two occasions, either when the user reads the eventlog buffer using the `gr1553rt_evlog_read()` function or from the interrupt handler, see the interrupt section for more information. The `gr1553rt_evlog_read()` function is called by the user to read the eventlog, it simply copies the current logged entries to a user buffer. The user must empty the driver eventlog in time to avoid entries to be overwritten. A certain descriptor or SA may be logged to help the application implement communication protocols.

The eventlog is typically sized depending the frequency of the log input (logged transfers) and the frequency of the log output (task reading the log). Every logged transfer is described with a 32-bit word, making it quite compact.

The memory of the eventlog does not require as tight latency requirement as the SA-table and descriptors. However the user still is provided the ability to put the eventlog at a custom address, or letting the driver dynamically allocate it. When providing a custom address the start address is given, the area must have room for the configured number of entries and have the hardware required alignment.

Note that the alignment requirement of the eventlog varies depending on the eventlog length.

13.2.1.7. Interrupt service

The RT core can be programmed to interrupt the CPU on certain events, transfers and errors (SA-table and DMA). The driver divides transfers into two different types of events, mode codes and data transfers. The three types of events can be assigned custom callbacks called from the driver's interrupt service routine (ISR), and custom argument can be given. The callbacks are registered per RT device using the functions `gr1553rt_irq_err(), gr1553rt_irq_mc(), gr1553rt_irq_sa()`. Note that the three different callbacks have different arguments.

Error interrupts are discovered in the ISR by looking at the IRQ event register, they are handled first. After the error interrupt has been handled by the user (user interaction is optional) the RT core is stopped by the driver.

Data transfers and Mode Code transfers are logged in the eventlog. When a transfer-triggered interrupt occurs the ISR starts processing the event log from the first event that caused the IRQ (determined by hardware register) calling the mode code or data transfer callback for each event in the log which has generated an IRQ (determined by the IRQSR bit). Even though both the ISR and the eventlog read function `r1553rt_evlog_read()` processes the eventlog, they are completely separate processes - one does not affect the other. It is up to the user to make sure that events that generated interrupt are not double processed. The callback functions are called in the same order as the event was generated.

Is is possible to configure different callback routines and/or arguments for different sub addresses (1..30) and transfer types (RX/TX). Thus, 60 different callback handlers may be registered for data transfers.

13.2.1.8. Indication service

The indication service is typically used by the user to determine how many descriptors have been processed by the hardware for a certain SA and transfer type. The `gr1553rt_indication()` function returns the next

descriptor number which will be used next transfer by the RT core. The indication function takes a sub address and an RT device as input, By remembering which descriptor was processed last the caller can determine how many and which descriptors have been accessed by the BC.

13.2.1.9. Mode Code support

The RT core a number of registers to control and interact with mode code commands. See hardware manual which mode codes are available. Each mode code can be disabled or enabled. Enabled mode codes can be logged and interrupt can be generated upon transmission events. The `gr1553rt_config_init()` function is used to configure the aforementioned mode code options. Interrupt caused by mode code transmissions can be programmed to call the user through an callback function, see the interrupt Section 13.2.1.7.

The mode codes "Synchronization with data", "Transmit Bit word" and "Transmit Vector word" can be interacted with through a register interface. The register interface can be read with the `gr1553rt_status()` function and selected (or all) bits of the bit word and vector word can be written using `gr1553rt_set_vecword()` function.

Other mode codes can interacted with using the Bus Status Register of the RT core. The register can be read using the `gr1553rt_status()` function and written selectable bit can be written using `gr1553rt_set_bussts()`.

13.2.1.10. RT Time

The RT core has an internal time counter with a configurable time resolution. The finest time resolution of the timer counter is one microsecond. The resolution is configured using the `gr1553rt_config_init()` function. The current time is read by calling the `gr1553rt_status()` function.

13.2.2. Application Programming Interface

The RT driver API consists of the functions in the table below.

Table 13.2. Data structures

Prototype	Description
`void *gr1553rt_open(int minor)`	Open an RT device by instance number. Returns a handle identifying the specific RT device. The handle is given as input in most functions of the API
`void gr1553rt_close(void *rt)`	Close a previously opened RT device

`int gr1553rt_config_init(`
`  void *rt,`
`  struct gr1553rt_cfg *cfg)`
	Configure the RT device driver

Configure the RT device driver and allocate device memory	
`int gr1553rt_config_free(void *rt)`	Free allocated device memory
`int gr1553rt_start(void *rt)`	Start RT communication, enables Interrupts
`void gr1553rt_stop(void *rt)`	Stop RT communication, disables interrupts

`void gr1553rt_status(`
`  void *rt,`
`  struct gr1553rt_status *status)`
	Get Time, Bus/RT Status and mode code status

`int gr1553rt_indication(`
`  void *rt,`
`  int subadr,`
`  int *txeno,`
`  int *rxeno)`
	Get the next descriptor that will processed of an RT sub-address and transfer type

`int gr1553rt_evlog_read(`
`  void *rt,`
`  unsigned int *dst,`
`  int max)`
	Copy contents of event log to a user provided data buffer

`void gr1553rt_set_vecword(`
`  void *rt,`
`  unsigned int mask,`
`  unsigned int words)`
	Set all or a selection of bits in the Vector word and Bit word used by the "Transmit Bit word" and "Transmit Vector word" mode codes

Prototype	Description

`void gr1553rt_set_bussts(`
`  void *rt,`
`  unsigned int mask,`
`  unsigned int sts)`
	Modify a selection of bits in the RT Bus Status register

`void gr1553rt_sa_setopts(`
`  void *rt,`
`  int subadr,`
`  unsigned int mask,`
`  unsigned int options)`
	Configures a sub address control word located in the SA-table.

`void gr1553rt_list_sa(`
`  struct gr1553rt_list *list,`
`  int *subadr,`
`  int *tx)`
	Get the Sub address and transfer type of a scheduled list

`void gr1553rt_sa_schedule(`
`  void *rt,`
`  int subadr,`
`  int tx,`
`  struct gr1553rt_list *list)`
	Schedule a RX or TX descriptor list on a sub address of a certain transfer type

`int gr1553rt_irq_err(`
`  void *rt,`
`  gr1553rt_irqerr_t func,`
`  void *data)`
	Assign an Error Interrupt handler callback routine and custom argument

`int gr1553rt_irq_mc(`
`  void *rt,`
`  gr1553rt_irqmc_t func,`
`  void *data)`
	Assign a Mode Code Interrupt handler callback routine and custom argument

`int gr1553rt_irq_sa(`
`  void *rt,`
`  int subadr,`
`  int tx,`
`  gr1553rt_irq_t func,`
`  void *data)`
	Assign a Data Transfer Interrupt handler callback routine and custom argument to a certain sub address and transfer type

`int gr1553rt_list_init(`
`  void *rt,`
`  struct gr1553rt_list **plist,`
`  struct gr1553rt_list_cfg *cfg)`
	Initialize a descriptor List according to configuration. The List can be used for RX/TX on any sub address.

`int gr1553rt_list_alloc(`
`  void *rt,`
`  struct gr1553rt_list **plist,`
`  struct gr1553rt_list_cfg *cfg)`
	Initialize and allocate a descriptor List according to configuration. The List can be used for RX/TX on any sub address.

`int gr1553rt_bd_init(`
`  struct gr1553rt_list *list,`
`  unsigned short entry_no,`
`  unsigned int flags,`
`  uint16_t *dptr,`
`  unsigned short next)`
	Initialize a Descriptor in a List identified by number.

`int gr1553rt_bd_update(`
`  struct gr1553rt_list *list,`
`  int entry_no,`
`  unsigned int *status,`
`  uint16_t **dptr)`
	Update the status and/or the data buffer pointer of a descriptor.

13.2.2.1. Data structures

The `gr1553rt_cfg` data structure is used to configure an RT device. The configuration parameters are described in the table below.

```
struct gr1553rt_cfg {
    unsigned char rtaddress;
    unsigned int modecode;
    unsigned short time_res;
    void *satab_buffer;
    void *evlog_buffer;
    int evlog_size;
    int bd_count;
    void *bd_buffer;
    void *bd_sw_buffer;
};
```

Table 13.3. gr1553rt_cfg member descriptions

Member	Description
rtaddress	RT Address on 1553 bus [0..30]

Member	Description
modecode	Mode codes `enable/disable/IRQ/EV-Log`. Each mode code has a 2-bit configuration field. Mode Code Control Register in hardware manual
time_res	Time tag resolution in microseconds
satab_buffer	Sub Address Table (SA-table) allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of SA-table is given, the address must be aligned to 10-bit (1KiB) boundary and at least 16*32 bytes.
evlog_buffer	Eventlog DMA buffer allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of eventlog is given, the address must be of `evlog_size` and aligned to `evlog_size`. See hardware manual.
evlog_size	Length in bytes of Eventlog, must be a multiple of 2. If set to zero event log is disabled, note that enabling logging in SA-table or descriptors will cause failure when eventlog is disabled.
bd_count	Number of descriptors for RT device. All descriptor lists share the descriptors. Maximum is 65K descriptors.
bd_buffer	Descriptor memory area allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of descriptors is given, the address must be aligned to 32 bytes and of *`(32 * bd_count)`* bytes size.
bd_sw_buffer	Descriptor memory area allocation for internal usage. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of descriptors is given, the area must be of *`(4 * bd_count)`* bytes size.

The `gr1553rt_list_cfg` data structure hold the configuration parameters of a descriptor List.

```
struct gr1553rt_list_cfg {
    unsigned int bd_cnt;
};
```

Table 13.4. gr1553rt_list_cfg member descriptions

Member	Description
bd_cnt	Number of descriptors in List

The current status of the RT core is stored in the `gr1553rt_status` data structure by the function **gr1553rt_status()**. The fields are described below.

```
struct gr1553rt_status {
    unsigned int status;
    unsigned int bus_status;
    unsigned short synctime;
    unsigned short syncword;
    unsigned short time_res;
    unsigned short time;
};
```

Table 13.5. gr1553rt_status member descriptions

Member	Description
status	Current value of RT Status Register
bus_status	Current value of RT Bus Status Register
synctime	Time Tag when last synchronize with data was received
syncword	Data of last mode code synchronize with data
time_res	Time resolution in microseconds (set by config)
time	Current Time Tag. `(time_res * time)` gives the number of microseconds since last time overflow.

13.2.2.2. gr1553rt_open

Opens a GR1553B RT device identified by instance number, *`minor`*. The instance number is determined by the order in which GR1553B cores with RT functionality are found, the order of the Plug & Play.

A handle is returned identifying the opened RT device, the handle is used internally by the RT driver, it is used as an input parameter rt to all other functions that manipulate the hardware.

Close and Stop an RT device identified by input argument rt previously returned by **gr1553rt_open()**.

13.2.2.3. gr1553rt_close

Close and Stop an RT device identified by input argument rt previously returned by **gr1553rt_open()**.

13.2.2.4. gr1553rt_config_init

Configure memory for an RT device. The configuration parameters are stored in the location pointed to by *cfg*. The layout of the parameters must follow the `gr1553rt_cfg` data structure, described in Table 13.3.

This function will not allocate any memory. Replace this function call with gr1553rt_config_alloc() if you want the driver to allocate memory. If any of the data pointers are NULL, then this function will return a negative result. On success zero is returned.

13.2.2.5. gr1553rt_config_alloc

Configure and allocate memory for an RT device. The configuration parameters are stored in the location pointed to by *cfg*. The layout of the parameters must follow the `gr1553rt_cfg` data structure, described in Table 13.3.

If memory allocation fails (in case of dynamic memory allocation) the function return a negative result, on success zero is returned.

13.2.2.6. gr1553bm_config_free

Free allocated memory.

13.2.2.7. gr1553rt_start

Starts RT communication by enabling the core and enabling interrupts. The user must have configured the driver (RT address, Mode Code, SA-table, lists, descriptors, etc.) before calling this function.

After the RT has been started the configuration function can not be called.

On success this function returns zero, on failure a negative result is returned.

13.2.2.8. gr1553rt_stop

Stops RT communication by disabling the core and disabling interrupts. Further 1553 commands to the RT will be ignored.

13.2.2.9. gr1553rt_status

Read current status of the RT core. The status is written to the location pointed to by status in the format determined by the `gr1553rt_status` structure described in Table 13.5.

13.2.2.10. gr1553rt_indication

Get the next descriptor that will be processed for a specific sub address. The descriptor number is looked up from the descriptor address found the SA-table for the sub address specified by *subadr* argument.

The descriptor number of respective transfer type (RX/TX) will be written to the address given by *txeno* and/or *rxeno*. If end-of-list has been reached, -1 is stored into *txeno* or *rxeno*.

If the request is successful zero is returned, otherwise a negative number is returned (bad sub address or descriptor).

13.2.2.11. gr1553rt_evlog_read

Copy up to *max* number of entries from eventlog into the address specified by *dst*. The actual number of entries read is returned. It is important to read out the eventlog entries in time to avoid data loss, the eventlog can be sized so that data loss can be avoided.

Zero is returned when entries are available in the log, negative on failure.

13.2.2.12. gr1553rt_set_vecword

Set a selection of bits in the RT Vector and/or Bit word. The words are used when,

- Vector Word is used in response to "Transmit vector word" BC commands
- Bit Word is used in response to "Transmit bit word" BC commands

The argument *mask* determines which bits are written, and *words* determines the value of the bits written. The lower 16-bits are the `Vector Word`, the higher 16-bits are the Bit Word.

13.2.2.13. gr1553rt_set_bussts

Set a selection of bits of the Bus Status Register. The bits written is determined by the mask bit-mask and the values written is determined by *sts*. Operation:

```
bus_status_reg = (bus_status_reg & ~mask) | (sts & mask)
```

13.2.2.14. gr1553rt_sa_setopts

Configure individual bits of the SA Control Word in the SA-table. One may for example Enable or Disable a SA RX and/or TX. See hardware manual for SA-Table configuration options.

The *mask* argument is a bit-mask, it determines which bits are written and *options* determines the value written.

The *subadr* argument selects which sub address is configured.

Note that SA-table is all zero after configuration, every SA used must be configured using this function.

13.2.2.15. gr1553rt_list_sa

This function looks up the SA and the transfer type of the descriptor list given by `list`. The SA is stored into `subadr`, the transfer type is written into `tx` (TX=1, RX=0).

13.2.2.16. gr1553rt_sa_schedule

This function associates a descriptor list with a sub address (given by *subadr*) and a transfer type (given by *tx*). The first descriptor in the descriptor list is written to the SA-table entry of the SA.

13.2.2.17. gr1553rt_irq_err

his function registers an interrupt callback handler of the Error Interrupt. The handler `func` is called with the argument data when a DMA error or SA-table access error occurs. The callback must follow the prototype of `gr1553rt_irqerr_t` :

```
typedef void (*gr1553rt_irqerr_t)(int err, void *data);
```

Where *err* is the value of the GR1553B IRQ register at the time the error was detected, it can be used to determine what kind of error occurred.

13.2.2.18. gr1553rt_irq_mc

This function registers an interrupt callback handler for Logged Mode Code transmission Interrupts. The handler `func` is called with the argument data when a Mode Code transmission event occurs, note that interrupts must be enabled per Mode Code using **gr1553rt_config_init()**. The callback must follow the prototype of `gr1553rt_irqmc_t`:

```
typedef void (*gr1553rt_irqmc_t)(
    int mcode,
    unsigned int entry,
    void *data
    );
```

Where *mcode* is the mode code causing the interrupt, entry is the raw event log entry.

13.2.2.19. gr1553rt_irq_sa

Register an interrupt callback handler for data transfer triggered Interrupts, it is possible to assign a unique function and/or data for every SA (given by *subadr*) and transfer type (given by *tx*). The handler *func* is called with the argument *data* when a data transfer interrupt event occurs. Interrupts is configured on a descriptor or SA basis. The callback routine must follow the prototype of gr1553rt_irq_t:

```
typedef void (*gr1553rt_irq_t)(
    struct gr1553rt_list *list,
    unsigned int entry,
    int bd_next,
    void *data
    );
```

Where *list* indicates which descriptor list (Sub Address, transfer type) caused the interrupt event, *entry* is the raw event log entry, *bd_next* is the next descriptor that will be processed by the RT for the next transfer of the same sub address and transfer type.

13.2.2.20. gr1553rt_list_init

Configure a list structure according to configuration given in *cfg*, see the `gr1553rt_list_cfg` data structure in Table 13.4. Assign the list to an RT device, however not to a sub address yet. The *rt* handle is stored within list.

This function will not allocate any memory. Replace this function call with gr1553rt_list_alloc() if you want the driver to allocate the memory.

Note that descriptor are allocated from the RT device, so the RT device itself must be configured using `gr1553rt_config_init()` before calling this function.

A negative number is returned on failure, on success zero is returned.

13.2.2.21. gr1553rt_list_alloc

Allocate and configure a list structure according to configuration given in *cfg*, see the `gr1553rt_list_cfg` data structure in Table 13.4. Assign the list to an RT device, however not to a sub address yet. The *rt* handle is stored within list.

The resulting descriptor list is written to the location indicated by the *plist* argument.

Note that descriptor are allocated from the RT device, so the RT device itself must be configured using `gr1553rt_config_alloc()` before calling this function.

A negative number is returned on failure, on success zero is returned.

13.2.2.22. gr1553rt_bd_init

Initialize a descriptor entry in a list. This is typically done prior to scheduling the list. The descriptor and the next descriptor is given by descriptor indexes relative to the list (*entry_no and next*), see table below for options on *next*. Set bit 30 of the argument *flags* in order to set the IRQEN bit of the descriptors Control/Status Word. The argument *dptr* is written to the descriptors Data Buffer Pointer Word.

Note that the data pointer is accessed by the GR1553B core and must therefore be a valid address for the core. This is only an issue if the GR1553B core is located on a AMBA- over-PCI bus, the address may need to be translated from CPU accessible address to hardware accessible address.

Table 13.6. gr1553rt_bd_init next argument description

Values of *next*	Description
0xffff	Indicate to hardware that this is the last entry in the list, the next descriptor is set to end-of-list mark (0x3).
0xfffe	Next descriptor (entry_no+1) or 0 is last descriptor in list.
other	The index of the next descriptor.

A negative number is returned on failure, on success a zero is returned.

13.2.2.23. gr1553rt_bd_update

Manipulate and read the Control/Status and Data Pointer words of a descriptor.

If `status` is non-zero, the Control/Status word is swapped with the content pointed to by `status`.

If `dptr` is non-zero, the Data Pointer word is swapped with the content pointed to by `dptr`.

A negative number is returned on failure, on success a zero is returned.

14. GR1553B Bus Monitor Driver

14.1. Introduction

This section describes the GRLIB GR1553B Bus Monitor (BM) device driver interface. The driver relies on the GR1553B driver. The reader is assumed to be well acquainted with MIL-STD-1553 and the GR1553B core.

14.1.1. GR1553B Remote Terminal Hardware

The GR1553B core supports any combination of the Bus Controller (BC), Bus Monitor (BM) and Remote Terminal (RT) functionality. This driver supports the BM functionality of the hardware, it can be used simultaneously with the RT or BC functionality, but not both simultaneously. When the BM is used together with the RT or BC interrupts are shared between the drivers.

The three functions (BC, BM, RT) are accessed using the same register interface, but through separate registers. In order to shared hardware resources between the three GR1553B drivers, the three depends on a lower level GR1553B driver, see GR1553B driver section.

The driver supports the on-chip AMBA bus and the AMBA-over-PCI bus.

14.1.2. Driver registration

The driver registration is handled by the GR1553B driver, see Chapter 11.

14.2. User Interface

14.2.1. Overview

The BM software driver provides access to the BM core and help with accessing the BM log memory buffer. The driver provides the services list below,

- Basic BM functionality (Enabling/Disabling, etc.)
- Filtering options
- Interrupt support (DMA Error, Timer Overflow)
- 1553 Timer handling
- Read BM log

The driver sources and definitions are listed in the table below, the path is given relative to the extracted distribution archive.

Table 14.1. BM driver Source location

Filename	Description
src/libdrv/src/gr1553b/gr1553bm.c	GR1553B BM Driver source
src/libdrv/src/include/gr1553bm.h	GR1553B BM Driver interface declaration

14.2.1.1. Accessing a BM device

In order to access a BM core a specific core must be identified (the driver support multiple devices). The core is opened by calling `gr1553bm_open()`, the open function allocates a BM device by calling the lower level GR1553B driver and initializes the BM by stopping all activity and disabling interrupts. After a BM has been opened it can be configured `gr1553bm_config_init()` and then started by calling `gr1553bm_start()`. Once the BM is started the log is filled by hardware and interrupts may be generated. The logging can be stopped by calling `gr1553bm_stop()`.

When the application no longer needs to access the BM driver services, the BM is closed by calling `gr1553bm_close()`.

14.2.1.2. BM Log memory

The BM log memory is written by the BM hardware when transfers matching the filters are detected. Each command, Status and Data 16-bit word takes 64-bits of space in the log, into the first 32-bits the current 24-bit 1553

timer is written and to the second 32-bit word status, word type, Bus and the 16-bit data is written. See hardware manual.

The BM log DMA-area can be dynamically allocated by the driver or assigned to a custom location by the user. Assigning a custom address is typically useful when the GR1553B core is located on an AMBA-over-PCI bus where memory accesses over the PCI bus will not satisfy the latency requirements by the 1553 bus, instead a memory local to the BM core can be used to shorten the access time. Note that when providing custom addresses the 8-byte alignment requirement of the GR1553B BM core must be obeyed. The memory areas are configured using the `gr1553bm_config()` function.

14.2.1.3. Accessing the BM Log memory

The BM Log is filled as transfers are detected on the 1553 bus, if the log is not emptied in time the log may overflow and data loss will occur. The BM log can be accessed with the functions listed below.

- gr1553bm_available()
- gr1553bm_read()

A custom handler responsible for copying the BM log can be assigned in the configuration of the driver. The custom routine can be used to optimize the BM log read, for example one may not perhaps not want to copy all entries, search the log for a specific event or compress the log before storing to another location.

14.2.1.4. Time

Th BM core has a 24-bit time counter with a programmable resolution through the `gr1553bm_config_init()` function. The finest resolution is a microsecond. The BM driver maintains a 64-bit 1553 time. The time can be used by an application that needs to be able to log for a long time. The driver must detect every overflow in order maintain the correct 64-bit time, the driver gives users two different approaches. Either the timer overflow interrupt is used or the user must guarantee to call the `gr1553bm_time()` function at least once before the second time overflow happens. The timer overflow interrupt can be enabled from the `gr1553bm_config_init()` function.

The current 64-bit time can be read by calling `gr1553bm_time()`.

The application can determine the 64-bit time of every log entry by emptying the complete log at least once per timer overflow.

14.2.1.5. Filtering

The BM core has support for filtering 1553 transfers. The filter options can be controlled by fields in the configuration structure given to `gr1553bm_config_init()`.

14.2.1.6. Interrupt service

The BM core can interrupt the CPU on DMA errors and on Timer overflow. The DMA error is unmasked by the driver and the Timer overflow interrupt is configurable. For the DMA error interrupt a custom handler may be installed through the `gr1553bm_config_init()` function. On DMA error the BM logging will automatically be stopped by a call to `gr1553bm_stop()` from within the ISR of the driver.

14.2.2. Application Programming Interface

The BM driver API consists of the functions in the table below.

Table 14.2. function prototypes

Prototype	Description
`void *gr1553bm_open(int minor)`	Open a BM device by instance number. Returns a handle identifying the specific BM device opened. The handle is given as input parameter *`bm`* in all other functions of the API
`void gr1553bm_close(void *bm)`	Close a previously opened BM device

`int gr1553bm_config_init(`
`  void *bm,`
`  struct gr1553bm_cfg *cfg)`
	Configure the BM device driver BM log DMA-memory

Prototype	Description
`int gr1553bm_config_alloc( void *bm, struct gr1553bm_cfg *cfg)`	Configure the BM device driver and allocate BM log DMA-memory
`void gr1553bm_config_free(void *bm)`	Free allocated memory
`int gr1553bm_start(void *bm)`	Start BM logging, enables Interrupts
`void gr1553bm_stop(void *bm)`	Stop BM logging, disables interrupts
`void gr1553bm_time( void *bm, uint64_t *time)`	Get 1553 64-bit Time maintained by the driver. The lowest 24-bits are taken directly from the BM timer register, the most significant 40-bits are taken from a software counter.
`int gr1553bm_available( void *bm, int *nentries)`	The current number of entries in the log is stored into `nentries`.
`int gr1553bm_read( void *bm, struct gr1553bm_entry *dst, int *max)`	Copy contents a maximum number (*`max`*) of entries from the BM log to a user provided data buffer (dst). The actual number of entries copied is stored into *`max`*.

14.2.2.1. Data structures

The `gr1553bm_cfg` data structure is used to configure the BM device and driver. The configuration parameters are described in the table below.

```
struct gr1553bm_config {
    uint8_t time_resolution;
    int time_ovf_irq;
    unsigned int filt_error_options;
    unsigned int filt_rtadr;
    unsigned int filt_subadr;
    unsigned int filt_mc;
    unsigned int buffer_size;
    void *buffer_custom;
    bmcopy_func_t copy_func;
    void *copy_func_arg;
    bmisr_func_t dma_error_isr;
    void *dma_error_arg;
};
```

Table 14.3. gr1553bm_config member descriptions.

Member	Description
time_resolution	8-bit time resolution, the BM will update the time according to this setting. 0 will make the time tag be of highest resolution (no division), 1 will make the BM increment the time tag once for two time ticks (div with 2), etc.
time_ovf_irq	Enable Time Overflow IRQ handling. Setting this to 1 makes the driver to update the 64-bit time by it self, it will use time overflow IRQ to detect when the 64-bit time counter must be incremented. If set to zero, the driver expect the user to call gr1553bm_time() regularly, it must be called more often than the time overflows to avoid an incorrect time.
filt_error_options	Bus error log options: bit0,4-31 = reserved, set to zero Bit1 = Enables logging of Invalid mode code errors Bit2 = Enables logging of Unexpected Data errors Bit3 = Enables logging of Manchester/parityerrors
filt_rtadr	RT Subaddress filtering bit mask, bit definition: 31: Enables logging of mode commands on subadr 31 1..30: BitN enables/disables logging of RT subadr N 0: Enables logging of mode commands on subadr 0
filt_mc	Mode code Filter, is written into "BM RT Mode code filter" register, please see hardware manual for bit declarations.
buffer_size	Size of buffer in bytes, must be aligned to 8-byte boundary.
buffer_custom	Custom BM log buffer location, must be aligned to 8-byte and be of buffer_size length. If NULL dynamic memory allocation is used.

Member	Description
copy_func	Custom Copy function, may be used to implement a more effective/ custom way of copying the DMA buffer. For example the DMA log may need to processed at the same time when copying.
copy_func_arg	Optional Custom Data passed onto `copy_func()`
dma_error_isr	Custom DMA error function, note that this function is called from Interrupt Context. Set to NULL to disable this callback.
dma_error_arg	COptional Custom Data passed on to `dma_error_isr()`

```
struct gr1553bm_entry {
    uint32_t time;
    uint32_t data;
};
```

Table 14.4. gr1553bm_entry member descriptions.

Member	Description
time	Time of word transfer entry. Bit31=1, bit 30..24=0, bit 23..0=time
data	Transfer status and data word

Bits	Description
31	Zero
30..20	Zero
19	0=BusA, 1=BusB
18..17	Word Status: 00=Ok, 01=Manchester error, 10=Parity error
16	Word type: 0=Data, 1=Command/ Status
15..0	16-bit Data on detected on bus

14.2.2.2. gr1553bm_open

Opens a GR1553B BM device identified by instance number, `minor`. The instance number is determined by the order in which GR1553B cores with BM functionality are found, the order of the Plug & Play.

A handle is returned identifying the opened BM device, the handle is used internally by the driver, it is used as an input parameter *`bm`* to all other functions that manipulate the hardware.

This function initializes the BM hardware to a stopped/disable level.

14.2.2.3. gr1553bm_close

Close and Stop a BM device identified by input argument bm previously returned by `gr1553bm_open()`.

14.2.2.4. gr1553bm_config_init

Configure the log DMA-memory for a BM device. The configuration parameters are stored in the location pointed to by *`cfg`*. The layout of the parameters must follow the `gr1553bm_config` data structure, described in Table 14.3.

This function will not allocate any memory. Replace this function call with gr1553bm_config_alloc() if you want the driver to allocate memory. If BM device is started or any of the data pointers are NULL, then this function will return a negative result. On success zero is returned.

14.2.2.5. gr1553bm_config_alloc

Configure and allocate the log DMA-memory for a BM device. The configuration parameters are stored in the location pointed to by *`cfg`*. The layout of the parameters must follow the `gr1553bm_config` data structure, described in Table 14.3.

If BM device is started or memory allocation fails (in case of dynamic memory allocation), then this function will return a negative result. On success zero is returned.

14.2.2.6. gr1553bm_config_free

Free allocated memory.

14.2.2.7. gr1553bm_start

Starts 1553 logging by enabling the core and enabling interrupts. The user must have configured the driver (log buffer, timer, filtering, etc.) before calling this function.

After the BM has been started the configuration function can not be called.

On success this function returns zero, on failure a negative result is returned.

14.2.2.8. gr1553bm_stop

Stops 1553 logging by disabling the core and disabling interrupts. Further 1553 transfers will be ignored.

14.2.2.9. gr1553bm_time

This function reads the driver's internal 64-bit 1553 Time. The low 24-bit time is acquired from BM hardware, the MSB is taken from a software counter internal to the driver. The counter is incremented every time the Time overflows by:

- using "Time overflow" IRQ if enabled in user configuration
- by checking "Time overflow" IRQ flag (IRQ is disabled), it is required that user calls this function before the next timer overflow. The software can not distinguish between one or two timer overflows. This function will check the overflow flag and increment the driver internal time if overflow has occurred since last call.

This function update software time counters and store the current time into the address indicated by the argument time.

14.2.2.10. gr1553bm_available

Copy up to `max` number of entries from BM log into the address specified by `dst`. The actual number of entries read is returned in the location of `max` (zero when no entries available). `The max` argument is thus in/out. It is important to read out the log entries in time to avoid data loss, the log can be sized so that data loss can be avoided.

Zero is returned on success, on failure a negative number is returned.

14.2.2.11. gr1553bm_read

Copy up to max number of entries from BM log into the address specified by dst. The actual number of entries read is returned in the location of max (zero when no entries available). The max argument is thus in/out. It is important to read out the log entries in time to avoid data loss, the log can be sized so that data loss can be avoided.

Zero is returned on success, on failure a negative number is returned.

15. GR716 memory protection unit driver

15.1. Introduction

This section describes the driver used to control the two memory protection units (MEMPROT) available in GR716.

15.1.1. User Interface

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

Controlling the driver and device is done with functions provided by the driver prefixed with `memprot_`. All driver functions take a device handle returned by `memprot_open` as the first parameter. All supported functions and their data structures are defined in the driver's header file `drv/memprot.h`.

15.1.2. Features

- Global enable and disable
- Per-segment configuration
- Automatic locking and unlocking

15.1.3. Limitations

The GR716 master-to-APB *grant* interface is not directly supported by the driver. Register structures definitions are available in the header file.

15.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 5.

Table 15.1. Driver registration functions

Registration method	Function
Register one device	`memprot_register()`
Register many devices	`memprot_init()`

15.3. Examples

Examples are available in the `src/libdrv/examples` directory in the Zephyr distribution.

15.4. Opening and closing device

A MEMPROT 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 `memprot_dev_count`. A particular device can be opened using `memprot_open` and closed `memprot_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 `osal_ldstub` from the OSAL. Protection is used by all MEMPROT devices on opening and closing. It is assumed that at most one thread operates on one MEMPROT device at a time.

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

- The device is marked opened to protect the caller from other users of the same device.
- Internal data structures are initialized.
- The device is locked using the `PCR.PROT` field.

The example below prints the number of MEMPROT devices to screen then opens and closes the first MEMPROT device present in the system.

```
int print_memprot_devices(void)
{
        struct memprot_priv *device;
        int count;
```

```
        count = memprot_dev_count();
        printf("%d MEMPROT device(s) present\n", count);

        device = memprot_open(0);
        if (!device) {
                return -1; /* Failure */
        }

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

Table 15.2. `memprot_dev_count` function declaration

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

Table 15.3. `memprot_open` function declaration

Proto	`struct memprot_priv *memprot_open(int dev_no)`	
About	Opens a MEMPROT device. The 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 `memprot_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 MEMPROT device.

Table 15.4. `memprot_close` function declaration

Proto	`int memprot_close(struct memprot_priv *d)`
About	Closes a previously opened device.
Param	*`d`* [IN] pointer Device identifier. Returned from `memprot_open`.
Return	int. DRV_OK

Memory protection configuration is not changed by the open and close functions. In particular, memory protection is not disabled by close.

15.5. Operation mode

The driver always operates in one of two modes: *started* or *stopped*,

This translates directly to whether the memory protection unit is enabled or disabled.

- *Started* is equivalent to `PCR.EN=1`. It means that the memory protection unit is enabled.
- *Stopped* is equivalent to `PCR.EN=0`. It means that the memory protection unit is disabled.

All API functions are available in both operation modes.

15.5.1. Starting and stopping

The `memprot_start()` function places the driver in started mode. The function `memprot_stop()` makes the driver core leave the started mode and enter stopped mode. `memprot_isstarted()` is used to determine the driver operation mode.

Table 15.5. `memprot_start` function declaration

Proto	`int memprot_start(struct memprot_priv *priv)`	
About	Start driver.	
Param	*d* [IN] pointer Device handle returned by `memprot_open`.	
Return	int.	
	Value	Description
	DRV_OK	Device was started by the function call.
	DRV_BUSY	Device already in started mode. Nothing performed.

Table 15.6. `memprot_stop` function declaration

Proto	`int memprot_stop(struct memprot_priv *priv)`	
About	Stop driver.	
Param	*d* [IN] pointer Device handle returned by `memprot_open`.	
Return	int.	
	Value	Description
	DRV_OK	Device was stopped by the function call.
	DRV_BUSY	Device already in stopped mode. Nothing performed.

Table 15.7. `memprot_isstarted` function declaration

Proto	`int memprot_isstarted(struct memprot_priv *d)`	
About	Get current MEMPROT driver running state	
Param	*d* [IN] Pointer Device identifier. Returned by `memprot_open`.	
Return	int. Status	
	Value	Description
	0	Stopped
	1	Started

15.6. Reset

Opening the driver does not change any of the units configuration. To reset the memory protection unit to a known accept-all state, the function `memprot_reset()` can be used.

Table 15.8. `memprot_reset` function declaration

Proto	`int memprot_reset(struct memprot_priv *d)`
About	Reset memory protection unit. This function disables the unit and disables all segment configurations.
Param	*d* [IN] Pointer Device identifier. Returned by `memprot_open`.
Return	int. DRV_OK

15.7. Segment configuration

15.7.1. Number of segments

The number of implemented segments can be retrieved with the function `memprot_nseg()`.

Table 15.9. `memprot_nseg` function declaration

Proto	`int memprot_nseg(struct memprot_priv *d)`
About	Retrieve number of implemented memory segments for memory protection device.
Param	*`d`* [IN] Pointer Device identifier. Returned by `memprot_open`.
Return	int. Number of memory segments supported. This is the value of the constant register field `PCR.NSGEG`.

15.7.2. Data structures

`struct memprot_seginfo` is used by the application to describe individual memory protection segments. The structure is available in `drv/memprot.h` and describes how the driver shall configure the segment.

```
/* User representation of one memory protection segment */
struct memprot_seginfo{
        uintptr_t start;
        uintptr_t end;
        uint32_t g;
        int en;
};
```

Table 15.10. memprot_seginfo data structure declaration

start	Start address	
end	End address	
g	Exclusive write grant `Gi`. This is a bit mask. See GR716-DS-UM for bit definitions of `Gi`.	
	Bit	Description
	`0`	`G0` - Grant master 0 exclusive write access.
	`1`	`G1` - Grant master 1 exclusive write access.
	`i`	`Gi` - Grant master *i* exclusive write access.
en	Disable or enable segment.	
	Value	Description
	`0`	Disable this segment.
	`1`	Enable this segment.

15.7.3. Set

An individual memory segment can be configured by calling the function `memprot_set()` with a user supplied as `struct memprot_seginfo` parameter. The following example configures segment 2.

```
struct memprot_seginfo si;
si.start        = 0x80004000;
si.end          = 0x800040ff;
si.g            = 1 << 2;
si.en           = 1;

memprot_reset(dev);
memprot_set(dev, 2, &si);
memprot_start(dev);
```

For any segment configuration to be in effect, the device must be in started operation mode.

Closing the driver does not cancel the configured memory protections.

Table 15.11. `memprot_set` function declaration

Proto	`int memprot_set(struct memprot_priv *d, int segment, const struct memprot_seginfo *seginfo)`
About	Configure a memory protection segment.

	The information contained in the *seginfo* is installed in the hardware registers corresponding to the *segment* number.
Param	*d* [IN] Pointer Device identifier. Returned by memprot_open.
Param	*segment* [IN] Integer Target segment number. Must be in the range 0 to memprot_nseg()-1.
Param	*seginfo* [IN] Pointer User representation of segment configuration.
Return	int. DRV_OK

15.7.4. Get

Memory protection segments can be read back from hardware into a struct memprot_seginfo record with the function memprot_get(). Everything in the record is qualified with the en field.

Protection segments are not affected when opening the driver which means that the previous configuration can be read out.

Table 15.12. memprot_get function declaration

Proto	int memprot_get(struct memprot_priv *d, int segment, struct memprot_seginfo *seginfo)
About	Read back memory protection segment configuration from hardware. The configuration contained in the hardware registers corresponding segment indexed by *segment* is read back and written to the *seginfo*.
Param	*d* [IN] Pointer Device identifier. Returned by memprot_open.
Param	*segment* [IN] Integer Target segment number. Must be in the range 0 to memprot_nseg()-1.
Param	*seginfo* [OUT] Pointer User representation of segment configuration.
Return	int. DRV_OK

15.7.4.1. Example

The following example function printall() prints information on all memory protection segment of a particular device. In addition to the en field, isstarted() can be used as a global qualifier to determine if a segment is in effect.

```
static void printsi(const struct memprot_seginfo *si)
{
        printf("  start = %08x\n", (unsigned) si->start);
        printf("  end   = %08x\n", (unsigned) si->end);
        printf("  g     = %08x\n", (unsigned) si->g);
        printf("  en    = %d (%s)\n", si->en, si->en ? "enabled" : "disabled");
}

void printall(struct memprot_priv *dev)
{
        const int nseg = memprot_nseg(dev);
        for (int i = 0; i < nseg; i++) {
                struct memprot_seginfo si;
                printf("SEGMENT %d\n", i);
                memprot_get(dev, i, &si);
                printsi(&si);
                puts("");
```

```
        }
}
```

16. Memory scrubber

16.1. Introduction

This section describes the Memory Scrubber (MEMSCRUB) driver for SPARC/LEON processors.

16.1.1. Hardware Support

The MEMSCRUB core hardware interface is documented in the GRIP Core User's manual. The MEMSCRUB core is used to monitor the memory AHB bus and can be programmed to scrub a memory area.

16.1.2. Driver sources

The driver sources and definitions are listed in the table below, the path is given relative to the driver source tree `src/libdrv`.

Table 16.1. MEMSCRUB driver source location

Location	Description
`src/include/drv/memscrub.h`	MEMSCRUB user interface definition
`src/memscrub`	MEMSCRUB driver implementation

16.1.3. Examples

There is an example available that uses the MEMSCRUB driver to scrub a memory area and log the different events. The example is part of the driver distribution, it can be found under `examples/memscrub`.

16.2. Software design overview

The driver provides a function interface, an API, to the user.

The API is not designed for multi-threading, i.e. multiple threads operating on the driver independently. The driver does not contain any lock or protection for SMP environments. Changing the MEMSCRUB configuration is not intended to be done extensively at runtime or independently of the rest of the system, since it usually has a system-level impact. Therefore the user must take care of any impact that the different actions might have on other parts of the system (such as threads, CPUs, DMAs, ...).

16.2.1. Driver usage

The driver provides a set of functions that allow to start and stop the scrubber in different modes. The first step is to setup the memory range (or memory ranges) in which the scrubber is going to act (see Section 16.3.3).

After setting up the range we can start the scrubber in one of the three modes available (see Section 16.3.4):

- Init mode: Initialize the memory area.
- Scrub mode: Scrub the memory area.
- Regen mode: Regenerate the memory area. Similar to scrub mode, but has an optimized access pattern for correcting many errors.

Note that scrub and regen mode can be changed on the fly.

The driver provides functions to check if the scrubber is active and to stop it (see Section 16.3.4).

When dealing with errors, the drivers provides two different interfaces:

- Interrupts (see Section 16.3.6): Allows the user to install an Interrupt Service Routine (ISR) that will be executed whenever an error exceeds its corresponding threshold. Also the MEMSCRUB core allows to generate an interrupt when its done.
- Polling (see Section 16.3.7): Allows the user to poll the error status to check if an error have occurred.

Only one of these interfaces can be used at a given time.

The different errors that the MEMSCRUB can report are:

- AHB correctable error.
- AHB uncorrectable error.

- Scrubber run count errors.
- Scrubber block count errors.

There are functions that allow to configure the error count thresholds for each type of error individually (see Section 16.3.5). When the error count for a certain type exceeds the threshold, the error status is updated and an interrupt is generated. If a threshold is disabled, the error status is not updated and no interrupt is generated.

16.3. Memory scrubber user interface

16.3.1. Return values

```
MEMSCRUB_ERR_OK
MEMSCRUB_ERR_EINVAL
MEMSCRUB_ERR_ERROR
```

All the driver function calls return the following values when an error occurred:

- MEMSCRUB_ERR_OK - Successful execution.
- MEMSCRUB_ERR_EINVAL - Invalid input parameter. One of the input values checks failed.
- MEMSCRUB_ERR_ERROR - Internal error. Can have different causes.

16.3.2. Opening and closing device

A MEMSCRUB 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 `memscrub_dev_count`. A particular device can be opened using `memscrub_open` and closed `memscrub_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 `osal_ldstub` from the OSAL. Protection is used by all MEMSCRUB devices on opening and closing. It is assumed that at most one thread operates on one MEMSCRUB device at a time.

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

- The device is marked opened to protect the caller from other users of the same device.
- Internal data structures are initialized.
- Error and interrupt status is cleared.

The example below prints the number of MEMSCRUB devices to standard output. It then opens and closes the first MEMSCRUB device present in the system.

```
int print_memscrub_devices(void)
{
        struct memscrub_priv *device;
        int count;

        count = memscrub_dev_count();
        printf("%d MEMPROT device(s) present\n", count);

        device = memscrub_open(0);
        if (!device) {
                return -1; /* Failure */
        }

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

Table 16.2. `memscrub_dev_count` function declaration

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

Table 16.3. `memscrub_open` function declaration

Proto	`struct memscrub_priv *memscrub_open(int dev_no)`

About	Opens a MEMSCRUB device. The 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 `memscrub_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 MEMPROT device.

Table 16.4. `memscrub_close` function declaration

Proto	`int memscrub_close(struct memscrub_priv *d)`
About	Closes a previously opened device.
Param	*d* [IN] pointer Device identifier. Returned from `memscrub_open`.
Return	int. MEMSCRUB_ERR_OK

Hardware configuration is not changed by the `memscrub_open()` function, apart from clearing the error and interrupt status at open. `memscrub_close()` does not change the current hardware configuration.

16.3.3. Configuring the memory range

```
int memscrub_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )
int memscrub_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )
int memscrub_secondary_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )
int memscrub_secondary_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )
int memscrub_scrub_position( struct memscrub_priv *priv, uint32_t * position )
```

The driver uses these functions to setup the primary and secondary memory ranges of the MEMSCRUB core. The scrubber will act on the range from address *start* to *end*, both inclusive.

The position function shows the actual position of the MEMSCRUB within the memory range.

These functions return a negative value if something went wrong, as explained in Section 16.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 16.5. `memscrub_range_set` function declaration

Proto	`int memscrub_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )`
About	Set the primary memory range for the MEMSCRUB core. The range is defined by the memory addresses *start* and *end*, both inclusive. See Section 16.3.3.
Param	*start* [IN] Integer 32-bit start address. The address bits below the burst size alignment are constant ‘0’.
Param	*end* [IN] Integer 32-bit end address. The address bits below the burst size alignment are constant ‘1’.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

Table 16.6. `memscrub_range_get` function declaration

Proto	`int memscrub_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )`

About	Get the primary memory range for the MEMSCRUB core. The range is defined by the memory addresses *start* and *end*, both inclusive. See Section 16.3.3.
Param	*start* [IN] Pointer Pointer to the 32-bit start address. The address bits below the burst size alignment are constant '0'.
Param	*end* [IN] Pointer Pointer to the 32-bit end address. The address bits below the burst size alignment are constant '1'.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

Table 16.7. `memscrub_secondary_range_set` function declaration

Proto	`int memscrub_secondary_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )`
About	Set the primary memory range for the MEMSCRUB core. The range is defined by the memory addresses *start* and *end*, both inclusive. See Section 16.3.3.
Param	*start* [IN] Integer 32-bit start address. The address bits below the burst size alignment are constant '0'.
Param	*end* [IN] Integer 32-bit end address. The address bits below the burst size alignment are constant '1'.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

Table 16.8. `memscrub_secondary_range_get` function declaration

Proto	`int memscrub_secondary_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )`
About	Get the secondary memory range for the MEMSCRUB core. The range is defined by the memory addresses *start* and *end*, both inclusive. See Section 16.3.3.
Param	*start* [IN] Pointer Pointer to the 32-bit start address. The address bits below the burst size alignment are constant '0'.
Param	*end* [IN] Pointer Pointer to the 32-bit end address. The address bits below the burst size alignment are constant '1'.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

Table 16.9. `memscrub_scrub_position` function declaration

Proto	`int memscrub_scrub_position( struct memscrub_priv *priv, uint32_t * position )`
About	Get the position of the scrubber within the memory range. See Section 16.3.3.
Param	*position* [IN] Pointer Pointer to the 32-bit position address.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

16.3.4. Starting/stopping different modes.

```
int memscrub_init_start( struct memscrub_priv *priv, uint32_t value, uint8_t delay, int options )
int memscrub_scrub_start( struct memscrub_priv *priv, uint8_t delay, int options )
int memscrub_regen_start( struct memscrub_priv *priv, uint8_t delay, int options )
int memscrub_stop( struct memscrub_priv *priv )
int memscrub_active( struct memscrub_priv *priv )
```

The driver uses these functions to start or stop the different modes of the MEMSCRUB core:

- Init mode: Initialize the memory area.
- Scrub mode: Scrub the memory area.
- Regen mode: Regenerate the memory area. Similar to scrub mode, but has an optimized access pattern for correcting many errors.

All the modes act on the configured memory range (see Section 16.3.3).

The active functions checks if the scrubber is currently running.

These functions return a negative value if something went wrong, as explained in Section 16.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 16.10. `memscrub_init_start` function declaration

Proto	`int memscrub_init_start( struct memscrub_priv *priv, uint32_t value, uint8_t delay, int options )`	
About	Start the initialization mode of the scrubber. See Section 16.3.4.	
Param	*`value`* [IN] Integer 32-bit value to be written into each memory position.	
Param	*`delay`* [IN] Integer 8-bit delay value. Processor cycles delay time between processed blocks.	
Param	*`options`* [IN] Integer Options.	
	Value	Description
	MEMSCRUB_OPTIONS_INTERRUPTDONE_ENABLE	Enable interrupt when done.
	MEMSCRUB_OPTIONS_INTERRUPTDONE_DISABLE	Disable interrupt when done (default).
	MEMSCRUB_OPTIONS_EXTERNALSTART_ENABLE	Enable external start.
	MEMSCRUB_OPTIONS_EXTERNALSTART_DISABLE	Disable external start (default).
	MEMSCRUB_OPTIONS_LOOPMODE_ENABLE	Enable loop mode.
	MEMSCRUB_OPTIONS_LOOPMODE_DISABLE	Disable loop mode (default).
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_ENABLE	Enable secondary memory range.
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_DISABLE	Disable secondary memory range (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.	

Table 16.11. `memscrub_scrub_start` function declaration

Proto	`int memscrub_scrub_start( struct memscrub_priv *priv, uint8_t delay, int options )`	
About	Start the scrubbing mode of the scrubber. See Section 16.3.4.	
Param	*`delay`* [IN] Integer 8-bit delay value. Processor cycles delay time between processed blocks.	
Param	*`options`* [IN] Integer Options.	
	Value	Description
	MEMSCRUB_OPTIONS_INTERRUPTDONE_ENABLE	Enable interrupt when done.
	MEMSCRUB_OPTIONS_INTERRUPTDONE_DISABLE	Disable interrupt when done (default).

	MEMSCRUB_OPTIONS_EXTERNALSTART_ENABLE	Enable external start.
	MEMSCRUB_OPTIONS_EXTERNALSTART_DISABLE	Disable external start (default).
	MEMSCRUB_OPTIONS_LOOPMODE_ENABLE	Enable loop mode.
	MEMSCRUB_OPTIONS_LOOPMODE_DISABLE	Disable loop mode (default).
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_ENABLE	Enable secondary memory range.
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_DISABLE	Disable secondary memory range (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.	

Table 16.12. `memscrub_regen_start` function declaration

Proto	`int memscrub_regen_start( struct memscrub_priv *priv, uint8_t delay, int options )`	
About	Start the regeneration mode of the scrubber. See Section 16.3.4.	
Param	*`delay`* [IN] Integer 8-bit delay value. Processor cycles delay time between processed blocks.	
Param	*`options`* [IN] Integer Options.	
	Value	Description
	MEMSCRUB_OPTIONS_INTERRUPTDONE_ENABLE	Enable interrupt when done.
	MEMSCRUB_OPTIONS_INTERRUPTDONE_DISABLE	Disable interrupt when done (default).
	MEMSCRUB_OPTIONS_EXTERNALSTART_ENABLE	Enable external start.
	MEMSCRUB_OPTIONS_EXTERNALSTART_DISABLE	Disable external start (default).
	MEMSCRUB_OPTIONS_LOOPMODE_ENABLE	Enable loop mode.
	MEMSCRUB_OPTIONS_LOOPMODE_DISABLE	Disable loop mode (default).
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_ENABLE	Enable secondary memory range.
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_DISABLE	Disable secondary memory range (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.	

Table 16.13. `memscrub_stop` function declaration

Proto	`int memscrub_stop( struct memscrub_priv *priv )`
About	Stop the scrubber. See Section 16.3.4.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

Table 16.14. `memscrub_active` function declaration

Proto	`int memscrub_active( struct memscrub_priv *priv )`
About	Returns the active status of the scrubber. When the scrubber is active, it returns a non-zero positive value. When the scrubber is stopped, it returns zero. See Section 16.3.4.
Return	int. Positive value when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

16.3.5. Setting up error thresholds

```
int memscrub_ahberror_setup(struct memscrub_priv *priv, int uethres, int cethres, int options)
int memscrub_scruberror_setup(struct memscrub_priv *priv, int blkthres, int runthres, int options)
```

The driver uses these functions to setup the thresholds for AHB and scrub errors respectively. The following thresholds can be enabled or disabled:

- AHB correctable error.
- AHB uncorrectable error.
- Scrubber run count errors.
- Scrubber block count errors.

If a threshold is disabled, no error status or interrupt will be generated for that type of error. If a threshold is enabled, the error status or interrupt will be triggered when the error count exceeds the threshold value.

These functions return a negative value if something went wrong, as explained in Section 16.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 16.15. `memscrub_ahberror_setup` function declaration

Proto	`int memscrub_ahberror_setup( struct memscrub_priv *priv, int uethres, int cethres, int options )`	
About	Setup the AHB correctable and uncorrectable error thresholds for the MEMSCRUB core. See Section 16.3.5.	
Param	*`uethres`* [IN] Integer AHB uncorrectable error threshold value (only 8 LSB used).	
Param	*`cethres`* [IN] Integer AHB correctable error threshold value (only 10 LSB used).	
Param	*`options`* [IN] Integer Options.	
	Value	Description
	MEMSCRUB_OPTIONS_AHBERROR_CORTHRES_ENABLE	Enable AHB correctable error threshold.
	MEMSCRUB_OPTIONS_AHBERROR_CORTHRES_DISABLE	Disable AHB correctable error threshold (default).
	MEMSCRUB_OPTIONS_AHBERROR_UNCORTHRES_ENABLE	Enable AHB uncorrectable error threshold.
	MEMSCRUB_OPTIONS_AHBERROR_UNCORTHRES_DISABLE	Disable AHB uncorrectable error threshold (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.	

Table 16.16. `memscrub_scruberror_setup` function declaration

Proto	`int memscrub_scruberror_setup( struct memscrub_priv *priv, int blkthres, int runthres, int options )`
About	Setup the scrubber run and block count error thresholds for the MEMSCRUB core. See Section 16.3.5.
Param	*`blkthres`* [IN] Integer Block count error threshold value (only 8 LSB used).
Param	*`runthres`* [IN] Integer Run count error threshold value (only 10 LSB used).
Param	*`options`* [IN] Integer

	Options.	
	Value	Description
	MEMSCRUB_OPTIONS_SCRUBERROR_RUNTHRES_ENABLE	Enable run count error threshold.
	MEMSCRUB_OPTIONS_SCRUBERROR_RUNTHRES_DISABLE	Disable run count error threshold (default).
	MEMSCRUB_OPTIONS_SCRUBERROR_BLOCKTHRES_ENABLE	Enable block count error threshold.
	MEMSCRUB_OPTIONS_SCRUBERROR_BLOCKTHRES_DISABLE	Disable block count error threshold (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.	

16.3.6. Registering an ISR

```
typedef void (*memscrub_isr_t) (
        void *arg,
        uint32_t ahbaccess,
        uint32_t ahbstatus,
        uint32_t scrubstatus
)
int memscrub_isr_register( struct memscrub_priv *priv, memscrub_isr_t isr, void * data )
int memscrub_isr_unregister( struct memscrub_priv *priv )
```

The driver uses these functions to register and unregister an ISR for error interrupts. When registering an ISR, interrupts are enabled. To set the error thresholds that trigger interrupts use the functions described in Section 16.3.5.

These functions return a negative value if something went wrong, as explained in Section 16.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 16.17. `memscrub_isr_register` function declaration

Proto	`int memscrub_isr_register( struct memscrub_priv *priv, memscrub_isr_t isr, void * arg )`
About	Registers an ISR for the MEMSCRUB core. See Section 16.3.6.
Param	*`isr`* [IN] Pointer The ISR function pointer.
Param	*`arg`* [IN] Pointer The ISR argument pointer.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

Table 16.18. `memscrub_isr_unregister` function declaration

Proto	`int memscrub_isr_unregister( struct memscrub_priv *priv )`
About	Unregisters an ISR for the MEMSCRUB core. See Section 16.3.6.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

16.3.7. Polling the error status

```
int memscrub_error_status( struct memscrub_priv *priv, uint32_t * ahbaccess, uint32_t * ahbstatus, uint32_t * scrubstatus )
```

The driver uses this function to poll the error status and clear the error status in case an error is found. To set the error thresholds that trigger error status use the functions described in Section 16.3.5.

This function returns a negative value if something went wrong, as explained in Section 16.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 16.19. `memscrub_error_status` function declaration

Proto	`int memscrub_error_status( struct memscrub_priv *priv, uint32_t * ahbaccess, uint32_t * ahbstatus, uint32_t * scrubstatus )`
About	Poll the state of the error status registers. Returns the status registers and the AHB failing access register. If a error has been detected the function automatically clears the status in order to catch new errors. See Section 16.3.7.
Param	*`ahbaccess`* [OUT] Pointer The value pointed will be updated with the AHB failing access.
Param	*`ahbstatus`* [OUT] Pointer The value pointed will be updated with the AHB error status register content.
Param	*`scrubstatus`* [OUT] Pointer The value pointed will be updated with the scrub error status register content.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 16.3.1.

16.4. API reference

This section lists all functions part of the MEMSCRUB 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 16.1.2.

Table 16.20. MEMSCRUB function reference

Prototype	Section
`int memscrub_range_get(struct memscrub_priv *priv, uint32_t *start, uint32_t *end)`	16.3.3
`int memscrub_range_set(struct memscrub_priv *priv, uint32_t start, uint32_t end)`	16.3.3
`int memscrub_secondary_range_get(struct memscrub_priv *priv, uint32_t *start, uint32_t *end)`	16.3.3
`int memscrub_secondary_range_set(struct memscrub_priv *priv, uint32_t start, uint32_t end)`	16.3.3
`int memscrub_scrub_position(struct memscrub_priv *priv, uint32_t *position)`	16.3.3
`int memscrub_init_start(struct memscrub_priv *priv, uint32_t value, uint8_t delay, int options)`	16.3.4
`int memscrub_scrub_start(struct memscrub_priv *priv, uint8_t delay, int options)`	16.3.4
`int memscrub_regen_start(struct memscrub_priv *priv, uint8_t delay, int options)`	16.3.4
`int memscrub_stop(struct memscrub_priv *priv)`	16.3.4
`int memscrub_active(struct memscrub_priv *priv)`	16.3.4
`int memscrub_ahberror_setup(struct memscrub_priv *priv, int uethres, int cethres, int options)`	16.3.5
`int memscrub_scruberror_setup(struct memscrub_priv *priv, int blkthres, int runthres, int options)`	16.3.5
`int memscrub_isr_register(struct memscrub_priv *priv, memscrub_isr_t isr, void * data)`	16.3.6
`int memscrub_isr_unregister(struct memscrub_priv *priv)`	16.3.6

Prototype	Section
`int memscrub_error_status(struct memscrub_priv *priv, uint32_t *ahbaccess, uint32_t *ahbstatus, uint32_t *scrubstatus)`	16.3.7

17. SpaceWire Router Driver

17.1. Introduction

The SpaceWire router connects external SpaceWire ports and internal AMBA ports together using a non-blocking switch matrix which can connect any input port to any output port. A single routing table is used for the whole router. This chapter describes the API used configure the router. The AMBA port interfaces are controlled by the SpaceWire driver (Chapter 6).

17.2. Driver sources

The driver sources and definitions are listed in Table 17.1. The path is given relative to the driver source tree at `src/libdrv`.

Table 17.1. SpaceWire Router driver source location

Location	Description
src/include/drv/grspwrouter.h	SpaceWire Router driver interface
src/grspwrouter	SpaceWire Router driver implementation

17.3. Routing

Packets can enter into the router from either the external SpaceWire ports or the internal AMBA ports. The router looks at the first byte of the packet, the destination address, to determine where the package shall be routed. If it is below 32, it is treated as a physical address and will be routed to either a SpaceWire port, an AMBA port, or be spilled if there is no port available at the address. For logical addresses (32 and above), the router needs to be provided route information to know to which port the packet shall be routed.

It is also possible to configure the router to do static routing, where all incoming packets on a specific port are routed to a specific output port, regardless of the destination address in the packet.

When routing a packet, the router can be instructed to drop the address byte (called header deletion). This can for example be used to do path addressing, where the packet starts with the entire path through the network and the first address in the path is dropped after every link to reveal the next step in the path.

17.4. Register and access driver

This driver uses the driver registration mechanism described in Chapter 5.

Table 17.2. `grspwrouter_autoinit` function declaration

Name	grspwrouter_autoinit()
Proto	`int grspwrouter_autoinit( )`
About	Register SpaceWire router devices using Plug-n-Play Registers any available SpaceWire router devices and returns the number of devices found.
Return	`int` - The number of devices found and registered

Table 17.3. `grspwrouter_register` function declaration

Name	grspwrouter_register()
Proto	`drvret grspwrouter_register( struct grspwrouter_devcfg * devcfg )`
About	Manually register a single SpaceWire router device The configuration must include the location of the register area and the interrupt number in `devcfg->regs`. The `devcfg->dev` member is used be the driver to store information. The memory used by the `devcfg` argument must never be freed.
Param	`devcfg` - [in] - Settings defining the router device

Name	grspwrouter_register()
Return	`drvret` - DRV_OK on success

When the driver has been registered a device can be accessed by calling `grspwrouter_open()`. The function needs to be provided the system and SpaceWire frequency (in MHz) to be able to configure the scalers used to set up the correct link rate used for initialisation and optional timeouts. The function will configure the timer prescaler so that all router timers operate at 10KHz. This is done to be able to set reasonable timeout values using the API.

Table 17.4. `grspwrouter_open` function declaration

Name	grspwrouter_open()
Proto	`grspwrouter_dev * grspwrouter_open( uint32_t index, uint32_t spw_freq, int32_t sys_freq )`
About	Initialize handle to SpaceWire router driver This function returns a handle to SpaceWire router driver for the device specified by `index`. The `spw_freq` argument shall specify the SpaceWire clock frequency (in MHz) provided to the router. This value is used to configure the initialization bit rate for the all the SpaceWire links. It is also used by `grspwrouter_port_link_start` to set the run state speed of individual links. Use the value 0 to keep the existing value. The `sys_freq` arguments shall specify the system clock frequency (in MHz). This value is used to configure the various timeout functionality provided by the router. This function will set the timer scaler so that all timers run at 10KHz. Use the value 0 to keep the existing value. For the GR740 the default internal SpaceWire clock frequency is 400MHz. This corresponds to an external clock frequency for a SPW_CLK of 50 MHz if the default PLL configuration of 8x is used.
Param	`index` - Index of the SpaceWire router device
Param	`spw_freq` - SpaceWire clock frequency
Param	`sys_freq` - System clock frequency
Return	`grspwrouter_dev *` • `grspwrouter_dev` - on success • `NULL` - if no device with the provided index, or if already opened

Table 17.5. `grspwrouter_close` function declaration

Name	grspwrouter_close()
Proto	`drvret grspwrouter_close( grspwrouter_dev * dev )`
About	Closes a previously opened device The provided handle must have been previously opened by `grspwrouter_open()`.
Param	`dev` - [in] - A valid device handle
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if not previously opened by `grspwrouter_open`

17.5. Setup routing table

The router looks at the address of each incoming packet and uses that as an index in a routing table with information on where to route the packet. The routing information for a specific address is set using the `grspwrouter_route_set()`. It is possible to specify one or multiple target ports.

For each route it is possible to set the following options:

- Enable/disable header deletion

- Spill or wait if output port's link interface is not in run-state
- Set normal / high priority
- Enable packet distribution or group adaptive

Table 17.6. `grspwrouter_route_set` function declaration

Name	**grspwrouter_route_set()**
Proto	`drvret grspwrouter_route_set( grspwrouter_dev * dev, uint8_t address, uint32_t to_mask, bool header_deletion, bool spill_packet, uint32_t options )`
About	Set up a route for incoming packets based on destination address Incoming packets with the destination address `address` will be routed to the first available output port of the ones specified in the `to_mask`. If packet distribution has been enabled the same packet will be sent on all specified output ports. The `to_mask` argument can be built using a mask where each bit index corresponds to the SpaceWire port with the same index. The `GRSPWROUTER_PORT()` define can be used for this: `to_mask = GRSPWROUTER_PORT(3) | GRSPWROUTER_PORT(4)` On the GR740 the following defines can be used: • AMBA port 0 (GRSPWROUTER_GR740_AMBA_0) • AMBA port 1 (GRSPWROUTER_GR740_AMBA_1) • AMBA port 2 (GRSPWROUTER_GR740_AMBA_2) • AMBA port 3 (GRSPWROUTER_GR740_AMBA_3) • SpaceWire port 1 (GRSPWROUTER_GR740_SPW_1) • SpaceWire port 2 (GRSPWROUTER_GR740_SPW_2) • SpaceWire port 3 (GRSPWROUTER_GR740_SPW_3) • SpaceWire port 4 (GRSPWROUTER_GR740_SPW_4) • SpaceWire port 5 (GRSPWROUTER_GR740_SPW_5) • SpaceWire port 6 (GRSPWROUTER_GR740_SPW_6) • SpaceWire port 7 (GRSPWROUTER_GR740_SPW_7) • SpaceWire port 8 (GRSPWROUTER_GR740_SPW_8) Packets sent to the AMBA ports are handled by the SpaceWire driver. The router can be configured to automatically remove the first byte of the packet, the byte that contains the destination address. This is called header deletion. If the output port's link interface is not in run-state the router can be ordered to wait until the link is up or to spill the packet. The `options` argument can be built by or:ing the following defines: • Set high priority when more than one packet is competing for the same output port (`GRSPWROUTER_ROUTE_PRIORITY`) • Enable packet distribution (default group adaptive) (`GRSPWROUTER_PACKET_DIST`)
Param	`dev` - [in] - Valid router device handle
Param	`address` - Route incoming packets with this destination address
Param	`to_mask` - Route packets to these output ports
Param	`header_deletion` - Remove the first byte of the packet when routing it
Param	`spill_packet` - Spill the packet if the output port's link interface is not in run-state
Param	`options` - Enable high priority (GRSPWROUTER_ROUTE_PRIORITY) and/or packet distribution (GRSPWROUTER_PACKET_DIST)
Return	`drvret` • `DRV_OK` - on success

Name	grspwrouter_route_set()
	• DRV_INVAL - if address is 0

Table 17.7. `grspwrouter_route_disable` function declaration

Name	grspwrouter_route_disable()
Proto	`drvret grspwrouter_route_disable( grspwrouter_dev * dev, uint8_t address )`
About	Disable a route for incoming packets based on destination address PrSTATUS the router from routing packets with a specific destination address. Only logical addresses can be blocked. Packets with a physical destination address will still be routed.
Param	`dev` - [in] - Valid router device handle
Param	`address` - Packets with this logical destination address will not be routed (32 - 255)
Return	`drvret` • DRV_OK - on success • DRV_INVAL - on non-logical address

The router also supports static routing in which all packets received on a certain port are always forwarded unmodified to a specified port regardless of the target address in the packet. Static routing is enabled for a port by `grspwrouter_static_route_set()`.

Table 17.8. `grspwrouter_port_static_route_set` function declaration

Name	grspwrouter_port_static_route_set()
Proto	`drvret grspwrouter_port_static_route_set( grspwrouter_dev * dev, uint8_t port, uint32_t destination, bool use_route_info )`
About	Set up a static route for incoming packets on a specific port This function enables static routing for a port where incoming packets are always routed unmodified to a specific output port regardless of the address in the packet. By setting `use_route_info` to `true` it is possible to use the normal route information to route the packet to multiple ports.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid port
Param	`destination` - Target port
Param	`use_route_info` - Use the target addresses configured by `grspwrouter_route_set` for the target
Return	`drvret` • DRV_OK - on success • DRV_INVAL - if static routing not supported, or invalid port or destination

Table 17.9. `grspwrouter_port_static_route_disable` function declaration

Name	grspwrouter_port_static_route_disable()
Proto	`drvret grspwrouter_port_static_route_disable( grspwrouter_dev * dev, uint8_t port )`
About	Disable static routing for the port Disable static routing for the port.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid port
Return	`drvret` • DRV_OK - on success

Name	grspwrouter_port_static_route_disable()
	• DRV_INVAL - if invalid port

17.5.1. GR716B

The SpaceWire router in GR716B can only use 1 logical address at a time. The current logical address that is mapped can be read by `gr716b_grspwrouter_mapped_adr_get()`

If a logical address has already been selected then in order to change the currently mapped address it must first be reset with `gr716b_grspwrouter_mapped_adr_reset()`

After the mapped address has been reset a new route can then be created with `grspwrouter_route_set()`

Table 17.10. `gr716b_grspwrouter_mapped_adr_get` function declaration

Name	gr716b_grspwrouter_mapped_adr_get()
Proto	`uint8_t gr716b_grspwrouter_mapped_adr_get( grspwrouter_dev * dev )`
About	Return the current mapped address Returns routers current mapped address. GR716b only.
Param	`dev` - [in] - Valid router device handle
Return	`uint8_t` - Current mapped address

Table 17.11. `gr716b_grspwrouter_mapped_adr_reset` function declaration

Name	gr716b_grspwrouter_mapped_adr_reset()
Proto	`uint8_t gr716b_grspwrouter_mapped_adr_reset( grspwrouter_dev * dev )`
About	Resets the current mapped adress Reset the currently mapped address on GR716B. The currently mapped address needs to be reset before a new address can be mapped.
Param	`dev` - [in] - Valid router device handle
Return	`drvret` • DRV_OK - on success

17.6. Link handling

A SpaceWire link can be started with a desired link rate by calling the `grspwrouter_port_link_start()` function.

Table 17.12. `grspwrouter_port_link_start` function declaration

Name	grspwrouter_port_link_start()
Proto	`drvret grspwrouter_port_link_start( grspwrouter_dev * dev, uint8_t port, uint32_t link_rate )`
About	Start the SpaceWire link Configure the link rate to use and enable the link. The link rate shall be specified in MBits/s. This function can only be called on a SpaceWire port, not an AMBA port.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Param	`link_rate` - The requested run-state link rate
Return	`drvret` • DRV_OK - on success

Name	grspwrouter_port_link_start()
	• DRV_INVAL - port is not a SpaceWire port or invalid link rate

Table 17.13. `grspwrouter_port_link_stop` function declaration

Name	grspwrouter_port_link_stop()
Proto	`drvret grspwrouter_port_link_stop( grspwrouter_dev * dev, uint8_t port )`
About	Stops the SpaceWire port link This function can only be called on a SpaceWire port, not an AMBA port.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Return	`drvret` • DRV_OK - on success • DRV_INVAL - port is not a SpaceWire port

The current state of the link can be checked by using `grspwrouter_port_link_status()`. Possible states are:

- error reset (GRSPWROUTER_LINK_ERROR_RESET)
- error wait (GRSPWROUTER_LINK_ERROR_WAIT)
- ready (GRSPWROUTER_LINK_READY)
- started (GRSPWROUTER_LINK_STARTED)
- connecting (GRSPWROUTER_LINK_CONNECTING)
- run state (GRSPWROUTER_LINK_RUN_STATE)

Table 17.14. `grspwrouter_port_link_status` function declaration

Name	grspwrouter_port_link_status()
Proto	`drvret grspwrouter_port_link_status( grspwrouter_dev * dev, uint8_t port, link_state * status )`
About	Returns the link state of the SpaceWire port This function can only be called on a SpaceWire port, not an AMBA port.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Param	`status` - [out] - The current link state
Return	`drvret` • DRV_OK - on success • DRV_INVAL - port is not a SpaceWire port

An overview of the run state of all links can be read out by `grspwrouter_link_status()`, which return a bitmask indicating which links are in run state.

Table 17.15. `grspwrouter_link_status` function declaration

Name	grspwrouter_link_status()
Proto	`void grspwrouter_link_status( grspwrouter_dev * dev, uint32_t * run_state )`
About	Return list of SpaceWire ports with links in runstate The mask returned by the function indicates which SpaceWire port links are in runstate. Bit 1 is SpaceWire port 1, bit 2 is SpaceWire port 2, and so on.

Name	grspwrouter_link_status()
Param	`dev` - [in] - Valid router device handle
Param	`run_state` - [out] - Mask indicating runstate of each link

The status of a port can be checked with `grspwrouter_port_status()`. This includes information on any error events that have occurred and if the port is currently transmitting or receiving data.

Table 17.16. `grspwrouter_port_status` function declaration

Name	grspwrouter_port_status()
Proto	`drvret grspwrouter_port_status( grspwrouter_dev * dev, uint8_t port, uint32_t * status )`
About	Return the status of the port This function returns the value of the status register for the port. The status value can be parsed using the following defines: • port type (SpaceWire/AMBA/FIFO/Custom) (GRSPWROUTER_STATUS_PORT_TYPE(status)) • a packet for which this port was the input port has been spilled due to the packet length truncation feature (GRSPWROUTER_STATUS_ERR_TRUNC) • a packet for which this port was the input port has been spilled due to the time-code / distributed interrupt code truncation feature (GRSPWROUTER_STATUS_ERR_INTTRUNC) • an RMAP / SpaceWire Plug-and-Play command received on this port was spilled by the configuration port (GRSPWROUTER_STATUS_ERR_RMAP) • a packet received on this port was spilled due to the spill-if-not-ready feature (GRSPWROUTER_STATUS_ERR_NOTRDY) • this port either was started, or currently is trying to start, due to the link-start-on-request feature (GRSPWROUTER_STATUS_START_REQUEST) • a packet that is incoming on this port currently is being spilled (GRSPWROUTER_STATUS_SPILL) • a packet arrives at this port and the port has been given access to the routing table (GRSPWROUTER_STATUS_ACTIVE_STATUS) • the active SpaceWire ports if dual ports is implemented (GRSPWROUTER_STATUS_ACTIVE_PORT) • a packet for which this port was the input port was spilled due to a packet timeout (GRSPWROUTER_STATUS_ERR_TIMEOUT) • a memory error occur while accessing the on-chip memory in the ports (GRSPWROUTER_STATUS_ERR_MEM) • transmit FIFO on this port is full (GRSPWROUTER_STATUS_TX_FIFO_FULL) • receive FIFO on this port is empty (GRSPWROUTER_STATUS_RX_FIFO_EMPTY) • current link state (GRSPWROUTER_STATUS_LINK_STATE(status)) • the number of the input port for the current or last packet transfer on this port (GRSPWROUTER_STATUS_INPUT_PORT(status)) • port is the input port of an ongoing packet transfer (GRSPWROUTER_STATUS_RX_BUSY) • port is the output port of an ongoing packet transfer (GRSPWROUTER_STATUS_TX_BUSY) • an invalid address error occurred on this port (GRSPWROUTER_STATUS_ERR_ADRS) • a credit error has occurred (GRSPWROUTER_STATUS_ERR_CREDIT) • an escape error has occurred (GRSPWROUTER_STATUS_ERR_ESCAPE) • a disconnect error has occurred (GRSPWROUTER_STATUS_ERR_DISCON) • a parity error has occurred (GRSPWROUTER_STATUS_ERR_PARITY)
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid port
Param	`status` - [out] - The port status register

Name	grspwrouter_port_status()
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - port is not a valid port

17.7. Error handling

The `grspwrouter_isr_register()` function can be used to install a handler that will be called when specified error events occur on the port, or when a link enters run state. It is possible to specify for which events the handler should be called, and for which ports.

Table 17.17. `grspwrouter_isr_register` function declaration

Name	grspwrouter_isr_register()
Proto	`void grspwrouter_isr_register( grspwrouter_dev * dev, uint32_t err_mask, uint32_t port_mask, grspwrouter_isr_func isr, void * arg )`
About	Register handler for port events Register a handler for the selected interrupt types. The defines below can be or:ed together to form the `mask` argument: • Generate an interrupt when a SpaceWire Plug and Play error has been detected in the configuration port (GRSPWROUTER_INTERRUPT_CONF_PNP) • Generate an interrupt when a packet has been spilled because of the spill-if-not-ready feature (GRSPWROUTER_INTERRUPT_NOTRDY) • Generate an interrupt when a SpaceWire link enters run-state (GRSPWROUTER_INTERRUPT_RUN_STATE) • Generate an interrupt when a packet has been spilled because of the time code / distributed interrupt code truncation feature (GRSPWROUTER_INTERRUPT_INTTRUNC) • Generate an interrupt when a packet has been spilled due to the packet length truncation feature (GRSPWROUTER_INTERRUPT_TRUNC) • Generate an interrupt when a packet has been spilled due to the timeout mechanism (GRSPWROUTER_INTERRUPT_TIMEOUT) • Generate an interrupt when either a header CRC error, protocol ID error, packet type error, early EOP, or early EEP has been detected in the configuration port (GRSPWROUTER_INTERRUPT_CONF_PORT) • Generate an interrupt when an error has been detected in the configuration port for an RMAP command such that the PSTS.EC field is set to a non-zero value (GRSPWROUTER_INTERRUPT_CONF_RMAP) • Generate an interrupt when an invalid address error has occurred on a port (GRSPWROUTER_INTERRUPT_ADRS) • Generate an interrupt when a link error (parity, escape, credit, disconnect) has been detected on a SpaceWire port (GRSPWROUTER_INTERRUPT_LINK) • Generate an interrupt when a memory error occur in any of the router's on-chip memories (GRSPWROUTER_INTERRUPT_MEM) The define GRSPWROUTER_INTERRUPT_ALL can be used to enable all interrupt types and GRSPWROUTER_INTERRUPT_NONE to disable all interrupt types.
Param	`dev` - [in] - Valid router device handle
Param	`err_mask` - Interrupts that the handler should trigger on
Param	`port_mask` - Ports that the interrupts can be generated for
Param	`isr` - [in] - Interrupt handler function pointer
Param	`arg` - [in] - Custom argument to interrupt handler

17.8. Time codes

To make it possible to send time codes the time code support needs to be enabled both globally in the router and for each port that shall send or receive them. The router will keep track of the current time code, but initiating a time code change or handling interrupts codes must be done via an AMBA port using the SpaceWire driver (Chapter 6).

Time codes are enabled globally by `grspwrouter_tc_enable()` and per port by `grspwrouter_port_tc_enable()`. Using the latter function the router can be configured to ignore the time code values it receives from the AMBA port and instead always use its internal time representation.

Table 17.18. `grspwrouter_tc_enable` function declaration

Name	**grspwrouter_tc_enable()**
Proto	`void grspwrouter_tc_enable( grspwrouter_dev * dev )`
About	Enable the handling of time codes Enable the router time code support. Also needs to be enabled for each port that intend to use time codes using `grspwrouter_port_tc_enable`.
Param	`dev` - [in] - Valid router device handle

Table 17.19. `grspwrouter_tc_disable` function declaration

Name	**grspwrouter_tc_disable()**
Proto	`void grspwrouter_tc_disable( grspwrouter_dev * dev )`
About	Disable time code support Disable the router time code support.
Param	`dev` - [in] - Valid router device handle

Table 17.20. `grspwrouter_port_tc_enable` function declaration

Name	**grspwrouter_port_tc_enable()**
Proto	`drvret grspwrouter_port_tc_enable( grspwrouter_dev * dev, uint8_t port, bool router_time )`
About	Enable time code support This function enables time codes to be sent and received via the port. If `router_time` is `true` the router will not look at the timer value and instead use its internal time representation. Time code support also needs to be enabled globally using `grspwrouter_tc_enable`.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid port
Param	`router_time` - If true, always use the routers time, never the incoming time
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if invalid port

Table 17.21. `grspwrouter_port_tc_disable` function declaration

Name	**grspwrouter_port_tc_disable()**
Proto	`drvret grspwrouter_port_tc_disable( grspwrouter_dev * dev, uint8_t port )`
About	Disable time code support Disables support for time codes for the port. Any time codes received will be dropped.
Param	`dev` - [in] - Valid router device handle

Name	grspwrouter_port_tc_disable()
Param	`port` - Index of a valid SpaceWire port
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if invalid port

The internal time representation can be read out with `grspwrouter_tc_get()` and set to 0 with `grspwrouter_tc_reset()`.

Table 17.22. `grspwrouter_tc_get` function declaration

Name	grspwrouter_tc_get()
Proto	`uint8_t grspwrouter_tc_get( grspwrouter_dev * dev )`
About	Return the current time code Returns routers internal time representation.
Param	`dev` - [in] - Valid router device handle
Return	`uint8_t` - Current time code

Table 17.23. `grspwrouter_tc_reset` function declaration

Name	grspwrouter_tc_reset()
Proto	`void grspwrouter_tc_reset( grspwrouter_dev * dev )`
About	Set the current time code to 0 Sets the routers internal time representation to 0.
Param	`dev` - [in] - Valid router device handle

17.9. Interrupt codes

The routing of interrupt-codes needs to be enabled both for the router and per port. For the router it is enabled by `grspwrouter_ic_enable()`. When enabling the interrupt code support it is possible to set a time out that will trigger an interrupt if an acknowledge reply is not received within the specified time period (100µs - 6.5s).

It also possible to set a cooldown period to protect against being flooded by interrupt codes (100µs - 25ms). A new interrupt-code will not be registered until the cooldown has expired. Both the timeout and cooldown are optional and can be disabled by setting the time period to 0.

Table 17.24. `grspwrouter_ic_enable` function declaration

Name	grspwrouter_ic_enable()
Proto	`drvret grspwrouter_ic_enable( grspwrouter_dev * dev, uint32_t timeout, uint32_t cooldown )`
About	Enable interrupt code support Enable the router interrupt code support. Also needs to be enabled for each port that intend to send or receive interrupt codes using `grspwrouter_port_ic_enable`. A timer can be configured that will trigger an interrupt when an acknowledge reply is not received within the specified time period (100µs - 6.5s). A cooldown period can be configured that prevents new interrupts from being submitted until the specified time period has passed (100µs - 3.1ms). Set the timeout to zero to disable.
Param	`dev` - [in] - Valid router device handle
Param	`timeout` - Timeout in microseconds (or 0 to disable) (100 - 6553500 in steps of 100)

Name	grspwrouter_ic_enable()
Param	`cooldown` - Cooldown period in microseconds (or 0 to disable) (100 - 3100 in steps of 100)
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if the timeout or cooldown value is too big

Table 17.25. `grspwrouter_ic_disable` function declaration

Name	grspwrouter_ic_disable()
Proto	`void grspwrouter_ic_disable( grspwrouter_dev * dev )`
About	Disable interrupt code support Disable interrupt code support for all ports in router
Param	`dev` - [in] - Valid router device handle

The per port interrupt-code support is enabled by `grspwrouter_port_ic_enable()`. By default it enables forwarding of both interrupt codes and interrupt acknowledgement codes in both directions, but it is possible to disable the transmission or reception of interrupt or interrupt acknowledgement codes.

Table 17.26. `grspwrouter_port_ic_enable` function declaration

Name	grspwrouter_port_ic_enable()
Proto	`drvret grspwrouter_port_ic_enable( grspwrouter_dev * dev, uint8_t port, uint32_t options )`
About	Enable interrupt code support for port By default forwarding of both interrupt codes and interrupt acknowledgement codes in both direction are enabled. This can be changed by or:ing the defines below together to form an `options` argument: • Disable the transmission of interrupt codes (GRSPWROUTER_IC_DIS_TX_INT) • Disable the reception of interrupt codes (GRSPWROUTER_IC_DIS_RX_INT) • Disable the transmission of interrupt acknowledgement codes (GRSPWROUTER_IC_DIS_TX_ACK) • Disable the reception of interrupt acknowledgement codes (GRSPWROUTER_IC_DIS_RX_ACK) Interrupt code support also needs to be enabled globally using `grspwrouter_ic_enable`.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Param	`options` - Options mask
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if invalid port

Table 17.27. `grspwrouter_port_ic_disable` function declaration

Name	grspwrouter_port_ic_disable()
Proto	`drvret grspwrouter_port_ic_disable( grspwrouter_dev * dev, uint8_t port )`
About	Disable interrupt code support for port Disables support for interrupt codes for the port. Any interrupt codes received will be dropped.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port

Name	grspwrouter_port_ic_disable()
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if invalid port

Using the `grspwrouter_port_code_truncation()` function it is possible to abort the currently received packet when an interrupt code or time code with a specified value is received. The packet will be truncated and marked with an EEP.

Table 17.28. `grspwrouter_port_code_truncation` function declaration

Name	grspwrouter_port_code_truncation()
Proto	`drvret grspwrouter_port_code_truncation( grspwrouter_dev * dev, uint8_t port, bool enable, uint8_t value, uint8_t mask )`
About	Abort packet on time/interrupt code Configure the port to abort the current packet if a time or interrupt code with the specified value is received.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Param	`enable` - Enable packet truncation
Param	`value` - The value that can cause truncation
Param	`mask` - Mask for the value
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if invalid port

17.10. Configure timeouts

The packet timeout functionality is enabled by `grspwrouter_port_timeout()`. It possible to enable it for overruns (when the input port has data available, but the output port(s) can not accept data fast enough) and underruns (when the output port(s) can accept more data, but the input port can not provide data fast enough). It is also possible to use it to automatically stop the link if it has not been used within the specified time.

Table 17.29. `grspwrouter_port_timeout` function declaration

Name	grspwrouter_port_timeout()
Proto	`drvret grspwrouter_port_timeout( grspwrouter_dev * dev, uint8_t port, uint32_t timeout, bool overrun, bool underrun, bool autodisconnect )`
About	Enable timeouts Enable a timeout for packets transfers (overrun and underrun) and auto-disconnect per port. An overrun timeout occurs when the input port has data available but the output port(s) can not accept data fast enough. An underrun timeout occurs when the output port(s) can accept more data but the input port can not provide data fast enough. The timeout can be set to between 100µs - 6.5s.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Param	`timeout` - The timeout in microseconds (100 - 6553500 in steps of 100)
Param	`overrun` - Enable for overrun
Param	`underrun` - Enable for underrun
Param	`autodisconnect` - Enable for auto disconnect (Only for SpaceWire ports)

Name	grspwrouter_port_timeout()
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if trying to enable auto disconnect on non-SpaceWire port, or if invalid port, or if the timeout value is too big

17.11. Configure packet max length

A max packet length can be configured for each port. If a packet exceeds this length it will be truncated by the router and get an error end of packet (EEP). The max packet length is set by the `grspwrouter_port_max_length()` function.

Table 17.30. `grspwrouter_port_max_length` function declaration

Name	grspwrouter_port_max_length()
Proto	`drvret grspwrouter_port_max_length( grspwrouter_dev * dev, uint8_t port, uint32_t length )`
About	Set the maximum length of packets If an incoming packets is larger it will be truncated and marked with an EEP. Use the length 0 to accept any length.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Param	`length` - The maximum length of the packet or 0 to disable
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if invalid port

17.12. Configure Plug-and-Play

The router supports the SpaceWire Plug-and-Play protocol which can be used to discover devices on the network. The `grspwrouter_pnp_set()` function is used to set the vendor id, product id, and serial number of the device which is presented to any device scanning the network using the protocol.

Table 17.31. `grspwrouter_pnp_set` function declaration

Name	grspwrouter_pnp_set()
Proto	`void grspwrouter_pnp_set( grspwrouter_dev * dev, uint16_t vendor_id, uint16_t product_id, uint32_t serial, bool keep_instance_id )`
About	Set the SpaceWire Plug-and-Play information Sets the serial number, vendor id, and product id that is presented when accessing this device using the SpaceWire Plug-and-Play protocol. Bits 3:0 of the serial number can be set using the INSTANCEID[7:0] signal. Use `keep_instance_id` to preserve this part of the serial number.
Param	`dev` - [in] - Valid router device handle
Param	`vendor_id` - Custom vendor id
Param	`product_id` - Custom product id
Param	`serial` - Custom serial number
Param	`keep_instance_id` - Use reset value for bits 3:0 of serial number

17.13. Read out credit counters

The credit counter for a SpaceWire port can be read out using `grspwrouter_port_cred()`. It can only be called on a SpaceWire port and will return an error if used on an AMBA port.

Table 17.32. `grspwrouter_port_cred` function declaration

Name	**grspwrouter_port_cred()**
Proto	`drvret grspwrouter_port_cred( grspwrouter_dev * dev, uint8_t port, uint8_t * in, uint8_t * out )`
About	Read the credit counters for the port Returns the current credit counters for the SpaceWire port. Can not be used on an AMBA port.
Param	`dev` - [in] - Valid router device handle
Param	`port` - Index of a valid SpaceWire port
Param	`in` - [out] - Incoming credit
Param	`out` - [out] - Outgoing credit
Return	`drvret` • `DRV_OK` - on success • `DRV_INVAL` - if invalid SpaceWire port

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