RCC Driver Manual

Table of Contents

1. Introduction 9
2. GRLIB AMBA Plug&Play bus 10
    2.1. Introduction 10
        2.1.1. AMBA Plug&Play terms and names 10
        2.1.2. Sources 10
    2.2. Overview 11
    2.3. Initialization 11
    2.4. Finding AMBAPP devices by Plug&Play 11
    2.5. Allocating a device structure 12
    2.6. Name database 12
    2.7. Frequency of a device 12
3. Driver Manager 13
    3.1. Introduction 13
        3.1.1. Driver manager terms and names 13
        3.1.2. Sources 13
    3.2. Overview 13
        3.2.1. Bus and bus driver 14
        3.2.2. Root driver 16
        3.2.3. Device driver 16
        3.2.4. Device 17
        3.2.5. Driver resources 17
        3.2.6. Driver interface 18
    3.3. Configuration 18
        3.3.1. Available LEON drivers 19
    3.4. Initialization 20
        3.4.1. LEON3/4 BSP 20
    3.5. Interrupt 20
    3.6. Address translation 21
    3.7. Function Interface 22
4. RMAP Stack 23
    4.1. Introduction 23
        4.1.1. Examples 23
    4.2. Driver Interface 23
    4.3. Logical and Path addressing 23
    4.4. Zero-copy implementation 23
    4.5. RMAP GRSPW driver 23
    4.6. Thread-safe 24
    4.7. User interface 24
        4.7.1. Data structures 24
        4.7.2. Function interface description 25
5. SpaceWire Network model 27
    5.1. Introduction 27
    5.2. Overview 27
    5.3. Requirements 27
    5.4. Node Description 27
        5.4.1. The Node ID 27
    5.5. Read and write operation 27
    5.6. Interrupt handling 28
    5.7. Using the spacewire bus driver 28
6. AMBA over SpaceWire 29
    6.1. Introduction 29
    6.2. Overview 29
    6.3. Requirements 29
    6.4. Interrupt handling 29
    6.5. Memory allocation on target 29

6.6. Differences between on-chip AMBA drivers 29
7. LEON PCI host briedge drivers 31
7.1. Introduction 31
7.1.1. Examples 31
7.2. Sources 31
7.3. Configuration 31
7.3.1. GRPCI 31
7.3.2. GRPCI2 32
7.3.3. AT697 32
7.4. User interface 32
7.4.1. PCI address space 32
7.4.2. PCI interrupt 33
7.4.3. PCI endianess 33
8. GR-RASTA-ADCDAC PCI peripheral 34
9. GR-RASTA-IO PCI peripheral 35
10. GR-RASTA-TMTC PCI peripheral 36
11. GR-RASTA-SPW_ROUTER PCI Peripheral 37
12. GR-CPCI-LEON4-N2X PCI Peripheral 38
12.1. Driver registration 38
12.2. Driver resource configuration 38
13. GRSPW Packet driver 39
13.1. Introduction 39
13.1.1. GRSPW packet driver vs. old GRSPW driver 39
13.1.2. Hardware Support 39
13.1.3. Driver sources 39
13.1.4. Show routines 39
13.1.5. Examples 39
13.1.6. Known driver limitations 39
13.2. Software design overview 40
13.2.1. Overview 40
13.2.2. Driver resource configuration 40
13.2.3. Initialization 40
13.2.4. Link control 41
13.2.5. Time Code support 41
13.2.6. RMAP support 42
13.2.7. Port support 42
13.2.8. SpaceWire node address configuration 42
13.2.9. SpaceWire Interrupt Code support 42
13.2.10. User DMA buffer handling 43
13.2.11. Driver DMA buffer handling 43
13.2.12. Polling and blocking mode 45
13.2.13. Interrupt and work-task 45
13.2.14. Starting and stopping DMA 46
13.2.15. Thread concurrency 46
13.2.16. SMP Support 47
13.3. Device Interface 48
13.3.1. Opening and closing device 48
13.3.2. Hardware capabilities 49
13.3.3. Link Control 50
13.3.4. Node address configuration 52
13.3.5. Time Code support 53
13.3.6. Port Control 54
13.3.7. RMAP Control 56
13.3.8. Statistics 57
13.4. DMA interface 58
13.4.1. Opening and closing DMA channels 58
13.4.2. Starting and stopping DMA operation 60
13.4.3. Packet buffer description 61

13.4.4. Blocking/Waiting on DMA activity 62
13.4.5. Sending packets 64
13.4.6. Receiving packets 66
13.4.7. Transmission queue status 69
13.4.8. Statistics 70
13.4.9. DMA channel configuration 72
13.5. API reference 73
13.5.1. Data structures 73
13.5.2. Device functions 73
13.5.3. DMA functions 74
14. GRSPW GRLIB SpaceWire driver 76
14.1. Introduction 76
14.1.1. Software driver 76
14.1.2. Examples 76
14.2. User interface 76
14.2.1. Driver registration 76
14.2.2. Driver resource configuration 76
14.2.3. Opening the device 77
14.2.4. Closing the device 77
14.2.5. I/O Control interface 78
14.2.6. Transmission 85
14.2.7. Reception 85
14.3. Receiver example 86
15. SpaceWire router 87
15.1. Introduction 87
15.1.1. SpaceWire Router register driver 87
15.1.2. AMBA port driver 87
16. SpaceWire router register driver 88
16.1. Introduction 88
16.2. User interface 88
16.2.1. Driver registration 88
16.2.2. Driver resource configuration 88
16.2.3. Opening the device 88
16.2.4. Closing the device 88
16.2.5. I/O Control interface 89
17. GR1553B GRLIB MIL-STD-1553B driver 94
17.1. Introduction 94
17.2. GR1553B Hardware 94
17.3. Software driver 94
17.4. Driver Registration 94
17.5. Examples 94
18. GR1553B remote terminal driver 95
18.1. Introduction 95
18.1.1. GR1553B Remote Terminal Hardware 95
18.1.2. Examples 95
18.2. User Interface 95
18.2.1. Overview 95
18.2.2. Application Programming Interface 98
19. GR1553B bus monitor driver 105
19.1. Introduction 105
19.1.1. GR1553B Remote Terminal Hardware 105
19.1.2. Examples 105
19.2. User Interface 105
19.2.1. Overview 105
19.2.2. Application Programming Interface 107
20. GR1553B bus controller driver 110
20.1. Introduction 110
20.1.1. GR1553B Bus Controller Hardware 110

20.1.2. Software driver 110
20.1.3. Examples 111
20.2. BC Device Handling 111
20.2.1. Device API 111
20.3. Descriptor List Handling 113
20.3.1. Overview 113
20.3.2. Example: steps for creating a list 114
20.3.3. Major Frame 115
20.3.4. Minor Frame 115
20.3.5. Slot (Descriptor) 115
20.3.6. Changing a scheduled BC list (during BC-runtime) 116
20.3.7. Custom Memory Setup 116
20.3.8. Interrupt handling 117
20.3.9. List API 117
21. B1553BRM GRLIB Actel Core1553BRM driver 125
21.1. Introduction 125
21.1.1. BRM Hardware 125
21.1.2. Software Driver 125
21.1.3. Supported OS 125
21.2. User Intrerface 125
21.2.1. Driver registration 125
21.2.2. Driver resource configuration 126
21.2.3. Opening the device 126
21.2.4. Closing the device 127
21.2.5. I/O Control interface 127
21.2.6. Configuration 130
21.2.7. Remote Terminal operation 132
21.2.8. Bus Controller operation 132
21.2.9. Bus monitor operation 133
22. B1553RT GRLIB Actel Core1553 RT driver 134
22.1. Introduction 134
22.1.1. RT Hardware 134
22.1.2. 1.1.2 Examples 134
22.2. User interface 134
22.2.1. Driver registration 134
22.2.2. Driver resource configuration 134
22.2.3. Opening the device 135
22.2.4. Closing the device 135
22.2.5. I/O Control interface 135
22.2.6. 136
22.2.7. Remote Terminal operation 138
23. GRCAN CAN driver 139
23.1. User interface 139
23.1.1. Driver registration 139
23.1.2. Driver resource configuration 139
23.1.3. Opening the device 139
23.1.4. Closing the device 140
23.1.5. I/O Control interface 140
23.1.6. Transmission 145
23.1.7. Reception 145
24. CAN_OC GRLIB Opencores CAN driver 147
24.1. Introduction 147
24.1.1. CAN Hardware 147
24.1.2. Software Driver 147
24.1.3. Examples 147
24.2. User interface 147
24.2.1. Driver registration 147
24.2.2. Driver resource configuration 147

24.2.3. Opening the device 148
24.2.4. Closing the device 148
24.2.5. I/O Control interface 148
24.2.6. Reception 153
25. SatCAN driver (SatCAN) 154
25.1. Introduction 154
25.1.1. SatCAN Hardware Wrapper 154
25.1.2. Software Driver 154
25.1.3. Examples 154
25.2. User interface 154
25.2.1. Driver registration 154
25.2.2. Opening the device 155
25.2.3. Closing the device 156
25.2.4. Reading from the device 156
25.2.5. Writing to the device 156
25.2.6. I/O Control interface 157
26. CAN_MUX driver (CAN_MUX) 162
26.1. Introduction 162
26.1.1. CAN_MUX Hardware 162
26.1.2. Software Driver 162
26.1.3. Examples 162
26.2. User interface 162
26.2.1. Driver registration 162
26.2.2. Opening the device 162
26.2.3. Closing the device 162
26.2.4. I/O Control interface 162
27. GRASCS driver 164
27.1. Introduction 164
27.1.1. Software driver 164
27.1.2. Examples 164
27.2. User interface 164
27.2.1. ASCS_init 164
27.2.2. ASCS_input_select 165
27.2.3. ASCS_etr_select 165
27.2.4. ASCS_start 165
27.2.5. ASCS_stop 165
27.2.6. ASCS_iface_status 165
27.2.7. ASCS_TC_send 166
27.2.8. ASCS_TC_send_block 166
27.2.9. ASCS_TC_sync_start 166
27.2.10. ASCS_TC_sync_stop 166
27.2.11. ASCS_TM_recv 167
27.2.12. ASCS_TM_recv_block 167
27.3. Examples code 167
28. APBUART console driver 169
28.1. Introduction 169
28.1.1. Functioning mode 169
28.1.2. Hardware support 169
28.2. User interface 169
28.2.1. Driver registration 169
28.2.2. Driver resource configuration 169
28.2.3. Opening the device 170
28.2.4. Closing the device 170
28.2.5. Control interface 170
28.2.6. Startup-parameter inheritance 171
28.2.7. Transmission 171
28.2.8. Reception 171
29. APBUART - Raw UART driver interface 172

29.1. Limitations 172
29.2. User interface 172
29.2.1. Driver registration 172
29.2.2. Driver resource configuration 172
29.2.3. Opening the device 172
29.2.4. Closing the device 173
29.2.5. I/O Control interface 173
29.2.6. Transmission 175
29.2.7. Reception 175
30. SPICTRL GRLIB SPI master driver 177
30.1. Introduction 177
30.1.1. SPI Hardware 177
30.1.2. Examples 177
30.2. User interface 177
30.2.1. Driver registration 177
30.2.2. Accessing the SPI bus 177
30.2.3. Extensions to the standard RTEMS interface 178
31. I2CMST GRLIB I2C Master driver 181
31.1. Introduction 181
31.1.1. I2C Hardware 181
31.1.2. Examples 181
31.2. User interface 181
31.2.1. Driver registration 181
31.2.2. Accessing the I2C bus 181
32. GPIO Library 182
32.1. Introduction 182
32.1.1. Examples 182
32.2. Driver interface 182
32.3. User interface 182
32.3.1. Accessing a GPIO port 182
32.3.2. Interrupt handler registration 183
32.3.3. Data structures 183
32.3.4. Function prototype description 183
33. GRGPIO GRLIB GPIO driver 185
33.1. Introduction 185
33.1.1. GPIO Hardware 185
33.1.2. Examples 185
33.2. User interface 185
33.2.1. Driver registration 185
33.2.2. Driver resource configuration 185
33.2.3. Accessing GPIO ports 185
34. GRADCDAC GRLIB ADC/DAC driver 188
34.1. Introduction 188
34.1.1. ADC/DAC Hardware 188
34.1.2. Examples 188
34.2. User interface 188
34.2.1. Driver registration 188
34.2.2. Driver resource configuration 188
34.2.3. Accessing ADC/DAC 188
34.2.4. Interrupt handler registration 189
34.2.5. Data structures 189
34.2.6. Function prototype description 191
35. GRTC GRLIB CCSDS Telecommand driver 195
35.1. INTRODUCTION 195
35.1.1. TC Hardware 195
35.1.2. Software Driver 195
35.2. User interface 195
35.2.1. Driver registration 195

35.2.2. Opening the device 195
35.2.3. Closing the device 196
35.2.4. I/O Control interface 196
35.2.5. Operating mode 204
35.2.6. Reception in FRAME mode 204
35.2.7. Reception using RAW mode 204
36. GRTM GRLIB CCSDS Telemetry Driver 206
36.1. Introduction 206
36.1.1. TM Hardware 206
36.1.2. Software Driver 206
36.2. User interface 206
36.2.1. Driver registration 206
36.2.2. Opening the device 207
36.2.3. Closing the device 207
36.2.4. I/O Control interface 207
36.2.5. Transmission 214
37. GRCTM driver 215
37.1. Introduction 215
37.1.1. Examples 215
37.1.2. User interface 215
38. SPWCUC driver 218
38.1. Introduction 218
38.1.1. Examples 218
38.2. User interface 218
38.2.1. Overview 218
38.2.2. Application Programming Interface 218
39. GRPWRX GRLIB PacketWire Receiver driver 222
39.1. Introduction 222
39.1.1. Software Driver 222
39.2. User interface 222
39.2.1. Driver registration 222
39.2.2. Opening the device 222
39.2.3. Closing the device 223
39.2.4. I/O Control interface 223
39.2.5. Reception 228
40. GRAES GRLIB AES DMA driver 229
40.1. Introduction 229
40.1.1. Software Driver 229
40.2. User interface 229
40.2.1. Driver registration 229
40.2.2. Opening the device 229
40.2.3. Closing the device 230
40.2.4. I/O Control interface 230
40.2.5. De/encryption 235

1. Introduction

This document contain a compilation of documents describing most of the LEON3 and LEON2 drivers included in the Gaisler RTEMS distribution. Each driver is described in a separate chapter.

Most of the drivers for GRLIB cores relies on the RTEMS Driver Manager for a number of services. The manager is responsible to unite a driver with the hardware the driver is intended for and creating a device instance. The driver manager is documented in a separate chapter.

Gaisler RTEMS samples and a common makefile can be found under `/opt/rtems-4.10/src/samples` in the distribution. The examples are often composed of a transmitting task and a receiving task communicating to one another. The tasks are either intended to run on the same board requiring two cores, or run on different boards requiring multiple boards with one core each, or both. The tasks use the console to print their progress and status.

2. GRLIB AMBA Plug&Play bus

2.1. Introduction

The AMBA bus that GRLIB is built upon supports Plug&Play probing of device information. This section gives an overview of the AMBA Plug&Play (AMBAPP) routines that comes with the RCC distribution. Systems without on-chip AMBA Play&Play support (AT697 for example) may use the library when accessing remote GRLIB systems over SpaceWire or PCI.

The AMBAPP Layer is used by the AMBAPP Bus driver used to interface the AMBAPP bus to the driver manager. Note that the AMBAPP Bus is not documented here.

2.1.1. AMBA Plug&Play terms and names

Throughout this document some software terms and names are frequently used, below is table that summarizes some of them.

Table 2.1. AMBA Layer terms and names

Term	Description
AMBAPP, AMBA PnP	AMBA Plug&Play bus. See AHBCTRL and APBCTRL in GRIP documentation.
AMBA	AMBA bus without Plug&Play information, typically used in LEON2 designs
device	AMBA AHB Master, AHB Slave or APB Slave interface. The ambapp_dev structure describe any of the interfaces.
core	A AMBA IP core often consists of multiple AMBA interfaces but not more than one interface of the same type. The ambapp_core structure is used to describe a AMBA core as a unit with up to three interfaces all of different type.
bus	All AMBA AHB and APB buses in a system in one ambapp_bus structure. See scanning.
ambapp_plb	The processor local AMBA PnP bus in LEON3 BSP, RAM description of first Plug&Play bus at 0xFFF00000.
scanning	Process where the AMBA PnP bus is searched for all AMBA interfaces and a description is created in RAM, the RAM copy makes it easier to access the PnP information rather than accessing directly.
depth	Number of levels of AHB-AHB bridges from topmost AHB bus.

2.1.2. Sources

The sources of the driver manager is found according to the table below.

Table 2.2. AMBAPP Layer Sources

Path	Description
ambapp.h	Include path of AMBAPP layer header-file definitions
ambapp_ids.h	Vendor and Device IDs auto generated from GRLIB devices.vhd
libbsp/sparc/shared/amba	Path within RTEMS sources to AMBAPP sources
ambapp.c	Scanning routine `ambapp_scan()` and function to iterate over all devices `ambapp_for_each()`
ambapp_alloc.c	Mark ownership of devices
ambapp_count.c	Helper function to get number of devices found in Plug & Play
ambapp_depth.c	Function to get bus depth of a device
ambapp_find_by_idx.c	Helper function `ambapp_find_by_idx()` used as input to `ambapp_for_each()` to search for a matching device by index.

Path	Description
ambapp_freq.c	Functions to initialize AMBAPP bus frequency and get the frequency of a device
ambapp_names.c	Vendor and device ID name database
ambapp_old.c	Old AMBAPP interface, reimplemented on top of ambapp.c. Deprecated.
ambapp_parent.c	Get device parent bridge by searching the device tree
ambapp_show.c	Print AMBAPP bus RAM description information onto terminal, for debugging

2.2. Overview

The AMBAPP layer provides functions for device drivers to access the AMBA Plug&Play information in an easy way by reading a RAM description rather than accessing the Plug&Play ROM information directly. It is also beneficial to have a RAM description for remote systems over SpaceWire or PCI where scanning often must be performed once at initialization.

The AMBAPP interface is defined in `ambapp.h` and vendor/device IDs in `ambapp_ids.h`.

2.3. Initialization

Before accessing the AMBAPP interface one must initialize the ambapp_bus RAM description by scanning the AMBA Plug&Play information for all buses, bridges and devices. The bus is scanned by calling `ambapp_scan()` with prototype as listed below, the RAM description will be written to abus. The function takes an optional access function memfunc called when the AMBA library read the PnP information, the abus argument is passed along to memfunc which makes it possible for the caller to have a custom argument to memfunc. If addresses found in the Plug&Play information must be translated (as with AMBA-over-PCI for example) the mmaps array must point to address translation information. The scanning routine starts scanning at (ioarea | 0x000ff00), the default Plug&Play area is located at 0xFFF0000.

```
int ambapp_scan(
    struct ambapp_bus *abus,
    unsigned int ioarea,
    ambapp_memcpy_t memfunc,
    struct ambapp_mmap *mmaps
);
```

A bus and device tree is created in abus during initialization, cores (struct ambapp_core) are not created by the layer. The AMBAPP layer is used from the AMBAPP Bus driver in the driver manager, it creates AMBAPP cores by finding AMBA devices that comes from the same IP core.

The frequency of the AMBAPP bus can not be read from the Plug&Play information, however how different AMBA AHB buses frequency relates to each can be found at respective AHB-AHB bridge. In order for the frequency function `ambapp_freq_get()` to report a correct frequency the user is required to register the frequency of one AMBAPP device calling the `ambapp_freq_init()` function, prototype listed below. The LEON3 BSP determines the frequency by assuming that the first GPTIMER clock frequency has been initialized to 1MHz by boot loader, the BSP registers the frequency of the GPTIMER APB device.

```
/* Initialize the frequency [Hz] of all AHB Buses from knowing the
 * frequency of one particular APB/AHB Device.
 */
void ambapp_freq_init(
      struct ambapp_bus *abus,
      struct ambapp_dev *dev,
      unsigned int freq);

/* Returns the frequency [Hz] of a AHB/APB device */
unsigned int ambapp_freq_get(
      struct ambapp_bus *abus,
      struct ambapp_dev *dev);
```

2.4. Finding AMBAPP devices by Plug&Play

After the Plug&Play information has been scanned the user can search for AMBA devices in the RAM description without accessing the Plug&Play ROM by calling `ambapp_for_each()`, see prototype below. The user provided function is called every time the search options matches a AMBA device in the device tree. The

ambapp_for_each() function can search for a any combination of [VENDOR, DEVICE] ID, device types (AHB MST, AHB SLV and/or APB SLV), free or allocated devices. If a VENDOR/DEVICE ID of -1 is given the function will match any vendors/devices.

```
/* Iterates through all AMBA devices previously found, it calls func
 * once for every device that match the search arguments.
 *
 * SEARCH OPTIONS
 * All search options must be fulfilled, type of devices searched (options)
 * and AMBA Plug&Play ID [VENDOR,DEVICE], before func() is called. The options
 * can be use to search only for AMBA APB or AHB Slaves or AHB Masters for
 * example. Note that when VENDOR=-1 or DEVICE=-1 it will match any vendor or
 * device ID, this means setting both VENDOR and DEVICE to -1 will result in
 * calling all devices matches the options argument.
 *
 * \param abus     AMBAPP Bus to search
 * \param options  Search options, see OPTIONS_* above
 * \param vendor   AMBAPP VENDOR ID to search for
 * \param device   AMBAPP DEVICE ID to search for
 * \param func     Function called for every device matching search options
 * \param arg      Optional argument passed on to func
 *
 * func return value affects the search, returning a non-zero value will
 * stop the search and ambapp_for_each will return immediately returning the
 * same non-zero value.
 *
 * Return Values
 * 0 - all devices was scanned
 * non-zero - stopped by user function returning the non-zero value
 */

int ambapp_for_each(
       struct ambapp_bus *abus,
       unsigned int options,
       int vendor,
       int device,
       ambapp_func_t func,
       void *arg);
```

2.5. Allocating a device structure

A device can be marked allocated so that other parts of the code knows that the device has been taken, this feature is not used by the LEON BSPs. The ambapp_dev.owner field is set to a non-zero value to mark that the device is allocated, use `ambapp_alloc_dev()` and `ambapp_free_dev()` to set allocation mark.

2.6. Name database

In `ambapp_names.c` AMBA Plug&Play vendor and device names are stored in a name database. The names are taken from `device.vhd` in GRLIB distribution. Names can be requested by calling appropriate function listed below.

```
/* Get Device Name from AMBA PnP name database */
char *ambapp_device_id2str(int vendor, int id);

/* Get Vendor Name from AMBA PnP name database */
char *ambapp_vendor_id2str(int vendor);

/* Set together VENDOR_DEVICE Name from AMBA PnP name database. Return length
 * of C-string stored in buf not including string termination '\0'.
 */
int ambapp_vendev_id2str(int vendor, int id, char *buf);
```

2.7. Frequency of a device

As described in the initialization section every AHB bus may have a unique bus frequency, APB buses always have the same frequency as the AHB bus it is situated on. Since a core may consist of a AHB master, AHB slave and a APB slave interface the frequencies of the different interfaces may vary. The AMBAPP layer provides a function `ambapp_freq_get()` that returns the frequency in Hz of a single device interface.

```
/* Returns the frequency [Hz] of a AHB/APB device */
unsigned int ambapp_freq_get(
      struct ambapp_bus *abus,
      struct ambapp_dev *dev);
```

3. Driver Manager

3.1. Introduction

This section describes the Driver Manager available in RCC-1.2 distribution. The sources are located in `cpukit/libdrvmgr` in the RTEMS source release. The driver manager is used to simplify the handling of buses, devices, bus drivers, device driver, configuration of device instances and providing a common programming interface where possible for drivers regardless of bus architecture.

3.1.1. Driver manager terms and names

Throughout this document some terms and names are frequently used, below is table that summarizes some of them.

Table 3.1. Driver Manager terms and names

Term	Description
Bus	Describes a bus with child devices.
Device	Describes a hardware device situated on a bus, bus driver.
Bridge device	A device with a child bus.
Bus drives	Software that handles a bus, implements the bus.
Device driver	Software that handles hardware devices.
Root device	Topmost device in driver manager tree, has no parent bus.
Root bus	Topmost bus, the root device exports, has no parent bus.
Register bus	Process where the driver manager is informed about the existence of a new bus.
Register device	Process where the driver manager is informed about the existence of a new device.
Register driver	Before driver manager initialization, drivers are added into a internal driver list.
Unite device and driver	Process where the driver manager finds a device driver for a device.
Separate device and driver	Process where a device driver is requested to never use the device any more, for example before a device is removed.
Unregister bus or device	Inform driver manager about that a bus or device (and all child buses/devices) should be removed from the device tree and related drivers be informed.
Init level	The device driver and bus driver initialization process is performed in multiple stages, called initialization levels.

3.1.2. Sources

The sources of the driver manager is found according to the table below.

Table 3.2. Driver Manager Sources

Path	Description
`cpukit/libdrvmgr`	Path within RTEMS sources. Driver manager sources
`drvmgr/drvmgr.h`	Include path of driver manager definitions
`drvmgr/drvmgr_confdefs.h`	Include patch of driver configuration

3.2. Overview

The driver manager works with the concepts bus, bus driver, device, device driver and driver resources. Since everything is tied together somehow it is quite difficult to start describing the driver manager, instead each com-

ponent is described in a separate section below and the following text assumes that the reader has knowledge of respective component.

The driver manager manages all buses and devices in a system by using a tree structure. The root of the tree starts with the root device created by the root bus driver. The root device creates a bus which is called the root bus, it is an ordinary bus without a parent bus. All buses have a linked list of devices which are situated directly on the bus, if a device is a bridge to another bus that device registers a child bus and the bus pointer in the device is set appropriately. At the moment of writing a bridge device can only have one child bus. During the boot process the device/bus tree is created either dynamically by bus drivers reading plug and play or from hard coded information.

The BSP or user must register a root bus driver in order for the driver manager to create and initialize the root device. The function `drvmgr_root_drv_register()` must be called before the driver manager initialization process starts. Buses and devices are initialized in a four step process called levels (1, 2, 3, 4). The driver manager guarantees that the bus is always initialized before to the same or higher level than devices on that bus, that the devices are initialized in the same order as they are registered in, and that child buses are initialized after all devices on the parent bus are initialized to the level. If a bus or device fails to initialize the children (devices or child bus) are never initialized further, instead they are put on a inactive list for later inspection. Dependencies between buses and devices are hence easily managed by the fact that drivers are not allowed to access certain APIs until a certain level is reached.

Drivers are registered before the driver manager initialization starts with `drvmgr_drv_register()`, the manager keeps a list of drivers which is accessed to find a suitable driver for a device. Every time a new device is registered by the bus driver the driver manager searches the driver list for a suitable driver, the bus is asked (bus.ops->unite) if the driver is compatible with the device, if so the manager unites the driver with its device and inserts the device into the initialization procedure. The driver's initialization routines will be called for all its devices at every level. If a driver was not found, the device is never initialized.

The driver manager is either initialized by the BSP during startup or manually by the user from the Init task where interrupt is enabled. The BSP initialization is enabled by passing –drvmgr to configure when building the RTEMS kernel, in that case RTEMS_DRVMGR_STARTUP is defined in [system.h]. When custom initialization is selected interrupt is enabled during the driver manager initialization and drivers initialized during RTEMS boot (system clock timer and system console UART for example) can not rely on the driver manager.

When the driver manager is initialized during boot, the `rtems_initialize_device_drivers()` function puts the manager into level 1 before RTEMS I/O drivers are initialized, so that drivers relying on the manager for device discovery are able to register devices to the I/O subsystem in time. At time of initialization most of RTEMS APIs are available for drivers, for example `malloc()` is available.

3.2.1. Bus and bus driver

A bus driver is responsible to make the driver manager aware of hardware devices, simply called devices, by scanning Plug & Play information or by any other approach. It finds, creates and registers devices in a deterministic order. The manager help bus drivers with new devices, insertion into the device tree and device numbering for example. Each device is described in a bus architecture independent way and with bus specific device information like register addresses, interrupt numbers and bus frequency information. Drivers targeting devices on the bus must know how to extract valuable information from the specific information.

All buses have a linked list of devices which are situated directly on the bus (bus.children), if a device is a bridge to another bus that device registers another device (dev.bus), a bus does maintain a list of child buses.

```
/*! Bus information. Describes a bus. */
struct drvmgr_bus {
  int           obj_type;   /*!< DRVMGR_OBJ_BUS */
  unsigned char bus_type;   /*!< Type of bus */
  unsigned char depth;      /*!< Bus level distance from root bus */
  struct drvmgr_bus   *next;      /*!< Next Bus */
  struct drvmgr_dev   *dev; /*!< Bridge device */
  void           *priv;     /*!< BUS driver Private */
  struct drvmgr_dev    *children;/*!< devices on this bus */
  struct drvmgr_bus_ops *ops;      /*!< Bus operations of bus driver */
  struct drvmgr_func *funcs; /*!< Extra operations */
  int            dev_cnt;    /*!< Number of devices this bus has */
```

```
  struct drvmgr_bus_res *reslist; /*!< Bus resources, head of a linked
                                     list of resources. */
  struct drvmgr_map_entry *maps_up; /*!< Map Translation, array of
                                    address spaces upstreams to CPU */
  struct drvmgr_map_entry *maps_down; /*!< Map Translation, array of
                                address spaces downstreams to Hardware */
  /* Bus status */
  int            level;      /*!< Initialization Level of Bus */
  int            state;      /*!< Init State of Bus, BUS_STATE_* */
  int            error;      /*!< Return code from bus.ops->initN() */
};
```

A device driver can be configured per device instance using driver resources, the resources are managed per bus as a linked list of bus resources (bus.reslist). A bus resource is an array of driver resources assigned by the bus driver. The resources are described in a separate section below.

Bus bridges often interfaces parts of an address space onto the child bus and vice versa. For example in a LEON system one linear region of the PCI memory space may be accessed through the PCI Host's PCI Window from the processor's AMBA memory space side. The `bus.maps_up` and `bus.maps_down` fields can be used to describe the bridge address regions used to access buses in upstreams or downstreams direction. The driver manager provides address translation functions that is implemented using the region descriptions.

Every bus driver implements a number of functions that provide an interface to the driver manager, device drivers or to the user. The function interface is listed below. Every bus has number of init functions similar to device drivers where the bus is responsible for finding, creating, low level initialization and registration of new devices. If a bus driver require some feature from the parent bus that is available in a certain level the bus can assume that the parent bus and all its devices has already reached a higher level or the same as the bus is requested to enter.

```
/*! Bus operations */
struct drvmgr_bus_ops {
  /* Functions used internally within driver manager */
  int          (*init[DRVMGR_LEVEL_MAX])(struct drvmgr_bus *);
  int          (*remove)(struct drvmgr_bus *);
  int          (*unite)(struct drvmgr_drv *, struct drvmgr_dev *);

  /* Functions called indirectly from drivers */
  int          (*int_register)(struct drvmgr_dev *, int index, const char *info,
drvmgr_isr isr, void *arg);
  int          (*int_unregister)(struct drvmgr_dev *, int index, drvmgr_isr isr,
void *arg);
  int          (*int_clear)(struct drvmgr_dev *, int index);
  int          (*int_mask)(struct drvmgr_dev *, int index);
  int          (*int_unmask)(struct drvmgr_dev *, int index);
  int          (*get_params)(struct drvmgr_dev *, struct drvmgr_bus_params *);
  int          (*freq_get)(struct drvmgr_dev*, int, unsigned int*);

  /*! Function called to request information about a device. The bus
   * driver interpret the bus-specific information about the device.
   */
  void         (*info_dev)(struct drvmgr_dev *, void (*print)(void *p, char *str),
void *p);
};
```

If a bus supports interrupt it can hide the actual implementation in the bus driver by implementing all or some of the int_* routines listed in the table below. Device drivers are accessing interrupts using the generic interrupt functions of the driver manager. The index determines which interrupt number the device requests, for example 0 means the first interrupt of the device, 1 the second interrupt of the device and so on, it is possible for the bus driver to determine the absolute interrupt number usually by looking at the bus specific device information. If a negative interrupt number is given it is considered to be an absolute interrupt number and should not be translated, for example an index of -3 means IRQ3 on the AMBA bus or INTC# of the PCI bus.

Table 3.3. Interrupt backend inteface of driver manager

Operation	**Description**
int_register	Register an interrupt service routine (ISR) and unmask(enable) appropriate interrupt source
int_unregister	Unregister ISR and mask interrupt source
int_clear	Manual interrupt source acknowledge at the interrupt controller
int_mask	Manual mask (disable) interrupt source at interrupt controller

Operation	Description
int_unmask	Manual unmask (enable) interrupt source at interrupt controller

3.2.1.1. Bus specific device information

A bus provide a bus dependent way to describe devices on that bus (register address for example). The information is created by the bus driver from plug & play or hardcoded information. The information may for example be used by the bus driver to unite a device with a suitable device driver and by a device driver to get information about a certain device instance.

Each bus has its own device properties, for example a PCI device have up to 6 BARs or variable size and a GRLIB AMBA AHB device has up to four different AHB areas of variable length. This kind of information is hidden by the bus driver into the bus specific area that device drivers targeting the bus type can access.

3.2.2. Root driver

The driver that is responsible for initialization of the root device and root bus. The driver manager needs to know what driver should handle the root (often CPU local) bus. The root bus driver is registered by the BSP (--drvmgr option) or by the user before the driver manager is initialized. One can say it is the starting point of finding the system's all devices.

3.2.3. Device driver

Driver for one or multiple hardware devices, simply called devices here. It uses the driver manager services provided. The driver holds information to identify supported hardware device, it tells the driver manager what kind of bus is supported and bus specific information so that the bus driver can pinpoint devices supported by driver. The bus specific information may for example be a plug & play Vendor and Device ID used to identify certain hardware.

```
/*! Information about a device driver */
struct drvmgr_drv {
  int            obj_type;   /*!< DRVMGR_OBJ_DRV */
  struct drvmgr_drv   *next;     /*!< Next Driver */
  struct drvmgr_dev   *dev;  /*!< Devices using this driver */
  uint64_t drv_id;    /*!< Unique Driver ID */
  char           *name;      /*!< Name of Driver */
  int            bus_type;   /*!< Type of Bus this driver supports */
  struct drvmgr_drv_ops *ops;    /*!< Driver operations */
  struct drvmgr_func *funcs; /*!< Extra Operations */
  unsigned int   dev_cnt;    /*!< Number of devices in dev */
  unsigned int   dev_priv_size; /*!< If non-zero DRVMGR will allocate
                           memory for dev->priv */
};
```

Every driver must be assigned a unique driver ID by the developer, the bus driver provides a macro to generate the ID. The ID is used to identify driver resources to a specific driver, only the driver knows how the resources are interpreted. The driver provides operations executed per device in drv.ops that is called by the driver manager at certain events such as device initialization and removal.

The driver manager manages a list of devices assigned to the driver order according to driver minor number. The driver minor number is assigned as the lowest free number starting at 0. A device driver can lookup a device pointer from knowing the minor number. The number of devices currently present is counted in drv.dev_cnt.

The driver manager can optionally allocate zeroed memory for the device private data structure and place a pointer in dev.priv, this is done by setting [drv.dev_priv_size]to a non-zero value.

The driver information above does not contain a bus specific device information needed to detect suitable devices. Bus drivers provide extended driver structures containing this additional bus specific information, for example the PCI bus has a pointer to an array of PCI device identifications:

```
struct pci_dev_id_match {
  uint16_t        vendor;
  uint16_t        device;
  uint16_t        subvendor;
  uint16_t        subdevice;
```

```
  uint32_t        class; /* 24 lower bits */
  uint32_t        class_mask; /* 24 lower bits */
};

struct pci_drv_info {
  struct drvmgr_drv          general;   /* General bus info */
  /* PCI specific bus information */
  struct pci_dev_id_match     *ids; /* Supported hardware */
};
```

3.2.4. Device

Represents a hardware device found by the bus driver, in this document called device. A device is found, created and registered by the bus driver, once registered the driver manager will insert it into the bus device tree, assign a bus minor number (depending on the registration order) and tries to find driver that supports the hardware. If a suitable driver is found it will unite the device with the driver. In the process of uniting the manager will assign insert the device into the driver's device list, give a driver minor number to the device (lowest free number), optionally allocate zeroed memory for driver private, queue the device for initialization.

The bus driver must have given the device a bus specific description in dev.businfo if before registering it. The driver can use the information to get register addresses, interrupt number etc.

During the first level of initialization the device driver may register a child bus, in that case the bus will be queued for initialization.

```
/*! Device information */
struct drvmgr_dev {
  int             obj_type;   /*!< DRVMGR_OBJ_DEV */
  struct drvmgr_dev    *next;      /*!< Next device */
  struct drvmgr_dev    *next_in_bus;/*!< Next device on the same bus */
  struct drvmgr_dev    *next_in_drv;/*!< Next device using the same driver*/

  struct drvmgr_drv    *drv; /*!< The driver owning this device */
  struct drvmgr_bus    *parent; /*!< Bus that this device resides on */
  short           minor_drv;/*!< Device number within driver */
  short           minor_bus;/*!< Device number on bus (for device
                      separation) */
  char            *name;      /*!< Name of Device Hardware */
  void            *priv;      /*!< Driver private device structure */
  void            *businfo;   /*!< Host bus specific information */
  struct drvmgr_bus    *bus; /*!< Bus, set only if this is a bridge */

/* Device Status */
unsigned int    state;       /*!< State of device, see DEV_STATE_* */
int             level;       /*!< Init Level */
int             error;       /*!< Error state returned by driver */
};
```

3.2.5. Driver resources

A driver resource is a read-only configuration option used by a driver for a certain device instance. The resource may be an integer with value 65 called "numberTxDescriptors". The driver resources are grouped together in arrays targeting one device instance, the arrays are grouped together into a bus resource. It is up to the bus driver to install the bus resource, some bus drivers may use a predefined bus resource or it may provide an interface for the user to provide its own configuration. Below is the

```
/* Key Data Types */
#define KEY_TYPE_NONE   0
#define KEY_TYPE_INT         1
#define KEY_TYPE_STRING 2
#define KEY_TYPE_POINTER     3

/*! Union of different values */
union drvmgr_key_value {
  unsigned int    i;    /*!< Key data type UNSIGNED INTEGER */
  char            *str; /*!< Key data type STRING */
  void            *ptr; /*!< Key data type ADDRESS/POINTER */
};

/* One key. One Value. Holding information relevant to the driver. */
struct drvmgr_key {
  char            *key_name;/* Name of key */
  int       key_type;   /* How to interpret key_value */
  union drvmgr_key_value key_value;/* The value or pointer to value */
```

```
};
```

A driver resource targets a device driver instance, not a device instance even this is in practise the same thing since there is only one driver for a device. Instead of using a bus specific device ID to identify a device instance a driver ID together with a instance minor number is used to target the driver instance. Below is a typical driver resource array with two configuration options:

```
/* GRSPW0 and GRSPW1 resources */
struct drvmgr_key grlib_grspw_0n1_res[] =
{

      {"txDesc", KEY_TYPE_INT, {(unsigned int)16}},
      {"rxDesc", KEY_TYPE_INT, {(unsigned int)32}},
      KEY_EMPTY
};
```

It is up to the driver to interpret the options, one should refer to the driver documentation for configuration options available and their format.

A bus resource in an array of device resources (driver resource arrays), the bus resource is assigned to the bus in a bus driver dependent way. In the below example the LEON3 BSP root bus is configured by simply defining a bus resource named [grlib_drv_resource], since the LEON3 root bus driver's defaults have been declared weak it can be overridden by the user project. In the example the GRSPW0 and GRSPW1 cores are configured with the same driver resources.

```
/* If RTEMS_DRVMGR_STARTUP is defined we override the "weak defaults"
 * that is defined by the LEON3 BSP.
 */
struct drvmgr_bus_res grlib_drv_resources = {
  .next = NULL,
  .resource = {
    {DRIVER_AMBAPP_GAISLER_GRSPW_ID, 0, &grlib_grspw_0n1_res[0]},
    {DRIVER_AMBAPP_GAISLER_GRSPW_ID, 1, &grlib_grspw_0n1_res[0]},
    RES_EMPTY /* Mark end of device resource array */
  }
};
```

3.2.6. Driver interface

Device drivers normally request a resource by name and type. The function `drvmgr_dev_key_get()` returns a pointer to a resource value for a specific device, see below prototype.

```
extern union drvmgr_key_value *drvmgr_dev_key_get(
    struct drvmgr_dev *dev,
    char *key_name,
    int key_type);
```

3.3. Configuration

The driver manager is configured by selecting drivers that will be registered to the manage, by registering a root bus driver prior to driver manager initialization and drivers may optionally be configured by using driver resources, see previous section.

The root bus device driver is registered by calling `drvmgr_root_drv_register()`, this must be done before the driver manager is initialized. When the BSP initializes the manager during the RTEMS boot process, nothing need to be done by user. For example calling `ambapp_grlib_root_register()` registers the GRLIB AMBA Plug & Play Bus as the root bus driver and also assigns the bus resources for the root bus.

Table 3.4. Root device driver entry points for LEON systems

System	Root driver
LEON3	`ambapp_bus_grlib.c`, register by calling `ambapp_grlib_root_register()`.
LEON2	leon2_amba_bus.c, register by calling `leon2_root_register()`.
GRLIB-LEON2	leon2_amba_bus.c, register by calling leon2_root_register(). Add LEON2_AMBA_AMBAPP_ID to the bus so that the GRLIB AMBA plug & play is found.

The drivers are selected by defining the array drvmgr_drivers, it contains one function pointer per driver that is responsible to register one or more drivers. The array is processed by `_DRV_Manager_initialization()` during startup or when calling `drvmgr_init()` from the Init task. .The drvmgr_drivers can be set up by defining CONFIGURE_INIT, selecting the appropriate drivers and including `drvmgr/drvmgr_confdefs.h`. This approach is similar to configuring a standard RTEMS project using `rtems/confdefs.h`. Below is an example how to select drivers. It is also possible to define up to ten drivers in the project by using the predefined CONFIGURE_DRIVER_CUSTOM macros.

```
#include <rtems.h>
#include <bsp.h>


#define CONFIGURE_INIT

/* Standard RTEMS set up */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_DRIVERS 32

#include <rtems/confdefs.h>

/* Driver manager set up */
#if defined(RTEMS_DRVMGR_STARTUP)/* if --drvmgr was given to configure */
 /* Add Timer and UART Driver for this example */
 #ifdef CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
  #define CONFIGURE_DRIVER_AMBAPP_GAISLER_GPTIMER
 #endif
 #ifdef CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
  #define CONFIGURE_DRIVER_AMBAPP_GAISLER_APBUART
 #endif
#endif
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_GRETH
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_GRSPW
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_GRCAN
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_OCCAN
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_B1553BRM
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_APBUART
#define CONFIGURE_DRIVER_AMBAPP_MCTRL
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_PCIF
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_GRPCI
#define CONFIGURE_DRIVER_PCI_GR_RASTA_IO
#define CONFIGURE_DRIVER_PCI_GR_RASTA_TMTC
#define CONFIGURE_DRIVER_PCI_GR_701

#include <drvmgr/drvmgr_confdefs.h>
```

3.3.1. Available LEON drivers

Below is a list of available drivers in the LEON3 BSP and the define that must be set before including [drvmgr_confdefs.h] to include the driver in the project. All drivers are preceded with CONFIGURE_DRIVER_.

Table 3.5. LEON device drivers available

Hardware	**Define to include driver**
GPTIMER	AMBAPP_GAISLER_GPTIMER
APBUART	AMBAPP_GAISLER_APBUART
GRETH	AMBAPP_GAISLER_GRETH
GRSPW	AMBAPP_GAISLER_GRSPW
GRCAN	AMBAPP_GAISLER_GRCAN
OCCAN	AMBAPP_GAISLER_OCCAN
GR1553B	AMBAPP_GAISLER_GR1553B
GR1553B RT	AMBAPP_GAISLER_GR1553BRT
GR1553B BM	AMBAPP_GAISLER_GR1553BBM
GR1553B BC	AMBAPP_GAISLER_GR1553BBC
B1553BRM	AMBAPP_GAISLER_B1553BRM

Hardware	Define to include driver
B1553RT	AMBAPP_GAISLER_B1553RT
GRTM	AMBAPP_GAISLER_GRTM
GRTC	AMBAPP_GAISLER_GRTC
PCIF PCI Host	AMBAPP_GAISLER_PCIF
GRPCI PCI Host	AMBAPP_GAISLER_GRPCI
GRPCI2 PCI Host	AMBAPP_GAISLER_GRPCI2
FTMCTRL and MCTRL	AMBAPP_MCTRL
SPICTRL	AMBAPP_GAISLER_SPICTRL
I2CMST	AMBAPP_GAISLER_I2CMST
GRGPIO	AMBAPP_GAISLER_GRGPIO
GRPWM	AMBAPP_GAISLER_GRPWM
GRADCDAC	AMBAPP_GAISLER_GRADCDAC
SPWCUC	AMBAPP_GAISLER_SPWCUC
GRCTM	AMBAPP_GAISLER_GRCTM
SPW_ROUTER	AMBAPP_GAISLER_SPW_ROUTER
AHBSTAT	AMBAPP_GAISLER_AHBSTAT
GRAES	AMBAPP_GAISLER_GRAES
GRPWRX	AMBAPP_GAISLER_GRPWRX
AT697 PCI Host	LEON2_AT697PCI
GRLIB-LEON2 AMBA PnP	LEON2_AMBAPP
GR-RASTA-ADCDAC	PCI_GR_RASTA_ADCDAC
GR-RASTA-IO PCI peripheral	PCI_GR_RASTA_IO
GR-RASTA-TMTC PCI peripheral	PCI_GR_RASTA_TMTC
GR-701 PCI peripheral	PCI_GR_701
GR-TMTC-1553 PCI peripheral	PCI_GR_TMTC_1553
GR-RASTA-SPW-ROUTER PCI peripheral	PCI_GR_RASTA_SPW_ROUTER
GR-LEON4-N2X PCI peripheral	PCI_GR_LEON4_N2X

3.4. Initialization

As described in the overview the driver manager the initialization of the driver manager is determined how the RTEMS kernel has been built. When –drvmgr has been used when configuring the kernel the manager is initialized by the BSP and RTEMS boot code, otherwise the driver manager is optional and may be initialized by the user calling drvmgr_init() after the root bus driver has been registered.

3.4.1. LEON3/4 BSP

In the RCC distribution the LEON3 BSP has been precompiled twice, once where the BSP initialized the driver manager (-qleon3, -qleon3mp) and once for custom initialization or no driver manager (-qleon3std). Please see RCC User's Manual for additional information about the gcc flags. Two different driver versions for the GPTIMER, APBUART and GRETH hardware is provided within the LEON3 BSP to support both initialization approaches.

3.5. Interrupt

The Driver manager provides a shared interrupt service. The device driver calls the driver manager which in turn rely on the bus driver to satisfy the request, that way the manager can maintain one interrupt interface regardless of bus.

For shared interrupt sources all registered interrupt handlers are called upon interrupt. The driver must itself detect if the IRQ was actually generated by its device and then decide to handle it or not.

The index of the interrupt functions determines which interrupt number the device requests, for example 0 means the first interrupt of the device, 1 the second interrupt of the device and so on, it is possible for the bus driver to determine the absolute interrupt number usually by looking at the bus specific device information. If a negative interrupt number is given it is considered to be an absolute interrupt number and should not be translated, for example an index of -3 means IRQ3 on the AMBA bus or INTC# of the PCI bus.

Table 3.6. Driver interrupt interface

Operation	Description
drvmgr_interrupt_register	Register an interrupt service routine (ISR) and unmas (enable) appropriate interrupt source
drvmgr_interrupt_unregister	Unregister ISR and mask interrupt source
drvmgr_interrupt_clear	Manual interrupt source acknowledge at the interrupt controller
drvmgr_interrupt_mask	Manual mask (disable) interrupt source at interrupt controller
drvmgr_interrupt_unmask	Manual unmask (enable) interrupt source at interrupt controller

The interrupt service route (ISR) must be of the format determined by [drvmgr_isr]. The argument is user defined per ISR and IRQ index.

```
/* Interrupt Service Routine (ISR) */
typedef void (*drvmgr_isr)(void *arg);

extern int drvmgr_interrupt_register(
      struct drvmgr_dev *dev,
      int index,
      const char *info,
      drvmgr_isr isr,
      void *arg);

extern int drvmgr_interrupt_unregister(
      struct drvmgr_dev *dev,
      int index,
      drvmgr_isr isr,
      void *arg);

extern int drvmgr_interrupt_clear(
      struct drvmgr_dev *dev,
      int index);

extern int drvmgr_interrupt_unmask(
      struct drvmgr_dev *dev,
      int index);

extern int drvmgr_interrupt_mask(
      struct drvmgr_dev *dev,
      int index);
```

3.6. Address translation

As described in the overview address regions can be translated between buses. It requires the bridge bus driver to set up address maps in at least one direction. If a bus does not support DMA for example, it might be that it is only the CPU that can access the bus but the bus can not access the CPU bus, hence the address translation will be unidirectional.

The translation software can translate addresses in up to four different ways using `drvmgr_translate()`, as listed in the table below. The function will return 0 if no map matches the translation requested, the length until the end of the matching map or 0xffffffff if no translation was needed. If a bridge has no map, the addresses are translated 1:1 (not changed) and 0xffffffff will be returned.

The `drvmgr_translate_check()` function can be called instead, it has the same functionality but verifies that the address range specified by the user is accessible. If not, the function will call `printk()` with an error message.

```
extern int drvmgr_translate(
```

```
struct drvmgr_dev *dev,
int options,
void *src_address,
void **dst_address);
```

Table 3.7. Translate options to `drvmgr_translate()`

Options argument	Translate direction	Example usage
CPUMEM_TO_DMA	Translate a CPU RAM address to an address that DMA unit can access	The CPU has a buffer in RAM, it translates the address to the PCI bus so that PCI devices can access it through the host's PCI target BAR
CPUMEM_FROM_DMA	Translate a CPU RAM address that a DMA unit can access into a an address that the CPU can access	The CPU reads out the the DMA address from a descriptor that the hardware use to access to CPU RAM, it can then translate it into the memory address the CPU can access since the memory is located at the CPU bus
DMAMEM_TO_CPU	Translate DMA unit local memory to an address that the CPU can access	PCI target BAR2 value (PCI bus address) is translated into an address which the CPU access (CPU bus address) in order to get to BAR2
DMAMEM_FROM_CPU	Translate DMA unit local memory address that the CPU can access, into an address that the DMA unit can access	

3.7. Function Interface

The driver manager provides an interface where device drivers and bus drivers can provide functions that can be looked up by knowing an associated function ID. The functions can be used to provide additional bus support over the driver manager structure, or a device driver can provide a function that the bus driver use.

For example some buses may require special access methods in order to access the hardware registers. Depending on the bus driver (bus architecture for example) is must be performed differently, the driver can request a function pointer to a WRITE_U32 function in to implement register accesses.

The `drvmgr/drvmgr.h` header file defines a number of read/write function ID numbers that drivers can use to get access routines on buses which define such operations.

4. RMAP Stack

4.1. Introduction

This section describes the RMAP stack function interface available for RTEMS. The RMAP stack provide a simple interface that can generate RMAP commands and transmit them over SpaceWire by relying on the RMAP stack driver layer. Read, read-modify-write and write with acknowledge or verification will block the caller until the transaction is completed. The features of the RMAP stack is summarized below:

- header and data CRC generation, if not generated by hardware
- logical addressing
- path addressing
- generate all read and write types defined by the RMAP specification.
- thread safe if requested
- driver layer to support multiple SpaceWire hardware
- driver for GRSPW driver
- zero-copy API

The two interfaces the RMAP stack implements can be found in the rmap header file (rmap.h), it contains definitions of all necessary data structures, bit masks, procedures and functions used when accessing the function interface.

This document describes the user interface, but not the driver interface.

4.1.1. Examples

The SpaceWire bus driver can be seen as an example, it can be found under `rtems-4.10/c/src/lib/libbsp/sparc/shared/drvmgr/spw_bus.c`.

4.2. Driver Interface

The driver interface is not described in this document.

4.3. Logical and Path addressing

The RMAP stack is by default configured to do logical addressing, however a custom callback function may be used to implement path addressing. The stack will call the function twice (one for distination path and one for return path) when the RMAP header is generated, the function is responsible to write the address path bytes directly into the header at the specified location.

4.4. Zero-copy implementation

The RMAP stack is zero-copy meaning that the data of the transfer is not copied, this improves performance. Note that when the RMAP driver does not support CRC generation the RMAP stack will write the data CRC after the input data, this means that the caller is responsible to reserve one byte of space when writing data. The RMAP stack will not write the data CRC after the data in cases where the RMAP driver that support CRC generation.

Note that even though the RMAP stack is zero-copy the RMAP driver may not be zero-copy, lowering the performance.

To get true zero-copy from user to SpaceWire transfer one can instead use the asynchonous RMAP layer part of the RMAP stack and interface that to the GRSPW Packet Driver.

4.5. RMAP GRSPW driver

A driver for the RTEMS GRSPW driver is provided with the RMAP stack, the driver automatically check if the GRSPW hardware has support for CRC generation.

The GRSPW driver is named `rmap_drv_grspw.c`.

4.6. Thread-safe

The RMAP stack can be configured to be thread safe, when entering the stack an internal semaphore will be obtained guaranteeing that multiple threads of execution can enter simultaneously. It is not needed when only one task is using the RMAP stack or if the RMAP driver itself is thread-safe.

A task may be blocked waiting for another task to complete the RMAP operation, when the RMAP stack is configured thread-safe.

4.7. User interface

The location of the RMAP stack is indicated in Table 2.1. All paths are given relative the RTEMS kernel source root.

Table 4.1. RMAP stack source location

Source description	Location
Interface implementation	`c/src/lib/libbsp/sparc/shared/spw/rmap.c`
Interface declaration	`c/src/lib/libbsp/sparc/shared/include/rmap.h`

4.7.1. Data structures

The [rmap_config] data structure is used to configure the RMAP stack, as an argument to `rmap_init()`. The data structure is defined in `rmap.h`.

```
typedef int (*rmap_route_t)(
   void *cookie,
   int dir,
   int srcadr,
   int dstadr,
   void *buf,
   int *len
   );

struct rmap_config {
   rmap_route_t      route_func;
   int               tid_msb;
   int               spw_adr;
   struct rmap_drv   *drv;
   int               max_rx_len;
   int               max_tx_len;
   int               thread_safe;
}
```

Table 4.2. rmap_config members

Member	Description
route_func	Function is a callback, called when the RMAP stack is about to generate the addressing to the target node address. It can be used to implement path addressing. Set the function pointer to NULL to make the stack use logical addressing.
tid_msb	Control the eight most significant bits in the TID field in the RMAP header. Set to -1 for normal operation, the RMAP stack will use all bits in TID for sequence counting. This option can be used when multiple RMAP stacks or other parts of the software sends RMAP commands but not using the RMAP stack. This requires, of course, a thread-safe RMAP driver.
spw_adr	The SpW Address of the SpW interface used.
drv	RMAP driver used for transmission.
max_rx_len	Maximum data length of received packets, this must match the RMAP driver's configuration.
max_tx_len	Maximum data length of transmitted packets, this must match the RMAP driver's configuration.

Member	Description
thread_safe	Set this to non-zero to enable the RMAP stack to create a semaphore used to protect the RMAP stack and the RMAP driver from multiple tasks entering the transfer function(s) of the stack at the same time.

A RMAP command is described the rmap_command structure, the type decide which parts of the union data is used when generating the RMAP header. In order to simplify for the caller three data structures avoiding the union are provided, they are named **rmap_command_write**, **rmap_command_read**, **rmap_command_rmw**. They can be used instead of **rmap_command** as argument to the function interface.

```
struct rmap_command {
      char            type;
      unsigned char         dstadr;
      unsigned char         dstkey;
      unsigned char         status;
      unsigned short        tid;
      unsigned long long    address;
      union {
            struct {
                  unsigned int length;
                  unsigned char *data;
            } write;
            struct {
                  unsigned int length;
                  unsigned int datalength;
                  unsigned int *data;
            } read;
            struct {
                  unsigned int length;
                  unsigned int data;
                  unsigned int mask;
                  unsigned int oldlength;
                  unsigned int olddata;
            } read_m_write;
      } data;
}
```

Table 4.3. rmap_command members

Member	Description
type	Type of RMAP transfer, Read/Write/Read-Modify-Write/Acked Write etc., see RMAP_CMD_*.
dstadr	Destination address of SpaceWire Node that the RMAP command should be execute upon.
dstkey	SpaceWire destination key of target node
status	Output from stack: Error/Status response. Zero if no response is successful
tid	Output from stack: TID assigned to packet header
address	40-bit address that the operation targets
data	A union of different input and output arguements depending on the type of command.

4.7.2. Function interface description

The table below sumarize all available functions in the RMAP stack.

Table 4.4. RMAP stack function prototypes

Prototype Name
void *rmap_init(struct rmap_config *config)
int rmap_ioctl(void *cookie, int command, void *arg)
int rmap_send(void *cookie, struct rmap_command *cmd)
int rmap_write(void *cookie, void *dst, void *buf , int length, int dstadr, int dstkey)
int rmap_read(void *cookie, void *src, void *buf , int length, int dstadr, int dstkey)

Prototype Name
unsigned char rmap_crc_calc(unsigned char *data , unsigned int len)

4.7.2.1. rmap_init

The RMAP stack must be initialized before other function may be called. Calling **rmap_init** initializes the RMAP stack. During the initialization the RMAP stack is configured as described by the **rmap_config** data structure, see the data structures section.

If successful, **rmap_init** will return a non-zero value later used as input argument (cookie) in other RMAP stack functions. The cookie is needed in order to support multiple RMAP stacks in parallel, the cookie identify a certain stack.

If the RMAP stack fail to initialize zero is returned.

The rmap_config structure is described in Table 2.2.

4.7.2.2. rmap_ioctl

Set run-time options such as blocking, time out, get configuration and operating the stack such as starting and stopping the communication link.

This function is not thread-safe.

If successful zero is returned.

4.7.2.3. rmap_send

Execute a command by sending the command, then wait for the response if a response is expected. This function will block until the response is received or if the timeout is expired. The timeout functionality may not be supported by the RMAP driver.

Note that when the RMAP stack is in non-blocking mode the stack will not wait for the response, however if the response is available the response is handled. If the response wasn't received -2 is returned.

Note that if the RMAP driver does not support CRC generation a byte will be written after the data provided by the user, please see zero-copy section.

If an error occurs -1 is returned. On success 0 is returned. Note that even though the RMAP request failed the RMAP stack may return zero, the RMAP status indicates the error response of the target, see the rmap_command structure in the data structures section.

4.7.2.4. rmap_crc_calc

This function is a help function used by the RMAP stack to calculate the CRC of the header and data when CRC generation is not provided by the RMAP driver.

4.7.2.5. rmap_write and rmap_read

The read and write functions are example functions that implement the most common read and write operations. The function will call **rmap_send** to execute the read and write request.

5. SpaceWire Network model

5.1. Introduction

This document describes the SpaceWire bus driver used to write device drivers for a SpaceWire Node accessed over SpaceWire with RMAP.

5.2. Overview

In order to provide a standardized way of writing drivers for Nodes on a SpaceWire network and to improve code reuse a Bus driver for a SpaceWire network as been written. The bus driver is written using the concepts of the Driver Manager.

The SpaceWire Bus driver provides services to the nodes in the network, some of the services are listed below:

- Read/Write access to target (using the RMAP protocol)
- Interrupt handling
- Per node resources

The hardware topology is organized by the driver manager's bus and device trees, the SpaceWire bus driver is attached to the SpaceWire core providing the actual SpaceWire interface in order to maintain the hardware topology. It is important that the on-chip devices and drivers are loaded and initialized before the SpaceWire network as the SpaceWire network depends on the on-chip devices. The bus driver initialization is controlled and started by the user after the driver manager has initialized the on-chip bus.

The SpaceWire driver requires the SpaceWire RMAP stack to perform read and write access to the SpaceWire Target Nodes.

The driver support Logical SpaceWire Addressing only at this point.

5.3. Requirements

The SpaceWire network must be Logical addressed and the SpaceWire bus driver requires the RMAP stack for target node access.

5.4. Node Description

The SpaceWire bus driver is a driver for the devices on the SpaceWire bus, in this particular case a device is called a SpaceWire Node, a node is described by the data structure spw_node. Each node has a Node ID, a name, and a list of optional keys. A SpaceWire node has the following configurable elements:

- Node ID (connected to driver)
- Node Name
- SpaceWire Destination key
- SpaceWire Node Address
- IRQ setup (up to four IRQs)

5.4.1. The Node ID

The Node ID identifies a type of target, not a certain Node. The Node ID in combination with the node index on the bus creates a Unique identifier. The Node ID is used to identify a driver that can handle the node. The node index is taken from the index in the Node table.

The NodeIDs are defined in `spw_bus_ids.h`.

5.5. Read and write operation

A SpaceWire target Node's memory and registers are accessed using RMAP commands. The RMAP protocol is implemented by the RMAP stack in a separate module.

The driver manager provide read and write operations to registers and memory for drivers, the SpaceWire Bus driver implements them for the SpaceWire bus. A node driver calls the standard read and write operations which

are translated into a SpaceWire bus read/write which is implemented using the RMAP stack. All operations are blocking until data is available, the return value indicates it the transfer was successful or not.

5.6. Interrupt handling

The RMAP protocol does not support interrupt handling, this is instead implemented by an separate interrupt line, the interrupt handling is an optional feature per SpaceWire node. Each SpaceWire node may have up to four interrupts connected to interrupt capable GPIO pins.

The user must setup a Virtual Interrupt Table, the table entries provide a way for the bus driver to translate a Virtual IRQ number to a GPIO pin. The GPIO pin is used to connect to the IRQ and receive the interrupt. In the node description a node may for example define it's IRQ1 to be connected to the SpaceWire bus Virtual IRQ 2, which in turn is connected to GPIO5.

Setting up and controlling interrupts for node drivers are similar to a on-chip device driver, however the interrupt service routine must take more things in to account. The ISR is expected to read and write to the node's registers over the SpaceWire bus, that would require that SpaceWire bus is not busy and that the SpaceWire request is executed very fast, non of these assumptions can be made. The ISR can thus not execute in interrupt context, instead a high priority ISR task is managed by the SpaceWire bus driver. This way the ISR can access the node over SpaceWire, however extra care must be taken in the node driver to avoid conflicts and races when the ISR is executing as a task, instead of locking interrupt as in tradition drivers one may use a semaphore to protect the critical regions.

5.7. Using the spacewire bus driver

The SpaceWire bus is registered to the driver manager for each SpaceWire network by calling the `spw_bus_register()` function with a configuration description. The configuration describe the nodes on the network, IRQ setup, driver resources

The SpaceWire bus is attached to a on-chip GRSPW driver, the core that provides access to the SpaceWire bus via the RMAP stack.

There is an example of how to configure and use the SpaceWire bus driver in **config_spw_bus.c**.

SpaceWire Node drivers must set the bus type to DRVMGR_BUS_TYPE_SPW_RMAP and define an array with all devices nodes that are supported by the driver. The AMBA PnP RMAP may be considered as an example node driver.

6. AMBA over SpaceWire

6.1. Introduction

This document describes the AMBA Plug&Play bus driver used to write device drivers for AMBA cores accessed over SpaceWire. The driver rely on the SpaceWire network bus driver.

6.2. Overview

The AMBA Plug&Play bus driver for the SpaceWire network is a generic driver for all GRLIB systems by using the Plug&Play functionality provided by GRLIB systems. The address of the Plug&Play area start address is configurable. The driver is a driver for a SpaceWire Node on a SpaceWire Network.

The system is accessed using RMAP commands and interrupt handling is performed when the IRQMP core is found.

The services provided to device drivers on the AMBA bus accessed over SpaceWire are listed below:

- AMBA Plug&Play scanning over SpaceWire
- Interrupt management (driver for IRQMP)
- Read and Write registers and memory over SpaceWire
- Memory allocating (`ambapp_rmap_partition_memalign()`)
- Driver resources

6.3. Requirements

The SpaceWire bus driver is required.

6.4. Interrupt handling

See the interrupt service routine of the AMBA Plug&Play bus is executed on the SpaceWire bus driver's ISR task. See the SpaceWire Bus driver's documentation about the constraints of the interrupt handling.

6.5. Memory allocation on target

Two functions are provided by the AMBA RMAP driver to simplify memory allocation of target memory, `ambapp_rmap_partition_create()` and `ambapp_rmap_partition_memalign()`.

A partition symbolize a memory area with certain properties. For example, partition 0 might be SRAM and partition 1 might be on-chip RAM. A memory controller driver typically registers a partition after it has initialized the memory controller and perhaps washed the memory, other drivers may then request memory from a certain partition. The partition number that a driver request memory from may be configured from driver resources making it possible for the user to easily control which parts of the memory is used. For example a descriptor table may be required to be located in on-chip RAM.

Drivers request memory with memory alignment requirements by calling `ambapp_rmap_partition_memalign()`. The device structure is passed along when creating partitions and when allocating memory, making it possible for the AMBA RMAP bus driver to allocate memory from the same bus.

6.6. Differences between on-chip AMBA drivers

There some differences when writing drivers for a remote target accessed over SpaceWire using the AMBA RMAP driver, this section identifies the most common differences.

- Read and Write access (memory and registers) must be through functions rather than direct, functions are provided
- Error handling of failed read/write accesses, this may also be handled on a global level (by the SpaceWire bus driver)
- Memory allocation of target memory

- ISR may block (executed in task context)
- Lock out ISR method is different
- Drivers must set bus type to DRVMGR_BUS_TYPE_AMBAPP_RMAP

7. LEON PCI host briedge drivers

7.1. Introduction

This section describes PCI Host support in RTEMS for SPARC/LEON processors. The supported PCI Host hardware are listed below

- GRLIB GRPCI2
- GRLIB GRPCI
- GRLIB PCIF
- AT697 PCI

The PCI drivers require the Driver Manager and PCI Library available in the RCC distribution. The PCI Library documentation is available in the doc/user directory in the RCC kernel source distribution. Note that the PCI Library is not available in the official RTEMS distribution.

7.1.1. Examples

There is a simple example available that initializes the PCI Bus, lists the PCI configuration and demonstrates how to write a PCI device driver. The example is part of the RCC distribution, it can be found under `/opt/rtems-4.10/src/samples/rtems-pci.c`. The `rtems-shell.c` sample found in the same directory also demonstrates PCI with RTEMS, note that there is a **pci** command which can be used to get information about the PCI set up.

7.2. Sources

The drivers can be found in the RTEMS SPARC BSP shared directory and in the LEON2 BSP. See table below.

Table 7.1. PCI driver source location

Location	Description
`.../libbsp/sparc/leon2/pci/at697_pci.c`	AT967 PCI
`.../libbsp/sparc/shared/pci/grpci2.c`	GRPCI2
`.../libbsp/sparc/shared/pci/grpci.c`	GRPCI
`.../libbsp/sparc/shared/pci/pcif.c`	GRLIB PCIF, ACTEL PCI AMBA wrapper
`cpukit/libpci`	PCI Library
`cpukit/libpci/pci_bus.*`	PCI Bus driver for driver manager
`doc/user/libpci.t`	PCI Library documentation

7.3. Configuration

The PCI interrupt assignment can be configured to override the Plug & Play information. The PCI driver is configured using any combination of the driver resources in the table below, see samples or driver manager documentation how driver resources are assigned.

Table 7.2. PCI Host driver parameter description

Name	Type	Parameter Description
INTA#	INT	Select system IRQ for PCI interrupt pin INTA#
INTB#	INT	Select system IRQ for PCI interrupt pin INTB#
INTC#	INT	Select system IRQ for PCI interrupt pin INTC#
INTD#	INT	Select system IRQ for PCI interrupt pin INTD#

7.3.1. GRPCI

GRLIB designs using the GRPCI PCI Host bridge has in addition to the INTX# configuration options the below options.

Name	Type	Parameter Description
tgtbar1	INT	PCI target Base Address Register (BAR) 0 (defaults is 0x40000000)
byteTwisting	INT	Enable (1) or Disable (0=default) PCI bytes twisting

7.3.2. GRPCI2

GRLIB designs using the GRPCI2 PCI Host bridge has in addition to the INTX# configuration options the below options.

The GRPCI2 host has up to 6 BARs, each with a configurable size. The driver uses only the first BAR by default, it is set to start of RTEMS RAM memory and 256MBytes. The tgtBarCfg option is an address to an array of 6 `struct grpci2_pcibar_cfg` descriptions, each describing one BAR's size and PCI address and AMBA address the PCI access is translated into. Thus, the programmer has full flexibility of where DMA capable PCI targets should access. A size of 0 disables the BAR, see `grpci2.h` for the structure definition.

Name	Type	Parameter Description
tgtBarCfg	INT	PCI target Base Address Registers (BAR) configuration
byteTwisting	INT	Enable (1) or Disable (0=default) PCI bytes twisting

7.3.3. AT697

The AT697 PCI Host driver has additional configuration parameters to set up interrupts which is routed through GPIO pins. The GPIO registers will be configured, and when a PCI target driver enables/disables IRQ the system IRQ will be unmasked/masked.

Name	Type	Parameter Description
INTA#_PIO	INT	Select PIO pin connected to PCI interrupt pin INTA#
INTB#_PIO	INT	Select PIO pin connected to PCI interrupt pin INTB#
INTC#_PIO	INT	Select PIO pin connected to PCI interrupt pin INTC#
INTD#_PIO	INT	Select PIO pin connected to PCI interrupt pin INTD#

The two AT697 PCI target BARs are configurable from driver resources as below. A PCI target BAR determines at which PCI address the AT697 AMBA space is accessed on, the AT697 has two 16Mbytes base address registers. The default value is set to 0x40000000 (base of SRAM) and 0x60000000 (base of SDRAM).

Name	Type	Parameter Description
tgtbar2	INT	PCI target Base Address Register (BAR) 0
tgtbar2	INT	PCI target Base Address Register (BAR) 1

7.4. User interface

The PCI drivers are not accessed directly instead the user calls the PCI Library that translates into a call to the active PCI host driver. When the drivers are initialized they register a backend to the PCI library, all PCI devices are initialized using the PCI configuration library, then a PCI Bus is registered which is implemented on top of the PCI Library. That way the PCI Bus is independent of PCI host driver. The driver manager will find all PCI devices and assign a suitable driver for them, and so on.

Please see the PCI Library documentation.

7.4.1. PCI address space

The PCI Library supports the following PCI address spaces:

- 16-bit I/O Space (IO)
- non-prefetchable memory space (MEMIO)
- prefetchable memory space (MEM)

- configuration space (CFG)

On LEON hardware the address spaces are accessed over dedicated AHB areas as ordinary AMBA memory accesses and it will be transformed into appropriate PCI access type depending on which AHB area (window) was accessed and of which AMBA access type (burst, single access). Note that LEON hardware have only one memory window which can do both MEM and MEMIO access types, so the PCI Library is configured with one MEMIO Window. No special instructions are required to access I/O or configuration space. The location of the PCI Windows are determined by looking at AMBA plug and play information for the PCI Host core. The AT697 PCI MEM Window is defined to 0xA0000000-0xF0000000.

The PCI Library is informed about the PCI windows location and size. PCI BARs are allocated within the MEM, MEMIO and I/O windows.

7.4.2. PCI interrupt

For every PCI target board found by the PCI Library the PCI driver is asked to provide a system IRQ for the target's PCI Interrupt pin number. The interrupt is normally taken from AMBA Plug & Play interrupt number assigned to the PCI Host hardware itself. However it can be overridden using driver resources as described in section Section 7.4.

After the PCI Library has allocated memory for all targets BARs and assigned IRQ. The PCI bus driver can access the IRQ number from configuration space and connect a PCI Target driver with its system interrupt source. The PCI target drivers use the Driver Manager interrupt register routine.

When a PCI target driver enable interrupt using the Driver Manager interrupt enable routine, the system IRQ for the PCI target is unmasked. AT697 PCI interrupt is not routed through the PCI core but through user selectable GPIO. Enabling IRQ will only cause the system IRQ to be unmasked, the PCI driver will not change GPIO parameters, this is required by the user to set up. PCI is level triggered.

PCI interrupts must be acknowledge after being handled to ensure that the interrupt handler is not executed twice. The Driver Manager interrupt clear routine can be used to clear the pending bit in the LEON interrupt controlled after the interrupt has been handled by the PCI target Driver.

When the LEON takes the PCI IRQ the LEON IRQ controller is acknowledged, however the PCI target is still driving the IRQ line causing the LEON IRQ controller being set once again. This is because PCI is level triggered (level is still low), the other IRQs on the LEON is edge triggered. The solution is to acknowledge the LEON IRQ controller after the PCI target has stop driving the PCI IRQ line, only then will the driver be able to stop the last already handled IRQ to occur. This must be done in the PCI ISR of the target device driver after the hardware causing the IRQ has been acknowledge.

7.4.3. PCI endianess

The PCI bus is defined little-endian whereas the SPARC and AMBA bus are defined big-endian, this imposes a problem where the CPU has to byte-swap the data in PCI accesses. The GRPCI and GRPCI2 host controllers has support for doing byte-swapping in hardware for us,, it is enabled/disabled using the byteTwisting configuration option. The AT697 PCI and PCIF does not have this option, the software defaults to the PCI bus being non-standard big-endian instead. Please see more information about this in hardware manuals and the PCI Library documentation.

8. GR-RASTA-ADCDAC PCI peripheral

This section describes the GR-RASTA-ADCDAC PCI peripheral driver.

The GR-RASTA-ADCDAC driver require the Driver Manager and that the PCI bus is big endian.

The GR-RASTA-ADCDAC driver is a bus driver providing an AMBA Plug & Play bus. The driver first sets up the target PCI register such as PCI Master enable and the address translation registers. Once the PCI target is set up the driver creates an `ambapp_bus` that scans the bus and assigns the appropriate drivers. This driver provides interrupt handling and memory address translation on the internal AMBA bus so that the drivers can function as expected.

The driver resources of the AMBA bus created by the GR-RASTA-ADCDAC driver can be assigned by calling gr_rasta_adcdac_set_resource as defined by gr_rasta_adcdac.h.

The driver resources of the AMBA bus created by the GR-RASTA-ADCDAC driver can be assigned by overriding the weak default bus resource array `gr_rasta_adcdac_resources[]` of the driver. It contains a array of pointers to bus resources where index=N determines the bus resources for GR-RASTA-ADCDAC[N] board. The array is declared in gr_rasta_adcdac.h. The driver resources can be used to set up the memory parameters, configure locations of the DMA areas and other parameters of GRCAN, GRADCDAC and all other supported cores. Please see respective driver for available configuration options.

9. GR-RASTA-IO PCI peripheral

This section describes the GR-RASTA-IO PCI peripheral driver.

The GR-RASTA-IO driver require the Driver Manager and that the PCI bus is big endian.

The GR-RASTA-IO driver is a bus driver providing an AMBA Plug & Play bus. The driver first sets up the target PCI register such as PCI Master enable and the address translation registers. Once the PCI target is set up the driver creates an **ambapp_bus** that scans the bus and assigns the appropriate drivers. This driver provides interrupt handling and memory address translation on the internal AMBA bus so that the drivers can function as expected.

The driver resources of the AMBA bus created by the GR-RASTA-IO driver can be assigned by overriding the weak default bus resource array `gr_rasta_io_resources[]` of the driver. It contains a array of pointers to bus resources where index=N determines the bus resources for GR-RASTA-IO[N] board. The array is declared **in gr_rasta_io.h**. The driver resources can be used to set up the memory parameters and configure locations of the DMA areas of 1553BRM, GRCAN, GRSPW cores. Please see respective driver for available configuration options.

10. GR-RASTA-TMTC PCI peripheral

This section describes the GR-RASTA-TMTC PCI peripheral driver.

The GR-RASTA-TMTC driver require the Driver Manager and that the PCI bus is big endian.

The GR-RASTA-TMTC driver is a bus driver providing an AMBA Plug & Play bus. The driver first sets up the target PCI register such as PCI Master enable and the address translation registers. Once the PCI target is set up the driver creates an **ambapp_bus** that scans the bus and assigns the appropriate drivers. This driver provides interrupt handling and memory address translation on the internal AMBA bus so that the drivers can function as expected.

The driver resources of the AMBA bus created by the GR-RASTA-TMTC driver can be assigned by overriding the weak default bus resource array `gr_rasta_tmtc_resources[]` of the driver. It contains a array of pointers to bus resources where index=N determines the bus resources for GR-RASTA-TMTC[N] board. The array is declared in **gr_rasta_tmtc.h**. The driver resources can be used to set up the memory parameters and configure locations of the DMA areas of GRTC, GRTM, GRSPW cores. Please see respective driver for available configuration options.

11. GR-RASTA-SPW_ROUTER PCI Peripheral

This section describes the GR-RASTA-SPW_ROUTER PCI peripheral driver.

The GR-RASTA-SPW_ROUTER driver require the RTEMS Driver Manager and that the PCI bus is big endian.

The GR-RASTA-SPW-ROUTER driver is a bus driver providing an AMBA Plug & Play bus. The driver first sets up the target PCI register such as PCI Master enable and the address translation registers. Once the PCI target is set up the driver creates an ambapp_bus that scans the bus and assigns the appropriate drivers. This driver provides interrupt handling and memory address translation on the internal AMBA bus so that the drivers can function as expected.

The driver resources of the AMBA bus created by the GR-RASTA-SPW_ROUTER driver can be assigned by overriding the weak default bus resource array gr_rasta_spw_router_resources[] of the driver. It contains a array of pointers to bus resources where index=N determines the bus resources for GR-RASTA-SPW_ROUTER[N] board. The driver resources can be used to set up the memory parameters and configure locations of the DMA areas of GRSPW2 AMBA port cores. Please see GRSPW driver documentation for available configuration options.

12. GR-CPCI-LEON4-N2X PCI Peripheral

This section describes the GR-CPCI-LEON4-N2X PCI peripheral driver.

The GR-CPCI-LEON4-N2X driver require the RTEMS Driver Manager and that the PCI bus is big endian.

The GR-CPCI-LEON4-N2X driver is a bus driver providing an AMBA Plug & Play bus. The driver first sets up the target PCI register such as PCI Master enable and the address translation registers. The clock gating unit is by default set up so that all functionality is enabled. Once the PCI target is set up the driver creates an **ambapp_bus** that scans the bus and assigns the appropriate drivers. This driver provides interrupt handling and memory address translation on the internal AMBA bus so that the drivers can function as expected.

The driver resources of the AMBA bus created by the driver can be assigned by overriding the weak default bus resource array `gr_cpci_leon4_n2x_resources[]` of the driver. It contains a array of pointers to bus resources where index=N determines the bus resources for GR-CPCI-LEON4-N2X[N] board. The array is declared in `gr_cpci_leon4_n2x.h`. The driver resources can be used to set up the memory parameters and for configuring other driver options such as the base DMA area address of the SpaceWire cores. Please see respective driver for available configuration options.

12.1. Driver registration

The driver must be registered to the driver manager by adding the CONFIGURE_DRIVER_PCI_GR_LEON4_N2X define in the RTEMS project configuration. This process is described in the driver manager chapter.

12.2. Driver resource configuration

The driver can be configured using driver resources as described in the driver manager chapter. Below is a description of configurable driver parameters. The driver parameters is unique per PCI device and configured in the PCI bus driver resources array. The parameters are all optional, the parameters only overrides the default values. However the ambaFreq paramter is typically required.

Table 12.1. GR-CPCI-LEON4-N2X driver parameter description

Name	Type	Parameter Description
ahbmst2pci	INT	PCI base address of the 1Gbyte AMBA->PCI window. Default to RAM start address.
ambaFreq	INT	Frequency in Hz of the LEON4-N2X AMBA bus. Defaults to 200MHz.
cgEnMask	INT	Clock gating enable/disable mask. Each bit in the mask corresponds to one bit the the clock gate unit (one clock tree), set to 1 to enable or 0 to disable individual clock trees.
bar0	INT	PCI target BAR0 AMBA access address. Defaults to 0x00000000 (L2-cache main memory)
bar1	INT	PCI target BAR1 AMBA access address. Defaults to 0xf0000000 (L2-cache registers)

13. GRSPW Packet driver

13.1. Introduction

This section describes the GRSPW packet driver for RTEMS The packet driver will replace the older GRSPW driver in the future.

It is an advantage to understand the SpaceWire bus/protocols, GRSPW hardware and software driver design when developing using the user interface in Section 13.3 and Section 13.4. The Section 13.2.1 describes the overall software design of the driver.

13.1.1. GRSPW packet driver vs. old GRSPW driver

This driver is a complete redesign of the older GRSPW driver. The user interfaces to GRSPW devices using an API rather than using the standard UNIX file procedures like open(), read(), ioctl() and so on. The driver uses linked lists of packet buffers to receive and transmit SpaceWire packets. Before the user called read() or write() to copy data into/from the GRSPW DMA buffers, where each call received or transmitted a single packet at a time. The packet driver implements a new API that allows efficient custom data buffer handling providing zero-copy ability, SMP support and multiple DMA channel support. The link control handling has been separated from the DMA handling, just to name a few improvements.

13.1.2. Hardware Support

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

- GRSPW, GRSPW2 and GRSPW2_DMA (router AMBA port)
- Multiple DMA channels
- Time Code
- Link Control
- Port Control
- RMAP Control
- SpaceWire Interrupt codes
- Interrupt handling
- Multi-processor SMP support (not supported by RTEMS-4.10)

13.1.3. Driver sources

The driver sources and definitions are listed in the table below, the path is given relative to the RTEMS source tree `rtems-4.10/c/src/lib/libbsp/sparc`.

Table 13.1. Source Location

Filename	Description
`shared/include/grspw_pkt.h`	GRSPW user interface definition
`shared/spw/grspw_pkt.c`	GRSPW driver implementation

13.1.4. Show routines

There are currently no show routines.

13.1.5. Examples

Examples are available in the `src/samples/spw/` directory in the RCC distribution.

13.1.6. Known driver limitations

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

- SMP is not supported yet.
- The shutdown of the work thread when destroying the message queue may be problematic.

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

13.2. Software design overview

13.2.1. Overview

The driver has been implemented using the Driver Manager Framework. The driver provides a kernel function interface, an API, rather than implementing a IO system device. The API is designed for multi-threadding allowing multiple threads to operate on the link, RX and TX DMA channel interfaces independently. The driver API has been split up in two major parts listed below:

- Device interface, see Section 13.3.
- DMA channel interface, see Section 13.4.

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.

13.2.2. Driver resource configuration

It is possible to configure the GRSPW driver by driver resources assigned at compile time. The resources are set individually per GRSPW device. The table below shows all options.

Table 13.2. GRSPW packet driver resources

Name	Type	Parameter description
`nDma`	INT	Number of DMA channels to present to user. This is used to limit the number of DMA channels and thereby save memory. This option does not have an effect if it is less than one or greater than the number of DMA channels present in the hardware.
`bdDmaArea`	INT	Custom RX and TX DMA descriptor table area address. The driver always requires 0x800 Bytes memory per DMA channel. This means that at least (*`nDMA`* * 0x400 * 2) Bytes must be available. The address must be aligned to 0x400.

13.2.3. 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 and potentially overridden by *nDMA* configuration option, see Section 13.2.2.
- Device index determined.

13.2.4. Link control

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

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

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

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

To activate the GRSWP interrupt routine when any link error occurs, the bitmask option *Enable Error Link IRQ* (*LINKOPTS_EIRQ*) shall be set. The bitmask options described as *Disable Link on XX Error* (*LINKOPTS_DIS_ON_**) are used to select which events shall actually cause link disable in the interrupt routine and inform the work-task of a shutdown stop.

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

Statistics about the link errors can be read from the driver, see Section 13.3.8.

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

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

Function names prefix: `grspw_link_*()`.

13.2.5. Time Code support

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

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

Function names prefix: `grspw_tc_*()`

13.2.6. RMAP support

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

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

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

The SpaceWire node address, which also affects the RMAP target, is controlled from the address configuration routines, see Section 13.2.8.

Function names prefix: `grspw_rmap_*()`

13.2.7. 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_*()`

13.2.8. 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_*()`

13.2.9. SpaceWire Interrupt Code support

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

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

Function names prefix: `grspw_ic_*()`

13.2.10. 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 , there is a helper library distributed together with the examples that do buffer allocation and handling . 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_*()`

13.2.10.1. Buffer List help routines

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

Function names prefix: `grspw_list_*()`

13.2.11. Driver DMA buffer handling

The driver represents packets with the struct grspw_pkt packet structure, see Table 13.32. 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.

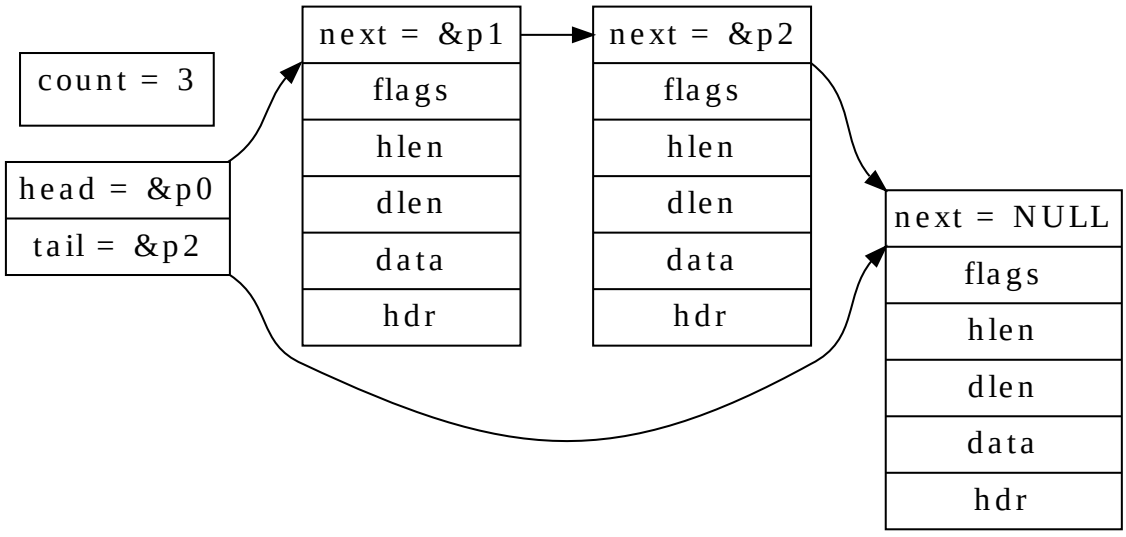


Figure 13.1. Queue example - linked list of three grspw_pkt packets

13.2.11.1. DMA Queues

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

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

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

The current number of packets in respective queue can be read by doing function calls using the DMA API, see Section 13.4.7. The user can for example use this to determine to wait or continue with packet processing.

13.2.11.2. DMA Queue operations

The user can control how the RX READY and TX SEND queue is populated, by providing packet buffers. The user can control how and when packets are moved from RX READY and TX SEND queues into the RX SCHED or TX SCHED by enabling the work-task and interrupt driven DMA or by manually trigger the moving calling reception and transmission routines as described in Section 13.4.6 and Section 13.4.5.

The packets always flow in one direction from RX READY -> RX SCHED -> RX RECV. Likewise the TX packets flow TX SEND -> TX SCHED -> TX SENT. The procedures triggering queue packet moves are listed below and in Figure 13.2 and Figure 13.3. The interface of theses procedures are described in the DMA channel API.

- USER -> RX READY queue - rx_prepare, Section 13.4.6.
- RX RECV -> USER - rx_recv, Section 13.4.6.
- USER -> TX SEND - tx_send, Section 13.4.5.
- TX SEND -> USER - tx_reclaim, Section 13.4.5.

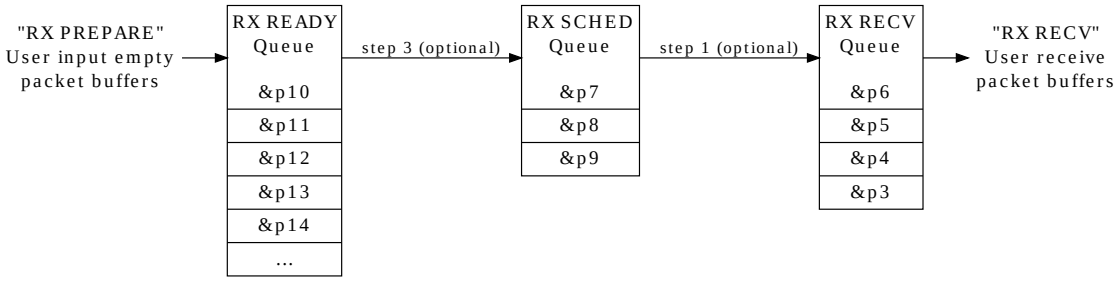


Figure 13.2. RX queue packet flow and operations

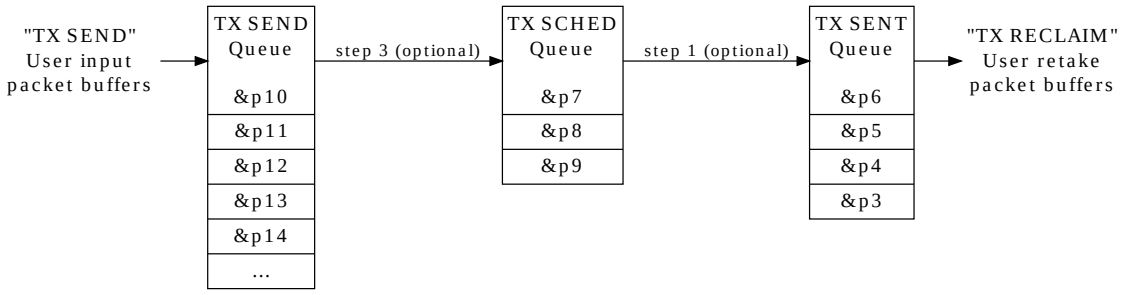


Figure 13.3. TX queue packet flow and operations

13.2.12. Polling and blocking mode

Both polling and blocking transfer modes are supported. Blocking mode is implemented using DMA interrupt and a work-task for processing the descriptor tables to avoid loading the CPU in interrupt context. One common work-task handles all GRSPW devices DMA handling triggered by DMA interrupt. In polling mode the user is responsible for processing the DMA descriptor tables at a user defined interval by calling reception and transmit routines of the driver.

DMA interrupt is generated every N received/transmitted packets or controlled individually per packet. The latter is configured in the packet data structures and the former using the DMA channel configuration. See Section 13.4.3 and Section 13.4.9 for more information.

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

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

13.2.13. Interrupt and work-task

The driver can optionally spawn one work-task that is used to service all GRSPW devices. The work-task execution is resumed/triggered from the GRSPW ISR at certain user configurable events, at link errors or DMA transmissions completed. The ISR sends messages to the work-task using the RTEMS Message API. When the work-task has been scheduled work for a specific device or DMA channel the ISR has turned off the specific interrupt that the work-task handles, once the work has been completed the work-task re-enables interrupt again for the specific event. This is to lower the number of interrupts.

When the work-task is used to process DMA descriptor tables the priority of the work-task must be considered. The priority must be selected so that the work-task is allowed to execute in time. Normally a high priority should be selected to lower the latency and for higher DMA throughput. When using the RX/TX DMA Wait interface the waiting tasks will be woken first after the work-task has processed the DMA descriptor table. If the work-task never gets CPU resources due to other higher-priority tasks always ready it may appear to dead-lock. To avoid dead-lock or wait timeouts the priority must be set with other task priorities in mind. When the priority is set to -1 the work-task is never created. The functionality of the ISR sending messages to the work-task, the work-task DMA and link error handling and RX/TX DMA wait interface are not available to the user.

The priority is controlled by the user at compile time by defining the variable `grspw_work_task_priority`. Its default value is 100 declared by the driver using the a weak variable:

```
int grspw_work_task_priority __attribute__((weak)) = 100;
```

NOTE: The priority of the work-task is user configurable and must be assigned with the rest of the application's task priorities in mind so that it is allowed to execute and wake up the waiting/blocked user task(s). Otherwise the user task may be blocked forever or for too long.

The work-task can also be used to automatically stop DMA operation on certain link errors. This feature is enabled by activating the different *Disable Link on XX Error* (`LINKOPTS_DIS_ON_*`) options from the device API link control interface. See Section 13.2.4. For the configured link errors the GRSPW interrupt handler will trigger the shutdown work to start which will stop all DMA channels by calling `grspw_dma_stop()`.

13.2.14. Starting and stopping DMA

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

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

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

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

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

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

13.2.15. Thread concurrency

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

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

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

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

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

Table 13.3. DMA channel semaphore operations.

Function	Operation	Semaphore	Description
`dma_open`	Init semaphores	RX TX	RX and TX semaphores are initialized to 1.
`dma_close`	Free semaphores	RX TX	Both RX and TX semaphores are taken and left in locked state or deleted on a successful close. From this point the user can not enter other DMA functions than `dma_open`.
`dma_start`	Init semaphores	RX-WAIT TX-WAIT	The wait semaphores are initialized to 0 (locked) state. From this point onwards the RX/TX wait interface is available.
`dma_stop`	Shutdown DMA	RX TX RX-WAIT TX-WAIT	The RX and TX semaphores are taken and returned in sequence during stopping a DMA channel. The RX-WAIT and TX-WAIT semahpores are signalled in order for potential locked tasks to be worken up and return to caller with an error code or indicating DMA stopped (1) error code.
`dma_rx_recv` `dma_rx_prepare` `dma_rx_count`	RX DMA operations	RX	Holds the RX semahpore while performing RX operations.
`dma_tx_send` `dma_tx_reclaim` `dma_tx_count`	RX DMA operations	RX	Holds the RX semaphore while performing TX operations.
`dma_tx_wait`	Wait for TX DMA.	TX TX-WAIT	Takes the TX semaphore to initialize the wait structures. TX-WAIT is taken to block the calling thread until the worker, DMA shutdown or timeout awakens the thread again.
`dma_rx_wait`	Wait for RX DMA.	RX RX-WAIT	Takes the RX semaphore to initialize the wait structures. RX-WAIT is taken to block the calling thread until the worker, DMA shutdown or timeout awakens the thread again.
DMA work	Normal DMA descriptor list processing.	RX TX RX-WAIT TX-WAIT	RX and TX locks taken in sequence. RX-WAIT and TX-WAIT given on matching conditions.
DMA work error	DMA AHB error handling.	RX TX	DMA RX/TX AHB errors leads to calling `grspw_dma_stop()` for one DMA channel. The work-task does not hold any locks itself.
Link work error	Link error handling.	RX TX	SpaceWire link errors configured to generate interrupt may be handled by worker to call `grspw_dma_stop()` for all DMA channels.

13.2.16. SMP Support

The driver has been designed with SMP in mind, but not has not been implemented yet. However setting up scheuler and interrupt affinity will make it possible to use the driver in SMP mode under these limitations. Data structures, interrupt handling routine and GRSPW control register accesses should be protected by spin-locks when SMP is enabled in the future.

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

As described in Section 13.1.6 the SMP support is not available. RTEMS-4.10 does not support SMP.

13.3. Device Interface

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

13.3.1. 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. See driver resource configuration options described in Section 13.2.2. The descriptor table length is always the maximum 0x400 Bytes for RX and TX.
- Internal resources like spin-locks and data structures are initialized.
- The GRSPW device Interrupt Service Routine (ISR) is installed and enabled. However hardware does not generate interrupt until the user configures the device or DMA channel to generate interrupts.
- The driver is configured to clear all link status events from the ISR.
- The device is marked opened to protect the caller from other users of the same device.

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

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

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

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

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

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

Table 13.4. `grspw_dev_count` function declaration

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

Table 13.5. `grspw_open` function declaration

Proto	`void *grspw_open(int dev_no)`
About	Opens a GRSPW device. The GRSPW device is identified by index. The returned value is used as input argument to all functions operating on the device.

Param	*dev_no* [IN] Integer Device identification number. Devices are indexed by the registration order to the driver, normally dictated by the Plug & Play order. Must be equal or greater than zero, and smaller than that returned by `grspw_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device semaphore fails or device already is open.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which GRSPW device.
Notes	May blocking until other GRSPW device operations complete.	

Table 13.6. `grspw_close` function declaration

Proto	`int grspw_close(void *d)`	
About	Closes a previously opened device. All DMA channels must have been stopped and closed by the user prior to calling this function. See the documentation for `grspw_dma_stop` and `grspw_dma_close`.	
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.	
Return	Value. Description	
	0	Device was successfully closed, or already previously closed.
	1	Failure due to a DMA channel is open for this device.
	-1	Failure due to invalid input arguments or unknown semaphore error.

13.3.2. Hardware capabilities

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

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

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

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

Table 13.7. `grspw_hw_support` function declaration

Proto	`void grspw_hw_support(void *d, struct grspw_hw_sup *hw)`
About	Read hardware capabilities of GRSPW device and write them in an easy to use format described by the grspw_hw_sup data structure. The data structure is described by Table 13.8.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*hw* [OUT] pointer Address to where the driver will write the hardware capabilities. Pointer must point to memory and be valid.

Return	None. Always success, input is not range checked.

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

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

Table 13.8. grspw_hw_sup data structure declaration

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

13.3.3. Link Control

The SpaceWire link is controlled and configured using the device API functions described below. The link control functionality is described in Section 13.2.4.

Table 13.9. `grspw_link_ctrl` function declaration

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

	START	Read/Set start link option.
	AUTOSTART	Read/Set enable/disable link auto-start option.
	DIS_ONERR	Read/Set *disable DMA transmitters when a link error occurs* option.
	EIRQ	Read/Set interrupt generation on link error option.
	DIS_ON_CE	Read/Set disable link on credit error option.
	DIS_ON_ER	Read/Set disable link on escape error option.
	DIS_ON_DE	Read/Set disable link on disconnect error option.
	DIS_ON_PE	Read/Set disable link on parity error option.
	DIS_ON_WE	Read/Set disable link on write synchronization error option (GRSPW1 only).
	DIS_ON_EE	Read/Set disable link on early EOP/EEP error option.
Param	`stscfg` [IO] pointer to bitmask If `stscfg` points to -1, the link status configuration is only read, otherwise it is updated according to the value in memory pointer to by `stscfg`. Use LINKSTS_* defines for `stscfg` bit declarations. The status configuration selects which link status bits to clear by the driver ISR. Bits in the link status register are cleared by the driver interrupt service routine if and only if the corresponding bit is set in the `stscfg` parameter.	
	Bitmask	Description (prefixed LINKSTS_)
	CE	Read/Set clear status from ISR for credit error
	ER	Read/Set clear status from ISR for escape error
	DE	Read/Set clear status from ISR for disconnect error
	PE	Read/Set clear status from ISR for parity error
	WE	Read/Set clear status from ISR for write synchronization error (GRSPW1 only)
	IA	Read/Set clear status from ISR for invalid address
	EE	Read/Set clear status from ISR for early EOP/EEP
Param	`clkdiv` [IO] pointer to integer If `clkdiv` points to -1, the clock divisor fields are only read from the I/O registers, otherwise it is updated according to the value in memory pointed to by `clkdiv`.	
Return	None.	

Table 13.10. `grspw_link_state` function declaration

Proto	`spw_link_state_t grspw_link_state(void *d)`	
About	Read and return current SpaceWire link status.	
Param	`d` [IN] pointer Device identifier. Returned from `grspw_open`.	
Return	enum spw_link_state_t. SpaceWire link status according to SpaceWire standard FSM state machine numbering. The possible return values are listed below, all numbers must be prefixed with SPW_LS_ declared by enum spw_link_state_t.	
	Value	Description.
	ERRRST	Error reset.
	ERRWAIT	Error Wait state.
	READY	Error Wait state.
	CONNECTING	Connecting state.
	STARTED	Stated state.
	RUN	Run state - link and DMA is fully operational.

Table 13.11. `grspw_link_status` function declaration

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

Table 13.12. `grspw_link_status_clr` function declaration

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

13.3.4. 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 13.2.8. The data structures and functions involved in controlling the node address configuration are listed below.

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

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

Table 13.13. grspw_addr_config data structure declaration

promiscuous	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. This field shall the set to the *inverse* of the effective 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	If separate node address is enabled this option sets the node address for DMA channel N (determined by array index).
	node_mask	If separate node address is enabled this option sets the node address mask for DMA channel N (determined by array index). This field shall the set to the *inverse* of the effective node address mask.

Table 13.14. `grspw_addr_ctrl` function declaration

Proto	`void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)`
About	Always read and optionally set the node addresses configuration. The GRSPW device is either configured to have one single node address or a range of addresses by masking. The *`cfg`* input memory layout is described by the grspw_addr_config data structure in Table 13.13. When using multiple DMA channels one must assign each DMA channel a unique node address or a unique range by masking. Each DMA channel is represented by the input *`dma_nacfg[N]`*.
Param	*`d`* [IN] pointer Device identifier. Returned from grspw_open.
Param	*`cfg`* [IO] pointer Address to where the driver will read or write the address configuration from. If the *`promiscous`* field is set to -1 the hardware is not written, instead the current configuration is only read and memory content updated accordingly.
Return	None.

13.3.5. Time Code support

SpaceWire Time Code handling is controlled and configured using the device API functions described below. The Time Code functionality is described in Section 13.2.5.

Table 13.15. `grspw_tc_ctrl` function declaration

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

Table 13.16. `grspw_tc_tx` function declaration

Proto	`void grspw_tc_tx(void *d)`
About	Generates a TimeCode Tick-In.

Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Return	None.

Table 13.17. `grspw_tc_isr` function declaration

Proto	`void grspw_tc_isr(void *d, void (*tcisr)(void *data, int tc), void *data)`
About	Assigns a Interrupt Service Routine (ISR) to handle TimeCode interrupt events. The ISR is called from the GRSPW device's interrupt handler, thus the isr is called in interrupt context and care needs to be taken. The ISR is called when a Tick-Out event has happened and an interrupt has been generated. The ISR is called with a custom argument *`data`* and the current value of the GRSPW TC register. The TC register contains TimeCode control flags and counter. The GRSPW interrupt handler always clears the GRSPW status field. It is performed after the ISR has been called. Note that even if the Tick-Out interrupt generation has not been enabled the ISR may still be called if other GRSPW interrupts are generated and the GRSPW status indicates that a Tick-Out has been received.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*`tcisr`* [IN] pointer to function If argument is NULL the Tick-Out ISR call is disabled. Otherwise the pointer will be used in a function call from interrupt context when a Tick-Out event is detected.
Param	*`data`* [IN] pointer to custom data This value is given as the first argument to the ISR.
Return	None.

Table 13.18. `grspw_tc_time` function declaration

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

13.3.6. 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 13.2.4.

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 13.19. `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 13.20. `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 13.21. `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.

13.3.7. RMAP Control

The device API described below is used to configure the hardware supported RMAP target. The RMAP support is described in Section 13.2.6.

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 13.32 for more information.

Table 13.22. `grspw_rmap_support` function declaration

Proto	`int grspw_rmap_support(void *d, char *rmap, char *rmap_crc)`	
About	Reads the RMAP hardware support of a GRSPW device. It is equivalent to use the `grspw_hw_support` function to get the RMAP functionality present in hardware.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Param	*`rmap`* [OUT] pointer If not NULL the RMAP configuration is stored into the address of *`rmap`*.	
	Value	Description
	0	RMAP target is not implemented in hardware.
	1	RMAP target is implemented in hardware.
Param	*`rmap_crc`* [OUT] pointer If not NULL the RMAP configuration is stored into the address of *`rmap`*.	
	Value	Description
	0	RMAP CRC algorithm is not implemented in hardware
	1	RMAP CRC algorithm is implemented in hardware
Return	None.	

Table 13.23. `grspw_rmap_ctrl` function declaration

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

Return	int. Status	
	0	Request successful.
	-1	Failed to enable RMAP handling in hardware. Not present in hardware.

13.3.8. Statistics

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

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

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

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

Table 13.24. grspw_core_stats data structure declaration

irq_cnt	Number of interrupts serviced for this GRSPW device.
err_credit	Number of credit errors experienced for this GRSPW device.
err_eeop	Number of Early EOP/EEP errors experienced for this GRSPW device.
err_addr	Number of invalid address errors experienced for this GRSPW device.
err_parity	Number of parity errors experienced for this GRSPW device.
err_disconnect	Number of disconnect errors experienced for this GRSPW device.
err_escape	Number of escape errors experienced for this GRSPW device.
err_wsync	Number of write synchronization errors experienced for this GRSPW device. This is only applicable for GRSPW cores.

Table 13.25. `grspw_stats_read` function declaration

Proto	`void grspw_stats_read(void *d, struct grspw_core_stats *sts)`
About	Reads the current driver statistics collected from earlier events by GRSPW device and driver usage. The statistics are stored to the address given by the second argument. The layout and content of the statistics are defined by the grspw_core_stats data structure described in Table 13.24. Note that the snapshot is taken without lock protection, as a consequence the statistics may not be synchronized with each other. This could be caused if the function is interrupted by a the GRSPW interrupt.
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*`sts`* [OUT] pointer If NULL no operating is performed. Otherwise a snapshot of the current driver statistics are copied to this user provided buffer. The layout and content of the statistics are defined by the grspw_core_stats data structure described in Table 13.24.
Return	None.

Table 13.26. `grspw_stats_clr` function declaration

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

13.4. 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.

13.4.1. 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 13.3.1 the GRSPW device the DMA channel belongs to must be opened and passed onto the DMA channel open routines.

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

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

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

- DMA channel I/O registers are initialized to a state where most are zero.
- Resources like semaphores used for the DMA channel implementation itself are allocated and initialized.
- The channel is marked opened to protect the caller from other users of the DMA channel.

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

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

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

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

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

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

  /* Initialize and activate DMA */
  if (grspw_dma_start(dma0)) {
```

```
   grspw_dma_close(dma0);
   grspw_close(device);
   return -3;
  }

  /* ... */

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

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

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

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

  /* ... */

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

Table 13.27. `grspw_dma_open` function declaration

Proto	`void *grspw_dma_open(void *d, int chan_no)`
About	Opens a DMA channel of a previously opened GRSPW device. The GRSPW device is identified by its device handle *d* and the DMA channel is identified by index *chan_no*. The returned value is used as input argument to all functions operating on the DMA channel.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Param	*chan_no* [IN] Integer DMA channel identification number. DMA channels are indexed by 0, 1, 2 or 3. Other input values cause NULL to be returned. The index must be equal or greater than zero, and smaller than the number of DMA channels reported by `grspw_hw_support`.
Return	Pointer. Status and driver's internal device identification.

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

Notes	May blocking until other GRSPW device operations complete.

Table 13.28. `grspw_dma_close` function declaration

Proto	`int grspw_dma_close(void *c)`
About	Closes a previously opened DMA channel. The specified DMA channel must be in stopped state before calling this function. Prior to closing the user is responsible for calling `grspw_dma_stop` to stop on-going DMA transfers and interrupts, free DMA channels resources and to unblock tasks waiting for RX/TX events on this DMA channel. Blocked tasks must have exited the device driver otherwise an error code is returned. If threads have been blocked within DMA operations they will be woken up and `grspw_dma_close` waits N ticks until they have returned to the caller with an error return value.

Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Return	int. Return code as indicated below.

Value	Description
0	Success.
1	Failure due to DMA channel is active (started) or tasks may be blocked within the driver by the RX/TX wait interface of this specific device.
-1	Failure due to invalid input arguments or unknown semaphore error.

13.4.2. Starting and stopping DMA operation

The start and stop operational modes are described in Section 13.2.14. 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 13.29, see function description for details on limitations.

Table 13.29. 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 13.4.6	Yes
`grspw_dma_rx_prepare`	Yes, with limitations, see Section 13.4.6	Yes
`grspw_dma_rx_count`	Yes, with limitations, see Section 13.4.7	Yes
`grspw_dma_rx_wait`	No	Yes
`grspw_dma_tx_send`	Yes, with limitations, see Section 13.4.5	Yes
`grspw_dma_tx_reclaim`	Yes, with limitations, see Section 13.4.5	Yes
`grspw_dma_tx_count`	Yes with limitations, see Section 13.4.7	Yes
`grspw_dma_tx_wait`	No	Yes
`grspw_dma_config`	Yes	No
`grspw_dma_config_read`	Yes	Yes
`grspw_dma_stats_read`	Yes	Yes
`grspw_dma_stats_clr`	Yes	Yes

Table 13.30. `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 already is in started mode, no action will be taken. The start routine clears and initializes the following: • DMA descriptor rings.

	• DMA queues. • Statistic counters. • Interrupt counters • I/O registers to match DMA configuration • Interrupt • DMA Status • Enables the receiver Even though the receiver is enabled the user is required to prepare empty receive buffers after this point, see `grspw_dma_rx_prepare`. The transmitter is enabled when the user provides send buffers that ends up in the TX SCHED queue, see `grspw_dma_tx_send`.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Return	int. Always returns zero.

Table 13.31. `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. Blocked tasks within the DMA channel will be woken up and return to caller with an error indication. This will cause the stop function to wait in N ticks until the blovked tasks have exited the driver. When no tasks have previously been blocked this function is not blocking either. Packets in the RX READY, RX SCHED queues will be moved to the RX RECV queue. The RXPKT_FLAG_RX packet flag is used to signal if the packet was received or just moved. Similarly, the packets in the TX SEND and TX SCHED queues are moved to the TX SENT queue and the TXPKT_FLAG_TX marks if the packet actually was transferred or not.
Param	*d* [IN] pointer Device identifier. Returned from `grspw_open`.
Return	None.

13.4.3. Packet buffer description

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

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

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

Table 13.32. grspw_pkt data structure declaration

next	The packet structure can be part of a linked list. This field is used to point out the next packet in the list. Set to NULL if this packet is the last in the list or a single packet.
pkt_id	User assigned ID. This field is never touched by the driver. It can be used to store a pointer or other data to help implement the user buffer handling.

<table>
<tr><td>flags</td><td>RX/TX transmission options and flags indicating resulting status. The bits described below is to be prefixed with TXPKT_FLAG_ or RXPKT_FLAG_ in order to match the TX or RX options definitions declared by the driver's header file.
<table>
<tr><td>Bits</td><td>TX Description (prefixed TXPKT_FLAG_)</td></tr>
<tr><td>NOCRC_MASK</td><td>Indicates to driver how many bytes should not be part of the header CRC calculation. 0 to 15 bytes can be omitted. Use NOCRC_LENN to select a specific length.</td></tr>
<tr><td>IE</td><td>Enable (1) or Disable (0) IRQ generation on packet transmission completed.</td></tr>
<tr><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>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>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>LINKERR</td><td>Set if a link error was exhibited during transmission of this packet.</td></tr>
<tr><td>Bits</td><td>RX Description (prefixed RXPKT_FLAG_)</td></tr>
<tr><td>IE</td><td>Enable (1) or Disable (0) IRQ generation on packet reception completed.</td></tr>
<tr><td>TRUNK</td><td>Set if packet was truncated.</td></tr>
<tr><td>DCRC</td><td>Set if data CRC error detected (only valid if RMAP CRC is enabled).</td></tr>
<tr><td>HCRC</td><td>Set if header CRC error detected (only valid if RMAP CRC is enabled).</td></tr>
<tr><td>EEOP</td><td>Set if an End-of-Packet error occurred during reception of this packet.</td></tr>
<tr><td>RX</td><td>Marks if packet was received or not.</td></tr>
</table></td></tr>
<tr><td>hlen</td><td>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>
<tr><td>dlen</td><td>Data payload length. The number of bytes hardware DMA read or written from/to the address indicated by the data pointer. On RX this is the complete packet data received.</td></tr>
<tr><td>data</td><td>Header Buffer Address. DMA will read from this. The address can be 4-byte or byte aligned depending on hardware.</td></tr>
<tr><td>hdr</td><td>Header Buffer Address. DMA will read hlen bytes from this. The address can be 4-byte or byte aligned depending on hardware. This field is not used by RX operation.</td></tr>
</table>

13.4.4. Blocking/Waiting on DMA activity

Blocking and polling mode are described in the Section 13.2.12. The functions described below are used to set up RX or TX wait conditions and blocks the calling thread until condition evaluates true.

Table 13.33. `grspw_dma_tx_wait` function declaration

Proto	`int grspw_dma_tx_wait(void *c, int send_cnt, int op, int sent_cnt, int timeout)`
About	Block until *`send_cnt`* or fewer packets are queued in TX "Send and Scheduled" queue, *`op`* (AND or OR), *`sent_cnt`* or more packet "have been sent" (Sent Q) condition is met. If a link error occurs and the "Disable on Link error" is defined, this function will also return to caller. The timeout argument is used to return after *`timeout`* ticks, regardless of the other conditions. If timeout is zero, the function will wait forever until the condition is satisfied. If IRQ of TX descriptors are not enabled conditions are never checked, this may hang infinitely unless a timeout has been specified.
Param	*`d`* [IN] pointer

	Device identifier. Returned from `grspw_open`.
Param	*`send_cnt`* [IN] int Sets the condition's number of packets in TX SEND queue.
Param	*`op`* [IN] boolean Condition operation. Set to zero for AND or one for OR.
Param	*`sent_cnt`* [IN] int Sets the condition's number of packets in TX SENT queue.
Param	*`timeout`* [IN] int Sets the timeout in number of system clock ticks. The operating system's semaphore service is used to implement the timeout functionality. Set to zero to disable timeout, negative value is invalid.
Return	Int. See return code below.

Value	Description
-1	Error.
0	Returning to caller because specified conditions are now fullfilled.
1	DMA stopped.
2	Timeout, conditions are not met.
3	Another task is already waiting. Service is Busy.

Table 13.34. `grspw_dma_rx_wait` function declaration

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

Value	Description
-1	Error.

	0	Returning to caller because specified conditions are now fullfilled.
	1	DMA stopped.
	2	Timeout, conditions are not met.
	3	Another task is already waiting. Service is Busy.

13.4.5. Sending packets

Packets are sent by adding packets to the SEND queue. Depending on the driver configuration and usage the packets eventually are put into SCHED queue where they will be assigned a DMA descriptor and scheduled for transmission. After transmission has completed the packet buffers can be retrieved to view the transmission status and to be able to reuse the packet buffers for new transfers. During the time the packet is in the driver it must not be accessed by the user.

Transmission of SpaceWire packets are described in Section 13.2.1.

In the below example Figure 13.4 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.

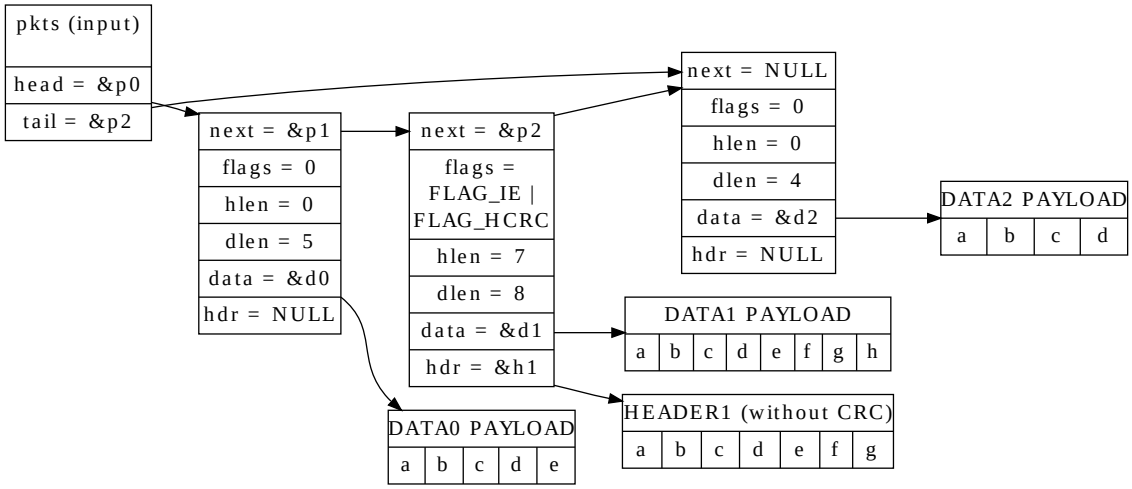


Figure 13.4. TX packet description `pkts` input to `grspw_tx_dma_send`

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

Table 13.35. `grspw_dma_tx_send` function declaration

Proto	`int grspw_dma_tx_send(void *c, int opts, struct grspw_list *pkts, int count)`
About	Schedules a list of packets for transmission at some point in future. The packets are put to the SEND queue of the driver. Depending on the input arguments a selection of the below steps are performed: 1. Move transmitted packets to SENT List (SCHED->SENT). 2. Add the requested packets to the SEND List (USER->SEND) 3. Schedule as many packets as possible for transmission (SEND->SCHED) Skipping both step 1 and 3 may be useful when IRQ is enabled, then the worker thread will be responsible for handling descriptors. The GRSPW transmitter is enabled when packets are added to the TX SCHED queue. The fastest solution in retrieving sent TX packets and sending new frames is to call: 1. grspw_dma_tx_reclaim(opts=0)

	2. grspw_dma_tx_send(opts=1) NOTE: the TXPKT_FLAG_TX flag must not be set in the packet structure.
Param	*c* [IN] pointer DMA channel identifier. Returned from grspw_dma_open.
Param	*opts* [IN] Integer bit-mask The above steps 1 and/or 3 may be skipped by setting *opts* argument according the description below.

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

Param	*pkts* [IN] pointer A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that *head* points to the first packet and *tail* points to the last. Call this function with *pkts* set to NULL to avoid step 2. Just doing step 1 and 3 as determined by *opts* is normally performed in polling-mode. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 13.32. Note that TXPKT_FLAG_TX of the *flags* field must not be set.
Param	*count* [IN] integer Number of packets in the packet list.
Return	Status. See return codes below

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

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

Table 13.36. grspw_dma_tx_reclaim function declaration

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

<table>
<tr><td></td><td colspan="2">The above steps 1 and/or 3 may be skipped by setting opts argument according the description below.</td></tr>
<tr><td></td><td>Bit</td><td>Description</td></tr>
<tr><td></td><td>0</td><td>Set to 1 to skip Step 1.</td></tr>
<tr><td></td><td>1</td><td>Set to 1 to skip Step 3.</td></tr>
<tr><td>Param</td><td colspan="2">pkts [OUT] pointer
The list will be initialized to contain the SpaceWire packets moved from the SENT queue to the packet list. The grspw_list structure will be initialized so that head points to the first packet, tail points to the last and the last packet (tail) next pointer is NULL.

Call this function with pkts set to NULL to avoid step 2. Just doing step 1 and 3 as determined by opts is normally performed in polling-mode.

The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 13.32. 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.</td></tr>
<tr><td>Param</td><td colspan="2">count [IO] pointer
Number of packets in the packet list.</td></tr>
<tr><td></td><td>Value</td><td>Input Description</td></tr>
<tr><td></td><td>NULL</td><td>Move all packets from the SENT queue to the packet list.</td></tr>
<tr><td></td><td>-1</td><td>Move all packets from the SENT queue to the packet list.</td></tr>
<tr><td></td><td>0</td><td>No packets are moved. Same as if pkts is NULL.</td></tr>
<tr><td></td><td>>0</td><td>Move a maximum of '*count' packets to the packet list.</td></tr>
<tr><td></td><td>Value</td><td>Output Description</td></tr>
<tr><td></td><td>NULL</td><td>Nothing performed.</td></tr>
<tr><td></td><td>others</td><td>'*count' is updated to reflect number of packets in packet list.</td></tr>
<tr><td>Return</td><td colspan="2">Status. See return codes below</td></tr>
<tr><td></td><td>Value</td><td>Description</td></tr>
<tr><td></td><td>-1</td><td>Error occurred, DMA channel may not be valid.</td></tr>
<tr><td></td><td>0</td><td>Successful. pkts list filled with all packets from sent list.</td></tr>
<tr><td></td><td>1</td><td>Indicates that DMA is stopped. Same as 0 but step 1 and 3 were never done.</td></tr>
<tr><td>Notes</td><td colspan="2">This function can only do step 1 and 2 to allow read out sent packets when in stopped mode. This is useful when a link goes down and the DMA activity is stopped by user of by driver automatically.</td></tr>
</table>

13.4.6. Receiving packets

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

Reception of SpaceWire packets are described in Section 13.2.1.

In the Figure 13.5 example three SpaceWire packets are received. The *count* parameters is set to three by the driver to reflect the number of packets. The first packet exhibited an early end-of-packet during reception which also resulted in header and data CRC error. All header points and header lengths have been set to zero by the user since they are no used, however the values in those fields does not affect the RX operations. The RX flag is set to indicate that DMA transfer was performed.

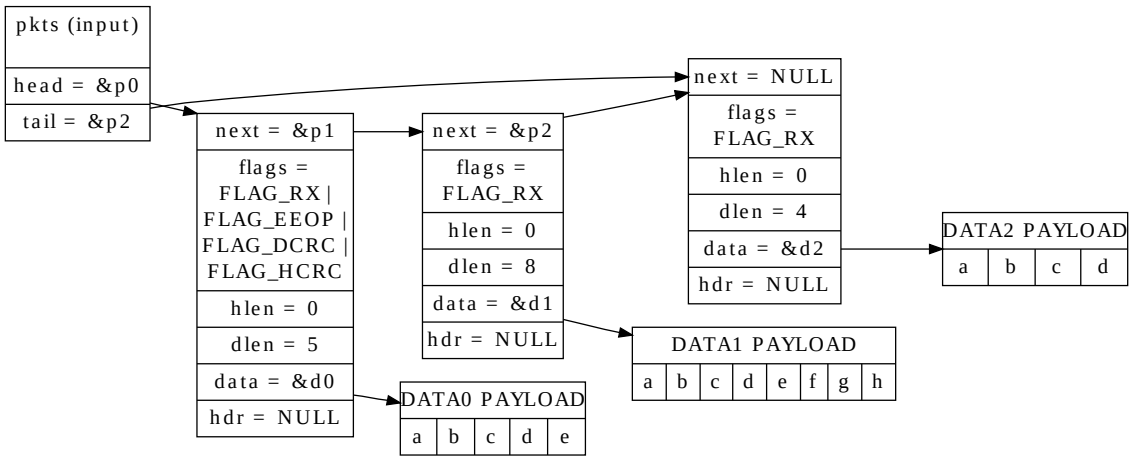


Figure 13.5. RX packet output from `grspw_rx_dma_recv`

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

Table 13.37. `grspw_dma_rx_prepare` function declaration

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

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

Param	*`pkts`* [IN] pointer A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that *`head`* points to the first packet and *`tail`* points to the last. Call this function with *`pkts`* set to NULL to avoid step 2. Just doing step 1 and 3 as determined by *`opts`* is normally performed in polling-mode. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 13.32. Note that RXPKT_FLAG_RX of the *`flags`* field must not be set.

<table>
<tr><td>Param</td><td>count [IN] integer
Number of packets in the packet list.</td></tr>
<tr><td>Return</td><td>Status. See return codes below<table><tr><td>Value</td><td>Description</td></tr><tr><td>-1</td><td>Error occurred, DMA channel may not be valid.</td></tr><tr><td>0</td><td>Successfully added pkts to RX READY/SCHED list.</td></tr><tr><td>1</td><td>DMA stopped. No operation.</td></tr></table></td></tr>
<tr><td>Notes</td><td>This function performs no operation when the DMA channel is stopped.</td></tr>
</table>

Table 13.38. `grspw_dma_rx_recv` function declaration

<table>
<tr><td>Proto</td><td><code>int grspw_dma_rx_recv(void *c, int opts, struct grspw_list *pkts, int *count)</code></td></tr>
<tr><td>About</td><td>Get received RX packet buffers that has previously been scheduled for reception with <code>grspw_dma_rx_prepare</code>. The packets in the RX RECV queue are moved to the pkts packet list. When the move has been completed the packet(s) can safely be reused again by the user. The packet structures have been updated with reception status to indicate transfer failures of individual packets, received packet length. The header pointer and length fields are not touched by the driver, all data ends up in the data area. Depending on the input arguments a selection of the below steps are performed:
Move scheduled packets to RECV List (SCHED->RECV).
Move all RECV packet to the callers list (RECV->USER).
Schedule as many free packet buffers as possible (READY->SCHED).
Skipping both step 1 and 3 may be useful when IRQ is enabled, then the worker thread will be responsible for descriptor processing. Skipping only step 2 can be useful in polling mode.

The fastest solution in retrieving received RX packets and preparing new packet buffers for future receive, is to call:
grspw_dma_rx_recv(opts=2, &recvlist) (Skip step 3)
grspw_dma_rx_prepare(opts=1, &freelist) (Skip step 1)
NOTE: the RXPKT_FLAG_RX flag indicates if a packet was received, thus if the data field contains new valid data or not.</td></tr>
<tr><td>Param</td><td>c [IN] pointer
DMA channel identifier. Returned from <code>grspw_dma_open</code>.</td></tr>
<tr><td>Param</td><td>opts [IN] Integer bit-mask
The above steps 1 and/or 3 may be skipped by setting opts argument according the description below.<table><tr><td>Bit</td><td>Description</td></tr><tr><td>0</td><td>Set to 1 to skip Step 1.</td></tr><tr><td>1</td><td>Set to 1 to skip Step 3.</td></tr></table></td></tr>
<tr><td>Param</td><td>pkts [OUT] pointer
The list will be initialized to contain the SpaceWire packets moved from the RECV queue to the packet list. The grspw_list structure will be initialized so that head points to the first packet, tail points to the last and the last packet (tail) next pointer is NULL.

Call this function with pkts set to NULL to avoid step 2. Just doing step 1 and 3 as determined by opts is normally performed in polling-mode.

The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 13.32. 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</td></tr>
</table>

	TRUNK, DCRC, HCRC and EEOP flags indicates if an error occurred during reception of the packet, regardless the RXPKT_FLAG_RX is set to indicate packet reception attempt.
Param	*count* [IO] pointer Number of packets in the packet list.

Value	Input Description
NULL	Move all packets from the RECV queue to the packet list.
-1	Move all packets from the RECV queue to the packet list.
0	No packets are moved. Same as if pkts is NULL.
>0	Move a maximum of '*count' packets to the packet list.
Value	Output Description
NULL	Nothing performed.
others	'*count' is updated to reflect number of packets in packet list.

Return	Status. See return codes below

Value	Description
-1	Error occurred, DMA channel may not be valid.
0	Successful. pkts list filled with all packets from recv list.
1	Indicates that DMA is stopped. Same as 0 but step 1 and 3 were never done.

Notes	This function can only do step 1 and 2 to allow read out received packets when in stopped mode. This is useful when a link goes down and the DMA activity is stopped by user or by driver automatically.

13.4.7. Transmission queue status

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

Table 13.39. `grspw_dma_tx_count` function declaration

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

Table 13.40. `grspw_dma_rx_count` function declaration

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

13.4.8. Statistics

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

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

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

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

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

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

Table 13.41. grspw_dma_stats data structure declaration

irq_cnt	Number of interrupts serviced for this DMA channel.

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

Table 13.42. `grspw_dma_stats_read` function declaration

Proto	`void grspw_dma_stats_read(void *d, struct grspw_dma_stats *sts)`
About	Reads the current driver statistics collected from earlier events by a DMA channel and DMA channel usage. The statistics are stored to the address given by the second argument. The layout and content of the statistics are defined by the grspw_dma_stats data structure is described in Table 13.41. Note that the snapshot is taken without lock protection, as a consequence the statistics may not be synchronized with each other. This could be caused if the function is interrupted by a the GRSPW interrupt or other tasks performing driver operations on the same DMA channel.
Param	*`c`* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*`sts`* [OUT] pointer A snapshot of the current driver statistics are copied to this user provided buffer. The layout and content of the statistics are defined by the grspw_dma_stats data structure is described in Table 13.41.
Return	None.

Table 13.43. `grspw_dma_stats_clr` function declaration

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

13.4.9. DMA channel configuration

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

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

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

Table 13.44. grspw_dma_config data structure declaration

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

Table 13.45. `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 identifier. Returned from `grspw_dma_open`.
Param	*`cfg`* [IN] pointer Address to where the driver will read or write the DMA channel configuration from. The configuration options are described in Table 13.44.

Return	int. Return code as indicated below.	
	Value	Description
	0	Success.
	-1	Failure due to invalid input arguments or DMA has already been started.

Table 13.46. `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 identifier. Returned from `grspw_dma_open`.
Param	*`sts`* [OUT] pointer The driver DMA channel configuration options are copied to this user provided buffer. The layout and content of the statistics are defined by the grpsw_dma_config data structure is described in Table 13.44.
Return	None.

13.5. API reference

This section lists all functions and data structures part of the GRSPW driver API, and in which section(s) they are described. The API is also documented in the source header file of the driver, see Section 13.1.3.

13.5.1. Data structures

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

Table 13.47. Data structures reference

Data structure name	Section
struct grspw_pkt	13.4.3
struct grspw_list	13.2.11
struct grspw_addr_config	13.3.4
struct grspw_hw_sup	13.3.2
struct grspw_core_stats	13.3.8
struct grspw_dma_config	13.4.9
struct grspw_dma_stats	13.4.8

13.5.2. 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 13.48. Device function reference

Prototype	Section
`int grspw_dev_count(void)`	13.3.1
`void *grspw_open(int dev_no)`	13.3.1
`void grspw_close(void *d)`	13.3.1
`void grspw_hw_support(void *d, struct grspw_hw_sup *hw)`	13.3.2
`void grspw_stats_read(void *d, struct grspw_core_stats *sts)`	13.3.8
`void grspw_stats_clr(void *d)`	13.3.8

Prototype	Section
`void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)`	13.3.4, 13.2.8
`spw_link_state_t grspw_link_state(void *d)`	13.3.3, 13.2.4
`void grspw_link_ctrl(void *d, int *options, int *clkdiv)`	13.3.3, 13.2.4
`unsigned int grspw_link_status(void *d)`	13.3.3, 13.2.4
`void grspw_link_status_clr(void *d, unsigned int mask)`	13.3.3, 13.2.4
`void grspw_tc_ctrl(void *d, int *options)`	13.3.5, 13.2.5
`void grspw_tc_tx(void *d)`	13.3.5, 13.2.5
`void grspw_tc_isr(void *d, void (*tcisr)(void *data, int tc), void *data)`	13.3.5, 13.2.5
`void grspw_tc_time(void *d, int *time)`	13.3.5, 13.2.5
`int grspw_rmap_ctrl(void *d, int *options, int *dstkey)`	13.3.7, 13.2.6
`void grspw_rmap_support(void *d, char *rmap, char *rmap_crc)`	13.3.7, 13.2.6, 13.3.2
`int grspw_port_ctrl(void *d, int *port)`	13.3.6, 13.2.7
`int grspw_port_count(void *d)`	13.3.6, 13.2.7, 13.3.2
`int grspw_port_active(void *d)`	13.3.6, 13.2.7

13.5.3. 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 13.49. DMA channel function reference

Prototype	Section
`void *grspw_dma_open(void *d, int chan_no)`	13.2.1, 13.4.1, 13.3.1
`void grspw_dma_close(void *c)`	13.2.1, 13.4.1, 13.3.1
`int grspw_dma_start(void *c)`	13.4.2, 13.2.14
`void grspw_dma_stop(void *c)`	13.4.2, 13.2.14

Prototype	Section
`int grspw_dma_rx_recv(void *c, int opts, struct grspw_list *pkts, int *count)`	13.4.6, 13.2.1
`int grspw_dma_rx_prepare(void *c, int opts, struct grspw_list *pkts, int count)`	13.4.6, 13.2.1
`void grspw_dma_rx_count(void *c, int *ready, int *sched, int *recv)`	13.4.7, 13.2.11.1
`int grspw_dma_rx_wait(void *c, int recv_cnt, int op, int ready_cnt, int timeout)`	13.4.4, 13.2.12
`int grspw_dma_tx_send(void *c, int opts, struct grspw_list *pkts, int count)`	13.4.5, 13.2.1
`int grspw_dma_tx_reclaim(void *c, int opts, struct grspw_list *pkts, int *count)`	13.4.5, 13.2.1
`void grspw_dma_tx_count(void *c, int *send, int *sched, int *sent)`	13.4.7, 13.2.11.1
`int grspw_dma_tx_wait(void *c, int send_cnt, int op, int sent_cnt, int timeout)`	13.4.4, 13.2.12
`int grspw_dma_config(void *c, struct grspw_dma_config *cfg)`	13.4.9
`void grspw_dma_config_read(void *c, struct grspw_dma_config *cfg)`	13.4.9
`void grspw_dma_stats_read(void *c, struct grspw_dma_stats *sts)`	13.4.8
`void grspw_dma_stats_clr(void *c)`	13.4.8

14. GRSPW GRLIB SpaceWire driver

14.1. Introduction

This document is intended as an aid in getting started developing with Gaisler GRSPW SpaceWire core using the GRSPW driver for RTEMS. It briefly takes the reader through some of the most important steps in using the driver such as setting up a connection, configuring the driver, reading and writing packets. The reader is assumed to be well acquainted with SpaceWire and RTEMS.

The cores supported are GRSPW, GRSPW2 and SpaceWire Router DMA interface.

The GRSPW driver require the RTEMS Driver Manager.

See the GRLIB IP Core User's Manual for GRSPW hardware details.

14.1.1. Software driver

The driver provides means for processes and threads to send and receive packets. Link errors can be detected by polling or by using a dedicated task sleeping until a link error is detected.

The driver is somewhat similar to an Ethernet driver. However, an Ethernet driver is referenced by an IP stack layer. The IP stack can detect missing or erroneous packets, since the user talks directly with the GRSPW driver it is up to the user to handle errors. The driver aims to be fully user space controllable in contrast to Ethernet drivers.

14.1.2. Examples

There is a example of how to use the GRSPW driver distributed together with the driver. The example demonstrates some fundamental approaches to access and use the driver. It is made up of two tasks communicating with each other through two SpaceWire devices. To be able to run the example one must have two GRSPW devices connected together on the same board or two boards with at least one GRSPW core on each board.

14.2. User interface

The RTEMS GRSPW driver supports the standard access routines to file descriptors such as **read**, **write** and **ioctl**. User applications should include the grspw driver's header file which contains definitions of all necessary data structures used when accessing the driver. The RTEMS GRSPW sample is called rtems-spwtest-2boards.c and it is provided in the Gaisler Research RTEMS distribution.

14.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when GRSPW hardware is found for the first time. The driver is called from the driver manager to handle detected GRSPW hardware. In order for the driver manager to unite the GRSPW driver with the GRSPW hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

14.2.2. Driver resource configuration

The driver can be configured using driver resources as described in the driver manager chapter. Below is a description of configurable driver parameters. The driver parameters is unique per GRSPW device. The parameters are all optional, the parameters only overrides the default values.

Table 14.1. GRSPW driver parameter description

Name	Type	Parameter Description
txBdCnt	INT	Number of transmit descriptors.
rxBdCnt	INT	Number of receive descriptors.
txDataSize	INT	Maximum transmit packet data size.
txHdrSize	INT	Maximum transmit packet header size.

Name	Type	Parameter Description
rxPktSize	INT	Maximum packet size of received packets.
rxDmaArea	INT	Custom receiver DMA area address. See note below.
txDataDmaArea	INT	Custom transmit Data DMA area address. See note below.
txHdrDmaArea	INT	Custom transmit Header DMA area address. See note below.

14.2.2.1. Custom DMA area parameters

The three DMA areas can be configured to be located at a custom address. The standard configuration is to leave it up to the driver to do dynamic allocation of the areas. However in some cases it may be required to locate the DMA area on a custom location, the driver will not allocate memory but will assume that enough memory is available and that the alignment needs of the core on the address given is fulfilled. The memory required can be calculated from the other parameters.

For some systems it may be convenient to give the addresses as seen by the GRSPW core. This can be done by setting the LSB bit in the address to one. For example a GR-RASTA-IO board with a GRSPW core doesn't read from the same address as the CPU in order to access the same data. This is dependent on the PCI mappings. Translation between CPU and GRPSW addresses must be done. The GRSPW driver automatically translates addresses in the descriptors. This requires the bus driver, in this case the GR-RASTA-IO driver, to set up translation addresses correctly.

14.2.3. Opening the device

Opening the device enables the user to access the hardware of a certain GRSPW device. Open reset the SpaceWire core and reads reset values of certain registers. With the **ioctl** command START it is possible to wait for the link to enter run state. The same driver is used for all GRSPW devices available. The devices are separated by assigning each device a unique name, the name is passed during the opening of the driver. Some example device names are printed out below.

Table 14.2. Device number to device name conversion

Device number	Filesystem name	Location
0	/dev/grspw0	On-Chip Bus
1	/dev/grspw1	On-Chip Bus
2	/dev/grspw2	On-Chip Bus
Depends on system configuration	/dev/rastaio0/grspw0	GR-RASTA-IO
Depends on system configuration	/dev/rastatmtc0/grspw1	GR-RASTA-TMTC

An example of an RTEMS open call is shown below.

```
fd = open("/dev/grspw0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 14.1.

Table 14.3. Open ERRNO values

ERRNO	Description
EINVAL	Illegal device name or not available
EBUSY	Device already opened
EIO	Error when writing to grspw registers

14.2.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the SpaceWire driver.

14.2.5. I/O Control interface

Changing the behaviour of the driver for a device is done via the standard system call **ioctl**. Most operating systems support at least two arguments to ioctl, the first being an integer which selects ioctl function and secondly a pointer to data that may be interpreted uniquely for each function. A typical ioctl call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global errno variable is set accordingly.

The commands may differ slightly between the operating systems but is mainly the same. The unique ioctl commands are described last in this section.

All supported commands and their data structures are defined in the GRSPW driver's header file `grspw.h`. In functions where only one argument in needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

14.2.5.1. Data structures

The `spw_ioctl_packetsize` data structure is used when changing the size of the driver's receive and transmit buffers.

```
typedef struct {
  unsigned int rxsize;
  unsigned int txdsize;
  unsigned int txhsize;
} spw_ioctl_packetsize;
```

Table 14.4. spw_ioctl_packetsize member descriptions.

Member	Description
rxsize	Sets the size of the receiver descriptor buffers.
txdsize	Sets the size of the transmitter data buffers.
txhsize	Sets the size of the transmitter header buffers.

The spw_ioctl_pkt_send struct is used for transmissions through the ioctl call. Se the transmission section for more information. The sent variable is set by the driver when returning from the ioctl call while the other are set by the caller.

```
typedef struct {
   unsigned int hlen;
   char *hdr;
   unsigned int dlen;
   char *data;
   unsigned int sent;
} spw_ioctl_pkt_send;
```

Table 14.5. spw_ioctl_pkt_send member descriptions.

Member	Description
hlen	Number of bytes that shall be transmitted from the header buffer.
hdr	Pointer to the header buffer.
dlen	Number of bytes that shall be transmitted from the data buffer.
data	Pointer to the data buffer.
sent	Number of bytes transmitted.

The spw_stats struct contains various statistics gathered from the GRSPW.

```
typedef struct {
  unsigned int tx_link_err;
  unsigned int rx_rmap_header_crc_err;
  unsigned int rx_rmap_data_crc_err;
  unsigned int rx_eep_err;
  unsigned int rx_truncated;
  unsigned int parity_err;
  unsigned int escape_err;
  unsigned int credit_err;
```

```
  unsigned int write_sync_err;
  unsigned int disconnect_err;
  unsigned int early_ep;
  unsigned int invalid_address;
  unsigned int packets_sent;
  unsigned int packets_received;
} spw_stats;
```

Table 14.6. spw_stats member descriptions.

Member	Description
tx_link_err	Number of link-errors detected during transmission.
rx_rmap_header_crc_err	Number of RMAP header CRC errors detected in received packets.
rx_rmap_data_crc_err	Number of RMAP data CRC errors detected in received packets.
rx_eep_err	Number of EEPs detected in received packets.
rx_truncated	Number of truncated packets received.
parity_err	Number of parity errors detected.
escape_err	Number of escape errors detected.
credit_err	Number of credit errors detected.
write_sync_err	Number of write synchronization errors detected.
disconnect_err	Number of disconnect errors detected.
early_ep	Number of packets received with an early EOP/EEP.
invalid_address	Number of packets received with an invalid destination address.
packets_sent	Number of packets transmitted.
packets_received	Number of packets received.

The `spw_config` structure holds the current configuration of the GRSPW.

```
typedef struct {
   unsigned int nodeaddr;
   unsigned int destkey;
   unsigned int clkdiv;
   unsigned int rxmaxlen;
   unsigned int timer;
   unsigned int disconnect;
   unsigned int promiscuous;
   unsigned int timetxen;
   unsigned int timerxen;
   unsigned int rmapen;
   unsigned int rmapbufdis;
   unsigned int linkdisabled;
   unsigned int linkstart;

   unsigned int check_rmap_err;
   unsigned int rm_prot_id;
   unsigned int tx_blocking;
   unsigned int tx_block_on_full;
   unsigned int rx_blocking;
   unsigned int disable_err;
   unsigned int link_err_irq;
   rtems_id event_id;

   unsigned int is_rmap;
   unsigned int is_rxunaligned;
   unsigned int is_rmapcrc;
} spw_config;
```

Table 14.7. spw_config member descriptions.

Member	Description
nodeaddr	Node address.
destkey	Destination key.
clkdiv	Clock division factor.

Member	Description
rxmaxlen	Receiver maximum packet length.
timer	Link-interface 6.4 us timer value.
disconnect	Link-interface disconnection timeout value.
promiscuous	Promiscuous mode.
rmapen	RMAP command handler enable.
rmapbufdis	RMAP multiple buffer enable.
linkdisabled	Linkdisabled.
linkstart	Linkstart.
check_rmap_error	Check for RMAP CRC errors in received packets.
rm_prot_id	Remove protocol ID from received packets.
tx_blocking	Select between blocking and non-blocking transmissions.
tx_blocking_on_full	Block when all transmit descriptors are occupied.
rx_blocking	Select between blocking and non-blocking receptions.
disable_err	Disable Link automatically when link-error interrupt occurs.
link_err_irq	Enable link-error interrupts.
event_id	Task ID to which event is sent when link-error interrupt occurs.
is_rmap	RMAP command handler available.
is_rxunaligned	RX unaligned support available.
is_rmapcrc	RMAP CRC support available.

14.2.5.2. Configuration

The GRSPW core and driver are configured using ioctl calls. Table 19 below lists all supported ioctl calls common to most operating systems. SPACEWIRE_IOCTRL_ should be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 for failure. Errno is set after a failure as indicated in Table 14.3.

```
result = ioctl(fd, SPACEWIRE_IOCTRL_SET_NODEADDR, 0xFE);
```

Operating system specific calls are described last in this section.

Table 14.8. ERRNO values for ioctl calls.

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	Only used for SEND. Returned when no descriptors are available in non-blocking mode.
ENOSYS	Returned for SET_DESTKEY if RMAP command handler is not available or if a non-implemented call is used.
ETIMEDOUT	Returned for SET_PACKETSIZE and START if the link could not be brought up.
ENOMEM	Returned for SET_PACKETSIZE if it was unable to allocate the new buffers.
EIO	Error when writing to grspw hardware registers.

Table 14.9. ioctl calls supported by the GRSPW driver.

Call Number	Description
START	Bring up link after open or STOP
STOP	Stops the SpaceWire receiver and transmitter, this makes the following read and write calls fail until START is called.
SET_NODEADDR	Change node address.

Call Number	Description
SET_RXBLOCK	Change blocking mode of receptions.
SET_DESTKEY	Change destination key.
SET_CLKDIV	Change clock division factor.
SET_TIMER	Change timer setting.
SET_DISCONNECT	Change disconnection timeout.
SET_COREFREQ	Calculates TIMER and DISCONNECT from a user provided SpaceWire core frequency. Frequency is given in KHz.
SET_PROMISCUOUS	Enable/Disable promiscuous mode.
SET_RMAPEN	Enable/Disable RMAP command handler.
SET_RMAPBUFDIS	Enable/Disable multiple RMAP buffer utilization.
SET_CHECK_RMAP	Enable/Disable RMAP CRC error check for reception.
SET_RM_PROT_ID	Enable/Disable protocol ID removal for reception.
SET_TXBLOCK	Change blocking mode of transmissions.
SET_TXBLOCK_ON_FULL	Change the blocking mode when all descriptors are in use.
SET_DISABLE_ERR	Enable/Disable automatic link disabling when link error occurs.
SET_LINK_ERR_IRQ	Enable/Disable link error interrupts.
SET_PACKETSIZE	Change buffer sizes.
GET_LINK_STATUS	Read the current link status.
SET_CONFIG	Set all configuration parameters with one call.
GET_CONFIG	Read the current configuration parameters.
GET_STATISTICS	Read the current configuration parameters.
CLR_STATISTICS	Clear all statistics
SEND	Send a packet with both header and data buffers.
LINKDISABLE	Disable the link.
LINKSTART	Start the link.
SET_EVENT_ID	Change the task ID to which link error events are sent.
SET_TCODE_CTRL	Control timecode interrupt and timecode reception/transmission enable.
SET_TCODE	Optionally set timecode register and optionally generate a tick-in.
GET_TCODE	Read GRSPW timecode register (get last timecode received).

14.2.5.2.1. START

This call try to bring the link up. The call returns successfully when the link enters the link state [run]. START is typically called after open and the ioctl commands SET_DISCONNECT, SET_TIMER or SET_COREFREQ. Calls to write or read will fail unless START is successfully called first.

Table 14.10. START argument description

Argument	Timeout function
-1	Default hard coded driver timeout. Can be set with a define.
less than -1	Wait for link forever, the link is checked every 10 ticks.
0	No timeout is used, if link is not up when entering START the call will fail with errno set to EINVAL.
positive	The argument specifies the number of clock ticks the driver will wait before START returns with error status. The link is checked every 10 ticks.

14.2.5.2.2. STOP

STOP disables the GRSPW receiver and transmitter it does not effect link state. After calling STOP subsequent calls to read and write will fail until START has successfully returned. The call takes no arguments. STOP never fail.

14.2.5.2.3. SET_NODEADDR

This call sets the node address of the device. It is only used to check the destination of incoming packets. It is also possible to receive packets from all addresses, see SET_PROMISCUOUS.

The argument must be an integer in the range 0 to 255. The call will fail if the argument contains an illegal value or if the register can not be written.

14.2.5.2.4. SET_RXBLOCK

This call sets the blocking mode for receptions. Setting this flag makes calls to read blocking when there is no available packets. If the flag is not set read will return EBUSY when there are no incoming packets available.

The argument must be an integer in the range 0 to 1. 0 selects non blocking mode while 1 selects blocking mode. The call will fail if the argument contains an illegal value.

14.2.5.2.5. SET_DESTKEY

This call sets the destination key. It can only be used if the RMAP command handler is available. The argument must be an integer in the range 0 to 255. The call will fail if the argument contains an illegal value, if the RMAP command handler is not available or if the register cannot be written.

14.2.5.2.6. SET_CLKDIV

This call sets the clock division factor used in the run-state. The argument must be an integer in the range 0 to 255. The call will fail if the argument contains an illegal value or if the register cannot be written.

14.2.5.2.7. SET_TIMER

This call sets the counter used to generate the 6.4 and 12.8 us time-outs in the link-interface FSM. The argument must be an integer in the range 0 to 4095. The call will fail if the argument contains an illegal value or if the register cannot be written. This value can be calculated by the driver, see SET_COREFREQ.

14.2.5.2.8. SET_DISCONNECT

This call sets the counter used to generate the 850 ns disconnect interval in the link-interface FSM. The argument must be an integer in the range 0 to 1023. The call will fail if the argument contains an illegal value or if the register cannot be written. This value can be calculated by the driver, see SET_COREFREQ.

14.2.5.2.9. SET_COREFREQ

This call calculates timer and disconnect from the GRSPW core frequency. The call take one unsigned 32-bit argument, see table below. This call can be used instead of the calls SET_TIMER and SET_DISCONNECT.

Table 14.11. SET_COREFREQ argument description

Argument Value	Function
0	The GRSPW core frequency is assumed to be equal to the system frequency. The system frequency is detected by reading the system tick timer or a hard coded frequency.
all other values	The argument is taken as the GRSPW core frequency in KHz.

14.2.5.2.10. SET_PROMISCUOUS

This call sets the promiscuous mode bit. The argument must be an integer in the range 0 to 1. The call will fail if the argument contains an illegal value or if the register cannot be written.

14.2.5.2.11. SET_RMAPEN

This call sets the RMAP enable bit. It can only be used if the RMAP command handler is available. The argument must be an integer in the range 0 to 1. The call will fail if the argument contains an illegal value, if the RMAP command handler is not available or if the register cannot be written.

14.2.5.2.12. SET_RMAPBUFDIS

This call sets the RMAP buffer disable bit. It can only be used if the RMAP command handler is available. The argument must be an integer in the range 0 to 1. The call will fail if the argument contains an illegal value, if the RMAP command handler is not available or if the register cannot be written.

14.2.5.2.13. SET_CHECK_RMAP

This call selects whether or not RMAP CRC should be checked for received packets. If enabled the header CRC error and data CRC error bits are checked and if one or both are set the packet will be discarded. The argument must be an integer in the range 0 to 1. 0 disables and 1 enables the RMAP CRC check. The call will fail if the argument contains an illegal value.

14.2.5.2.14. SET_RM_PROT_ID

This call selects whether or not the protocol ID should be removed from received packets. It is assumed that all packets contain a protocol ID so when enabled the second byte (the one after the node address) in the packet will be removed. The argument must be an integer in the range 0 to 1. 0 disables and 1 enables the RMAP CRC check. The call will fail if the argument contains an illegal value.

14.2.5.2.15. SET_TXBLOCK

This call sets the blocking mode for transmissions. The calling process will be blocked after each write until the whole packet has been copied into the GRSPW send FIFO buffer.

The argument must be an integer in the range 0 to 1. 0 selects non blocking mode while 1 selects blocking mode. The call will fail if the argument contains an illegal value.

14.2.5.2.16. SET_TXBLOCK_ON_FULL

This call sets the blocking mode for transmissions when all transmit descriptors are in use. The argument must be an integer in the range 0 to 1. 0 selects non blocking mode while 1 selects blocking mode. The call will fail if the argument contains an illegal value.

14.2.5.2.17. SET_DISABLE_ERR

This call sets automatic link-disabling due to link-error interrupts. Link-error interrupts must be enabled for it to have any effect. The argument must be an integer in the range 0 to 1. 0 disables automatic link- disabling while a 1 enables it. The call will fail if the argument contains an illegal value.

14.2.5.2.18. SET_LINK_ERR_IRQ

This call sets the link-error interrupt bit in the control register. The interrupt-handler sends an event to the task specified with the event_id field when this interrupt occurs. The argument must be an integer in the range 0 to 1. The call will fail if the argument contains an illegal value or if the register write fails.

14.2.5.2.19. SET_PACKETSIZE

This call changes the size of buffers and consequently the maximum packet sizes. The this cannot be done while the core accesses the buffers so first the receiver and the transmitter is disabled and ongoing DMA transactions is waited upon to finish. The time taken to wait for receiving DMA transactions to finish may vary depending on packet size and SpaceWire core frequency. The old buffers are reallocated and the receiver and transmitter is enabled again. The configuration before the call will be preserved (except for the packet sizes). The argument must be a pointer to a `spw_ioctl_packetsize` struct. The call will fail if the argument contains an illegal pointer, the requested buffer sizes cannot be allocated or the link cannot be re-started.

14.2.5.2.20. GET_LINK_STATUS

This call returns the current link status. The argument must be a pointer to an integer. The return value in the argument can be one of the following: 0 = Error-reset, 1 = Error-wait, 2 = Ready, 3 = Started, 4 = Connecting, 5 = Run. The call will fail if the argument contains an illegal pointer.

14.2.5.2.21. GET_CONFIG

This call returns all configuration parameters in a spw_config struct which is defined in `spacewire.h`. The argument must be a pointer to a `spw_config` struct. The call will fail if the argument contains an illegal pointer.

14.2.5.2.22. GET_STATISTICS

This call returns all statistics in a `spw_stats` struct. The argument must be a pointer to a `spw_stats struct`. The call will fail if the argument contains an illegal pointer.

14.2.5.2.23. CLR_STATISTICS

This call clears all statistics. No argument is taken and the call always succeeds.

14.2.5.2.24. SEND

This call sends a packet. The difference to the normal write call is that separate data and header buffers can be used. The argument must be a pointer to a spw_ioctl_send struct. The call will fail if the argument contains an illegal pointer, or the struct contains illegal values. See the transmission section for more information.

14.2.5.2.25. LINKDISABLE

This call disables the link (sets the linkdisable bit to 1 and the linkstart bit to 0). No argument is taken. The call fails if the register write fails.

14.2.5.2.26. LINKSTART

This call starts the link (sets the linkdisable bit to 0 and the linkstart bit to 1). No argument is taken. The call fails if the register write fails.

14.2.5.2.27. SET_EVENT_ID

This call sets the task ID to which an event is sent when a link-error interrupt occurs. The argument can be any positive integer. The call will fail if the argument contains an illegal value.

14.2.5.2.28. SET_TCODE_CTRL

This call is used to control the timecode functionality of the GRSPW core. The TR (Timecode RX Enable), TT (Timecode TX enable) and TQ (Tick-out IRQ) bits in the control register can be set or cleared, if tick-out IRQ is enabled global IRQ is also enabled. The argument is a 12-bit mask, the least significant four bits determines which bits are written (mask), the upper four bits determine the new register value of the enabled bits, please see the SPACEWIRE_TCODE_CTRL_* definitions.

When Tick-out interrupt is enabled the `grspw_timecode_callback` function is called for every tick-out that is received, it is called from the interrupt service routine in interrupt context. The function pointer must be set by the user to point to a function to handle tick-in interrupts. The function is global for all GRSPW devices, the arguments [regs] and [minor] both identify an unique GRSPW core and the [tc] argument determines the current value of the timecode register upon interrupt.

```
void (*grspw_timecode_callback)
   (void *pDev, void *regs, int minor, unsigned int tc);
```

14.2.5.2.29. SET_TCODE

This call sets the timecode register and/or generates a tick-in. The operation is controlled by the argument bit-mask, setting SPACEWIRE_TCODE_SET will result in the lower 8-bits will be written to the timecode register

whereas setting SPACEWIRE_TCODE_TX will result in a tick- in generation. If both operations are enabled the tick-in is generated after the timecode register is written.

14.2.5.2.30. GET_TCODE

This call reads the current GRSPW timecode register (unsigned int) and stores it in the location provided by the the user argument. Bit 8 is set if the timecode status register bit 0 (TO) is set, the status bit (TO) is cleared if set, this is to indicate if the timecode register has been updated since last call. Note that if interrupt is enabled and the callback function is assigned the TO bit is cleared by the interrupt handler. The argument must be a pointer to an unsigned integer. The call will fail if the argument contains an illegal pointer.

14.2.6. Transmission

Transmitting single packets are done with either the write [call] or a special **ioctl** call. There is currently no support for writing multiple packets in one call. Write calls are used when data only needs to be taken from a single contiguous buffer. An example of a write call is shown below:

```
result = write(fd, tx_pkt, 10))
```

On success the number of transmitted bytes is returned and -1 on failure. Errno is also set in the latter case. Tx_pkt points to the beginning of the packet which includes the destination node address. The last parameter sets the number of bytes that the user wants to transmit.

The call will fail if the user tries to send more bytes than is allocated for a single packet (this can be changed with the SET_PACKETSIZE ioctl call) or if a NULL pointer is passed. Write also fails if the link has not been started with the ioctl command START.

The write call can be configured to block in different ways. If normal blocking is enabled the call will only return when the packet has been transmitted. In non-blocking mode, the transmission is only set up in the hardware and then the function returns immediately (that is before the packet is actually sent). If there are no resources available in the non- blocking mode the call will return with an error.

There is also a feature called Tx_block_on_full which means that the write call blocks when all descriptors are in use.

The ioctl call used for transmissions is SPACEWIRE_IOCTRL_SEND. A spw_ioctl_send struct is used as argument and contains length, and pointer fields. The structure is shown in the data structures section. This ioctl call should be used when a header is taken from one buffer and datafrom another. The header part is always transmitted first. The hlen field sets the number of header bytes to be transmitted from the hdr pointer. The dlen field sets the number of data bytes to be transmitted from the data pointer. Afterwards the sent field contains the total number (header + data) of bytes transmitted.

The blocking behavior is the same as for write calls. The call fails if hlen+dlen is 0, one of the buffer pointer is zero and its corresponding length variable is nonzero.

Table 14.12. ERRNO values for write and ioctl send.

ERRNO	Description
EINVAL	An invalid argument was passed or link is not started. The buffers must not be null pointers and the length parameters must be larger that zero and less than the maximum allowed size.
EBUSY	The packet could not be transmitted because all descriptors are in use (only in non-blocking mode).

14.2.7. Reception

Reception is done using the read call. An example is shown below:

```
len = read(fd, rx_pkt, tmp);
```

The requested number of bytes to be read is given in tmp. The packet will be stored in rx_pkt. The actual number of received bytes is returned by the function on success and -1 on failure. In the latter case errno is also set.

The call will fail if a null pointer is passed.

The blocking behaviour can be set using ioctl calls. In blocking mode the call will block until a packet has been received. In non-blocking mode, the call will return immediately and if no packet was available -1 is returned and errno set appropriately. The table below shows the different errno values that can be returned.

Table 14.13. ERRNO values for read calls.

ERRNO	Description
EINVAL	A NULL pointer was passed as the data pointer, the length was illegal or the link hasn't been started with the ioctl command START.
EBUSY	No data could be received (no packets available) in non-blocking mode.

14.3. Receiver example

```
#include <grspw.h>

/* Open device */
fd = open("/dev/grspw0",O_RDWR);
if ( fd < 0 ) {
    printf("Error Opening /dev/grspw0, errno: %d\n",errno);
    return -1;
}

/* Set basic parameters */
if ( ioctl(fd, SPACEWIRE_IOCTRL_SET_COREFREQ,0) == -1 )
    printf("SPACEWIRE_IOCTRL_SET_COREFREQ, errno: %d\n",errno);

/* Make sure link is up */
while( ioctl(fd, SPACEWIRE_IOCTRL_START,0) == -1 ) {
    sched_yield();
}
/* link is up => continue */

/* Set parameters */
...

/* Set blocking receiving mode */
if ( ioctl(fd, SPACEWIRE_IOCTRL_SET_RXBLOCK,1) == -1 )
    printf("SPACEWIRE_IOCTRL_SET_RXBLOCK, errno: %d\n",errno);

/* Read/Write */
while(1) {
    unsigned char buf[256];
    if ( read(fd,buf,256) < 0 ) {
        printf("Error during read, errno: %d\n",errno);
        continue;
    }
    /* Handle incoming packet */
    ...
}
```

15. SpaceWire router

15.1. Introduction

This document describes how to use the Aeroflex Gaisler SpaceWire router device in RTEMS. The router does not have to be located on the same bus as the processor running RTEMS. The RTEMS driver manager abstracts the actual location of the device allowing application software to access the router resources always using the same API. Two different drivers, the SpaceWire router register driver and GRSPW driver, are needed to utilize the complete functionality of the router.

For details about each driver see their respective sections.

There is one example application available called rtems-spw-router-pci which can be used as a reference on how the router is used. In that particular case the router has 18 ports and a PCI interface through which it is connected to the host system. The host system can consist of either a LEON2, LEON3 or LEON4 running RTEMS and one of three PCI interfaces: PCIF, GRPCI or GRPCI2.

15.1.1. SpaceWire Router register driver

The main functionality of the router is to transfer packets between the SpaceWire ports. This ability is functional after reset without any configuration. To change the configuration, enable/disable links, collect statistics, fault detection etc the router configuration port has to be accessed. This is done through the SpaceWire router register driver.

The driver manager finds the configuration port interfaces automatically when the system is scanned. If the user application needs to use the configuration port it has to open a file handle to it. All the available router features can then be accessed using IOCTL calls through this file handle.

15.1.2. AMBA port driver

There are three different port types in the router: SpW ports, FIFO ports and AMBA ports. The data path of SpW and FIFO ports are not (directly) accessible from the processor. If the router has AMBA ports they can be used for transferring packets. The AMBA ports have identical interfaces to the GRSPW core so they use the same driver. To transfer packets through an AMBA port a file handle should be opened to it and then read and write calls can be used to receive and send packets. The driver also allows configuration and status options in the AMBA port to be accessed.

16. SpaceWire router register driver

16.1. Introduction

This document is intended as an aid in getting started developing with Aeroflex Gaisler SpaceWire Router Register driver for RTEMS. The driver provides applications with a SpaceWire Router configuration interface. The driver allows the user to configure the router and control the SpaceWire links through the AMBA AHB Registers.

The SpaceWire Router driver require the RTEMS IO Manager and Driver Manager.

See the SpaceWire Router Core User's Manual for hardware details.

16.2. User interface

The RTEMS SpaceWire Router driver supports the standard accesses to file descriptors [open], **ioctl** and [close]. User applications should include the router driver's header file which contains definitions of all necessary data structures used when accessing the driver.

16.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when Router hardware is found for the first time. The driver is called from the driver manager to handle detected Router devices. In order for the driver manager to unite the Router driver with the Router devices one must register the driver to the driver manager. This process is described in the driver manager chapter.

16.2.2. Driver resource configuration

This driver has no resource configuration options, it is configured using the **ioctl** interface.

16.2.3. Opening the device

Opening the device enables the user to access the hardware of a certain SpaceWire Router device. The same driver is used for all Router devices available. The devices are separated by assigning each device a unique name, the name is passed during the opening of the driver. Some example device names are printed out below.

Table 16.1. Device number to device name conversion

Device number	Filesystem name	Location
0	/dev/router0	On-Chip Bus
1	/dev/router1	On-Chip Bus
2	/dev/router2	On-Chip Bus
System dependent	/dev/spwrouter0/router0	GR-RASTA-SPW_ROUTER[0]
System dependent	/dev/spwrouter1/router0	GR-RASTA-SPW_ROUTER[1]

An example of an RTEMS [open] call is shown below.

```
fd = open("/dev/router0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 16.1.

Table 16.2. Open ERRNO values

ERRNO	Description
EINVAL	Illegal device name or not available
EBUSY	Device already opened

16.2.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the SpaceWire driver.

16.2.5. I/O Control interface

The APB register insterface of the Router can be accessed via the standard system call **ioctl**. The first argument is an integer which selects ioctl function and the second a pointer to data that may be interpreted uniquely for each function. A typical ioctl call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global errno variable is set accordingly.

All supported commands and their data structures are defined in the SpaceWire Router driver's header file `grspw_router.h`. In functions where only one argument in needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

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

- Input: Argument is an address. The driver reads data from the given address.
- Output: Argument is an address. The driver writes data to the given address.
- Input/Output: both above cases.
- Argument: 32-bit simple Argument, no data transferred between kernel/user.
- None: Argument ignored.

Table 16.3. ioctl calls supported by the SpaceWire Router driver.

ERRNO	**Description**		
HWINFO	Output	`struct router_hw_info *`	Copy hardware configuration of the router core, such as number of SpaceWire ports, number DMA port, number of FIFO port, etc.
CFG_SET	Input	`struct router_config *`	Configure the router by writing the configuration bit of the Control/Status register, setting the Instance ID, Start up Clock Divisor, Timer prescaler and the timer reload registers.
CFG_GET	Output	`struct router_config *`	Reads the current router configuration into the user specified memory area.
ROUTES_SET	Input	`struct router_routes *`	Configure the 224 words long router table.
ROUTES_GET	Output	`struct router_routes *`	Copy the current 224 words long router table to user provided buffer.
PS_SET	input	`struct router_ps *`	Configure the port setup registers according to user buffer.
PS_GET	Output	`struct router_ps *`	Copy the current port setup registers to user buffer.
WE_SET	Argument	[int]	If the argument's bit zero is one then the WE bit in the configuration write enable register is set, otherwise it is cleared. This enabled the user to write protect the current configuration.
PORT	Intput/Out-put	`struct router_port *`	Write and/or Read (in that order) the port control and port status registers of one port of the SpaceWire router. The flag field determines which operations should be performed. See ROUTER_PORTFLG_*. The port field selects which port is to be written/read.

ERRNO	Description		
CFGSTS_SET	Argument	`unsigned int *`	Writes the Config/Status register.
CFGSTS_GET	Output	`unsigned int *`	Copies the current value of the Config/Status register to the user provided buffer.
TC_GET	Output	`unsigned int *`	Copies the current value of the Time-code register to the user provided buffer.

Note that no detailed descriptions of the hardware register fields are given here. They are found in the GRSP-WROUTER IP core user manual or the Router FPGA/ASIC user manual. The information is not duplicated here to avoid potential inconsistencies. The driver provides no real intelligence with respect to the router functionality. It merely abstracts the access to the router resources so that the user does not need to know on which bus /network the router is located or the address and register bit locations in the router.

16.2.5.1. HWINFO

```
struct router_hw_info {
     unsigned char nports_spw;
     unsigned char nports_amba;
     unsigned char nports_fifo;
     char timers_avail;
     char pnp_avail;
     unsigned char ver_major;
     unsigned char ver_minor;
     unsigned char ver_patch;
     unsigned char iid;
};
```

Table 16.4. router_hw_info member descriptions.

Member	Description
nports_spw	Number of SpaceWire ports in the router
nports_amba	Number of AMBA ports in the router
nports_fifo	Number of FIFO ports in the router
timers_avail	1 if the router has timers 0 otherwise
pnp_avail	1 if the router has support for SpaceWire Plug and Play 0 otherwise
ver_major	Major version number of the router
ver_minor	Mino version number of the router
ver_patch	Patch number of the router
iid	Instance ID of the router

The router hardware information struct contains fields for static hardware parameters. This call reads the actual value for each field from hardware and stores them in the struct.

16.2.5.2. CFG_SET

```
struct router_config {
    unsigned int flags;
    unsigned int config;
    unsigned char iid;
    unsigned char idiv;
    unsigned int timer_prescaler;
    unsigned int timer_reload[32];
};
```

Table 16.5. router_config member descriptions.

Member	Description
flags	Flags that determine which of the configuration parameters should be written

Member	Description
config	Value written to configuration and status register. Bit 4 is always masked to 0
iid	Value written to instance ID field
idiv	Value written to instance ID field
timer_prescaler	Value written to timer prescaler field
timer_reload[32]	Value written to timer reload field of the port corresponding with the number corresponding to the field index

Table 16.6. router_config flag descriptions.

Flag	Description
ROUTER_FLG_IID	Write iid field
ROUTER_FLG_IDIV	Write idiv field
ROUTER_FLG_CFG	Write config field
ROUTER_FLG_TRPES	Write timer_prescaler field
ROUTER_FLG_TRLD	Write all timer_reload fields (one per port)

This call writes various configuration parameters in the router. A set of flags determine which of the parameters are written in each call. The flags should be set in the flags field of the struct.

For example setting “flags = ROUTER_FLG_IID“ will cause the instance ID field in the router to be written with the value from iid in the struct.

The flags can be or:ed so “flags = ROUTER_FLG_IID | ROUTER_FLG_CFG” will cause both the instance ID field and the config register to be written with iid and config respectively.

16.2.5.3. CFG_GET

Reads all the parameters in the router_config struct from hardware registers and stores them in the struct. The flags field is unused and all the registers are read in each call.

16.2.5.4. ROUTES_SET

```
struct router_routes {
     unsigned int route[224];
};
```

Table 16.7. router_routes member descriptions.

Member	Description
route[224]	Routing table entry for all logical addresses. Route[0] corresponds to address 32, route[1] to 33 etc.

This call sets up the complete routing table with the values in the router_routes struct. The value from each entry is written directly to the corresponding routing table location.

16.2.5.5. ROUTES_GET

Reads the complete routing table and stores the values in the router_routes struct. Single entries cannot be read on their own.

16.2.5.6. RS_SET

```
struct router_ps {
    unsigned int ps[31];
    unsigned int ps_logical[224];
};
```

Table 16.8. router_ps member descriptions.

Member	Description
ps[31]	Port setup physical addresses
ps_logical[224]	Port setup logical addresses

The port setup determines which ports a packet with a certain destination address should be transmitted on. One or more ports can be specified for a single address. Physical addresses should correspond to the port with the same number but the standard does allow group adaptive routing or packet distribution as well. Normally they should only be used with logical addresses though. The ps fields correspond to physical addresses 1 to 31 and ps_logical corresponds to logical addresses 32-255. Single addresses cannot be written individually.

Note that a port setup field corresponding to a logical address should not be nonzero if the routing table entry for the same address is not enabled or vice versa.

16.2.5.7. PS_GET

Reads the port setup entries for all physical and logical address and stores them in a router_ps struct. Single addresses cannot be read individually.

16.2.5.8. WE_SET

This call takes an integer value which should be either 0 or 1 and writes it to the configuration write enable bit (WE).

16.2.5.9. PORT

```
struct router_port {
    unsigned int flag;
    int port;
    unsigned int ctrl;
    unsigned int sts;
};
```

Table 16.9. router_port member descriptions.

Member	Description
flag	Flags that select the operation(s) that should be performed
port	Selects the port number that the operation(s) are performed on
ctrl	Value to be read/written from/to port control register
sts	Value to be read/written from/to port status register

Table 16.10. router_port flag descriptions.

Flag	Description
ROUTER_PORTFLG_GET_CTRL	Reads the port control register and stores the value in the ctrl field.
ROUTER_PORTFLG_GET_STS	Reads the port status register and stores the value in the sts field.
ROUTER_PORTFLG_SET_CTRL	Writes the port control register and stores the value in the ctrl field. If both a read and a write have been enabled the read is performed first and this value is returned in the ctrl field. After the read is finished the write is performed with the original value in the ctrl field.
ROUTER_PORTFLG_SET_STS	Writes the port status register and stores the value in the sts field. If both a read and a write have been enabled the read is performed first and this value is returned in the sts field. After the read is finished the write is performed with the original value in the sts field.

16.2.5.10. CFGSTS_SET

Takes an integer as input argument and writes it to the router configuration and status register.

16.2.5.11. CFGSTS_GET

Reads the router configuration and status register and stores the value in the argument which should be an integer.

16.2.5.12. TC_GET

Reads the time-code register and returns the value in the argument which should be an integer.

17. GR1553B GRLIB MIL-STD-1553B driver

17.1. Introduction

This document describes the RTEMS 4.10 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.

All the GR1553B drivers relies on the RTEMS Driver Manager for the services: device detection, driver loading/initialization and interrupt management. Driver Manager is responsible for creating GR1553B device instances and uniting GR1553B devices with the GR1553B low-level driver.

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.

17.2. 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.

17.3. 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 RTEMS SPARC BSP source tree `c/src/lib/libbsp/sparc`.

Table 17.1. Source Location

Filename	Description
shared/1553/gr1553b.c	GR1553B Driver source
share/include/gr1553b.h	GR1553B Driver interface declaration

17.4. Driver Registration

The driver must be registered to the driver manager. The registration is performed by calling the `gr1553_register()` function. The driver is automatically registered from the BC, BM and the RT device drivers registration procedure. This means that including the BC, BM and/or the RT driver will automatically include the GR1553B (this) driver.

17.5. Examples

The RCC distribution contains examples demonstrating how the GR1553B drivers are used. The examples can be found in `/opt/rtems-4.10/src/samples/1553`.

18. GR1553B remote terminal driver

18.1. Introduction

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

18.1.1. 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.

18.1.2. Examples

There is an example available that illustrates how the RT driver interface can be used to respond to 1553 BC commands. The example includes code for Interrupt handling, Event Logging, Synchronize and Synchronize With Data mode codes, RX/TX data transfers and various driver configuration options. The RT will respond on sub address 1,2 and 3, see comments in application and BC application. The RT example comes with a matching BC example that generates BC transfers that is understood by the RT. The RT application use the Eventlog to monitor certain transfers, the transfers are written to standard out.

The RT example includes a BM logger which can be used for debugging the 1553 bus. All 1553 transfers can be logged and sent to a Linux PC over a TCP/IP socket and saved to a raw text file for post processing. The default is however just to enable BM logging, for debugging one can quite easily read the raw BM log by looking at the BM registers and memory from GRMON.

In order to run all parts of the example a board with GR1553B core with BC and BM support, and a board with a GR1553B core with RT support is required.

The example is part of the Aeroflex Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/1553/rtems-gr1553rt.c`.

The example can be built by running:

```
$ cd /opt/rtems-4.10/src/samples/1553
$ make rtems-gr1553rt
```

18.2. User Interface

18.2.1. 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 RTEMS source tree `c/src/lib/lbbsp/sparc`.

Table 18.1. RT driver Source location

Filename	Description
shared/1553/gr1553rt.c	GR1553B RT Driver source
shared/include/gr1553rt.h	GR1553B RT Driver interface declaration

18.2.1.1. 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()`, 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()`.

18.2.1.2. 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()` function.

18.2.1.3. 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 lookup 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 18.2.1.8.

18.2.1.4. 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 descriptor's data buffer is one element in the circular buffer.

18.2.1.5. 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.

18.2.1.6. 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.

18.2.1.7. 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.

18.2.1.8. 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.

18.2.1.9. 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()` 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 18.2.1.7.

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()`.

18.2.1.10. 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()` function. The current time is read by calling the `gr1553rt_status()` function.

18.2.2. Application Programming Interface

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

Table 18.2. Data structures

Protoype	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(`
`  void *rt,`
`  struct gr1553rt_cfg *cfg)`
	Configure the RT device driver and allocate 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

`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 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,`
	Initialize a Descriptor in a List identified by number.

Protoype	Description
`unsigned short next)`	
`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.

18.2.2.1. 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;
};
```

Table 18.3. gr1553rt_cfg member descriptions

Member	Description
rtaddress	RT Address on 1553 bus [0..30]
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 (1kB) 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, length must be a power 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.

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

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

Table 18.4. 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 18.5. 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.

18.2.2.2. 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()**.

18.2.2.3. gr1553rt_close

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

18.2.2.4. gr1553rt_config

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 18.3.

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

18.2.2.5. 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.

18.2.2.6. gr1553rt_stop

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

18.2.2.7. 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 18.5.

18.2.2.8. 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).

18.2.2.9. 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.

18.2.2.10. 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.

18.2.2.11. 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)
```

18.2.2.12. 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.

18.2.2.13. 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).

18.2.2.14. 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.

18.2.2.15. 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.

18.2.2.16. 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()**. 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.

18.2.2.17. 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.

18.2.2.18. gr1553rt_list_init

Allocate and configure a list structure according to configuration given in *cfg*, see the `gr1553rt_list_cfg` data structure in Table 18.4. 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()` before calling this function.

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

18.2.2.19. 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 descriptor's Control/Status Word. The argument *dptr* is written to the descriptor's 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 18.6. 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.

18.2.2.20. 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.

19. GR1553B bus monitor driver

19.1. Introduction

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

19.1.1. 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.

19.1.2. Examples

There is an example available that illustrates how the BM driver interface can be used to log transfers seen on the 1553 bus. All 1553 transfers is be logged, by configuring the `config_bm.h` file the logger application can "compress" the log and send it to a Linux PC over a TCP/IP socket. The Linux application save the log to a raw text file for post processing.

The default BM example behaviour is however just to enable BM logging, for debugging one can quite easily read the raw BM log by looking at the BM registers and memory from GRMON.

The BM logger application can be run separately or together with the BC or RT examples.

In order to run all parts of the example a board with GR1553B core with BC and BM support, and a board with a GR1553B core with RT support is required.

The example is part of the Aeroflex Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/1553` named `rtems-gr1553bm.c`, `rtems-gr1553bcbm.c` or `rtems-gr1553rtbm.c`.

The example can be built by running:

```
$ cd /opt/rtems-4.10/src/samples/1553
$ make rtems-gr1553bm
```

19.2. User Interface

19.2.1. 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 interface definitions are listed in the table below, the path is given relative to the RTEMS source tree `c/src/lib/libbsp/sparc`.

Table 19.1. BM driver Source location

Filename	Description
shared/1553/gr1553bm.c	GR1553B BM Driver source
shared/include/gr1553bm.h	GR1553B BM Driver interface declaration

19.2.1.1. 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()` 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()`.

19.2.1.2. 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.

19.2.1.3. 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.

19.2.1.4. Time

Th BM core has a 24-bit time counter with a programmable resolution through the `gr1553bm_config()` 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()` 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.

19.2.1.5. 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()`.

19.2.1.6. 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()` function. On DMA error the BM logging will automatically be stopped by a call to `gr1553bm_stop()` from within the ISR of the driver.

19.2.2. Application Programming Interface

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

Table 19.2. 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(`
`  void *bm,`
`  struct gr1553bm_cfg *cfg)`
	Configure the BM device driver and allocate BM log DMA-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*.

19.2.2.1. 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 19.3. 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.

Member	Description
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.
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 19.4. 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

19.2.2.2. 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.

19.2.2.3. gr1553bm_close

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

19.2.2.4. gr1553bm_config

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 19.3.

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

19.2.2.5. 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.

19.2.2.6. gr1553bm_stop

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

19.2.2.7. 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.

19.2.2.8. 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.

19.2.2.9. 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.

20. GR1553B bus controller driver

20.1. Introduction

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

20.1.1. 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 17.

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

20.1.2. 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 RTEMS SPARC BSP source tree `c/src/lib/libbsp/sparc`.

Table 20.1. BC driver Source location

Filename	Description
shared/1553/gr1553bc.c	GR1553B BC Driver source
shared/1553/gr1553bc_list.c	GR1553B BC List handling source
shared/include/gr1553bc.h	GR1553B BC Driver interface declaration
shared/include/gr1553bc_list.h	GR1553B BC List handling interface declaration

20.1.3. Examples

There is an example available that illustrates how the BC driver interface can be used to communicate with one or more RTs. The descriptor list includes both transfer and conditional descriptors, time slot allocation, interrupt demonstration, read BC hardware currently executing descriptor by the indication service. The BC example does not require an RT to respond on the 1553 transfers, however it will be stuck in initialization mode of the 1553 bus. The BC example comes with a matching RT example that responds to the BC transfers.

The BC example includes a BM logger which can be used for debugging the 1553 bus. All 1553 transfers can be logged and sent to a Linux PC over a TCP/IP socket and saved to a raw text file for post processing.

In order to run all parts of the example a board with GR1553B core with BC and BM support, and a board with a GR1553B core with RT support is required.

The example is part of the Aeroflex Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/1553/rtems-gr1553bcbm.c`.

The example can be built by running:

```
$ cd /opt/rtems-4.10/src/samples/1553
$ make rtems-gr1553bcbm
```

20.2. 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`.

20.2.1. Device API

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

Table 20.2. Device API function prototyper

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.

20.2.1.1. 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 20.3. gr1553bc_status member descriptions

Member	Description
status	BC status register
time	BC Timer register

20.2.1.2. 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.

20.2.1.3. 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.

20.2.1.4. 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.

20.2.1.5. gr1553bc_pause

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

20.2.1.6. gr1553bc_resume

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

20.2.1.7. 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 20.4. 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.

20.2.1.8. 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.

20.2.1.9. 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 20.3.

20.2.1.10. 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.

20.2.1.11. 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.

20.3. 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.

20.3.1. 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.

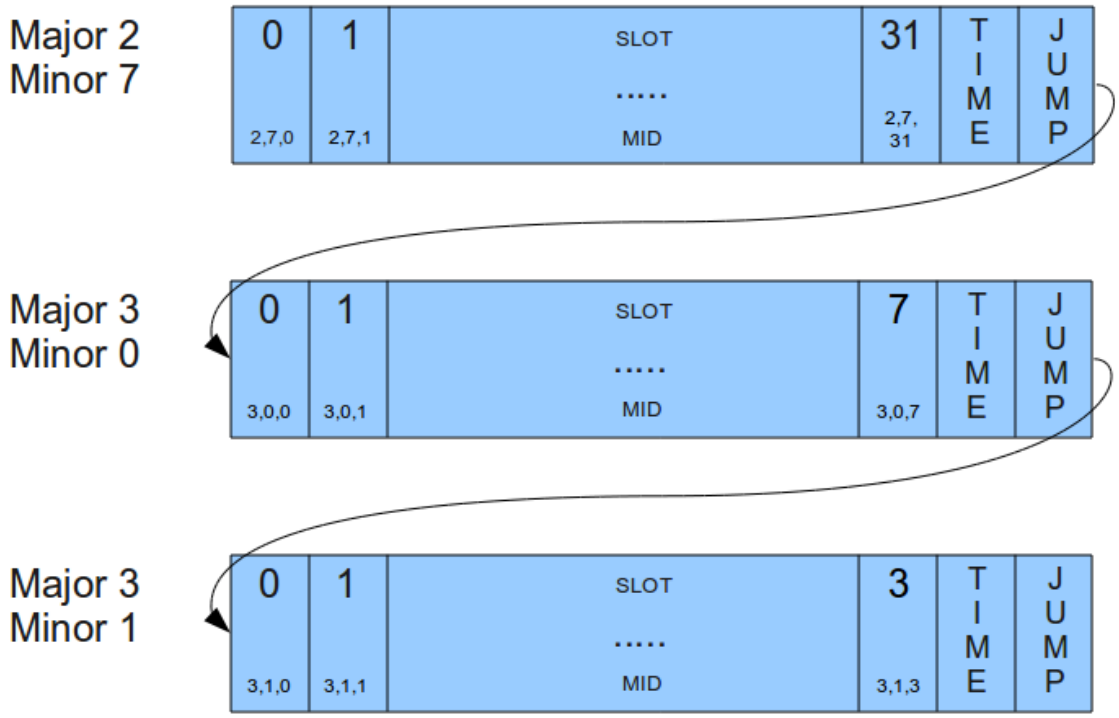


Figure 20.1. Three consecutive Minor Frames

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 20.5. 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)

20.3.2. 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 pre defined before starting:
  1. gr1553bc_slot_alloc(list, &MID, TIME_REQUIRED, ..)
  2. gr1553bc_slot_transfer(MID, ..)
- START BC HARDWARE BY SCHDULING ABOVE LIST
- Application operate on executing List

20.3.3. 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.

20.3.4. 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 20.2. 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).

20.3.5. 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 20.6. 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 20.7. 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

20.3.6. 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.

20.3.7. 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.

20.3.8. 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 lookup 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);
}
```

20.3.9. List API

Table 20.8. List API function prototypes

Prototype	Description

`int gr1553bc_list_alloc(`
`  struct gr1553bc_list **list,`
`  int max_major)`
	Allocate 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.

Prototype	Description

`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.

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

Prototype	Description

`unsigned int flags,`
`uint32_t word0,`
`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.

20.3.9.1. 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 20.9. gr1553bc_minor_cfg member descriptions.

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

Table 20.10. 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 20.11. 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

20.3.9.2. gr1553bc_list_alloc

Dynamically allocates 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.

20.3.9.3. gr1553bc_list_free

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

20.3.9.4. 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 20.11.

20.3.9.5. 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.

20.3.9.6. 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.

20.3.9.7. 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.

20.3.9.8. 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.

20.3.9.9. 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.

20.3.9.10. gr1553bc_list_table_free

Free previously allocated descriptor table memory.

20.3.9.11. 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.

20.3.9.12. gr1553bc_major_alloc_skel

Allocate 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 20.10.

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

20.3.9.13. 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.

20.3.9.14. 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.

20.3.9.15. gr1553bc_slot_free

Return Slot and timeslot allocated from the Minor Frame.

20.3.9.16. 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.

20.3.9.17. gr1553bc_slot_bd

Looks up descriptor address from MID.

20.3.9.18. 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.

20.3.9.19. 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.

20.3.9.20. 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.

20.3.9.21. gr1553bc_slot_jump

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

Table 20.12. gr1553bc_list_cfg member descriptions.

Argument	Description
list	List that the Slot is located at.
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.

20.3.9.22. 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.

20.3.9.23. 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 20.13. 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.

20.3.9.24. 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 descriptor's dummy bit is written.

Returns zero on success.

20.3.9.25. gr1553bc_slot_empty

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

Returns zero on success.

20.3.9.26. gr1553bc_slot_update

This function will update a transfer descriptor's 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.

20.3.9.27. 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 20.14. 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.

20.3.9.28. 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.

21. B1553BRM GRLIB Actel Core1553BRM driver

21.1. Introduction

This document is intended as an aid in getting started developing with GRLIB B1553BRM core using the driver described in this document. It briefly takes the reader through some of the most important steps in using the driver such as setting up a connection, configuring the driver, reading and writing messages between Bus Controllers (BC), Remote Terminals (RT) and Bus Monitors (BM). The reader is assumed to be well acquainted with MIL-STD-1553 and RTEMS.

The B1553BRM driver require the RTEMS Driver Manager.

21.1.1. BRM Hardware

The BRM hardware can operate in one of three modes, Bus Controller (BC), Remote Terminal (RT) or Bus Monitor (BM). All three modes are supported by the driver. The software interface of the BRM-RT is similar to the B1553RT software interface to simplify software development.

The B1553BRM core is present in GR712RC. In many newer systems the GR1553B IP core replaces the B1553BRM which has a different software interface documented in the Chapter 17.

21.1.2. Software Driver

The driver provides means for processes and threads to send, receive and monitor messages.

- Bus Controller
- Remote Terminal
- Bus monitor

21.1.3. Supported OS

There is a simple example available it illustrates how to set up a connection between a BC and a RT monitored by a BM. The BC sends the RT receive and transmit messages for a number of different sub addresses. The BM is set up to print messages from the BC and the RT. To be able to run the example one must have at least two boards connected together via the B1553BRM interfaces. To fully run the example three BRM boards is needed.

The example is part of the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/rtems-brm.c`, `brm_lib.c` and `brm_lib.h`.

The example can be built by running:

```
cd /opt/rtems-4.10/src/samples make clean rtems-brm_rt rtems-brm_bc rtems-brm_bm
```

21.2. User Intrerface

The RTEMS MIL-STD-1553B BRM driver supports standard accesses to file descriptors such as **read**, **write** and **ioctl**. User applications include the [brm] driver's header file which contains definitions of all necessary data structures and bit masks used when accessing the driver. An example application using the driver is provided in the examples directory.

The driver for the MIL-STD-1553 B BRM has three different operating modes, Remote Terminal, Bus Controller or Bus Monitor. It defaults to Remote Terminal (RT) with address 1, MIL- STD-1553 B standard, both buses enabled, and broadcasts enabled. The operating mode and settings can be changed with **ioctl** calls as described later.

21.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when B1553BRM hardware is found for the first time. The driver is called from the driver manager to handle detected

B1553BRM hardware. In order for the driver manager to unite the B1553BRM driver with the B1553BRM hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

21.2.2. Driver resource configuration

The driver can be configured using driver resources as described in the driver manager chapter. Below is a description of configurable driver parameters. The driver parameters is unique per B1553BRM device. The parameters are all optional, the parameters only overrides the default values.

Table 21.1. B1553BRM driver parameter description

Name	Type	Parameter description
clkSel	INT	Selects clock source (input value to the clock MUX)
clkDiv	INT	Selects clock prescaler, may not be available for all clock sources
coreFreq	INT	The input clock frequency to the BRM core. 0 = 12MHz, 1 = 16MHz, 2= 20MHz, 3 = 24MHz.
dmaArea	INT	Custom DMA area address. See note below.

21.2.2.1. Custom DMA area parameter

The DMA area can be configured to be located at a custom address. The standard configuration is to leave it up to the driver to do dynamic allocation of the areas. However in some cases it may be required to locate the DMA area on a custom location, the driver will not allocate memory but will assume that enough memory is available and that the alignment needs of the core on the address given is fulfilled. The memory required is either 16K or 128K bytes depending on how the driver has been compiled.

For some systems it may be convenient to give the addresses as seen by the B1553BRM core. This can be done by setting the LSB bit in the address to one. For example a GR-RASTA-IO board with a B1553BRM core doesn't read from the same address as the CPU in order to access the same data. This is dependent on the PCI mappings. Translation between CPU and B1553BRM addresses must be done. The B1553BRM driver automatically translates the DMA base address. This requires the bus driver, in this case the GR- RASTA-IO driver, to set up translation addresses correctly.

21.2.3. Opening the device

Opening the device enables the user to access the hardware of a certain BRM device. The driver is used for all BRM devices available. The devices is separated by assigning each device a unique name and a number called [minor]. The name is passed during the opening of the driver. Some example device names are printed out below.

Table 21.2. Device number to device name conversion

Device number	Filesystem name	Location
0	/dev/b1553brm0	On-Chip Bus
1	/dev/b1553brm1	On-Chip Bus
2	/dev/b1553brm2	On-Chip Bus
Depends on system configuration	/dev/rastaio0/b1553brm0	GR-RASTA-IO

An example of an RTEMS open call is shown below.

```
fd = open("/dev/b1553brm0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in table Table 21.3.

Table 21.3. Open errno values

Errno	Description
ENODEV	Illegal device name or not available

Errno	Description
EBUSY	Device already opened

21.2.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the [brm] driver.

21.2.5. I/O Control interface

Changing the behaviour of the driver for a device is done via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global [errno] variable is set accordingly.

All supported commands and their data structures are defined in the BRM driver's header file [brm.h]. In functions where only one argument is needed the pointer (...,void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

21.2.5.1. Data structures

21.2.5.1.1. Remote Terminal operating mode

The structure below is used for RT operating mode for all received events as well as to put data in the transmit buffer.

```
struct rt_msg {
    unsigned short miw;
    unsigned short time;
    unsigned short data[32];
    unsigned short desc;
};
```

Table 21.4. rt_msg member descriptions.

<table>
<tr><th>Member</th><th>Description</th></tr>
<tr><td>miw</td><td>
<table>
<tr><th>Bit(s)</th><th>Description</th></tr>
<tr><td>15-11</td><td>Word count / mode code - For subaddresses this is the number of received words. For mode codes it is the receive/transmit mode code.</td></tr>
<tr><td>10</td><td>-</td></tr>
<tr><td>9</td><td>A/B - 1 if message receive on bus A, 0 if received on bus B.</td></tr>
<tr><td>8</td><td>RTRT - 1 if message is part of an RT to RT transfer</td></tr>
<tr><td>7</td><td>ME - 1 if an error was encountered during message processing. Bit 4-0 gives the details of the error.</td></tr>
<tr><td>6-5</td><td>-</td></tr>
<tr><td>4</td><td>ILL - 1 if received command is illegalized.</td></tr>
<tr><td>3</td><td>TO - If set, the number of received words was less than the amount specified by the word count.</td></tr>
<tr><td>2</td><td>OVR - If set, the number of received words was more than amount specified by the word count.</td></tr>
<tr><td>1</td><td>PRTY - 1 if the RT detected a parity error in the received data.</td></tr>
</table>
</td></tr>
</table>

Member	Description
	Bit(s) / **Description** 0 / MAN - 1 if a Manchester decoding error was detected during data reception.
time	Time Tag - Contains the value of the internal timer register when the message was received.
data	An array of 32 16 bit words. The word count specifies how many data words that are valid. For receive mode codes with data the first data word is valid.
desc	Bit 6-0 is the descriptor used.

The last variable in the [struct rt_msg] shows which descriptor (i.e rx subaddress, tx subaddress, rx mode code or tx mode code) that the message was for. They are defined as shown in the table below:

Table 21.5. Descriptor table

Descriptor	Description
0	Reserved for RX mode codes
1-30	Receive subaddress 1-30
31	Reserved for RX mode codes
32	Reserved for TX mode codes
33-62	Transmit subaddress 1-30
63	Reserved for TX mode codes
64-95	Receive mode code
96-127	Transmit mode code

If there has occurred an event queue overrun bit 15 of this variable will be set in the first event read out. All events received when the queue is full are lost.

21.2.5.1.2. Bus Controller operating mode

When operating as BC the command list that the BC is to process is described in an array of BC messages as defined by the struct [bc_msg].

```
struct bc_msg {
    unsigned char rtaddr[2];
    unsigned char subaddr[2];
    unsigned short wc;
    unsigned short ctrl;
    unsigned short tsw[2];
    unsigned short data[32];
};
```

Table 21.6. struct bc_msg member description

Member	Description
rtaddr	Remote terminal address - For non RT to RT message only rtaddr[0] is used. It specifies the address of the remote terminal to which the message should be sent. For RT to RT messages rtaddr[0] specifies the receive address and rtaddr[1] the transmit address.
subaddr	The subaddr array works in the same manner as rtaddr but for the subaddresses.
wc	Word Count - Specifies the word count, or mode code if subaddress is 0 or 31.
ctrl	**Bit(s)** / **Description** 15 / Message Error. Set by BRM while traversing list if protocol error is detected.

<table>
<tr><th>Member</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Bit(s)</th><th>Description</th></tr>
<tr><td>14-6</td><td>-</td></tr>
<tr><td>5</td><td>END. Indicates end of list</td></tr>
<tr><td>4-3</td><td>Retry, Number of retries, 0=4, 1=1, 2=2, 3=3. BC will alternate buses during retries.</td></tr>
<tr><td>2</td><td>AB, 1 – Bus B, 0 - Bus A</td></tr>
<tr><td>1</td><td>1 RT to RT, 0 normal</td></tr>
<tr><td>0</td><td>0 RT Transmit, 1 RT receive (ignored for RT to RT)</td></tr>
</table>
</td></tr>
<tr><td>tsw</td><td>Status words</td></tr>
<tr><td>data</td><td>Data in message, not used for RT receive (ctrl.0 = 1).</td></tr>
</table>

21.2.5.1.3. Bus Monitor operationg mode

The structure below is used for BM operating mode for all received events as well as to put data in the transmit buffer.

```
struct bm_msg {
    unsigned short miw;
    unsigned short cw1;
    unsigned short cw2;
    unsigned short sw1;
    unsigned short sw2;
    unsigned short time;
    unsigned short data[32];
};
```

Table 21.7. struct bm_msg member description

<table>
<tr><th>Member</th><th>Description</th></tr>
<tr><td>miq</td><td>
<table>
<tr><th>Bit(s)</th><th>Description</th></tr>
<tr><td>15</td><td>Overrun- Indicates that the monitor message queue has been overrun.</td></tr>
<tr><td>14-10</td><td>-</td></tr>
<tr><td>9</td><td>Channel A/B -1 if message captured on bus A, 0 if captured on bus B.</td></tr>
<tr><td>8</td><td>RT to RT transfer - 1 if message is part of an RT to RT transfer</td></tr>
<tr><td>7</td><td>Message Error - 1 if an error was encountered during message processing. Bit 4-0 gives the details of the error.</td></tr>
<tr><td>6</td><td>Mode code without data - 1 if a mode code without data word was captured.</td></tr>
<tr><td>5</td><td>Broadcast - 1 if a broadcast message was captured.</td></tr>
<tr><td>4</td><td>-</td></tr>
<tr><td>3</td><td>Time out - If set, the number of captured data words was less than the amount specified by the word count.</td></tr>
<tr><td>2</td><td>Overrun -If set, the number of captured data words was more than amount specified by the word count.</td></tr>
<tr><td>1</td><td>Parity- 1 if the BM detected a parity error in the received data.</td></tr>
<tr><td>0</td><td>Manchester error - 1 if a Manchester decoding error was detected during data reception.</td></tr>
</table>
</td></tr>
<tr><td>cw1</td><td>1553 Command word 1</td></tr>
<tr><td>cw2</td><td>1553 Command word 2, only used for RT to RT transfers and then holds the transmit command.</td></tr>
</table>

Member	Description
sw1	1553 Status word 1
sw2	1553 Status word 2, is only used for RT to RT transfers and then holds the status word from the transmitting RT.
time	Time tag (time) Contains the value of the internal timer register when the message was captured.
data	An array of 32 16 bit words. The command word specifies how many data words that are valid. For receive mode codes with data the first data word is valid.

21.2.6. Configuration

The BRM core and driver are configured using **ioctl** calls. The Table 21.9 below lists all supported **ioctl** calls. BRM_ should be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 21.8.

An example is shown below where the operating mode is set to Bus Controller (BC) by using an **ioctl** call:

```
unsigned int mode = BRM_MODE_BC;
result = ioctl(fd, BRM_SET_MODE, &mode);
```

*Table 21.8. ERRNO values for **ioctl** calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The BRM hardware is not in the correct state to accept this command. Errno is set to EBUSY when issuing a BRM_DO_LIST before the last BRM_DO_LIST command has finished its execution.
ENOMEM	Not enough memory for driver to complete request.

*Table 21.9. **ioctl**calls supported by the BRM driver.*

Call Number	Description	ERRNO
SET_MODE	Set operating mode (0=BC, 1=RT, 2=BM)	EINVAL, ENOMEM
SET_BUS	Enable/disable buses	
SET_MSGTO	Set message timeout	
SET_RT_ADDR	Get Remote Terminal address	
SET_STD	Get bus standard	
SET_BCE	Enable/disable broadcasts	
TX_BLOCK	Set blocking/non-blocking mode for RT write calls and BC DO_LIST commands.	
RX_BLOCK	Set blocking/non-blocking mode for RT and BM read calls	
CLR_STATUS	Clear status flag	
GET_STATUS	Read status flag	EINVAL
SET_EVENTID	Set event id	
DO_LIST	Execute list (BC mode)	EINVAL, EBUSY
LIST_DONE	Wait for list to finish execution (BC mode)	EINVAL, EBUSY

All ioctl requests takes as parameter the address to an unsigned int where data will be read from or written to depending on the request.

There are two more ioctl requests but they are not for configuration and are described later in Bus Controller Operation.

21.2.6.1. SET_MODE

Sets the operating mode of the BRM. Data should be 0 for BC, 1 for RT and 2 for BM.

21.2.6.2. SET_BUS

For RT mode only. Sets which buses that are enabled.

0 - none, 1 - bus B, 2 - bus A and 3 both bus A and B.

21.2.6.3. SET_MSGTO

For BC and BM mode. Sets the RT no response time out. If in MIL-STD-1553 B mode it is either 14 us or 30 us. In MIL-STD-1553 A mode either 9 us or 21 us.

21.2.6.4. SET_RT_ADDR

Sets the remote address for the RT. 0 - 30 if broadcasts enabled, 0 - 31 otherwise.

21.2.6.5. BRM_SET_STD

Sets the bus standard. 0 for MIL-STD-1553 B, 1 for MIL-STD-1553 A.

21.2.6.6. BRM_SET_BCE

Enable/disable broadcasts. 1 enables them, 0 disables.

21.2.6.7. BRM_TX_BLOCK

Set blocking/non blocking mode for RT write calls and BC ioctls. Blocking is default.

21.2.6.8. BRM_RX_BLOCK

Set blocking/non blocking mode for RT read calls. Blocking is default.

21.2.6.9. BRM_CLR_STATUS

Clears status bit mask. No input is needed it always succeeds.

21.2.6.10. BRM_GET_STATUS

Reads the status bit mask. The status bit mask is modified when an error interrupt is received. This ioctl command can be used to poll the error status by setting the argument to an [unsigned int] pointer.

Table 21.10. Status bit mask

Bit(s)	Description	Modes
31-16	The last descriptor that caused an error. Is not set for hardware errors.	BC, RT
BRM_DMAF_IRQ	DMA Fail	all
BRM_WRAPF_IRQ	Wrap Fail	BC, RT
BRM_TAPF_IRQ	Terminal Address Parity Fail	RT
BRM_MERR_IRQ	Message Error	all
BRM_RT_ILLCMD_IRQ	Illegal Command	RT
BRM_BC_ILLCMD_IRQ	Illogical Command	BC
BRM_ILLOP_IRQ	Illogical Opcode	BC

21.2.6.11. BRM_EST_EVENTID

Sets the event id to an event id external to the driver. It is possible to stop the event signalling by setting the event id to zero.

When the driver notifies the user (using the event id) the bit mask that caused the interrupt is sent along as an argument. Note that it may be different from the status mask read with BRM_GET_STATUS since previous error interrupts may have changed the status mask. Thus there is no need to clear the status mask after an event notification if only the notification argument is read.

See table Table 21.10 for the description of the notification argument.

21.2.7. Remote Terminal operation

When operating as Remote Terminal (RT) the driver maintains a receive event queue. All events such as receive commands, transmit commands, broadcasts, and mode codes are put into the event queue. Each event is described using a `struct rt_msg` as defined earlier in the data structure subsection.

The events are read of the queue using the read() call. The buffer should point to the beginning of one or several `struct rt_msg`. The number of events that can be received is specified with the length argument. E.g:

```
struct rt_msg msg[2];
n = read(brm_fd, msg, 2);
```

The above call will return the number of events actually placed in msg. If in non- blocking mode -1 will be returned if the receive queue is empty and errno set to EBUSY. Note that it is possible also in blocking mode that not all events specified will be received by one call since the read call will seize to block as soon as there is one event available.

What kind of event that was received can be determined by looking at the [desc] member of a `rt_msg`. It should be interpreted according to Table 21.8. How the rest of the fields should be interpreted depends on what kind of event it was, e.g if the event was a reception to subaddress 1 to 30 the word count field in the message information word gives the number of received words and the data array contains the received data words.

To place data in the transmit buffers the write() call is used. The buffer should point to the beginning of one or several `struct rt_msg`. The number of messages is specified with the length argument. E.g:

```
struct rt_msg msg;
msg.desc = 33; /* transmit for subaddress 1 */
msg.miw = (16 << 11) | (1 << 9) /* 16 words on bus A */
msg.data[0] = 0x1234;
...
msg.data[15] = 0xAABB;
n = write(brm_fd, msg, 1);
```

The number of messages actually placed in the transmit queue is returned. If the device is in blocking mode it will block until there is room for at least one message. When the buffer is full and the device is in non-blocking mode -1 will be returned and [errno] set to EBUSY. Note that it is possible also in blocking mode that not all messages specified will be transmitted by one call since the write call will seize to block as soon as there is room for one message.

The transmit buffer is implemented as a circular buffer with room for 8 messages with 32 data words each. Each `write()` call appends a message to the buffer.

21.2.8. Bus Controller operation

To use the BRM as Bus Controller one first has to use an ioctl() call to set BC mode. Command lists that the BC should process are then built using arrays of `struct bc_msg` described earlier in the data structure subsection. To start the list processing the ioctl() request BRM_DO_LIST is used. The ioctl() request BRM_LIST_DONE is used to check when the list processing is done. It returns 1 in the supplied argument if operation has finished. Note that BRM_LIST_DONE must be called before traversing the list to check results since this operation also copies the results into the array. Errno is set to EBUSY when issuing a BRM_DO_LIST before the last BRM_DO_LIST command has finished its execution.

Example use:

```
struct bc_msg msg[2];
int done, data, k;
data = 0;
ioctl(brm_fd, BRM_SET_MODE, &data); /* set BC mode */
bc_msg[0].rtaddr[0] = 1;
```

```
bc_msg[0].subaddr[0] = 1;
bc_msg[0].wc = 32;
bc_msg[0].ctrl = BC_BUSA; /* rt receive on bus a */
for (k = 0; k < 32; k++)
  bc_msg[0].data[k] = k;
bc_msg[1].ctrl |= BC_EOL; /* end of list */
ioctl(brm_fd, BRM_DO_LIST, bc_msg);
ioctl(brm_fd, BRM_LIST_DONE, &done);
```

If in blocking mode the BRM_LIST_DONE ioctl will block until the BC has processed the list. When the BC is finished and BRM_LIST_DONE has returned 1 in the argument the status words and received data can be interpreted by the application. During blocking mode BRM_LIST_DONE may set errno to EINVAL if an illogical opcode or an illogical command is detected by the hardware during the list execution.

21.2.9. Bus monitor operation

When operating as Bus Monitor (BM) the driver maintains a capture event queue. All events such as receive commands, transmit commands, broadcasts, and mode codes are put into the event queue. Each event is described using a `struct bm_msg` as defined in the data structure subsection.

The events are read of the queue using the read() call. The buffer should point to the beginning of one or several `struct bm_msg`. The number of events that can be received is specified with the length argument. E.g:

```
struct bm_msg msg[2];
n = read(brm_fd, msg, 2);
```

The above call will return the number of events actually placed in [msg]. If in non- blocking mode -1 will be returned if the receive queue is empty and [errno] set to EBUSY. Note that it is possible also in blocking mode that not all events specified will be received by one call since the read call will seize to block as soon as there is one event available.

22. B1553RT GRLIB Actel Core1553 RT driver

22.1. Introduction

This section describes the B1553RT Remote Terminal driver available for RTEMS. The reader is assumed to be well acquainted with MIL-STD-1553 and RTEMS.

The B1553RT driver require the RTEMS Driver Manager.

22.1.1. RT Hardware

The B1553RT core operate at the same frequency as the bus, it must be 12, 16, 20 or 24MHz. It requires a 4KByte DMA buffer area that must be aligned properly.

22.1.2. 1.1.2 Examples

There is a simple example available, it illustrates how to set up RT for reception and transmission of messages sent by a BC. Received messages are handled by updating the transmission DMA Area for respective sub address. The example collects statistics for received mode codes that the BC can read at sub address 30.

The example is part of the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/rtems-b1553rt.c`.

22.2. User interface

The RTEMS MIL-STD-1553B RT driver supports standard accesses to file descriptors such as **read**, **write** and **ioctl**. User applications include the rt driver's header file which contains definitions of all necessary data structures and bit masks used when accessing the driver. An example application using the driver is provided in the examples directory.

22.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when B1553RT hardware is found for the first time. The driver is called from the driver manager to handle detected B1553RT hardware. In order for the driver manager to unite the B1553RT driver with the B1553RT hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

22.2.2. Driver resource configuration

The driver can be configured using driver resources as described in the driver manager chapter. Below is a description of configurable driver parameters. The driver parameters is unique per B1553RT device. The parameters are all optional, the parameters only overrides the default values.

Table 22.1. B1553RT driver parameter description

Name	Type	Parameter description
coreFreq	INT	The input clock frequency to the RT core. 0 = 12MHz, 1 = 16MHz, 2= 20MHz, 3 = 24MHz. The default is 24MHz. The driver auto detect the bus frequency and override the default if the bus frequency is 20MHz, 16MHz or 12MHz. This parameter override the default and the auto detected value.
dmaBaseAdr	INT	Custom DMA area address. See note below.

22.2.2.1. Custom DMA area parameter

The DMA area can be configured to be located at a custom address. The standard configuration is to leave it up to the driver to do dynamic allocation of the areas. However in some cases it may be required to locate the DMA area on a custom location, the driver will not allocate memory but will assume that enough memory is available and that the alignment needs of the core on the address given is fulfilled. The memory required is either 4K bytes.

For some systems it may be convenient to give the addresses as seen by the B1553RT core. This can be done by setting the LSB bit in the address to one. For example a PCI Target board with a AMBA bus with a B1553RT core doesn't read from the same address as the CPU in order to access the same data. This is dependent on the PCI mappings. Translation between CPU and B1553RT addresses must be done. The B1553RT driver automatically translates the DMA base address. This requires the bus driver, in this case the PCI Target driver, to set up translation addresses correctly.

22.2.3. Opening the device

Opening the device enables the user to access the hardware of a certain RT device. The driver is used for all RT devices available. The devices is separated by assigning each device a unique name and a number called [minor]. The name is passed during the opening of the driver. Some example device names are printed out below.

Table 22.2. Device number to device name conversion.

Device number	Filesystem name	Location
0	/dev/b1553rt0	On-Chip Bus
1	/dev/b1553rt1	On-Chip Bus
2	/dev/b1553rt2	On-Chip Bus

An example of an RTEMS open call is shown below.

```
fd = open("/dev/b1553rt0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 22.3.

Table 22.3. Open[errno] values.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened

22.2.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the rt driver.

22.2.5. I/O Control interface

Changing the behaviour of the driver for a device is done via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global [errno] variable is set accordingly.

All supported commands and their data structures are defined in the RT driver's header file `b1553rt.h`.

22.2.5.1. Data structures

22.2.5.1.1. Remote Terminal operating mode

The structure below is used for all received events as well as to put data in the transmit buffer.

```
struct rt_msg {
    unsigned short miw;
    unsigned short time;
    unsigned short data[32];
```

```
    unsigned short desc;
};
```

Table 22.4. [rt_msg] member descriptions.

Member	Description
miw	Message Information Word. **Bit(s)** / **Description** 15-11: Word count / mode code - For sub addresses this is the number of received words. For mode codes it is the receive/transmit mode code. 10: - 9: A/B - 1 if message receive on bus A, 0 if received on bus B. 8: reserved 7: ME - 1 if an error was encountered during message processing. Bit 4-0 gives the details of the error. 6-5: - 4: ILL - 1 if received command is illegalized. 3: reserved 2: reserved 1: PRTY - 1 if the RT detected a parity error in the received data. 0: MAN - 1 if a Manchester decoding error was detected during data reception.
time	Time Tag - Contains the value of the internal timer register when the message was received.
data	An array of 32 16 bit words. The word count specifies how many data words that are valid. For receive mode codes with data the first data word is valid.
desc	Bit 6-0 is the descriptor used. Bit 15 indicates software buffer overrun when set, the messages was not read out in time which lead to the driver needed to skip at least one received message.

The last variable in the struct rt_msg shows which descriptor (i.e rx subaddress, tx subaddress, rx mode code or tx mode code) that the message was for. They are defined as shown in the table below:

Table 22.5. Descriptor table

Descriptor	Description
0	Reserved for RX mode codes
1-30	Receive subaddress 1-30
31	Reserved for RX mode codes
32	Reserved for TX mode codes
33-62	Transmit subaddress 1-30
63	Reserved for TX mode codes
64-95	Receive mode code
96-127	Transmit mode code

If there has occurred an event queue overrun bit 15 of this variable will be set in the first event read out. All events received when the queue is full are lost.

The RT core and driver are configured using ioctl calls. The table Table 22.7 below lists all supported ioctl calls. RT_ should be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 22.6.

An example is shown below where the Remote Terminal Address is set to one by using an **ioctl** call:

*Table 22.6. ERRNO values for **ioctl** calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
ENOSYS	Invalid request, now such **ioctl** command.

*Table 22.7. ERRNO values for **ioctl** calls.*

Call Number	Description	ERRNO
SET_ADDR	Set Remote Terminal address	
SET_BCE	Enable/disable broadcast	
SET_VECTORW	Set VECTOR WORD register in RT core	
SET_EXTMDATA	Set/Clear EXTMDATA bit in RT core	
RX_BLOCK	Set blocking/non-blocking mode for read calls	
CLR_STATUS	Reset status flag	
GET_STATUS	Read status flag	EINVAL
SET_EVENTID	Set event id used to signal detected errors with	

All **ioctl** requests takes as parameter the address to [an unsigned int] where data will be read from or written to depending on the request.

22.2.6.1. RT_SET_ADDR

Sets the remote address for the RT. 0 - 30 if broadcasts enabled, 0 - 31 otherwise.

22.2.6.2. RT_SET_BCE

Enable/disable broadcasts. 1 enables them, 0 disables.

22.2.6.3. RT_SET_VECTORW

Set the vector word register in the RT core. This might not have an effect depending on how the RT core register have been set up.

22.2.6.4. RT_RX_BLOCK

Set blocking/non blocking mode for RT read calls. Blocking is default.

22.2.6.5. RT_SET_EXTMDATA

Set or clear the EXTMDATA bit of the RT core. The input is a pointer to a integer which determines the EXTMDATA bit.

22.2.6.6. RT_SET_STATUS

Clears status bit mask. No input is needed it always succeeds.

22.2.6.7. RT_GET_STATUS

Reads the status bit mask. The status bit mask is modified when an error interrupt is received. This ioctl command can be used to poll the error status by setting the argument to an `unsigned int` pointer.

Table 22.8. Status bit mask

Bit(s)	Description
31-16	The last descriptor that caused an error. Is not set for hardware errors.
RT_DMAF_IRQ	DMA Fail, AHB error from AMBA wrapper or Memory failure indicated by the RT Core.

Bit(s)	Description
RT_MERR_IRQ	Message Error
RT_ILLCMD_IRQ	Illegal Command

22.2.6.8. RT_SET_EVENTID

Sets the event id to an event id external to the driver. It is possible to stop the event signalling by setting the event id to zero.

When the driver notifies the user (using the event id) the bit mask that caused the interrupt is sent along as an argument. Note that it may be different from the status mask read with RT_GET_STATUS since previous error interrupts may have changed the status mask. Thus there is no need to clear the status mask after an event notification if only the notification argument is read.

See table Table 22.8 for the description of the notification argument.

22.2.7. Remote Terminal operation

The Remote Terminal (RT) driver maintains a receive event queue. All events such as receive commands, transmit commands, broadcasts, and mode codes are put into the event queue. Each event is described using a `struct rt_msg` as defined earlier in the data structure subsection.

The events are read of the queue using the read() call. The buffer should point to the beginning of one or several struct `rt_msg`. The number of events that can be received is specified with the length argument. E.g:

```
struct rt_msg msg[2];
n = read(rt_fd, msg, 2);
```

The above call will return the number of events actually placed in msg. If in non-blocking mode -1 will be returned if the receive queue is empty and errno set to EBUSY. Note that it is possible also in blocking mode that not all events specified will be received by one call since the read call will seize to block as soon as there is one event available.

What kind of event that was received can be determined by looking at the [desc] member of a `rt_msg`. It should be interpreted according to table 8. How the rest of the fields should be interpreted depends on what kind of event it was, e.g if the event was a reception to subaddress 1 to 30 the word count field in the message information word gives the number of received words and the data array contains the received data words.

To place data in the transmit sub addresses the `write()` call is used. The buffer should point to the beginning of one `struct rt_msg`. The number of messages is specified with the length argument, it must be specified to one. E.g:

```
struct rt_msg msg;
msg.desc = 33; /* transmit for subaddress 1 */
msg.miw = (16 << 11); /* 16 words */
msg.data[0] = 0x1234;
...
msg.data[15] = 0xAABB;
n = write(rt_fd, msg, 1);
```

Regardless of the blocking mode the message will be copied directly into the RT DMA area and the write call will return directly.

23. GRCAN CAN driver

23.1. User interface

The RTEMS CAN driver supports the standard accesses to file descriptors such as **read**, **write** and **ioctl**. User applications include the GRCAN driver's header file (`grcan.h`) which contains definitions of all necessary data structures and bit masks used when accessing the driver.

The GRCAN driver require the RTEMS Driver Manager.

23.1.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when CAN hardware is found for the first time. The driver is called from the driver manager to handle detected CAN hardware. In order for the driver manager to unite the CAN driver with the CAN hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

23.1.2. Driver resource configuration

The driver can be configured using driver resources as described in the driver manager chapter. Below is a description of configurable driver parameters. The driver parameters is unique per CAN device. The parameters are all optional, the parameters only overrides the default values.

Table 23.1. GRCAN driver parameter description

Name	Type	Parameter description
txBufSize	INT	Length of TX DMA area. Must be a multiple of 64 bytes, four messages.
rxBufSize	INT	Length of RX DMA area. Must be a multiple of 64 bytes, four messages.
txBufAdr	INT	Custom TX DMA area address. See note below.
rxBufAdr	INT	Custom RX DMA area address. See note below.

23.1.2.1. Custom DMA area parameters

The DMA area can be configured to be located at a custom address. The standard configuration is to leave it up to the driver to do dynamic allocation of the areas. However in some cases it may be required to locate the DMA area on a custom location, the driver will not allocate memory but will assume that enough memory is available and that the alignment needs of the core on the address given is fulfilled.

For some systems it may be convenient to give the addresses as seen by the CAN core. This can be done by setting the LSB bit in the address to one. For example a GR- RASTA-IO board with a CAN core doesn't read from the same address as the CPU in order to access the same data. This is dependent on the PCI mappings. Translation between CPU and CAN addresses must be done. The CAN driver automatically translates the required addresses. This requires the bus driver, in this case the GR-RASTA-IO driver, to set up translation addresses correctly.

23.1.3. Opening the device

Opening the device enables the user to access the hardware of a certain CAN core. The driver is used for all GRCAN cores available. The cores are separated by assigning each device a unique name and a number called [minor]. The name is passed during the opening of the driver. Some example device names are printed out below.

Table 23.2. Core number to device name conversion.

Device number	Filesystem name	Location
0	/dev/grcan0	On-Chip Bus
1	/dev/grcan1	On-Chip Bus
2	/dev/grcan2	On-Chip Bus

Device number	Filesystem name	Location
Depends on sytem configuration	/dev/rastaio0/grcan0	GR-RASTA-IO

An example of an RTEMS open call is shown below.

```
fd = open("/dev/grcan0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 23.1.

Table 23.3. Open ERRNO values

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

23.1.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the [grcan] driver.

23.1.5. I/O Control interface

Changing the behaviour of the driver for a device is done via the standard system call **ioctl**. Two arguments must be provided to ioctl, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global [errno] variable is set accordingly.

All supported commands and their data structures are defined in the CAN driver's header file `grcan.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

23.1.5.1. Data structures

The grcan_filter structure is used when changing acceptance filter of the CAN receiver and the SYNC Rx/Tx Filter.

Note that the two different **ioctl** commands use this data structure differently.

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

Table 23.4. grcan_filter member description

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

The CANMsg struct is used when reading and writing messages. The structure describes the drivers view of a CAN message. The structure is used for writing and reading. See the transmission and reception section for more information.

```
typedef struct {
  char extended;
  char rtr;
  char unused;
  unsigned char len;
```

```
  unsigned char data[8];
  unsigned int id;
} CANMsg;
```

Table 23.5. CANMsg member description

Member	Description
extended	Indicates whether message has 29 or 11 bits ID tag. Extended or Standard frame.
rtr	Remote Transmission Request bit.
len	Length of data.
data	Message data, data[0] is the most significant byte – the first byte.
Id	The ID field of the message. An extended frame has 29 bits whereas a standard frame has only 11-bits. The most significant bits are not used.

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

```
typedef struct {
     /* tx/rx stats */
     unsigned int passive_cnt;
     unsigned int overrun_cnt;
     unsigned int rxsync_cnt;
     unsigned int txsync_cnt;
     unsigned int ints;
} grcan_stats;
```

Table 23.6. grcan_stats member description

Member	Description
passive_cnt	Number of error passive mode detected.
overrun_cnt	Number of reception over runs.
rxsync_cnt	Number of received SYNC messages (matching SYNC filter)
txsync_cnt	Number of transmitted SYNC messages (matching SYNC filter)
ints	Number of times the interrupt handler has been invoked.

The grcan_timing data structure is used when setting the configuration register manually of the CAN core. The timing 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 23.7. 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 grcan_selection data structure is used to select active channel. Each channel has one transceiver that can be inactivated or activated 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 {
   unsigned char selection;
   unsigned char enable0;
   unsigned char enable1;
};
```

Table 23.8. 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

23.1.5.2. Configuration

The CAN core and driver are configured using **ioctl** calls. The Table 23.4 below lists all supported **ioctl** calls. GRCAN_IOC_ must be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 23.3.

An example is shown below where the driver's read call changes behaviour. After this call the driver will block the calling thread until free space in the receiver's circular buffer are available:

```
result = ioctl(fd, GRCAN_IOC_SET_RXBLOCK, 1);
```

*Table 23.9. ERRNO values for **ioctl** calls*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The CAN hardware is not in the correct state. Many **ioctl** calls need the CAN device to be in reset mode. One can switch state by calling START or STOP.
ENOMEM	Not enough memory to complete operation. This may cause other **ioctl** commands to fail.
ENODEV	The call has been aborted by another call or due to a state change. Is returned when the driver has blocked the calling thread but needs to wake it in order to avoid a dead lock. This may be due to another thread closing the driver or a detected hardware error.

23.1.5.2.1. START

This ioctl command places the CAN core in running mode. Settings previously set by other **ioctl** commands are written to hardware just before leaving reset mode. It is necessary to enter running mode to be able to read or write messages on the CAN bus.

The command will fail if receive or transmit buffers are not correctly allocated or if the CAN core already is in running mode.

23.1.5.2.2. STOP

This call makes the CAN core leave operating mode and enter reset mode. After calling STOP further calls to **read** and **write** will result in errors.

It is necessary to enter reset 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 command will fail if the CAN core already is in reset mode.

23.1.5.2.3. ISSTARTED

Is used to determine the driver state. Returns the error state EBUSY when the driver is in stopped mode. It returns 0 and errno is not set when the driver is started.

23.1.5.2.4. FLUSH

This call blocks the calling thread until all messages in the driver's buffers has been processed by the CAN hardware.

The flush command may fail if the state is changed, the driver is closed, or an error is detected by hardware. Errno is set to ENODEV to identify such a case.

23.1.5.2.5. SET_SILENT

This command set the SILENT bit in the configuration register of the CAN hardware. If the SILENT bit is set the CAN core operates in listen only mode. **write** calls fails and **read** calls proceed.

This call fail if the driver is in running mode. Errno is set to EBUSY when in running mode.

23.1.5.2.6. SET_ABORT

This command set 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 call never fail.

23.1.5.2.7. SET_SELECTION

This command selects active channel used during communication. The SET_SELECTION command takes a second argument, a pointer to a `grcan_selection` data structure described in the data structures section.

This call will fail if the driver is in running mode. The errno variable will be set to EBUSY and -1 is returned from **ioctl**.

23.1.5.2.8. SET_BTRS

This call sets the timing registers manually. See the CAN hardware documentation for a detailed description of the timing parameters. The SET_BTRS call takes a pointer to a `grcan_timing` data structure containing all available timing parameters. The grcan_timing data structure is described in the data structure section.

This call fail if the CAN core is in running mode, in that case [errno] will be set to EBUSY and ioctl will return -1.

23.1.5.2.9. SET_RXBLOCK

This call changes the behaviour of read calls to blocking or non-blocking mode. When in blocking mode the calling thread will be blocked until there is data available to read. It may return after any number of bytes has been read. Use the RXCOMPLETE for controlling the driver's blocking mode behaviour further.

For non-blocking mode the calling thread will never be blocked returning a zero length of data.

The RXCOMPLETE has no effect during non-blocking mode.

This call never fails, it is valid to call this command in any mode.

23.1.5.2.10. SET_TXBLOCK

This call changes the behaviour of **write** calls to blocking or non-blocking mode. When in blocking mode the calling thread will be blocked until at least one message can be written to the driver's circular buffer. It may return after any number of messages has been written. Use the TXCOMPLETE for controlling the driver's blocking mode behaviour further.

For non-blocking mode the calling thread will never be blocked which may result in **write** returning a zero length when the driver's internal buffers are full. The TXCOMPLETE has no effect during non-blocking mode.

This call never fails, it is valid to call this command in any mode.

23.1.5.2.11. SET_TXCOMPLETE

This command disables or enables the write command to block until all messages specified by the caller are copied to driver's internal buffers before returning.

This option is only relevant in TX blocking mode.

This call never fail.

23.1.5.2.12. SET_RXCOMPLETE

This command disables or enables the **read** command to block until all messages specified by the caller are read into the user specified buffer.

This option is only relevant in RX blocking mode.

This call never fails.

23.1.5.2.13. GET_STATS

This call copies the driver's internal counters to a user provided data area. The format of the data written is described in the data structure subsection. See the `grcan_stats` data structure.

The call will fail if the pointer to the data is invalid.

23.1.5.2.14. CLR_STATS

Clears the driver's collected statistics.

This call never fail.

23.1.5.2.15. SET_AFILTER

Set Acceptance filter matched by receiver for every message that is received. Let the second argument point to a `grcan_filter` data structure or NULL to disable filtering to let all messages pass the filter. Messages matching the below function are passed and possible to read from user space:

```
(Id XOR Code) AND Mask = 0
```

This command never fail.

23.1.5.2.16. SET_SFILTER

Set Rx/Tx SYNC filter matched by receiver for every message that is received. Let the second argument point to a `grcan_filter` data structure or NULL to disable filtering to let all messages pass the filter. Messages matching the below function are treated as SYNC messages:

```
(Id XOR Code) AND Mask = 0
```

This command never fail.

23.1.5.2.17. GET_STATUS

This call stores the current status of the CAN core to the address pointed to by the argument given to **ioctl**. This call is typically used to determine the error state of the CAN core. The 4 byte status bit mask can be interpreted as in table above.

Table 23.10. Status bit mask

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 councter value, 8-bit
GRCAN_STAT_TXERRCNT	Transmission error councter value, 8-bit

This call never fail.

23.1.6. Transmission

Transmitting messages are done with the **write** call. It is possible to write multiple packets in one call. An example of a write call is shown below:

```
result = write(fd, &tx_msgs[0], sizeof(CANMsg)*msgcnt));
```

On success the number of transmitted bytes is returned and -1 on failure. Errno is also set in the latter case. Tx_msgs points to the beginning of the CANMsg structure which includes id, type of message, data and data length. The last parameter sets the number of CAN messages that will be transmitted it must be a multiple of CANMsg structure size.

The write call can be configured to block when the software fifo is full. In non-blocking mode write will immediately return either return -1 indicating that no messages were written or the total number of bytes written (always a multiple of CANMsg structure size). Note that 3 message write request may end up in only 2 written, the caller is responsible to check the number of messages actually written in non-blocking mode.

If no resources are available in non-blocking mode the call will return with an error. The **errno** variable is set according to the table given below.

*Table 23.11. ERRNO values for **write***

ERRNO	Description
EINVAL	An invalid argument was passed. The buffer length was less than a single CANMsg structure size.
EBUSY	The link is not in operating mode, but in reset mode. Nothing done.
ETIMEDOUT	In non-blocking mode.
ENODEV	Calling task was woken up from blocking mode by a bus off error. The CAN core has entered reset mode. Further calls to **read** or **write** will fail until the **ioctl** command START is issued again.

Each Message has an individual set of options controlled in the CANMsg structure. See the data structure subsection for structure member descriptions.

23.1.7. Reception

Reception of CAN messages from the CAN bus can be done using the read call. An example is shown below:

```
CANMsg rx_msgs[5];

len = read(fd, rx_msgs, sizeof(rx_msgs));
```

The requested number of bytes to be read is given in the third argument. The messages will be stored in rx_msgs. The actual number of received bytes (a multiple of sizeof(CANMsg)) is returned by the function on success and -1 on failure. In the latter case **errno** is also set.

The CANMsg data structure is described in the data structure subsection.

The call will fail if a null pointer is passed, invalid buffer length, the CAN core is in stopped mode or due to a bus off error in blocking mode.

The blocking behaviour can be set using **ioctl** calls. In blocking mode the call will block until at least one message has been received. In non-blocking mode, the call will return immediately and if no message was available -1 is returned and **errno** set appropriately. The table below shows the different **errno** values returned.

*Table 23.12. ERRNO values for **read** calls*

ERRNO	**Description**
EIVNAL	A NULL pointer was passed as the data pointer or the length was illegal.
EBUSY	CAN core is in reset mode. Switch to operating mode by issuing a START **ioctl** command.
ETIMEOUT	In non-blocking mode no messages were available in the software receive FIFO.
EIO	A blocking read was interrupted by a bus off error. The CAN core has entered reset mode. Further calls to **read** or **write** will fail until the **ioctl** command START is issued again.

24. CAN_OC GRLIB Opencores CAN driver

24.1. Introduction

This document is intended as an aid in getting started developing with GRLIB wrapper for Opencores CAN core using the driver described in this document. It briefly takes the reader through some of the most important steps in using the driver such as setting up a connection, configuring the driver, reading and writing CAN messages. The reader is assumed to be well acquainted with CAN and RTEMS.

The OC_CAN driver require the RTEMS Driver Manager.

24.1.1. CAN Hardware

The OC_CAN core can operate in different modes providing the same register interfaces as other well known CAN cores. The OC_CAN driver supports PeliCAN mode only.

24.1.2. Software Driver

The driver provides means for processes and threads to send and receive messages. Errors can be detected by polling the status flags of the driver. Bus off errors cancels the ongoing transfers to let the caller handle the error.

The driver supports filtering received messages id fields by means of acceptance filters, runtime timing register calculation given a baud rate. However not all baud rates may be available for a given system frequency. The system frequency is hard coded and must be set in the driver.

24.1.3. Examples

There is a simple example available, it illustrates how to set up a connection, reading and writing messages using the OC_CAN driver. It is made up of two tasks communicating with each other through two OC_CAN devices. To be able to run the example one must have two OC_CAN devices externally connected together on the different or the same board.

The example is part of the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/examples/samples/rtems-occan.c, occan_lib.c and occan_lib.h .`

The example can be built by running:

```
cd /opt/rtems-4.10/src/examples/samples
make clean rtems-occan rtems-occan_tx rtems-occan_rx
```

Where rtems-occan is intended for boards with two OC_CAN cores and rtems-occan_* is for set ups including two boards with one OC_CAN core each.

24.2. User interface

The RTEMS OC_CAN driver supports the standard accesses to file descriptors such as **read**, **write** and **ioctl**. User applications include the occan driver's header file which contains definitions of all necessary data structures and bit masks used when accessing the driver. An example application using the driver is provided in the examples directory.

24.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when CAN hardware is found for the first time. The driver is called from the driver manager to handle detected CAN hardware. In order for the driver manager to unite the CAN driver with the CAN hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

24.2.2. Driver resource configuration

This driver does not have any configurable resources. All configuration can be made though the **ioctl** interface.

24.2.3. Opening the device

Opening the device enables the user to access the hardware of a certain OC_CAN device. The driver is used for all OC_CAN devices available. The devices is separated by assigning each device a unique name and a number called [minor]. The name is passed during the opening of the driver. The first 3 names are printed out:

Table 24.1. Device number to device name conversion.

Device number	Filename name
0	/dev/occan0
1	/dev/occan1
2	/dev/occan2

An example of an RTEMS open call is shown below.

```
fd = open("/dev/occan0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 24.1.

*Table 24.2. Open **ERRNO** values.*

ERRNO	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

24.2.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the **occan** driver.

24.2.5. I/O Control interface

Changing the behaviour of the driver for a device is done via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly.

All supported commands and their data structures are defined in the OC_CAN driver's header file `occan.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

24.2.5.1. Data structures

The occan_afilter struct is used when changing acceptance filter of the OC_CAN receiver.

```
struct occan_afilter {
  unsigned int code[4];
  unsigned int mask[4];
  int single_mode;
};
```

Table 24.3. occan_afilter member descriptions.

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

Member	Description
single_mode	Set to none-zero for a single filter - single filter mode, zero selects dual filter mode.

The CANMsg struct is used when reading and writing messages. The structure describes the driver's view of a CAN message. The structure is used for writing and reading. The **sshot** fields lacks meaning during reading and should be ignored. See the transmission and reception section for more information.

```
 typedef struct {
   char extended;
   char rtr;
   char sshot;
   unsigned char len;
   unsigned char data[8];
   unsigned int id;
} CANMsg;
```

Table 24.4. CANMsg member descriptions.

Member	Description
extended	Indicates whether message has 29 or 11 bits ID tag. Extended or Standard frame.
rtr	Remote Transmission Request bit.
sshot	Single Shot. Setting this bit will make the hardware skip resending the message on transmission error.
len	Length of data.
data	Message data, data[0] is the most significant byte – the first byte.
Id	The ID field of the message. An extended frame has 29 bits whereas a standard frame has only 11-bits. The most significant bits are not used.

The occan_stats struct contains various statistics gathered from the OC_CAN hardware.

```
typedef struct {
   /* tx/rx stats */
   unsigned int rx_msgs;
   unsigned int tx_msgs;

   /* Error Interrupt counters */
   unsigned int err_warn;
   unsigned int err_dovr;
   unsigned int err_errp;
   unsigned int err_arb;
   unsigned int err_bus;

   /* ALC 4-0 */
   unsigned int err_arb_bitnum[32];

   /* ECC 7-6 */
   unsigned int err_bus_bit; /* Bit error */
   unsigned int err_bus_form; /* Form Error */
   unsigned int err_bus_stuff; /* Stuff Error */
   unsigned int err_bus_other; /* Other Error */

   /* ECC 5 */
   unsigned int err_bus_rx;
   unsigned int err_bus_tx;

  /* ECC 4:0 */
  unsigned int err_bus_segs[32];

  /* total number of interrupts */
  unsigned int ints;

  /* software monitoring hw errors */
  unsigned int tx_buf_error;
} occan_stats;
```

Table 24.5. occan_stats member descriptions.

Member	Description
rx_msgs	Number of CAN messages received.

Member	Description
tx_msgs	Number of CAN messages transmitted.
err_warn	Number of error warning interrupts.
err_dovr	Number of data overrun interrupts.
err_errp	Number of error passive interrupts.
err_arb	Number of times arbitration has been lost.
err_bus	Number of bus errors interrupts.
err_arb_bitnum	Array of counters, err_arb_bitnum[index] is incremented when arbitration is lost at bit **index**.
err_bus_bit	Number of bus errors that was caused by a bit error.
err_bus_form	Number of bus errors that was caused by a form error.
err_bus_stuff	Number of bus errors that was caused by a stuff error.
err_bus_other	Number of bus errors that was not caused by a bit, form or stuff error.
err_bus_tx	Number of bus errors detected that was due to transmission.
err_bus_rx	Number of bus errors detected that was due to reception.
err_bus_segs	Array of 32 counters that can be used to see where the frame transmission often fails. See hardware documentation and header file for details on how to interpret the counters.
ints	Number of times the interrupt handler has been invoked.

24.2.5.2. Configuration

The OC_CAN core and driver are configured using **ioctl** calls. The Table 24.4 below lists all supported **ioctl** calls. OCCAN_IOC_ should be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 24.3.

An example is shown below where the receive and transmit buffers are set to 32 respective 8 by using an **ioctl** call:

```
result = ioctl(fd, OCCAN_IOC_SET_BUFLEN, (8<<16) | 32);
```

*Table 24.6. ERRNO values for **ioctl**calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The CAN hardware is not in the correct state. Many **ioctl** calls need the CAN device to be in reset mode. One can switch state by calling START or STOP.
ENOMEM	Not enough memory to complete operation. This may cause other **ioctl** commands to fail.

*Table 24.7. **ioctl** calls supported by the OC_CAN driver.*

Call Number	Call Mode	Description
START	Reset	Exit reset mode, brings up the link. Enables **read** and **write**.
STOP	Running	Exit operating mode, enter reset mode. Most of the settings can only be set when in reset mode.
GET_STATS	Don't care	Get Stats.
GET_STATUS	Don't care	Get status of device. Bus off can be read out.
GET_SPEED	Reset	Set baud rate.
GET_BLK_MODE	Don't Care	Set blocking or non-blocking mode for **read** and **write**.
GET_BUFLEN	Reset	Set receive and transmit buffer length.

Call Number	Call Mode	Description
GET_BTRS	Reset	Set timing registers manually.

24.2.5.2.1. START

This **ioctl** command places the CAN core in operating mode. Settings previously set by other **ioctl** commands are written to hardware just before leaving reset mode. It is necessary to enter operating mode to be able to read or write messages on the CAN bus.

The command will fail if receive or transmit buffers are not correctly allocated or if the CAN core already is in operating mode.

24.2.5.2.2. STOP

This call makes the CAN core leave operating mode and enter reset mode. After calling STOP further calls to **read** and **write** will result in errors.

It is necessary to enter reset 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 command will fail if the CAN core already is in reset mode.

24.2.5.2.3. GET_STATS

This call copies the driver's internal counters to a user provided data area. The format of the data written is described in the data structure subsection. See the `occan_stats` data structure.

The call will fail if the pointer to the data is invalid.

24.2.5.2.4. GET_STATUS

This call stores the current status of the CAN core to the address pointed to by the argument given to **ioctl**. This call is typically used to determine the error state of the CAN core. The 4 byte status bit mask can be interpreted as in Table 24.2 above.

Table 24.8. Status bit mask.

Mask	Description
OCCAN_STATUS_RESET	Core is in reset mode
OCCAN_STATUS_OVERRUN	Data overrun
OCCAN_STATUS_WARN	Has passed the error warning limit (96)
OCCAN_STATUS_ERR_PASSIVE	Has passed the error Passive limit (127)
OCCAN_STATUS_ERR_BUSOFF	Core is in reset mode due to a bus off (255)

This call never fail.

24.2.5.2.5. SET_SPEED

The SET_SPEED **ioctl** call is used to set the baud rate of the CAN bus. The timing register values are calculated for the given baud rate. The baud rate is given in Hertz. For the baud rate calculations to function properly one must define SYS_FREQ to the system frequency. It is located in the driver source `occan.c`.

If the timing register values could not be calculated -1 is returned and the **errno** value is set to EINVAL.

24.2.5.2.6. SET_BTRS

This call sets the timing registers manually. It is encouraged to use this function over the SET_SPEED.

This call fail if CAN core is in operating mode, in that case **errno** will be set to EBUSY.

24.2.5.2.7. SET_BLK_MODE

This call sets blocking mode for receive and transmit operations, i.e. **read** and **write**. Input is a bit mask as described in the table below.

*Table 24.9. SET_BLK_MODE **ioctl** arguments*

Bit number	Description
OCCAN_BLK_MODE_RX	Set this bit to make **read** block when no messages can be read.
OCCAN_BLK_MODE_TX	Set this bit to make **write** block until all messages has been sent or put info software fifo.

This call never fail.

24.2.5.2.8. SET_BUFLEN

This call sets the buffer length of the receive and transmit software FIFOs. To set the FIFO length the core needs to be in reset mode. In the table below the input to the **ioctl/** command is described.

*Table 24.10. SET_BLK_MODE **ioctl** arguments*

Mask	Description
0x0000ffff	Receive buffer length in number of `CANMsg` structures.
0xffff0000	Transmit buffer length in number of `CANMsg` structures.

Errno will be set to ENOMEM when the driver was not able to get the requested memory amount. EBUSY is set when the core is in operating mode.

24.2.5.2.9. Transmission

Transmitting messages are done with the **write** call. It is possible to write multiple packets in one call. An example of a **write** call is shown below:

```
result = write(fd, &tx_msgs[0], sizeof(CANMsg)*msgcnt))
```

On success the number of transmitted bytes is returned and -1 on failure. Errno is also set in the latter case. Tx_msgs points to the beginning of the CANMsg structure which includes id, type of message, data and data length. The last parameter sets the number of CAN messages that will be transmitted it must be a multiple of CANMsg structure size.

The call will fail if the user tries to send more bytes than is allocated for a single packet (this can be changed with the SET_PACKETSIZE ioctl call) or if a NULL pointer is passed.

The write call can be configured to block when the software fifo is full. In non-blocking mode write will immediately return either return -1 indicating that no messages was written or the total number of bytes written (always a multiple of CANMsg structure size). Note that 3 message write request may end up in only 2 written, the caller is responsible to check the number of messages actually written in non-blocking mode.

If no resources are available in non-blocking mode the call will return with an error. The **errno** variable is set according to the table given below.

*Table 24.11. ERRNO values for **write***

ERRNO	Description
EINVAL	An invalid argument was passed. The buffer length was less than a single CANMsg structure size.
EBUSY	The link is not in operating mode, but in reset mode. Nothing done.
ETIMEDOUT	In non-blocking mode.

ERRNO	Description
EIO	Calling task was woken up from blocking mode by a bus off error. The CAN core has entered reset mode. Further calls to **read** or **write** will fail until the **ioctl** command START is issued again.

Each Message has an individual set of options controlled in the CANMsg structure. See the data structure subsection for structure member descriptions.

24.2.6. Reception

Reception is done using the read call. An example is shown below:

```
CANMsg rx_msgs[5];

len = read(fd, rx_msgs, sizeof(rx_msgs));
```

The requested number of bytes to be read is given in the third argument. The messages will be stored in rx_msgs. The actual number of received bytes (a multiple of sizeof(CANMsg)) is returned by the function on success and -1 on failure. In the latter case **errno** is also set.

The CANMsg data structure is described in the data structure subsection.

The call will fail if a null pointer is passed, invalid buffer length, the CAN core is in reset mode or due to a bus off error in blocking mode.

The blocking behaviour can be set using **ioctl** calls. In blocking mode the call will block until at least one packet has been received. In non-blocking mode, the call will return immediately and if no packet was available -1 is returned and **errno** set appropriately. The table below shows the different **errno** values is returned.

*Table 24.12. ERRNO values for **read**calls.*

ERRNO	Description
EINVAL	AA NULL pointer was passed as the data pointer or the length was illegal.
EBUSY	CAN core is in reset mode. Swtich to operating mode by issuing a START **ioctl** command.

25. SatCAN driver (SatCAN)

25.1. Introduction

This document is intended as an aid in getting started developing with the GRLIB wrapper for the SatCAN core using the driver described in this document. It briefly takes the reader through some of the most important steps in using the driver such as setting up a connection, configuring the driver, reading and writing CAN messages. The reader is assumed to be well acquainted with the operation of the SatCAN core and RTEMS.

25.1.1. SatCAN Hardware Wrapper

See the SatCAN wrapper manual.

25.1.2. Software Driver

The driver provides means for processes and threads to send and receive messages and provides callback functions for SatCAN wrapper interrupts.

All core registers can be accessed via Input/Output-control **(ioctl)** calls.

25.1.3. Examples

There is a simple example available, it illustrates how to set up a connection, reading and writing messages using the SATCAN driver. It is made up of two tasks communicating with each other where one task uses the OC_CAN driver and the other the SatCAN driver. To be able to run the example one must have the cores connected together. The current example is tailored for with a configuration matching GR712RC and also initializes the CAN_MUX RTEMS driver which is described in a separate document.

The example can be found under the samples directory and consists of the files `samples/rtems-occan.c`, `occan_lib.c` and `occan_lib.h`.

25.2. User interface

The RTEMS SATCAN driver supports the standard accesses to file descriptors such as **read**, **write** and **ioctl**. User applications should include the SATCAN driver's header file, `satcan.h`, which contains definitions of all necessary data structures and defines used when accessing the driver. An example application using the driver is provided in the samples directory.

25.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The function `satcan_register` whose prototype is provided in `satcan.h` is used for registering the driver. It returns 0 on success and 1 on failure. A typical register call from the LEON3 Init task:

```
if ( satcan_register(&satcan_conf) )
     printf("SatCAN register Failed\n");
```

The second argument to the function is the SatCAN configuration structure. The contents of this structure is described below:

```
typedef struct {
      int nodeno;
      int dps;
      void (*ahb_irq_callback)(void);
      void (*pps_irq_callback)(void);
      void (*m5_irq_callback)(void);
      void (*m4_irq_callback)(void);
      void (*m3_irq_callback)(void);
      void (*m2_irq_callback)(void);
      void (*m1_irq_callback)(void);
      void (*sync_irq_callback)(void);
      void (*can_irq_callback)(unsigned int fifo);
} satcan_config;
```

Table 25.1. Members in satcan_config structure

Member	Description
nodeno	Integer containing the writeable bits if the node number. The four least significant bits of this member are written to the writeable part of the node number.
dps	Set to 0 if core is DPS, set to 1 of core is non-DPS i.e. slave.
ahb_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the AHB bit set.
pps_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the PPS bit set.
m5_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the M5 bit set.
m4_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the M4 bit set.
m3_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the M3 bit set.
m2_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the M2 bit set.
m1_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the M1 bit set.
sync_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the sync bit set.
can_irq_callback	Function pointer to function called when the core issues an interrupt and the wrapper interrupt pending register has the can bit set.

The last callback function, `can_irq_callback`, is called with an unsigned integer as argument. This integer contains the value of the SatCAN FIFO register read in the interrupt handler.

Each callback function is called whenever the corresponding status bit in the wrapper interrupt pending register is set, regardless of whether or not the interrupt is masked in the wrapper interrupt mask register. If the the user does not want to use a callback function the corresponding member in the `satcan_config` structure must be set to NULL. After the call to `satcan_register(..)` has returned the structure can be deallocated.

When the driver is registered the driver allocates its internal configuration structures and registers the name `/dev/satcan` with RTEMS. The SatCAN wrapper is initialized with the node number and DPS setting specified in the configuration structure and the core is reset. After the core has come out of reset the registers containing the memory address of the newly allocated 2K DMA memory area are initialized.

25.2.2. Opening the device

Opening the device enables the user to access the hardware of the SatCAN device. An example of an RTEMS open call is shown below.

```
fd = open("/dev/satcan", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 25.1.

*Table 25.2. Open **ERRNO** values.*

ERRNO	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

When the device is opened the driver enables the AHB and CAN interrupts in the SatCAN wrapper logic. Interrupts EOD1, EOD2 and CAN Critical are enabled in the SatCAN FPGA core. The SatCAN FPGA core is also configured

to use “CAN” interrupt for interrupt #0 (CAN_TODn_Int_sel is set to '1') and RX together with the RX DMA channel is enabled.

25.2.3. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

When the device is closed the SatCAN wrapper and SatCAN FPGA interrupt mask registers are cleared. CAN RX and all DMA channels are disabled. The driver's internal state is initialized to default values. Close always returns 0 (success) for the SATCAN driver.

25.2.4. Reading from the device

After the device has been successfully opened it can be accessed via calls to `read(...)`. Read expects a pointer to a `satcan_msg` structure, or list of structures, and only accepts a multiple of the size of satcan_msg as the number of bytes to read. The `satcan_msg` structure, defined in `satcan.h`, and a description of its members is given below:

```
 typedef struct {
     unsigned char header[SATCAN_HEADER_SIZE];
     unsigned char payload[SATCAN_PAYLOAD_SIZE];
} satcan_msg;
```

Table 25.3. Members in satcan_msg structure

Member	Description
header	Header of SatCAN message as described in SatCAN FPGA documentation. The default value of the define SATCAN_HEADER_SIZE is 4.
payload	Payload of SatCAN message as described in SatCAN FPGA documentation. The default value of the define SATCAN_HEADER_SIZE is 8.

The driver does not buffer received SatCAN messages but provides direct access to the SatCAN FPGA DMA area. Therefore the caller must specify which CAN ID the message should be read from. An example call reading a message received with ID 0x0040 looks like:

```
int i, size;
satcan_msg msg;

msg.header[0] = 0x40;
msg.header[1] = 0;
if ((size = read(fd, &msg, sizeof(satcan_msg))) !=
      sizeof(satcan_msg))
      printf("ERROR! read() returned %d\n", size);
```

The driver uses the value of `msg.header[1:0]` together with the current DMA setting (2K or 8K messages) determine where in the DMA area the message should be fetched. All elements in the `satcan_msg` structure are overwritten with data fetched from the DMA area. This includes the initialized members msg.header[1:0] which should keep their original value when the `read(..)` call returns. The `read` function returns sizeof(satcan_msg) on success and -1 on failure. In the latter case **errno** is also set.

*Table 25.4. ERRNO values for **read** calls.*

ERRNO	Description
EINVAL	A NULL pointer was passed as the data pointer or the length was illegal.

25.2.5. Writing to the device

Transmission of messages are performed with the write call. It is possible to write one or several messages in each call. The driver copies the messages to be sent from the specified `satcan_msg` structures to the DMA area.

A call to **write(..)** has different behavior depending on the DMA mode of the driver. The DMA mode is set using an Input/Output Control call described later in this document.

When the driver is in SATCAN_DMA_MODE_SYSTEM a call to **write(..)** will block until the core signals that it has completed DMA. When the driver is in SATCAN_DMA_MODE_USER a call to **write(..)** will return

immediately after the data has been placed in the DMA area. The driver will not activate any of the DMA TX channels and start of DMA transfers are left to the user using Input/Output Control calls.

On success the **write(..)** call returns number of transmitted bytes and -1 on failure. Errno is also set in the latter case.

An example call sending a Enable Override message is shown below:

```
int i, ret;
satcan_msg msg;

msg.header[0] = 0xE0;
msg.header[1] = 0;
msg.header[2] = 0x81;
msg.header[3] = 0xFF;
msg.payload[0] = 15;
for (i = 1; i < SATCAN_PAYLOAD_SIZE; i++)
      msg.payload[i] = 0;
ret = write(fd, &msg, sizeof(satcan_msg));
if (ret != sizeof(satcan_msg))
      printf("Write of override msg failed\n");
```

*Table 25.5. ERRNO values for **write***

ERRNO	Description
EINVAL	An invalid argument was passed. The buffer length was not equal to the satcan_msg structure size or no DMA channel is enabled.
EIO	Transmit DMA is activated. The driver requires that the **write(..)** call exclusively controls the DMA TX channels.

25.2.6. I/O Control interface

Changing the behaviour of the driver for a device is done via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly. All supported commands and their data structures are defined in the SatCAN driver's header file `satcan.h`.

25.2.6.1. Data structures

The satcan_regmod structure shown below is used to read and modify core registers.

```
typedef struct {
      unsigned int reg;
      unsigned int val;
} satcan_regmod;
```

Table 25.6. Member in satcan_regmod structure

Member	Description
reg	Register to be read or modify. The allowed values for this member are listed further down in this document.
val	When reading a register this member is utilized to return the register value. When modifying a register this member should be initialized with the new register value or mask.

25.2.6.2. Configuration

The SatCAN core and driver are configured using **ioctl** calls. The table Table 25.3 below lists all supported **ioctl** calls. SATCAN_IOC_ should be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 25.2.

*Table 25.7. **ioctl** calls supported by the SatCAN FPGA driver.*

Call Number	Description
DMA_2K	Instructs driver and core to use 2K DMA mode. Default setting.

Call Number	Description
DMA_8K	Instructs driver and core to use 8K DMA mode.
GET_REG	Provides direct read access to all core registers.
OR_REG	Writes register with current register value logical or specified value.
AND_REG	Writes register with current register value masked with a specified value.
EN_TX1_DIS_TX2	Changes internal driver state to enable DMA TX channel 1, Disable DMA TX channel 2.
EN_TX1_DIS_TX1	Changes internal driver state to enable DMA TX channel 1, Disable DMA TX channel 2.
GET_DMA_MODE	Returns the current DMA transmit mode.
SET_DMA_MODE	Sets the DMA transmit mode.
ACTIVATE_DMA	Activates a specified DMA channel.
DEACTIVATE_DMA	Deactivates a specified DMA channel.
GET_DOFFSET	Gets the data offset used for writing TX DMA messages.
SET_DOFFSET	Sets the data offset used for writing TX DMA messages.
GET_TIMEOUT	Get time out value used when waiting for DMA TX completion.
SET_TIMEOUT	Set time out value used when waiting for DMA TX completion.

25.2.6.2.1. DMA_2K

This **ioctl** command instructs the SatCAN core and driver to use a DMA area with room for 2048 messages. The driver is initialized in this state by default. This call disables the RX DMA channel and allocates a new memory area. After the new memory area has been successfully allocated the RX DMA channel is re-enabled.

25.2.6.2.2. DMA_8K

This **ioctl** command instructs the SatCAN core and driver to use a DMA area with room for 8192 messages. This call disables the RX DMA channel and allocates a new memory area. After the new memory area has been successfully allocated the RX DMA channel is re-enabled. The drivers default setting is to use a DMA area with room for 2K messages. The example below shows how to instruct the driver to use an 8K DMA area:

```
 if (ioctl(fd, SATCAN_IOC_DMA_8K)) {
      printf("ERROR: Failed to enable 8K DMA area\n");
}
```

25.2.6.2.3. GET_REG

This call provides read access to all the core's registers. Note that reading a register may affect the hardware state and may impact the correct function of the driver. The GET_REG call takes an register and an return pointer as additional arguments. Valid register values are listed in table 1.9. Note that some of the registers listed in the table a write only and a SATCAN_IOC_GET_REG call will return the read register that occupies the corresponding address. An example of reading the SatCAN CmdReg1:

```
satcan_regmod regmod;

regmod.reg = SATCAN_CMD1;
if (ioctl(fd, SATCAN_IOC_GET_REG, &regmod))
     printf("Failed to read CMD1 register\n");
printf("CMD1 register value: 0x%08x\n", regmod.val);
```

The contents of the `satcan_regmod` structure has been previously described. The **reg** member is initialized with a value from table 1.9. The contents of the specified register is returned in the structure's val member.

Table 25.8. Values used together with GET_REG and SET_REG

Register constant	Register name
SATCAN_SWRES	Software reset

Register constant	Register name
SATCAN_INT_EN	Interrupt enable
SATCAN_FIFO	FIFO read
SATCAN_FIFO_RES	FIFO reset
SATCAN_TSTAMP	Current time stamp
SATCAN_CMDO	Command register 0
SATCAN_CMD1	Command register 1
SATCAN_START_CTC	Start cycle time counter
SATCAN_RAM_BASE	RAM offset address
SATCAN_STOP_CTC	Stop cycle time counter
SATCAN_DPS_ACT	DPS active status
SATCAN_PLL_RST	DPLL reset
SATCAN_PLL_CMD	DPLL command
SATCAN_PLL_STAT	DPLL status
SATCAN_PLL_OFF	DPLL offset
SATCAN_DMA	DMA channel enable
SATCAN_DMA_TX_1_CUR	DMA channel 1 TX current address
SATCAN_DMA_TX_1_END	DMA channel 1 TX end address
SATCAN_DMA_TX_2_CUR	DMA channel 2 TX current address
SATCAN_DMA_TX_2_END	DMA channel 2 TX end address
SATCAN_RX	CAN RX enable
SATCAN_FILTER_START	Filter start ID
SATCAN_FILTER_SETUP	Filter setup
SATCAN_FILTER_STOP	Filter stop ID
SATCAN_WCTRL	Wrapper status/control register
SATCAN_WIPEND	Wrapper interrupt pending register
SATCAN_WIMASK	Wrapper interrupt mask register
SATCAN_WAHADDR	Wrapper AHB address register

25.2.6.2.4. SET_REG

This call writes a given value to a specified register. Note that assigning a register may interfere with the correct operation of the driver software. An example of writing a register is given below:

```
printf("Reset PLL\n");
regmod.reg = SATCAN_PLL_RST;
regmod.val = 1;
if (ioctl(fd, SATCAN_IOC_SET_REG, &regmod))
     printf("Reset PLL failed\n"):
```

25.2.6.2.5. SET_REG

This call modifies a specified register by performing a bitwise logical or operation with the specified value and the current register value. Note that assigning a register may interfere with the correct operation of the driver software. An example of masking in a value to a register is given below:

```
printf("Enable sync pulse and sync message\n");
regmod.reg = SATCAN_CMD1;
regmod.val = 0x30;
if (ioctl(fd, SATCAN_IOC_OR_REG, &regmod))
     printf("Failed to enable sync pulse sync msg\n");
```

25.2.6.2.6. AND_REG

This call modifies a specified register by performing a bitwise logical and operation with the specified value and the current register value. Note that assigning a register may interfere with the correct operation of the driver software. The use of this call follows the same syntax as the OR_REG call, described above.

25.2.6.2.7. AND_REG

This call enables transmit DMA channel 1 and disabled transmit DMA channel 2. It does not immediately modify the hardware registers. The DMA channels are only enabled during a call to write. This ioctl call only modifies the internal state of the driver. The example below shows how to enable DMA TX channel 1:

```
if (ioctl(fd, SATCAN_IOC_EN_TX1_DIS_TX2)) {
     printf("Failed to enable DMA TX channel 1\n");
}
```

25.2.6.2.8. EN_TX2_DIS_TX1

This call enables transmit DMA channel 2 and disables transmit DMA channel 1. It does not immediately modify the hardware registers. The DMA channels are only enabled during a call to write. This **ioctl** call only modifies the internal state of the driver.

25.2.6.2.9. GET_DMA_MODE

This call returns the current DMA mode of the driver. The driver has two modes for DMA operation. User mode (SATCAN_DMA_MODE_USER) and system mode (SATCAN_DMA_MODE_SYSTEM). In user mode calls to `write(..)` will place the messages in the DMA area bit will not activate any of the DMA TX channels and return immediately. In system mode the driver will activate the selected DMA TX channel and the call to `write(..)` will block until the core signals that it has completed the DMA operation.

25.2.6.2.10. SET_DMA_MODE

This call sets the driver DMA mode. Available values are [SATCAN_DMA_MODE_USER] and [SATCAN_DMA_MODE_SYSTEM]. See the previous description of GET_DMA_MODE and the description of the `write(..)` call for more information about the modes. An example call using [SET_DMA_MODE] is shown below:

```
int val;
val = SATCAN_DMA_MODE_USER;
if (ioctl(fd, SATCAN_IOC_SET_DMA_MODE, &val))
      printf("Failed to set DMA mode\n");
```

25.2.6.2.11. ACTIVATE_DMA

This call activates one of the DMA TX channels when the driver is set to user DMA mode. The user can not activate a DMA channel using this call if the driver is in system DMA mode. An example call activating DMA TX channel 2 is shown below:

```
int val;
val = SATCAN_DMA_ENABLE_TX2;
if (ioctl(fd, SATCAN_IOC_ACTIVATE_DMA, &val))
      printf("Task1:Could not enable DMA TX channel 2\n");
```

25.2.6.2.12. DEACTIVATE_DMA

This call deactivates one of the DMA TX channels when the driver is set to user DMA mode. The user can not deactivate a DMA channel using this call if the driver is in system DMA mode. An example call deactivating DMA TX channel 2 is shown below:

```
int val;
val = SATCAN_DMA_ENABLE_TX2;
if (ioctl(fd, SATCAN_IOC_DEACTIVATE_DMA, &val))
      printf("Could not disable DMA TX channel 2\n");
```

25.2.6.2.13. GET_DOFFSET

This call sets the offset used when writing TX messages via calls to `write(..)`. TX DMA messages are written at start of DMA buffer + data offset. The argument to this call is a pointer to the integer containing the offset.

25.2.6.2.14. SET_DOFFSET

This call returns the offset used when writing TX messages via calls to `write(..)`. TX DMA messages are written at start of DMA buffer + data offset. The argument to this call is a pointer to an integer. The integer is assigned the current offset.

25.2.6.2.15. GET_TIMEOUT

This call returns the time out value that the `write(..)` call uses when waiting for TX DMA completion. The argument is a pointer to an [rtems_interval] type.

25.2.6.2.16. SET_TIMEOUT

This call sets the time out value that the `write(..)` call uses when waiting for TX DMA completion. The argument is a pointer to an [rtems_interval] type.

26. CAN_MUX driver (CAN_MUX)

26.1. Introduction

This document is intended as an aid in getting started developing with GRLIB CAN_MUX core using the driver described in this document. It briefly takes the reader through some of the most important steps in using the driver such as configuring the driver and using Input/Output-control calls to modify the hardware state. The reader is assumed to be well acquainted with the operation of the CAN_MUX core and RTEMS.

26.1.1. CAN_MUX Hardware

See the GR712RC or CAN_MUX core documentation.

26.1.2. Software Driver

The driver provides means for setting the CAN_MUX MUX control register.

26.1.3. Examples

The rtems-satcan example uses the CAN_MUX driver.

26.2. User interface

The RTEMS CAN_MUX driver supports the standard accesses to file descriptors such as **read**, **write** and **ioctl**. The implementation of read and write calls are dummy functions. The driver is controlled exclusively via **ioctl**. User applications should include the CAN_MUX driver's header file, `canmux.h`, which contains definitions of all necessary values and functions used when accessing the driver.

26.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The function `canmux_register` whose prototype is provided in `canmux.h` is used for registering the driver. The function returns 0 on success. A typical register call from the LEON3 Init task:

```
if ( canmux_register(&amba_conf) )
     printf("CAN_MUX register failed\n");
```

26.2.2. Opening the device

Opening the device enables the user to access the hardware of the CAN_MUX core. An example of an RTEMS **open** call is shown below.

```
fd = open("/dev/canmux", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 26.1.

*Table 26.1. Open **errno** values.*

ERRNO	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened

26.2.3. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the CAN_MUX driver.

26.2.4. I/O Control interface

The driver and hardware is controlled via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global errno variable is set accordingly. The CAN_MUX driver does not use any additonal data except for the integer that selects the **ioctl** function. All supported commands are defined in the CAN_MUX driver's header file `canmux.h` and are described further down in this document.

26.2.4.1. Configuration

The CAN_MUX core and driver is controlled using **ioctl** calls. The Table 26.3 below lists all supported **ioctl** calls. OCCAN_IOC_ should be concatenated with the call name from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 26.2.

*Table 26.2. ERRNO values for **ioctl** calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.

*Table 26.3. **ioctl** calls supported by the CAN_MUX driver.*

Call Number	Description
BUSA_SATCAN	Routes bus A to SatCAN core
BUSA_OCCAN1	Routes bus A to OC-CAN 1 core
BUSB_SATCAN	Routes bus B to SatCAN core
BUSB_OCCAN2	Routes bus B to OC-CAN 2 core

27. GRASCS driver

27.1. Introduction

This document is intended as an introduction to the RTEMS driver for the Gaisler ASCS core. It is recommended that the reader also has access to the GRASCS IP core documentation when reading this document.

27.1.1. Software driver

The driver allows the developer of application software to communicate with the GRASCS core. It supplies the functions to initialize the core, send and receive data, start and stop synchronization etc. The complete user interface is described in more detail in Section 27.2 below. The driver is thread safe with the following two exceptions: ASCS_etr_select, ASCS_TC_sync_start, and ASCS_TC_sync_stop can not be called from different threads, and ASCS_start and ASCS_stop can not be called from different threads. The driver supports all the different configurations of the GRASCS core that is mentioned in the GRASCS IP core documentation.

27.1.2. Examples

A demonstration software which shows how to use the driver interface is distributed together with the driver. The software initialize the core, start the serial and synchronization interfaces, perform data writes and data reads and then stops the interfaces again. The software has been developed for pure demonstration purposes and the effects of the transactions performed on a real ASCS slave are unknown.

27.2. User interface

In Table 27.1 all the functions of the GRASCS driver interface are listed. To gain access to the functions a user application should include the GRASCS driver's header file.

Table 27.1. GRASCS driver interface

Function name	Described in	Short description
ASCS_init	Section 27.2.1	Initializes driver and GRASCS core
ASCS_input_select	Section 27.2.2	Selects slave
ASCS_etr_select	Section 27.2.3	Select source for synchronization pulse
ASCS_start	Section 27.2.4	Starts serial interface
ASCS_stop	Section 27.2.5	Stops serial interface
ASCS_iface_status	Section 27.2.6	Report status of serial and synchronization interfaces
ASCS_TC_send	Section 27.2.7	Performs a data write (TC)
ASCS_TC_send_block	Section 27.2.8	Performs a number of TCs
ASCS_TC_sync_start	Section 27.2.9	Starts synchronization interface
ASCS_TC_sync_stop	Section 27.2.10	Stops synchronization interface
ASCS_TM_recv	Section 27.2.11	Performs a data read (TM)
ASCS_TM_recv_block	Section 27.2.12	Performs a number of TMs

27.2.1. ASCS_init

Prototype	int ASCS_init()
Argument	This function does not take any arguments
Return value:	0 on success, -1 on failure
Description:	This function must be called before any other functions in the ASCS driver are called. ASCS_init initializes the driver and resets the core. When the function returns all of the cores registers will have their default values, which means that both the serial interface and synchronization interface are stopped.

27.2.2. ASCS_input_select

Prototype	int ASCS_input_select(int slave)
Argument	slave
Description	The number of the slave the core should listen to during a TM
Return value:	0 on success, -GRASCS_ERROR_CAPFAULT if slave value is invalid, -GRASCS_ERROR_TRANSACTIVE if a TM is in progress
Description:	This function sets the bits in the core's command register that control which slave data input is valid during a TM. Valid range of the input 0 – (nslaves-1), where nslaves is the number of slaves the core has been configured to communicate with (nslaves generic).

27.2.3. ASCS_etr_select

Prototype	int ASCS_etr_select(int etr, int freq)

Argument	**Description**
etr	The source for the etr signal, valid range 0 - 6
freq	The ETR frequency in Hz

Return value:	0 on success, -GRASCS_ERROR_CAPFAULT if arguments have invalid values, -GRASCS_ERROR_STARTSTOP if synchronization interface is running
Description:	This function need to be called if the source of the ETR synchronization pulse should be changed. The etr input specifies which source to use, where 0 means internal counter and 1 – 6 means external time marker 1 - 6. The freq input specifies the frequency of the etr signal. If etr is not 0 then the freq argument need to be the same as the frequency of the external time marker that is used. The core can not generate an ETR pulse of one frequency from an external time marker of a different frequency. This function, ASCS_TC_sync_start and ASCS_TC_sync_stop can not be called from different threads.

27.2.4. ASCS_start

Prototype	int ASCS_start()
Argument	This function does not take any arguments
Return value:	None
Description:	A call to this function starts the core's serial interface and the core is then ready to perform transactions. This function and ASCS_stop can not be called from different threads.

27.2.5. ASCS_stop

Prototype	int ASCS_stop()
Argument	This function does not take any arguments
Return value:	None
Description:	A call to this function stops the core's serial interface. This function will block until any possible call to ASCS_TC_send, ASCS_TC_send_block, ASCS_TM_recv or ASCS_TM_recv_block has returned. This function and ASCS_start can not be called from different threads.

27.2.6. ASCS_iface_status

Prototype	int ASCS_iface_status()
Argument	This function does not take any arguments
Return value:	0 if both serial interface and synchronization interface are stopped, 1 if serial interface is running and synchronization interface is stopped, 2 if serial interface is stopped and synchronization interface is running, 3 if both interfaces are running.

Description:	Uses the internal driver status and the value of the core's status register to report if serial and synchronization interfaces are running or stopped.

27.2.7. ASCS_TC_send

Prototype	int ASCS_TC_send(int *word)
Argument word ntrans	**Description** Pointer to data that should be sent as a telecommands. The argument is handled as a point erto a short int if the core is configured to send 16-bit words, or a char pointer for 8-bit words. The number of telecommands that should be sent
Return value:	0 on success, -GRASCS_ERROR_TRANSACTIVE if TC could not be started because some other transaction is in progress, - GRASCS_ERROR_STARTSTOP if TC could not be started because serial interface is stopped.
Description:	Sends a telecommand with the data pointed to by the word argument. If the TC is started the function blocks until the transaction is complete. If the TC can not be started the function returns with an error code. This function is thread safe.

27.2.8. ASCS_TC_send_block

Prototype	int ASCS_TC_send_block(int *block,int ntrans)
Argument blcok ntrans	**Description** Pointer to the start a block of data that should be sent as a number of telecommands. The block argument is handled as a point erto a block of short int if the core is configured to send 16-bit words, or a char pointer for 8-bit words. The number of telecommands that should be sent
Return value:	0 on success, -GRASCS_ERROR_TRANSACTIVE if TC could not be started because some other transaction is in progress, - GRASCS_ERROR_STARTSTOP if TC could not be started because serial interface is stopped.
Description:	Sends a number of telecommands with the data pointed to be the block argument. If the first TC is started the function blocks until all the transaction are complete. If the first TC can not be started the function returns with an error code. This function is thread safe.

27.2.9. ASCS_TC_sync_start

Prototype	int ASCS_TC_sync_start(void)
Argument	This function does not take any arguments
Return value:	None
Description:	Starts the synchronization interface. There might be a delay between the time this function is called and the time the interface is actually started, depending on whether a TM is active or not. Software can poll ASCS_iface_status to find out when interface is running. The first pulse on the synchronization interface might be delay with up to one period depending on the source used for the ETR signal. This function, ASCS_TC_sync_stop and ASCS_etr_select can not be called from different threads.

27.2.10. ASCS_TC_sync_stop

Prototype	int ASCS_TC_sync_stop(void)
Argument	this function does not take any arguments
Return value:	None

Description:	Stops the synchronization interface. In order not to prematurely abort a ETR pulse there might be a delay between the time this function is called and the time the interface is actually stopped. Software can poll ASCS_iface_status to find out when the interface is stopped. This function, ASCS_TC_sync_start and ASCS_etr_select can not be called from different threads.

27.2.11. ASCS_TM_recv

Prototype	int ASCS_TM_recv(int *word)
Argument: word	**Description**: Pointer to where data received in a TM should be stored. The argument is handled as a short int pointer if the core is configured to send 16-bit words, or a char pointer for 8-bit words.
Return value:	0 on success, -GRASCS_ERROR_TRANSACTIVE if TM could not be started because some other transaction is in progress, -GRASCS_ERROR_STARTSTOP if TM could not be started because serial interface is stopped.
Description:	Starts a TM and stores the incoming data at the address word points to. If the TM can not be started the function returns with an error code otherwise it blocks until the transaction is complete. This function is thread safe.

27.2.12. ASCS_TM_recv_block

Prototype	int ASCS_TM_recv_block(int *block, int ntrans)
Argument: block	**Description**: Pointer to the start of a block where data received in a number of TMs should be stored. The block argument is handled as a point erto a block of short int if the core is configured to send 16- bit words, or a char pointer for 8-bit words.
Argument: ntrans	**Description**: The number of TMs that should be sent
Return value:	0 on success, -GRASCS_ERROR_TRANSACTIVE if TM could not be started because some other transaction is in progress, -GRASCS_ERROR_STARTSTOP if TM could not be started because serial interface is stopped.
Description:	Starts a number of TMs and stores the incoming data with the beginning of the address that block points to. If the first TM can not be started the function returns with an error code otherwise it blocks until all the transactions are complete. This function is thread safe.

27.3. Examples code

To use the GRASCS driver its header file should be included:

```
#include <grascs.h>
```

The driver must first be initialized, and the return value must be checked to see that the initialization went well:

```
status = ASCS_init();

  if(status < 0) {

     printf("ERROR: Failed to initialize ASCS driver\n");

     exit(0);

}

printf("Successfully intialized ASCS driver\n");
```

When the ASCS_init function has been called the application can start calling the other functions as well. Below is an example of how to call ASCS_TC_send_block and send ten TCs.

```
retval = ASCS_TC_send_block((int*)block,10);

if(retval < 0) {

  if(retval == -GRASCS_ERROR_STARTSTOP)

    printf("ERROR: Failed to start TC because serial interface never
started\n");

  else if(retval == -GRASCS_ERROR_TRANSACTIVE)

    printf("ERROR: Failed to start TC because a transaction is in
progress\n");

   }
```

28. APBUART console driver

28.1. Introduction

The RTEMS Driver Manager APBUART console driver is implemented using the RTEMS POSIX Termios support. POSIX Termios is defined by `IEEE Std 1003.1-2008 (POSIX.1-2008)` which provides an operating system independent interface to terminal devices. The RTEMS kernel provides the Termios specific parts of the interface, while the APBUART console driver implements the low-level hardware access.

Even though the APBUART driver is referred to as a console driver, and the Termios interface is a general terminal interface, the driver can also be used as a general purpose UART driver.

28.1.1. Functioning mode

A device attached to the APBUART console driver can operate in one of three different functioning modes.

- Polled Mode (default)
- Interrupt Driven Mode
- Task Driven Mode

Polled mode is used by default and will leave calling tasks in a poll loop to wait for the requested operation to complete. Interrupt driven mode uses the APBUART interrupt and FIFO functionality to prevent blocking the task using the driver. In task driven mode, the RTEMS Termios layer creates daemon tasks which are activated on APBUART interrupt and carries out the processing in task context.

The functioning mode is set at compile time using driver manager resources and can not be changed afterwards. For more information on the operating modes, see the document *RTEMS BSP and Device Driver Development Guide*.

28.1.2. Hardware support

APBUART devices are available in different hardware configurations with different capabilities. This driver uses the following capababilities, if their presence is detected by the driver.

- RX/TX interrupt (depending on functioning mode as described below)
- RX/TX FIFO

28.2. User interface

The driver supports the standard accesses to file descriptors such as `read()`, `write()` and to Termios control functions such as `tcgetattr()`, `tcsetattr()` and `cfsetispeed()`. User applications include the Termios header file `<termios.h>` which provides prototypes and definitions of all necessary data structures and bit masks used when accessing a Termios file.

User operations specific to the APBUART driver are documented here, while common Termios operations are documented in `IEEE Std 1003.1-2008 (POSIX.1-2008)`.

The APBUART console driver require the RTEMS Driver Manager.

28.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when UART hardware is found for the first time. The driver is called from the driver manager to handle detected UART hardware. In order for the driver manager to unite the UART driver with the UART hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

28.2.2. Driver resource configuration

The driver can be configured at compile tile using driver resources as described in the driver manager chapter. Below is a description of configurable driver parameters. The driver parameters are unique per APBUART device. The parameters are all optional, the parameters only overrides the default values.

Table 28.1. APBUART console driver resources

Name	Type	Parameter description
syscon	INT	Suggest system console System console is by default assigned to the first APBUART in the system. The user can override this default behaviour by setting the *syscon* parameter to 0 on the first APBUART to force it to not be the system console and 1 on another to suggest it as system console.
mode	INT	Functioning mode • 0: Polled Mode (default) • 1: Interrupt Driven Mode • 2: Task Driven Mode

28.2.3. Opening the device

Opening the device enables the user to access the hardware of a APBUART device. Devices are identified by a file name which is constructed based on device bus location and order of discovery. The file name is used for opening a device. Some example file names are given in the following table.

Table 28.2. Example APBUART file names

Device number	Filesystem name	Location
0	`/dev/console`	On-Chip Bus
1	`/dev/console_b`	On-Chip Bus
2	`/dev/console_c`	On-Chip Bus
Depends on system configuration	`/dev/rastaio0/apbuart0`	GR-RASTA-IO
Depends on system configuration	`/dev/rastaio0/apbuart1`	GR-RASTA-IO

An example of an RTEMS `open()` call is shown below.

```
fd = open("/dev/console_b", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case *errno* is also set as defined by `open()`.

28.2.4. Closing the device

The device is closed using the `close()` function. An example is shown below.

```
res = close(fd)
```

28.2.5. Control interface

Changing the behaviour of the driver for a device is done via the Termios terminal parameter access functions defined by `<termios.h>`, such as `tcsetattr()`.

Table 28.3 describes the *struct termios* field *c_cflag* for hardware control supported by the APBUART console driver.

Table 28.3. Supported control modes

Mask name	Description
CLOCAL	Enable/disable flow control
CSIZE	Only 8 bit data per character is supported (*CS8*)
PARENB	Parity enabled if set, else disabled.
PARODD	Odd parity if set, else even. The parameter is qualified by *PARENB*

The device always sends one stop bit after the optional parity bit.

APBUART BAUD can be set to all values defined by `<termios.h>`, ranging from *`B50`* to *`B460800`*. The driver calculates a suitable APBUART scaler register value using the APBUART core frequency and the requested BAUD. Note that the target BAUD and its precision may be limited by the APBUART scaler in combination with the bus frequency. See documentation of the APBUART SCALER register in the *GRLIB IP Core User's Manual*.

The following example sets the APBUART BAUD to 9600, 8 bits per character and disables parity:

```
struct termios options;

tcgetattr(fd, &options);
cfsetispeed(&options, B9600);
cfsetospeed(&options, B9600);
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;
options.c_cflag &= ~PARENB;
tcsetattr(fd, TCSANOW, &options);
```

28.2.6. Startup-parameter inheritance

When the APBUART device configured with the *`syscon`* parameter set to 1 is opened by the user, its present hardware attributes are preserved. The preserved attributes are the ones listed in Table 28.3. This allows for the user to configure the console settings in a boot loader without the need to synchronize the settings in the application. For APBUART devices other than the system console, the user should configure the device with the Termios interface after open.

The file `/dev/console` is opened automatically as part of the RTEMS initialization and is associated with the streams *`stdin`*, *`stdout`* and *`stderr`*.

28.2.7. Transmission

Transmitting characters to the UART serial line can be done with the `write()` function. It is possible to write multiple bytes in one call. An example of a `write()` is shown below:

```
result = write(fd, &buffer[0], sizeof(buffer));
```

On success the number of transmitted bytes is returned and -1 on failure. *`errno`* is also set in the latter case. *`buffer`* points to the beginning of the character byte array. The last parameter sets the number of bytes taken from *`buffer`* that will be transmitted.

28.2.8. Reception

Reception of characters from the UART serial line can be done using the `read()` function. An example is shown below:

```
char buffer[16];

len = read(fd, buffer, 16);
```

The requested number of bytes to be read is given in the third argument. The received bytes will be stored in *`buffer`*. The actual number of received bytes is returned by the function on success and -1 on failure. In the latter case *`errno`* is also set.

29. APBUART - Raw UART driver interface

29.1. Limitations

This driver has been deprecated. The source code is available in the RCC kernel source tree and the driver documentation is available here. However, the driver object files are not provided pre-built to mark it deprected. It is recommended to use the APBUART console driver instead that provides a termios interface. It is disqabled to limit the number of APBUART drivers available and the conflict having two drivers for the same hardware installed at the same time. The APBUART termios driver is documented in Chapter 28.

29.2. User interface

The RTEMS "Raw" UART driver supports the standard accesses to file descriptors such as **read**, **write** and **ioctl**. User applications include the **apbuart** driver's header file (apbuart.h) which contains definitions of all necessary data structures and bit masks used when accessing the driver.

The APBUART driver require the RTEMS Driver Manager.

The UART driver is an interrupt driven "raw" character stream driver with the ability to add a "carriage return" (\r in C) after a "new line" (\n in C) has been detected in the output stream.

The UART interrupt handler copies received characters to a receive FIFO buffer placed in RAM to avoid overruns. Characters are then read from the RAM buffer by calling **read**.

Writing a number of characters when the hardware transmitter is full results in that the driver puts the characters into a software FIFO buffer located in RAM to be sent later on by the transmitter interrupt handler.

29.2.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The RTEMS I/O driver registration is performed automatically by the driver when UART hardware is found for the first time. The driver is called from the driver manager to handle detected UART hardware. In order for the driver manager to unite the UART driver with the UART hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

29.2.2. Driver resource configuration

This driver does not have any configurable resources. All configuration can be made though the **ioctl** interface.

29.2.3. Opening the device

Opening the device enables the user to access the hardware of a certain APBUART device. The driver is used for all APBUART devices available. The devices are separated by assigning each device a unique name and a number called **minor**. The name is passed during the opening of the driver. Some example device names are printed out below.

Table 29.1. Device number to device name conversion

Device number	Filesystem name	Location
0	/dev/apbuart0	On-Chip Bus
1	/dev/apbuart1	On-Chip Bus
2	/dev/apbuart2	On-Chip Bus
Depends on system configuration	/dev/rastaio0/apbuart0	GR-RASTA-IO
Depends on system configuration	/dev/rastaio0/apbuart1	GR-RASTA-IO

An example of an RTEMS open call is shown below.

```
fd = open("/dev/apbuart0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 29.1.

*Table 29.2. Open **errno** values.*

ERRNO	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Device failed to allocate necessary memory

29.2.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (succes) for the **apbuart** driver.

29.2.5. I/O Control interface

Changing the behaviour of the driver for a device is done via the standard system call **ioctl**. Two arguments must be provided to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly.

All supported commands and their data structures are defined in the UART driver's header file `apbuart.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

29.2.5.1. Configuration

The UART core and driver are configured using **ioctl** calls. The Table 29.3 below lists all supported **ioctl** calls. APBUART_IOC_ must be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 29.2.

An example is shown below where the driver's read call changes behaviour. After this call the driver will block the calling thread until free space in the receiver's circular buffer are available:

```
result = ioctl(fd, APBUART_IOC_SET_BAUDRATE, 115200);
```

*Table 29.3. ERRNO values for **ioctl** calls*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The APBUART hardware is not in the correct state. **ioctl** calls may need the UART to be in stopped mode to function correctly. One can switch state by calling START or STOP.
ENOMEM	Not enough memory to complete operation. This may cause other **ioctl** commands to fail.

*Table 29.4. **ioctl** calls supported by the APBUART driver.*

Call Number	Call Mode	Description
START	Stopped	Exit paused mode, brings enables receiver and transmitter. Enables read and write.
STOP	Running	Exit operating mode. Disables read and write, but enables user to change FIFO depth.

Call Number	Call Mode	Description
SET_RX_FIFO_LEN	Stopped	Sets software receiver FIFO length in number of bytes.
SET_TX_FIFO_LEN	Stopped	Sets software transmitter FIFO length in number of bytes.
SET_BAUDRATE	Don't Care	Sets baud rate of a UART channel.
SET_SCALER	Don't Care	Sets the baud rate manually by setting the [scaler] register of the APBUART core.
SET_BLOCKING	Don't Care	Set receive (read), transmit (write) blocking mode and TX-Flush mode which blocks until all characters have bee put in-to software transmit FIFO.
GET_STATS	Don't Care	Store UART driver statistics to a user defined buffer.
CLR_STATS	Don't Care	Resets the statistic counters.
SET_ASCII_MODE	Don't Care	et/unset ASCII mode. When ASCII mode is enabled a new line is replaced with a new line and a carriage return. '\n' => '\n\r'

29.2.5.1.1. START

This **ioctl** command enables the receiver and transmitter of the UART core. Settings previously set by other **ioctl** commands are written to hardware just before entering running mode. It is necessary to enter running mode to be able to read or write to/from the UART.

The command will fail if software receive or transmit buffers are not correctly allocated or if the UART driver already is in running mode.

29.2.5.1.2. STOP

This call makes the UART hardware leave running mode and enter stopped mode. After calling STOP further calls to **read** and **write** will result in errors.

It is necessary to enter stopped mode to change operating parameters of the UART driver to safely change configuration such as FIFO buffer lengths.

29.2.5.1.3. SET_RXFIFO_LEN

Sets the software receive FIFO length. The argument specifies the number of bytes for the new RX FIFO buffer.

This command may return ENOMEM if not enough memory was available to complete the request, this will make calls to START fail until a new buffer is allocated with SET_RX_FIFO_LEN.

29.2.5.1.4. SET_TX_FIFO_LEN

Sets the software transmit FIFO length. The argument specifies the number of bytes for the new TX FIFO buffer.

This command may return ENOMEM if not enough memory was available to complete the request, this will make calls to START fail until a new buffer is allocated with SET_TX_FIFO_LEN.

29.2.5.1.5. SET_BAUDRATE

Sets the baud rate of the UART hardware by specifying the rate in number of bits/second as argument. The SCALER register of the UART hardware is calculated by the driver using the UART core frequency and the requested baud rate.

This command fails if an out of range baud rate is given, maximum 115200 bits/second.

29.2.5.1.6. SET_SCALER

Makes it possible for the user to set the baud rate of the UART hardware manually. The UART SCALER register is documented in the IP Core manual. The new scaler register value is given as argument to this command.

29.2.5.1.7. SET_BLOCKING

Sets receive, transmit or transmit flush blocking mode. The argument to SET_BLOCKING is a bitmask as described in the table below.

Table 29.5. SET_BLOCKING Argument Bit Mask

Bit mask name	Function
BLK_RX	If set, enables blocking mode for read calls.
BLK_TX	If set, enables blocking mode for write calls.
BLK_FLUSH	If set, enables TX Flush mode. Blocks thread calling **write** until all requested data has been put into hardware transmission FIFO or software transmit FIFO.

29.2.5.1.8. GET_STATS

Stores the current driver statistics counters to a user defined data area. A pointer to the data area must be provided as argument. -1 will be returned and **errno** set to EINVAL if a invalid pointer is given.

29.2.5.1.9. CLR_STATS

Resets drivers statistics counters.

29.2.5.1.10. SET_ASCII_MODE

Sets ASCII mode of the driver. A non-zero argument enabled ASCII mode. In ASCII mode a "new line" character is replace with a "carriage return" and a "new line". This makes it easier to work with terminals.

29.2.6. Transmission

Transmitting characters to the UART serial line can be done with the **write** call. It is possible to write multiple bytes in one call. An example of a write call is shown below:

```
result = write(fd, &buffer[0], sizeof(buffer));
```

On success the number of transmitted bytes is returned and -1 on failure. Errno is also set in the latter case. **buffer** points to the beginning of the character byte array. The last parameter sets the number of bytes taken from **buffer** that will be transmitted.

The write call can be configured to block when the software FIFO is full. In non-blocking mode write will immediately return either return -1 indicating that no data were written or the total number of bytes written are returned. Note that a write request of 3 characters may end up in only 2 written, the caller is responsible to check the number of messages actually written.

If no resources are available the call will return with an error in non-blocking mode. The errno variable is set according to the table given below.

*Table 29.6. ERRNO values for**write***

ERRNO	Description
EINVAL	An invalid argument was passed. The buffer length was less than a single CANMsg structure size.
EBUSY	The link is not in operating mode, but in reset mode. Nothing done.
ETIMEDOUT	In non-blocking mode and driver was unable to put any bytes into the software transmit FIFO or the hardware transmit buffer.

29.2.7. Reception

Reception of characters from the UART serial line can be done using the read call. An example is shown below:

```
char buffer[16];
```

```
len = read(fd, buffer, 16);
```

The requested number of bytes to be read is given in the third argument. The received bytes will be stored in **buffer**. The actual number of received bytes is returned by the function on success and -1 on failure. In the latter case **errno** is also set.

The call will fail if a null pointer is passed, invalid buffer length, the UART core is in stopped mode or because the UART receive FIFO is empty in non-blocking mode.

The blocking behaviour can be set using **ioctl** calls. In blocking mode the call will block until at least one byte has been received. In non-blocking mode, the call will return immediately and if no message was available -1 is returned and **errno** set appropriately. The table below shows the different **errno** values returned.

*Table 29.7. ERRNO values for**read** calls.*

ERRNO	**Description**
EINVAL	A NULL pointer was passed as the data pointer or the length was illegal.
EBUSY	CAN core is in reset mode. Switch to started mode by issuing a START **ioctl** command.
ETIMEDOUT	In non-blocking mode no messages were available in the software receive FIFO.

30. SPICTRL GRLIB SPI master driver

30.1. Introduction

This section describes the SPICTRL Master driver available for RTEMS. The SPICTRL driver provides the necessary functions needed by the RTEMS I2C Library. The RTEMS I2C Library is used for both I2C and SPI. The RTEMS I2C Library is not documented here.

The SPICTRL driver require the RTEMS Driver Manager.

30.1.1. SPI Hardware

The SPICTRL core is documented in the GR-IP core's manual. The driver supports multiple SPI cores.

30.1.2. Examples

There are two examples available, one that read and write data to a standard SPI FLASH and one that access a SD Card FAT file system. The SPI driver initialize the I2C Library when a SPI core is found and the application initialize the higher level drivers.

The examples are part of the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/rtems-spi.c and rtems-spi-sdcard.c`.

30.2. User interface

The RTEMS SPICTRL SPI driver supports the RTEMS I2C Library operations and the simultaneous read/write operation available using the **ioctl** interface. The driver is united with SPICTRL cores by the driver manager as SPICTRL cores are found. During driver initialization the SPI driver initializes the RTEMS I2C Library and registers the driver. The driver is registered with the name `/dev/spi1`, `/dev/spi2` and so on.

An example application using the driver is provided in the samples directory distributed with the toolchain.

30.2.1. Driver registration

The registration of the driver is needed in order for the RTEMS I2C Library to know about the SPI hardware driver. The RTEMS I2C driver registration is performed automatically by the driver when SPICTRL hardware is found for the first time. The driver is called from the driver manager to handle detected SPICTRL hardware. In order for the driver manager to unite the SPICTRL driver with the SPICTRL hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

30.2.2. Accessing the SPI bus

The SPI bus can be accessed direct in RAW mode or by using a so called high level driver. The high level drivers must be connected with the SPICTRL driver by using the **rtems_libi2c_register_drv** function. The SD Card higher level driver does this automatically where as the memory driver needs the user to do this before initializing the memory driver. The location of the higher level drivers and the RTEMS I2C Library is indicated in Table 30.1. All paths are given relative the RTEMS kernel source root.

Table 30.1. SPI source location

Source description	Location
I2C Library	cpukit/libi2c
High level drivers	c/src/libchip/i2c
SPICTRL driver	c/src/lib/libbsp/sparc/shared/spi

When accessing the driver in RAW mode a device node must be created manually in the file system by calling `rtems_filesystem_make_dev_t` and `mknod` with the correct major and minor number identifying the SPICTRL driver. The major number must be the same as the RTEMS I2C Library I/O driver major number,

the minor number identify the SPICTRL driver. The macro RTEMS_LIBI2C_MAKE_MINOR can be used to generate a valid minor number.

After a device node is created either manually for the RAW mode or by I2C Library for the higher level driver the device node can be accessed using standard means such as **open**, **close**, **read**, **write** and **ioctl**.

30.2.3. Extensions to the standard RTEMS interface

The SPICTRL core supports automated periodic transfers if enabled in the hardware design. The driver provides means for accessing the extra features that the SPICTRL core implements through the **ioctl** interface. The additional features are optional, when ignored the driver operates as a standard RTEMS SPI driver.

The extra **ioctl** commands supported are listed in the table below. In periodic mode the SPI core is setup to execute one SPI request multiple times, each transfer is started on a constant interval or when an external trigger pulse is detected. In normal operation read and writes are done simultaneously, however in the automated (AM) periodic transfer mode multiple transfers are executed. Once the core has been set up to operate in periodic mode (via CONFIG), `libi2c_write()` and `ibi2c_read()` are replaced with calls to PERIOD_READ/PERIOD_WRITE `ioctl()`. In periodic mode the TX/RX FIFO can not be read, instead receive and transmit registers let us peek into the FIFO. Up to four mask registers controls which TX/RX registers are part of the transfers. Please see the SPICTRL hardware document for an overview of the AM periodic mode.

Table 30.2. Additional ioctl commands

Command	Description
PERIOD_START	Start periodic transfers
PERIOD_STOP	Stop periodic transfers
PERIOD_READ	Read receive registers and mask registers, in periodic mode only
PERIOD_WRITE	Write receive registers and mask registers, in periodic mode only
CONFIG	Configure periodic and non periodic transfers
STATUS	Return the current status, the event register of the core

Below is an example of the steps that can be used when accessing the driver in periodic mode.

1. libi2c_send_start()
2. libi2c_ioctl(SET_TRFMODE)
3. lib2ic_send_address()
4. libi2c_ioctl(CONFIG, &config) Enable periodic mode, configure SPICTRL periodic transfer options
5. libi2c_ioctl(PERIOD_WRITE, &period_io) Fills TX Registers and set MASK registers, note that this has some constraints. The content written here will be transmitted over and over again, according to the MASK register.
6. lib2ic_ioctl(PERIOD_START) Starts the periodic transmission of the content in the TX Registers selected by the MASK register
7. lib2ic_ioctl(PERIOD_READ, &period_io) Read one response of the transmitted data. It will hang until data is available. If hanging is not an option use lib2ic_ioctl(STATUS) to determine on beforehand if it will hang.
8. OPTIONAL: libi2c_ioctl(PERIOD_WRITE, &period_io) The transmitted data on the SPI wires can be changed by calling the PERIOD_WRITE, note that this method requires that TX registers beeing used are not overwritten.
9. Go back to 7. to read the content of one more transfer, stop by stepping to 10.
10. libi2c_ioctl(STOP) Stop to set up a new periodic or normal transfer.
11. libi2c_stop()

30.2.3.1. PERIOD_START

Start previously configured automatic periodic transfers. Starting periodic transfers can only be done after CONFIG has been called enabling automated periodic transfers, and after PERIOD_WRITE has been called to set up the MASK and TX registers. Once the transfers has been started STATUS can be called to indicate the current transfer status and PERIOD_READ can be called to read the current content of the receive registers.

30.2.3.2. PERIOD_STOP

Stops any ongoing period transfer by writing zero to the AM configuration register.

30.2.3.3. CONFIG

Configures the SPICTRL core in normal operation or in periodic operation. If periodic mode is enabled driver configure the periodic mode options by looking at the user provided argument, the argument is assumed to be a pointer to `spictrl_ioctl_config` data structure with the layout and properties indicated below.

```
/* SPICTRL_IOCTL_CONFIG argument */
struct spictrl_ioctl_config {
       int           clock_gap;
       unsigned int  flags;
       int           periodic_mode;
       unsigned int  period;
       unsigned int  period_flags;
       unsigned int  period_slvsel;
};
```

Table 30.3. spictrl_ioctl_config field description

Field	Description
clock_gap	Clock GAP on SPI bus between words, the FIFO word size is dependent on the software configuration
flags	Hardware options, such as enable Clock GAP and TAC mode
periodic_mode	non-zero enables automated periodic transfers
period	The period that might be used in periodic transfers
period_flags	AM Configuration register content. ACT bit has no effect. This controls the behaviour of libi2c_read().
period_slvsel	Slave chip select when no transfer is active.

30.2.3.4. STATUS

Copies the Event register of the SPICTRL core to a user provided buffer.

30.2.3.5. STATUS

Configures the SPICTRL TX and MASK registers. The registers are only used in periodic mode. The command may be called before or during periodic transfers are ongoing. The MASK register selects which registers will be used in the transfer process. Please see the SPI core hardware documentation how periodic mode is used.

Note that changing TX registers used in current transfers may create invalid SPI commands. One can make sure this does not happen by only changing content of unused TX registers, or by stopping the ongoing periodic transfers with PERIOD_STOP.

The command takes one argument, the argument is assumed to be a pointer to a `spictrl_period_io` data structure with the layout and properties indicated below.

The transmit register [N*32+M] corresponds to bit: masks[N] & (1<<M) .

```
/* SPICTRL_IOCTL_PERIOD_READ and SPICTRL_IOCTL_PERIOD_WRITE argument*/
struct spictrl_period_io {
       int             options;
       unsigned int    masks[4];
       void            *data;
};
```

Table 30.4. spictrl_period_io field description

Field	Description		
options	Selects operation performed by command		
	READ	BIT0	1=Read Mask registers into masks[].

Field	Description		
		BIT1	1=Read receive registers and store into data array. Only the registers specified by masks[] will be read. Note that the received registers are read after the masks[] registers has been updated, which if BIT0 will result in the active registers will be read into data.
	WRITE	BIT0	1=Write Mask registers with content of masks[]. Note that the MASK registers will be updated after the Transmit registers has been written.
		BIT1	1=Write transmit registers with values taken from data array. Only the registers specified by masks[] will be written/updated.
masks	An array of 4 32-bit words. Each bit corresponds to a Transmit or a Receive register. The masks array can be read from MASK registers or stored to MASK registers (if BIT0 is set), or only used to indicate which Transmit/Receive registers that should be Written/Read (if BIT0 is zero).		
data	Pointer to data array read (PERIOD_WRITE) or written (PERIOD_READ). The element size of the array depends on the configured word size (see CONFIG). The element size is either 8, 16 or 32-bits, the smallest possible that still fits the data words.		

The data pointer points to data in the format of an array with the same element size as the transfer bit-length configured. For example a 8-bit config will result in data being interpreted as an array of bytes, a 12-bit config in an array of 16-bit words etc. The order of the elements will be determined by: the lowest bit set in the mask will be the first, the second lowest the second in the array etc.

30.2.3.6. PERIOD_READ

This command Read the MASK registers and/or reads the Receive registers. The behaviour is controlled with **ioctl()** the argument provided by the user. The argument is a pointer to a data structure of the format `spictrl_period_io` described in Table 30.2.

By setting options to 0x3 will make the command read the receive registers activated only. The receive register [N*32+M] corresponds to bit: masks[N] & (1<<M).

31. I2CMST GRLIB I2C Master driver

31.1. Introduction

This section describes the I2C Master driver available for RTEMS. The I2CMST driver provides the necessary functions needed by the RTEMS I2C Library. The RTEMS I2C Library is not documented here.

The I2CMST driver require the RTEMS Driver Manager.

31.1.1. I2C Hardware

The I2CMST core is documented in the GR-IP core's manual. The driver supports multiple I2C cores.

31.1.2. Examples

There is an example available, it illustrates how to set up the I2C driver, initialize the I2C Library and access an I2C EEPROM. The EEPROM can be accessed with on of two different methods, either RAW mode or by using the high level driver.

The example is part of the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/rtems-i2cmst.c`.

31.2. User interface

The RTEMS I2CMST I2C driver supports the RTEMS I2C Library operations. The driver must be registered before it can be used. During driver registration the I2C driver initializes the RTEMS I2C Library and registers the driver. The driver is registered with the name /dev/i2c1, /dev/i2c2 and so on.

An example application using the driver is provided in the samples directory distributed with the toolchain.

31.2.1. Driver registration

The registration of the driver is needed in order for the RTEMS I2C Library to know about the I2CMST hardware driver. The RTEMS I2C driver registration is performed automatically by the driver when I2CMST hardware is found for the first time. The driver is called from the driver manager to handle detected I2CMST hardware. In order for the driver manager to unite the I2CMST driver with the I2CMST hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

31.2.2. Accessing the I2C bus

The I2C bus can be accessed direct in RAW mode or by using a so called high level driver. The high level drivers must be connected with the I2CMST driver by using the `rtems_libi2c_register_drv` function. The location of the higher level drivers and the RTEMS I2C Library is indicated in table 132. All paths are given relative the RTEMS kernel source root.

Table 31.1. I2C source location

Source description	Location
I2C Library	cpukit/libi2c
High level drivers	c/src/libchip/i2c
I2CMST driver	c/src/lib/libbsp/sparc/shared/i2c

When accessing the driver in RAW mode a device node must be created manually in the file system by calling `rtems_filesystem_make_dev_t` and `mknod` with the correct major and minor number identifying the I2CMST driver. The major number must be the same as the RTEMS I2C Library I/O driver major number, the minor number identify the I2CMST driver. The macro RTEMS_LIBI2C_MAKE_MINOR can be used to generate a valid minor number.

After a device node is created either manually for the RAW mode or by I2C Library for the higher level driver the device node can be accessed using standard means such as **open**, **close**, **read**, **write** and **ioctl**.

32. GPIO Library

32.1. Introduction

This section describes the GPIO Library available for RTEMS. The GPIO Library implements a simple function interface that can be used to access individual GPIO ports. The GPIO Library provides means to control and connect an interrupt handler for a particular GPIO port. The library itself does not access the hardware directly but through a GPIO driver, for example the GRGPIO driver. A driver must implement a couple of function operations to satisfy the GPIO Library. The drivers can register GPIO ports during runtime.

The two interfaces the GPIO Library implements can be found in the **gpiolib** header file (`gpiolib.h`), it contains definitions of all necessary data structures, bit masks, procedures and functions used when accessing the hardware and for the drivers implement GPIO ports.

This document describes the user interface rather than the driver interface.

32.1.1. Examples

There is an example available in the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/rasta-adcdac/gpio-demo.c`.

32.2. Driver interface

The driver interface is not described in this document.

32.3. User interface

The GPIO Library provides the user with a function interface per GPIO port. The interface is declared in `gpiolib.h`. GPIO ports are registered by GPIO drivers during runtime, depending on the registration order the GPIO port are assigned a port number. The port number can be used to identify a GPIO port. A GPIO port can also be referenced by a name, the name is assigned by the GPIO driver and is therefore driver dependent and not documented here.

GPIO ports which does not support a particular feature, for example interrupt generation, return error codes when tried to be accessed.

The location of the GPIO Library is indicated in Table 32.1. All paths are given relative the RTEMS kernel source root.

Table 32.1. GPIOLIB source location

Source description	Location
Interface implementation	c/src/lib/libbsp/sparc/shared/gpio/gpiolib.c
Interface declaration	c/src/lib/libbsp/sparc/shared/include/gpiolib.h

32.3.1. Accessing a GPIO port

The interface for one particular GPIO port is initialized by calling `gpiolib_open` with a port number or `gpiolib_open_by_name` with the device name identifying one port. The functions returns a pointer used when calling other functions identifying the opened GPIO port. If the device name can not be resolved to a GPIO port the open function return NULL. The prototypes of the initialization routines are shown below:

```
void *gpiolib_open(int port)

void *gpiolib_open_by_name(char *devName)
```

Note that this function must be called first before accessing other functions in the interface.

Note that the port naming is dependent of the GPIO driver used to access the underlying hardware.

32.3.2. Interrupt handler registration

Interrupt handlers can be installed to handle events as a result to GPIO pin states or state changes. Depending on the functions supported by the GPIO driver four interrupt modes are available, edge triggered on falling or rising edge and level triggered on low or high level. It is possible to register a handler per GPIO port by calling `gpiolib_irq_register` setting the arguments correctly as described in Table 32.2. Below is the prototype for the IRQ handler (ISR) install function.

```
int gpiolib_irq_register(
  void *handle,
  void *func,
  void *arg
)
```

The function takes three arguments described in the table below.

Table 32.2. gpiolib_irq_register argument description

Name	Description
handle	Handle used internally by the function interface, it is returned by the open function.
func	Pointer to interrupt service routine which will be called every time an interrupt is generated by the GPIO hardware.
arg	Argument passed to the `func ISR` function when called as the second argument.

To enable interrupt, the hardware needs to be initialized correctly, see functions described in the function prototype section. Also the interrupts needs to be unmasked.

32.3.3. Data structures

The data structure used to access the hardware directly is described below. The data structure `gpiolib_config` is defined in `gpiolib.h`.

```
 struct gpiolib_config {
     char mask;
     char irq_level;
     char irq_polarity;
}
```

Table 32.3. gpiolib_config members

Member	Description
mask	Mask controlling GPIO port interrupt generation 0 – Mask interrupt 1 – Unmask interrupt
irq_level	Level or Edge triggered interrupt 0 – Edge triggered interrupt 1 – Level triggered interrupt
irq_polarity	Polarity of edge or level 0 – Low level or Falling edge 1 – High level or Rising edge

32.3.4. Function prototype description

32.3.4.1. GPIO Library functions

A short summary to the functions are presented in the prototype lists below.

Table 32.4. GPIO per port functions

Prototype Name
void gpiolib_close(void *cookie)
int grpiolib_set_config (void *cookie, struct gpiolib_config *cfg)
int gpiolib_set (void *handle, int dir, int val)
int gpiolib_get(void *handle, int *inval)
int gpiolib_irq_clear(void *handle)
int gpiolib_irq_enable(void *handle)
gpiolib_irq_disable(void *handle)
int gpiolib_irq_force(void *handle)
int gpiolib_irq_register (void *handle, void *func, void *arg)
void gpiolib_show(int port, void *handle)

All functions takes a handle to a opened GPIO port by the argument handle. The handle is returned by the `gpiolib_open or gpiolib_open_by_name` function.

If a GPIO port does not support a particular operation, a negative value is returned. On success a zero is returned.

32.3.4.1.1. grpiolib_set_config

Configures one GPIO port according to the the `gpiolib_config` data structure.

The `gpiolib_config` structure is described in Table 32.3.

32.3.4.1.2. grpiolib_set

Set one GPIO port in output or input mode and set the GPIO Pin value. The third argument may not be used when [dir] indicated input. The direction of the GPIO port is controlled by the [dir] argument, 1 indicates output and 0 indicates input. The value driven by the GPIO port may be low by setting [val] to 0 or high by setting [val] to 1.

32.3.4.1.3. grpiolib_get

Get the input value of a GPIO port. The value is stored into the address indicated by the argument [inval].

32.3.4.1.4. grpiolib_irq_clear

Acknowledge any interrupt at the interrupt controller that the GPIO port is attached to. This may be needed in level sensitive interrupt mode.

32.3.4.1.5. grpiolib_irq_force

Force an interrupt by writing to the interrupt controller that the GPIO port is attached to.

32.3.4.1.6. grpiolib_irq_enable

Unmask GPIO port interrupt on the interrupt controller the GPIO port is attached to. This enables GPIO interrupts to pass though to the interrupt controller.

32.3.4.1.7. grpiolib_irq_disable

Mask GPIO port interrupt on the interrupt controller the GPIO port is attached to. This disable interrupt generation at the interrupt controller.

32.3.4.1.8. grpiolib_irq_register

Attaches a interrupt service routine to a GPIO port. Described separately above.

33. GRGPIO GRLIB GPIO driver

33.1. Introduction

This section describes the GRGPIO driver available for RTEMS. The GRGPIO driver provides the necessary functions needed by the GPIO Library. The GPIO Library is documented in Chapter 32.

The GRGPIO driver require the RTEMS Driver Manager.

33.1.1. GPIO Hardware

The GRGPIO core is documented in the GR-IP Core User's manual. The driver supports multiple GPIO cores.

The hardware may be configured to support interrupt generation on any combination of GPIO ports. The driver will fail with a return code when an interrupt is unmasked but the GPIO port does not support interrupt generation.

33.1.2. Examples

There is an example available in the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/rtems-gpio.c`.

33.2. User interface

The RTEMS GRGPIO GPIO driver supports the GPIO Library operations. The driver is united with GRGPIO cores by the driver manager as GRGPIO cores are found. During driver initialization the GPIO driver initializes the GPIO Library and registers the driver. Each GPIO port is handled separately using the GPIO Library.

An example application using the driver is provided in the samples directory distributed with the toolchain.

33.2.1. Driver registration

The registration of the driver is needed in order for the GPIO Library to know about the GPIO hardware driver. The GPIO driver registration is performed automatically by the driver when GRGPIO hardware is found for the first time. The driver is called from the driver manager to handle detected GRGPIO hardware. In order for the driver manager to unite the GRGPIO driver with the GRGPIO hardware one must register the driver to the driver manager. This process is described in the driver manager chapter.

33.2.2. Driver resource configuration

The driver can be configured using driver resources as described in the driver manager chapter. Below is a description of configurable driver parameters. The driver parameters is unique per GRPWM device. The parameters are all optional, the parameters only overrides the default values or behaviour.

Table 33.1. GRGPIO driver parameter description

Name	Type	Parameter description
nBits	INT	Tells the driver how many GPIO ports are available on this device, normally the driver auto detect the number of GPIO ports. The OUTPUT register of the GRGPIO core must be written in order to auto detect the number of GPIO ports, this can be a problem in some cases when the GPIO ports has already been initialized by the boot loader.
bypass	INT	This parameter specifies the BYPASS register content. If not available zero is written into the BYPASS register during driver initialization.

33.2.3. Accessing GPIO ports

The GPIO ports are accessed using the GPIO Library. Each GPIO port has a unique number which is assigned in the order the GPIO ports are registered. The GRGPIO GPIO ports are registered core wise, the first core in AMBA Plug & Play is registered first starting with PIO[0] to PIO[N], then all GPIO ports of the next GRGPIO core. See table below for an example.

Table 33.2. GRGPIO registration order

GRGPIO Core	GRGPIO I/O port	Registration order
0	PIO[0]	0
0	PIO[1]	1
0	PIO[2]	2
0	PIO[3]	3
0	PIO[4]	4
0	PIO[5]	5
0	PIO[6]	6
0	PIO[7]	7
1	PIO[0]	8
1	PIO[1]	9
1	PIO[2]	10
1	PIO[3]	11
1	PIO[4]	12
1	PIO[5]	13
1	PIO[6]	14
1	PIO[7]	15
2	PIO[0]	16
2	PIO[1]	17
2	PIO[2]	18
2	PIO[3]	19
2	PIO[4]	20
2	PIO[5]	21
2	PIO[6]	22
2	PIO[7]	23

The ports can also be referenced by using their names. The GRGPIO driver name the GPIO ports according to the following string,

```
"/dev/[SYSTEM_PREFIX]grgpio[SYSTEM_CORE_NR]/[PORT_NR]"
```

Table 33.3. GRGPIO registration order

MACRO	Description
SYSTEM_PREFIX	In systems where multiple AMBA buses exists it is convenient to reference a particular AMBA bus by a name. SYSTEM_PREFIX is substituted with the AMBA bus name that the GPIO core is attached to, for example on a GR-RASTA-IO PCI Target the AMBA bus is called rastaioN.
SYSTEM_CORE_NR	The core number on a particular AMBA system
PORT_NR	The port number on a particular GPIO core

The location of the GRGPIO drivers and the GPIO Library is indicated in table 137. All paths are given relative the RTEMS kernel source root.

Table 33.4. GRGPIO registration order

Source description	Location
GPIO Library	c/src/lib/libbsp/sparc/shared/gpio/gpiolib.c

Source description	Location
GRGPIO driver	c/src/lib/libbsp/sparc/shared/gpio/grgpio.c

34. GRADCDAC GRLIB ADC/DAC driver

34.1. Introduction

This section describes the GRADCDAC driver available for RTEMS. The GRADCDAC driver provides a function interface to the user with the ability to access the hardware directly. User applications include the `gradcdac` header file (`gradcdac.h`) which contains definitions of all necessary data structures, bit masks, procedures and functions used when accessing the hardware.

The GRADCDAC driver require the RTEMS Driver Manager.

34.1.1. ADC/DAC Hardware

The GRADCDAC core is documented in the GR-IP Core User's manual. The driver supports multiple GRADCDAC cores.

The GRADCADC core has two different IRQs, one ADC interrupt and one DAC interrupt.

34.1.2. Examples

There is an example available in the Gaisler RTEMS distribution, it can be found under `/opt/rtems-4.10/src/samples/rasta-adcdac/gradcdac-demo.c`.

34.2. User interface

The RTEMS GRADCDAC ADC/DAC driver provides the user with a function interface. The interface is declared in `gradcdac.h`. The driver is united with GRADCDAC cores by the driver manager as GRADCDAC cores are found. During driver initialization the ADCDAC driver initializes the ADC/DAC hardware to an initial state, for that point and onwards the function interface can be used to access the ADC/DAC hardware registers.

An example application using the driver is provided in the `samples/rasta-adcdac` directory distributed with the toolchain.

The location of the GRADCDAC driver is indicated in Table 34.1. All paths are given relative the RTEMS kernel source root.

Table 34.1. GRGPIO registration order

Source description	Location
GREDCDAC driver	c/src/lib/libbsp/sparc/shared/analog/gradcdac.c
Driver Interface	c/src/lib/libbsp/sparc/shared/include/gradcdac.h

34.2.1. Driver registration

The GRADCDAC is registered to the Driver Manager layer by setting the correct define in the project set up, see Driver Manager section.

The driver does not implement a I/O driver interface so the GRADCDAC does not register itself as a I/O driver, it implements a custom function interface that is available to the user.

34.2.2. Driver resource configuration

The driver does not support configurable resource parameters.

34.2.3. Accessing ADC/DAC

The Interface for one particular ADC/DAC core is initialized by calling `gradcdac_open` with the device name identifying one core. The function returns a pointer used when calling other functions identifying the opened ADC/

DAC core. If the device name can not be resolved to a ADC/DAC core the open function return NULL. The prototype of the initialization routine is shown below:

```
void *gradcdac_open(char *devname)
```

Note that this function must be called first before accessing other functions in the interface.

The GRADCDAC cores are be referenced by using their names, the names are generated according to the following string,

```
"/dev/[SYSTEM_PREFIX]gradcdac[SYSTEM_CORE_NR]"
```

Table 34.2. GRADCDAC core naming

MACRO	Description
SYSTEM_PREFIX	In systems where multiple AMBA buses exists it is convenient to reference a particular AMBA bus by a name. SYSTEM_PREFIX is substituted with the AMBA bus name that the ADC/DAC core is attached to, for example on a GR-RASTA-ADCDAC PCI Target the AMBA bus is called rastaadcdacN. This string is empty when the GRADCDAC is on the system AMBA bus.
SYSTEM_CORE_NR	The core number on a particular AMBA system.

34.2.4. Interrupt handler registration

Interrupt handlers can be installed to handle events as a result to AD/DA conversions. It is possible to register a handler for AD and or DA conversions by setting the adc argument appropriately as described in Table 34.3. Below is the prototype for the IRQ handler (ISR) install function.

```
int gradcdac_install_irq_handler(
  void *cookie,
  int adc,
  void (*isr)(int irq, void *arg),
  void *arg
)
```

The function takes three arguments described in the table below.

Table 34.3. gradcdac_install_irq_handler argument description

<table>
<tr><th>Name</th><th>Description</th></tr>
<tr><td>cookie</td><td>Handle used internally by the function interface, it is returned by the open function.</td></tr>
<tr><td>adc</td><td>
<table>
<tr><th>Value</th><th>Function</th></tr>
<tr><td>1</td><td>Register handler to ADC interrupt</td></tr>
<tr><td>2</td><td>Register handler to DAC interrupt</td></tr>
<tr><td>3</td><td>Register to both ADC and DAC interrupts</td></tr>
</table>
</td></tr>
<tr><td>isr</td><td>Pointer to interrupt service routine which will be called every time an interrupt is generated by the ADC/DAC hardware.</td></tr>
<tr><td>arg</td><td>Argument passed to the isr function when called as the second argument.</td></tr>
</table>

To enable interrupt the hardware needs to be initialized correctly see functions described in the function prototype section. Also the AD and or DA interrupts needs to be unmasked.

34.2.5. Data structures

The data structure used to access the hardware directly is described below. The data structure `gradcdac_regs` is defined in `gradcdac.h`.

```
struct gradcdac_regs {
     volatile unsigned int config;
     volatile unsigned int status;
     int unused0[2];
```

```
    volatile unsigned int adc_din;
    volatile unsigned int dac_dout;
    int unused1[2];
    volatile unsigned int adrin;
    volatile unsigned int adrout;
    volatile unsigned int adrdir;
    int unused2[1];
    volatile unsigned int data_in;
    volatile unsigned int data_out;
    volatile unsigned int data_dir;
}
```

The `gradcdac_config` data structure is used to read and write the ADC/DAC controllers configuration register.

```
struct gradcdac_config {
  unsigned char dac_ws;
  char wr_pol;
  unsigned char dac_dw;
  unsigned char adc_ws;
  char rc_pol;
  unsigned char cs_mode;
  char cs_pol;
  char ready_mode;
  char ready_pol;
  char trigg_pol;
  unsigned char trigg_mode;
  unsigned char adc_dw;
};
```

Table 34.4. gradcdac_config member and ADCONF reg definition

Member	Member type	ADCONF Bit start	Description
dac_ws	5-bit int	19	Number of DAC wait states.
wr_pol	Boolean	18	Polarity of DAC write strobe 0 Active low 1 Active High
dac_dw	2-bit selection	16	DAC data width 0 none 1 8-bit ADDATA [0:7] 2 16-bit ADDATA [0:15] 3 none/spare
adc_ws	5-bit int	11	Number of DAC wait states.
rc_pol	Boolean	10	Polarity of ADC read convert 0 Active low read 1 Active high read
cs_mode	2-bit selection	8	Mode of ADC chip select asserted ... 0 during conversion and read phases 1 during conversion phase

Member	Member type	ADCONF Bit start	Description
			2 during read phase 3 continuously during both phases
cs_pol	Boolean	7	Polarity of ADC chip select 0 Active low 1 Active high
ready_mode	Boolean	6	Mode of ADC ready 0 Falling edge 1 Rising edge
ready_pol	Boolean	5	Polarity of ADC ready 0 unused, open-loop 1 used, with time-out
trigg_pol	Boolean	4	Polarity of ADC triggers 0 falling edge 1 rising edge
trigg_mode	2-bit selection	2	ADC trigger source 0 none 1 ADTrig 2 32-bit Timer 1 3 32-bit Timer 2
adc_dw	2-bit selection	0	ADC data width 0 none 1 8-bit ADDATA[7:0] 2 16-bit ADDATA[15:0] 3 none/spare

34.2.6. Function prototype description

34.2.6.1. General ADC/DAC functions

A short summary to the functions are presented in the prototype lists below.

Table 34.5. General ADC/DAC functions

Prototype Name
void gradcdac_set_config (void *cookie, struct gradcdac_config *cfg)
void gradcdac_get_config (void *cookie, struct gradcdac_config *cfg)

Prototype Name
void gradcdac_set_cfg (void *cookie, unsigned int config)
unsigned int gradcdac_get_cfg(void *cookie)
unsigned int gradcdac_get_status(void *cookie)
void gradcdac_adc_convert_start(void *cookie)
unsigned int gradcdac_get_adrinput (void *cookie)
unsigned int gradcdac_get_adroutput (void *cookie)
void gradcdac_set_adroutput (void *cookie, unsigned int output)
unsigned int gradcdac_get_adrdir(void *cookie)
void gradcdac_set_adrdir (void *cookie, unsigned int dir)
unsigned int gradcdac_get_datainput (void *cookie, void)
unsigned int gradcdac_get_dataoutput (void *cookie, void)
void gradcdac_set_dataoutput (void *cookie, unsigned int output)
unsigned int gradcdac_get_datadir (void *cookie, void)
void gradcdac_set_datadir (void *cookie, unsigned int dir)

All functions takes a handle to the ADC/DAC core by the argument cookie. The handle is returned by the `gradcdac_open` function.

34.2.6.1.1. gradcdac_set_config

Writes the configuration register of the ADC / DAC controller from the `gradcdac_config` data structure.

The `gradcdac_config` structure is described in Table 34.4.

34.2.6.1.2. gradcdac_get_config

Reads the configuration from the controller's configuration register and converts into the data structure `gradcdac_config` pointed to by the user provided [cfg] argument.

The `gradcdac_config` structure is described in Table 34.4.

34.2.6.1.3. gradcdac_set_cfg

Sets the configuration register directly.

The bits of the ADCONF configuration register are described in Table 34.4.

34.2.6.1.4. gradcdac_get_cfg

Returns the current configuration register value as it is.

The bits of the ADCONF configuration register are described in Table 34.4.

34.2.6.1.5. gradcdac_get_status

Returns the current ADC / DAC controller's status register value.

34.2.6.1.6. gradcdac_get_adrinput

Returns the current address input register value.

34.2.6.1.7. gradcdac_get_adroutput

Returns the current address output register value.

34.2.6.1.8. gradcdac_set_adroutput

Sets the controller's address output register to the argument [output].

34.2.6.1.9. gradcdac_get_adrdir

Returns the current address direction register value.

34.2.6.1.10. gradcdac_set_adrdir

Sets the controller's address direction register to the argument [dir].

34.2.6.1.11. gradcdac_get_datainput

Returns the current data input register value.

34.2.6.1.12. gradcdac_get_dataioutput

Returns the current data output register value.

34.2.6.1.13. gradcdac_set_dataioutput

Sets the controller's data output register to the argument [output].

34.2.6.1.14. gradcdac_get_datadir

Returns the current data direction register value.

34.2.6.1.15. gradcdac_set_datadir

Sets the controller's data direction register to the argument [dir].

34.2.6.2. Status interpretation help function

A short summary to the functions are presented in the prototype lists below. Functions to help the interpretation of the status read with `gradcdac_get_status` are described in Table 34.5. The functions does not actually read or write any ADC/DAC register therefore the handle (cookie) is omitted.

Table 34.6. Status interpretation help functions

Prototypes	**Non-zero return meaning**
int gradcdac_DAC_ReqRej(unsigned int status)	DAC conversion request rejected
int gradcdac_DAC_isCompleted (unsigned int status)	DAC conversion complete
int gradcdac_DAC_isOngoing (unsigned int status)	DAC conversion is ongoing
int gradcdac_ADC_isTimeouted (unsigned int status)	ADC sample timed out
int gradcdac_ADC_ReqRej(unsigned int status)	ADC sample request rejected
int gradcdac_ADC_isCompleted (unsigned int status)	ADC conversion is completed
int gradcdac_ADC_isOngoing (unsigned int status)	ADC conversion is ongoing

34.2.6.3. ADC functions

A short summary to the functions are presented in the prototype lists below.

Table 34.7. ADC functions

Operating on all ports
void gradcdac_adc_convert_start(void)
int gradcdac_adc_convert_try (unsigned short *digital_value)

Operating on all ports
int gradcdac_adc_convert (unsigned short *digital_value)

34.2.6.3.1. gradcdac_adc_convert_start

Make the ADC circuitry initialize an analogue to digital conversion. The result can be read out by `gradcdac_adc_convert_try or gradcdac_adc_convert`.

34.2.6.3.2. gradcdac_adc_convert_try

Tries to read the conversion result previously started with `gradcdac_adc_convert_start`. If the circuitry is busy converting the function returns a non-zero value, if the conversion has successfully finished zero is returned.

Table 34.8. gradcdac_adc_convert_try return code

Return Code	Description
zero	ADC conversion complete, digital_value contain current conversion result.
Positive	ADC busy, digital value contain previous conversion result.
Negative	ADC conversion request failed.

34.2.6.3.3. gradcdac_adc_convert

Waits until the ADC circuity has finished a digital to analogue conversion. The waiting is implemented as a busy loop utilizing 100% CPU load. This function returns zero on success and a negative value on failure, a positive result is never returned. See Table 34.2 for a description of the return values.

34.2.6.4. DAC functions

A short summary to the functions are presented in the prototype lists below.

Table 34.9. DAC functions

Operates on a single port
int gradcdac_dac_convert_try (unsigned short digital_value)
void gradcdac_dac_convert (unsigned short digital_value)

For a more detailed description see each function's respective sub section.

34.2.6.4.1. grandcdac_dac_convert_try

Try to make the DAC circuitry initialize a digital to analogue conversion. The digital value to be converted is taken as the argument digital_value. If the circuitry is busy by a previous conversion the function returns a non-zero value, if the conversion is successfully initialized the function returns zero.

34.2.6.4.2. grandcdac_dac_convert

Initializes a digital to analogue conversion by waiting until any previous conversion is finished before proceeding with the conversion. The digital value to be converted is taken as the argument [digital_value]. The waiting is implemented as a busy loop utilizing 100% CPU load.

35. GRTC GRLIB CCSDS Telecommand driver

35.1. INTRODUCTION

This document is intended as an aid in getting started developing with Aeroflex Gaisler GRLIB GRTC Telecommand (TC) core using the driver described in this document. It describes accessing GRTC in a on-chip system and over PCI and SpaceWire. It briefly takes the reader through some of the most important steps in using the driver such as starting TC communication, configuring the driver and receiving TC frames. The reader is assumed to be well acquainted with TC and RTEMS.

35.1.1. TC Hardware

See the GRTC core manual. When the GRTC core is accessed over SpaceWire RMAP is used.

35.1.2. Software Driver

The driver provides means for threads to receive TC frames using standard I/O operations. There are two drivers, one that supports GRTC on an on-chip AMBA bus and an AMBA bus accessed over PCI (on a GR-RASTA-TMTC board for example) and one driver that supports accessing the GRTC over SpaceWire.

35.1.2.1. GRTC over SpaceWire

The SpaceWire capable GRTC driver introduces some limitations listed below:

1. RAW mode is not supported (the read call)
2. The GRTC DMA area accessed over SpaceWire is cached in RAM close to the CPU. The cached DMA area is equal in length to the GRTC DMA area. The cache is synchronized every time the user enters the receive function.
3. A field named `dma_partition` has been added to the `grtc_ioc_buf_params` structure identifying the partition used when allocating the DMA memory on the SpaceWire node. The `custom_buffer` option is still available, it determines where the cached area is located.

35.2. User interface

The RTEMS GRTC driver supports the standard accesses to file descriptors such as **open**, **read** and **ioctl**. User applications include the **grtc** driver's header file which contains definitions of all necessary data structures and bit masks used when accessing the driver.

The driver enables the user to configure the hardware and to receive TC frames. The driver can be operated in two different modes either in RAW mode giving the user the possibility to read the DMA area it self using the **read** call or in FRAME mode where the driver handles basic frame parsing by looking at the header length field and the control bytes from the TC core. In the FRAME mode the allocation of TC frames is handled by the user, empty frames are given to the driver that puts data and header of received TC frames into the user allocated frames in a two step process. In the first step the user provides the driver with unused frames queued in an driver internal queue, the second step is when the user retrieve the frames containing a complete received frame, filler is not copied in FRAME mode.

Note that RAW mode is not supported when operating the GRTC over SpaceWire.

35.2.1. Driver registration

The registration of the driver is crucial for threads to be able to access the driver using standard means, such as open. The function `grtc_register` whose prototype is provided in grtc.h is used for registering the driver. It returns 0 on success and 1 on failure. A typical register call from the LEON3 Init task:

```
if ( grtc_register(&amba_conf) )
     printf("GRTC register Failed\n");
```

35.2.2. Opening the device

Opening the device enables the user to access the hardware of a certain GRTC device. The driver is used for all GRTC cores available. The cores are separated by assigning each core a unique name and a number called [minor]. The name is given during the opening of the driver. The first three names are printed out:

Table 35.1. Core number to device name conversion.

Core number	Filesystem name
0	/dev/grtc0
1	/dev/grtc1
2	/dev/grtc2
0	/dev/rastatmtc0/grtc0
0	/dev/rmap_fe/grtc0

An example of an RTEMS [open] call is shown below.

```
fd = open("/dev/grtc0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 35.1.

Table 35.2. Open `errno` values.

ERRNO	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

35.2.3. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the `grtc` driver.

35.2.4. I/O Control interface

The behaviour of the driver and hardware can be changed via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly.

All supported commands and their data structures are defined in the GRTC driver's header file `grtc.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

35.2.4.1. Data structures

The grtc_ioc_buf_params struct is used for configuring the DMA area of the TC core and driver.

```
struct grtc_ioc_buf_params {
    unsigned int     length;
    void        *custom_buffer;
    int         dma_partition;
};
```

Table 35.3. grtc_ioc_buf_params member descriptions.

Member	Description
length	Length of custom buffer or length of DMA buffer requested to be allocated by driver.
custom_buffer	When custom_buffer is zero, a DMA buffer will be allocate using malloc() by the driver. Set this option to a non-zero buffer pointer to indicate that the buffer is allocated by the user (user custom buffer). custom_buffer is interpreted as the new DMA buffer address that the driver must use. Note that there are align-

Member	Description
	ment requirements that need to be met, see the hardware documentation. When the least significant bit is set to one the custom address is interpreted as an address local to the GRTC core. The GRTC driver will translate this address to an address that the CPU can read. This is useful when the GRTC core is not on the same bus as the CPU and translation is needed.
dma_partition	SpaceWire driver version only. This option select which partition the DMA area on the SpaceWire node is allocated from. The AMBA RMAP bus driver provides custom functions for allocating memory on the remote target, the memory is split into multiple partitions. Note that memory allocated cannot be returned/freed, this means that a memory leak may be created when configuring the memory more than once.

The grtc_ioc_config struct is used for configuring the driver and the TC core.

```
struct grtc_ioc_config {
     int    psr_enable;
     int    nrzm_enable;
     int    pss_enable;
     int    crc_calc;
};
```

Table 35.4. grtc_ioc_config member descriptions.

Member	Description
psr_enable	Enable Pseudo-De-Randomizer in the TC core. See hardware manual for more information.
nrzm_enable	Enable Non-Return-to-Zero Mark Decoder. See hardware manual for more information.
pss_enable	Enable ESA/PSS. See hardware manual for more information.
crc_calc	Reserved, set this to zero

The grtc_ioc_hw_status data structure is used to store the register values of some of the GRTC core's registers. See hardware manual for more information.

```
struct grtc_ioc_hw_status {
     unsigned int     sir;
     unsigned int     far;
     unsigned int     clcw1;
     unsigned int     clcw2;
     unsigned int     phir;
     unsigned int     str;
};
```

Table 35.5. grtc_ioc_hw_status member descriptions.

Member	Description
sir	Spacecraft Identifier register
far	Frame Acceptance Report Register
clcw1	CLCW Register 1
clcw2	CLCW Register 2
phir	Physical Interface Register
str	Status Register

The grtc_frame structure is used for adding unused frames as buffers to the TC driver and retrieving received frames, it is the driver's representation of a TC frame. A TC frame structure can be chained together using the `next` field in grtc_frame. The `data` field is only 3 bytes in the structure but when used the data field goes past the `grtc_frame` boundary making different sized frames possible. The frame structure may be allocated with the size [sizeof(struct grtc_frame) +DATA_LEN-3].

```
struct grtc_frame {
     struct grtc_frame      *next;
     unsigned short         len;
     unsigned short         reserved;
     struct grtc_frame_pool *pool;

     /* The Frame content */
     struct grtc_hdr        hdr;
     unsigned char          data[3];
};
```

Table 35.6. grtc_frame member descriptions.

Member	Description
next	Points to next TC frame in TC frame chain, NULL if last frame in chain. This field is used to make driver process multiple TC Frames at the same time, avoiding multiple ioctl calls.
len	Length of received TC Frame.
reserved	Reserved by the driver.
pool	Field internally used by driver, must not be changed by user.
hdr	Header of a TC Frame.
data	Start of TC Frame payload.

The grtc_list structure represents a linked list, a chain, of TC frames. The data structure holds the first frame and last frame in chain.

```
struct grtc_list {
     struct grtc_frame *head;
     struct grtc_frame *tail;
     int     cnt;
};
```

Table 35.7. grtc_list member descriptions.

Member	Description
head	First TC frame in chain
tail	Last TC frame in chain, last frame in list must have it's next field set to NULL
cnt	Number of frames in list

The grtc_ioc_pools_setup structure represents the set up of all frame pools used by the driver to select the shortest frame to put incoming TC frames into. The size of the data structure depends on the pool_cnt field, the size can be calculated as [sizeof(struct grtc_ioc_pools_setup) - 4 + 4*pool_cnt].

```
struct grtc_ioc_pools_setup {
     unsigned int     pool_cnt;
     unsigned int     pool_frame_len[1];
};
```

Table 35.8. grtc_ioc_pools_setup member descriptions.

Member	Description
pool_cnt	Number of frame pools in this setup
pool_frame_len	Array of frame lengths, one length per pool. Pool one has frame length pool_frame_len[0], Pool 2 pool_frame_len[1] and so on.

The grtc_ioc_assign_frm_pool structure hold a chain of frames all with the same minimum length, the length is specified by the `frame_len` field and the frame chain is pointed to by the field `frames`. This data structure is used by the driver to assign a common pool for all frames in the chain. This is to make the frame to pool insertion faster for unused frames.

```
struct grtc_ioc_assign_frm_pool {
     unsigned int     frame_len;
     struct grtc_frame *frames;
};
```

Table 35.9. grtc_ioc_assign_frm_pool member descriptions.

Member	Description
frame_len	Minimum length of all TC frames in the **frames** field
frames	Linked list of frames that will be assigned a pool by the driver

The grtc_ioc_stats structure contain statistics collected by the driver in FRAME mode.

```
struct grtc_ioc_stats {
     unsigned long long     frames_recv;

     /* Errors related to incoming data */
     unsigned int           err;
     unsigned int           err_hdr;
     unsigned int           err_payload;
     unsigned int           err_ending;
     unsigned int           err_abandoned;

     /* Errors related to the handling of incoming frames */
     unsigned int           dropped;
     unsigned int           dropped_no_buf;
     unsigned int           dropped_too_long;
};
```

Table 35.10. grtc_ioc_stats member descriptions.

Member	Description
frames_recv	Number of frames successfully received by the TC core.
err	Total number of errors related to incoming data, due to too early frame ending or abandoned frame.
err_hdr	Number of errors encountered during frame header processing.
err_payload	Number of errors encountered during frame payload processing.
err_ending	Number of errors encountered during filler and end of frame processing.
err_abandoned	reserved for future use, NOT IMPLEMENTED.
dropped	Number of dropped frames due to not the correct buffers were available when processing the frame.
dropped_no_buf	Number of frames dropped because no empty frames of this frame length were available upon reception.
dropped_too_long	Number of frames dropped because frame length too long to match any of the configured frame pools.

35.2.4.2. Configuration

The TC core and driver are configured using **ioctl** calls. The Table 35.4 below lists all supported **ioctl** calls. GRTC_IOC_ must be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 35.3.

An example is shown below where the statistics of the driver is copied to the user buffer **stats** by using an **ioctl** call:

```
struct grtc_ioc_stats stats;

result = ioctl(fd, GRTC_IOC_GET_STATS, &stats);
```

*Table 35.11. ERRNO values of **ioctl** calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The TC hardware is not in the correct state. Many **ioctl** calls need the TC core to be in stopped or started mode. One can switch state by calling START or STOP.

ERRNO	Description
ENOMEM	Not enough memory to complete operation. This may cause other **ioctl** commands to fail.
EIO	Writing to hardware failed. Feature not available in hardware.

*Table 35.12. **ioctl** calls supported by the GRTC driver.*

Call Number	Status	Mode	Description
START	Stopped	Both	Exit stopped mode, start the receiver.
STOP	Started	Both	Exit started mode, enter stopped mode. This stops the receiver. Most of the settings can only be set when in stopped mode.
ISSTARTED	Both	Both	Indicates operating status, started or stopped.
SET_BLOCKING_MODE	Both	RAW	Set blocking or non-blocking mode for read calls.
SET_TIMEOUT	Both	RAW	Set time out value used in blocking mode to wake up blocked task if **read** request takes too long time to complete.
SET_MODE	Stopped	Both	Select operating mode, RAW or FRAME mode. RAW is default.
SET_BUF_PARAM	Stopped	Both	Set DMA buffer parameters.
SET_CONFIG	Stopped	Both	Configure hardware and driver.
GET_CONFIG	Both	Both	Get current configuration previously set with SET_CONFIG or the driver defaults.
GET_BUF_PARAM	Both	Both	Get current DMA buffer parameters.
GET_BUF_STATUS	Both	Both	Get current GRTC hardware status.
GET_CLCW_ADR	Both	Both	Returns the address of the CLCWRx1 register, it can be used to get the current CLCW fields from hardware. For example can the no-RF and the No-Bit-Lock bit be read from this address. See hardware manual.
GET_STATS	Both	FRAME	Get statistics collected by driver.
CLR_STATS	Both	FRAME	Reset driver statistics.
POOLS_SETUP	Stopped	FRAME	Set up frame pool configuration.
ASSIGN_FRM_POOL	Both	FRAME	Assigns a chain of TC frame structures to a frame pool internal used by driver.
ADD_BUFF	Started	FRAME	Add a chain of free TC frames to the frame pools internal to the GRTC driver.
RECV	Both	FRAME	Get all complete processed TC frames from the ready queue internal to the GRTC driver.

35.2.4.2.1. START

This **ioctl** command enables the TC receiver and changes the driver's operating status to started. Settings previously set by other **ioctl** commands are written to hardware just before starting reception. It is necessary to enter started

mode to be able to receive TC frames using the **ioctl** command GRTC_IOC_RECV or to read the DMA data area by calling `read()`.

The command will fail if the receiver is unable to be brought up, the driver or hardware configuration is invalid or if the TC core already is started. In case of failure the return code is negative and **errno** will be set to EIO or EINVAL, see Table 35.3.

35.2.4.2.2. STOP

This call makes the TC core leave started mode and enter stopped mode. The receiver is stopped and no frames will be received. After calling STOP further calls to **read** and to **ioctl** using command such as ADD_BUFF, RECV, ISSTARTED, STOP will behave differently or result in error.

It is necessary to enter stopped mode to change major operating parameters of the TC core and driver. See SET_CONFIG for more details.

The command will fail if the TC driver already is in stopped mode.

35.2.4.2.3. ISSTARTED

Determines if driver and hardware is in started mode. Errno will be set to EBUSY in stopped mode and return successfully in started mode.

35.2.4.2.4. SET_BLOCKING_MODE

Changes the driver's **read** behaviour in RAW mode. This call has no effect for FRAME mode, FRAME mode is always non-blocking. Two modes are available blocking mode and polling mode, in polling mode the `read()` call always returns directly even when no DMA data is available. In blocking mode the task calling `read()` is blocked until at least one byte is available, it is also possible to make the blocked task time out after some time setting the timeout value using the SET_TIMEOUT **ioctl** command.

Input is set as as described in the table below.

*Table 35.13. SET_BLOCKING_MODE **ioctl** arguments*

Bit number	Description
GRTC_BLKMODE_POLL	Enables polling mode
GRTC_BLKMODE_BLK	Enables blocking mode

The driver's default is polling mode.

Note that the blocking mode is implemented using the CLTU stored interrupt.

This command never fail.

35.2.4.2.5. SET_TIMEOUT

Sets the blocking mode time out value, instead of blocking for eternity the task will be woken up after this time out expires. The time out value specifies the input to the RTEMS take semaphore operation `rtems_semaphore_obtain()`. See the RTEMS documentation for more information how to set the time out value.

Note that this option has no effect in polling mode.

This command never fail.

35.2.4.2.6. SET_MODE

Select RAW of FRAME mode. Argument must be either GRTC_MODE_RAW for RAW mode or GRTC_MODE_FRAME for FRAME mode. See the section Operating mode for more information about the modes.

The driver defaults to RAW mode.

This calls fails if driver is in started mode or due to an illegal input argument.

35.2.4.2.7. SET_BUF_PARAM

This command is used to configure the DMA buffer area of the TC core. The argument is a pointer to an initialized grtc_ioc_buf_params data structure described in the data structures section. The DMA buffer may be set to a custom location and length, or the driver may be requested to allocate a DMA buffer with the specified size. If the custom location lsb is set to one the address is interpreted as a remote address as viewed from the GRTC core, not the CPU. This can be useful for GRTC cores found on another bus than the CPU, for example for a GRTX core on a GR-RASTA-TMTC PCI board.

When GRTC is operated over SpaceWire an additional option is available, the `dma_partition` field, selecting from which memory partition the DMA area is allocated from. See AMBA Plug&Play SpaceWire bus driver for an description of memory allocation. The custom option described above is still available, however it identifies the cached memory area rather than the GRTC DMA area.

Trying to configure the DMA buffer area in started mode result in failure, and **errno** set to EBUSY. An invalid argument result in failure and **errno** set to EINVAL. The command will fail and **errno** set to ENOMEM when the driver is requested to allocate a buffer too large to be allocated by `malloc()`.

35.2.4.2.8. SET_CONFIG

Configures the driver and core. This call updates the configuration that will be used by the driver during the START command and during operation. Enabling features not implemented by the TC core will result in EIO error when starting the TC driver.

The input is a pointer to an initialized grtc_ioc_config structure described in Section 35.2.2.

This call fail if the TC core is in started mode, in that case **errno** will be set to EBUSY, or if a NULL pointer is given as argument, in that case **errno** will be set to EINVAL.

35.2.4.2.9. GET_CONFIG

Return the current configuration of the driver and hardware. The current configuration is either the driver and hardware defaults or the configuration previously set by the SET_CONFIG command.

The input to this **ioctl** command is a pointer to a data area of at least the size of a `grtc_ioc_config` structure. The data area will be stored according to the `grtc_ioc_config` data structure described in Section 35.2.2.

This command only fail if the pointer argument is invalid.

35.2.4.2.10. GET_BUF_PARAM

Get the current DMA buffer configuration. The argument is a pointer to an uninitialized grtc_ioc_buf_params data structure described in the data structures section.

This command will fail if the input argument is invalid, errno will be set to EINVAL in such cases.

35.2.4.2.11. GET_HW_STATUS

Read current TC hardware state, the argument is a pointer to a data area where the hardware status will be stored. The status is stored using the layout of the `grtc_ioc_hw_status` described in the data structures section.

This command only fail if the pointer argument is invalid.

35.2.4.2.12. GET_CLCW_ADR

The address of the GRTC register "GRTC CLCW Register 1" is stored into a user provided location. The register address may be used to access the current CLCW fields from the GRTC hardware. For example can the no-RF and the No-Bit-Lock bit be read from this address. See the hardware manual.

This command only fail if the pointer argument is invalid.

35.2.4.2.13. GET_STATS

This command copies the driver's internal statistics counters to a user provided data area. The format of the data written is described in the data structure subsection. See the `grtc_ioc_stats` data structure.

Note that the statistics only is available for the FRAME mode since it is only the FRAME mode that generate statistics such as number of frames received and errors in header, in RAW mode the data is never processed just copied to a user provided buffer.

The call will fail if the pointer to the data is invalid.

35.2.4.2.14. CLR_STATS

This command reset the driver's internal statistics counters.

This command never fail.

35.2.4.2.15. POOLS_SETUP

This command set up the frame pools internal to the driver. The frame pools must be configured before starting the receiver in FRAME mode. For more information about frame pools see section Operating mode. The pools are configured by the input argument pointing to an initialized `grtc_ioc_pools_setup` data structure described in the data structure subsection.

Note that the frame length must be sorted with the first frame pool having the shortest frame length.

The call will fail if the pointer to the data is invalid or if in RAW mode.

35.2.4.2.16. ASSIGN_FRM_POOL

Assigns a linked list of frames to a frame pool. The input argument is a pointer to a `grtc_ioc_assign_frm_pool` data structure containing the frame length identifying a pool and a linked list of frames that will be assign to the matching pool. All frames must be assigned to a frame pool before added to the driver's frame pools using the command ADD_BUF. For more information about frame pools and assigning a frame to a frame pool see section Operating mode. See section data structures for a description of `grtc_ioc_assign_frm_pool`.

The frame pools, using POOLS_SETUP, must be set up before assigning frames to a frame pool.

This command fail and **errno** set to EINVAL is the input argument is invalid, the driver is in RAW mode or no matching frame pool was found.

35.2.4.2.17. ADD_BUF

Adds a chain of frames to their respective frame pool for later use by the driver. The driver will use the added frames when frames are received. The input argument is a pointer to a `grtc_frame` data structure, the first frame in the chain, see the data section for a description of the `grtc_frame` structure.

Note, that the frame structure and any data pointed to by the frame added to the driver must not be accessed until the frame has been received using the **ioctl** command RECV.

The call will fail if the pointer to the data is invalid or if in RAW mode.

35.2.4.2.18. RECV

This command is used to process the DMA area and retrieve a linked list of successfully processed received frames. The input argument to RECV is a pointer to a `grtc_list` data structure, described in the data structure section. All currently processed frames will be put into data structure, **head** will point to the first and **tail** to the last frame in the chain, **cnt** will hold the number of frames in the list.

35.2.5. Operating mode

In RAW mode the user can read out the raw data from the TC DMA buffer set up by the driver using the standard `read()` call. This enables the user to do custom processing of incoming frames. All TC DMA data is read one control data byte for each frame data byte, for more information how to handle the data see the GRTC hardware manual. If the DMA buffer isn't read in time overflow may occur and data will be lost forcing the driver to stop the receiver.

When the driver is operated in FRAME mode the driver is responsible to determine the start and end of each frame. It does so by looking at the TC frame length field and the GRTC control bytes provided for each frame data byte. The header and data is copied into a free frame taken from a frame pool internal to the driver, see next section for information about frame pools, an put at the end of a linked list with received frames that can be read by the user using the **ioctl** command GRTC_IOC_RECV. After the user has processed the frame the frame is added again to the driver's frame pools using the **ioctl** command GRTC_IOC_ADD_BUFF. It is the users responsibility to make sure that there always is frames available for the TC driver to copy frames into, otherwise the TC driver will drop frames.

35.2.5.1. Driver frame pools

In FRAME mode a frame pool concept is used to group frames of equal frame length. Using multiple pools make it possible for the driver to select a frame with a frame length as short as possible that still fit the incoming frame data and header. The driver is configured with multiple pools with different frame lengths, the more frame pools the smaller is the difference of the incoming frame length to the taken buffer the driver selects. The pools are set up using the **ioctl** command GRTC_IOC_POOLS_SETUP.

Every time a frame is added to one of the driver's pool, using the GRTC_IOC_ADD_BUFF command, the correct frame pool must be found to put it in. To simplify and make the frame pool detection faster each frame must be assigned to a frame pool once before use, assigning a frame with a pool must done by using the ioctl command GRTC_IOC_ASSIGN_FRM_POOL.

35.2.6. Reception in FRAME mode

Receiving frames are done with the **ioctl** call using the command ADD_BUF and RECV. It is possible to receive multiple frames in one call, the frames are provided to the driver using a linked list of frames. See the **ioctl** command RECV and ADD_BUF for more information.

35.2.7. Reception using RAW mode

Reception is done using the **read** call. An example is shown below:

```
unsigned char tc_rx_buf[512];

len = read(fd, tc_rx_buf, sizeof(tc_rx_buf));
```

The requested number of bytes to be read is given in the third argument. The messages will be stored in tc_rx_buf. The actual number of received bytes is returned by the function on success and -1 on failure. In the latter case **errno** is also set.

The data formatting is described in the hardware manual.

The call will fail if a null pointer is given, invalid buffer length, the TC core is in stopped mode, no data available in non-blocking mode or due to a time out in blocking mode.

The blocking behaviour can be set using **ioctl** calls. In blocking mode the call will block until at least one byte has been received, unless a time out has been given and that time has expired causes the driver to return ETIMEDOUT. In non-blocking mode, the call will return immediately and if no data was available -1 is returned and **errno** set appropriately. The table below shows the different **errno** values is returned.

*Table 35.14. ERRNO values for **read** calls.*

ERRNO	Description
EINVAL	A NULL pointer was passed as the data pointer or the length was illegal.

ERRNO	Description
EBUSY	TC core is in stopped mode. Switch to started mode by issuing a START **ioctl** command.
ETIMEDOUT	In non-blocking mode no data were available in the DMA area, or in blocking mode and the time out has expired and still no data in DMA area.
ENODEV	A blocking read was interrupted by the TC receiver has been stopped. Further calls to **read** will fail until the **ioctl** command START is issued again.

36. GRTM GRLIB CCSDS Telemetry Driver

36.1. Introduction

This document is intended as an aid in getting started developing with Aeroflex Gaisler GRLIB GRTM Telemetry (TM) core using the driver described in this document. It describes accessing GRTM in a on- chip system and over PCI and SpaceWire. It briefly takes the reader through some of the most important steps in using the driver such as starting TM communication, configuring the driver and sending TM frames. The reader is assumed to be well acquainted with TM and RTEMS.

36.1.1. TM Hardware

See the GRTM core manual. When the GRTM core is accessed over SpaceWire RMAP is used.

36.1.2. Software Driver

The driver provides means for threads to send TM frames using standard I/O operations.

There are two drivers, one that supports GRTM on an on-chip AMBA bus and an AMBA bus accessed over PCI (on a GR-RASTA-TMTC board for example) and one driver that supports accessing the GRTM over SpaceWire.

36.1.2.1. GRTM over SpaceWire

There are some differences when the GRTM core is operated over SpaceWire, see below list for a summary.

- The GRTM driver manages one buffer per descriptor used to copy frame payload into. The payload is copied over SpaceWire by the GRTM driver. The maximal frame length must be given in order for the driver to know how much buffer space to allocate. It is controlled through the `maxFrameLength` driver resource.
- The driver has three new driver resources: `maxFrameLength` (maximal length of frames, used when allocating buffer space), `bdAllocPartition` (partition used when allocating descriptor table, see AMBA RMAP bus driver documentation) and `frameAllocPartition` (partition used when allocating buffer space, see AMBA RMAP bus driver documentation).
- TM frames has an additional option COPY_DATA, it determines if the payload is to be copied to the descriptor's buffer or if the address of the payload is an address that the GRTM core can read directly, for example the payload may already reside on the SpaceWire node's memory ready to be transmitted. In the latter case only the descriptor address pointer is written.
- The Frame options TRANSLATE and TRANSLATE_AND_REMEMBER has no effect.

36.2. User interface

The RTEMS GRTM driver supports the standard accesses to file descriptors such as **open**, **close** and **ioctl**. User applications include the **grtm** driver's header file which contains definitions of all necessary data structures and bit masks used when accessing the driver.

The driver enables the user to configure the hardware and to transmit TM frames. The allocation of TM frames is handled by the user and free frames are given to the driver that processes the frames for transmission in a two step process. In the first step the driver schedules frames for transmission using the DMA descriptors or they are put into an internal queue when all descriptors are in use, in the second step all sent frames are put into a second queue that is emptied when the user reclaims the sent frames. The reclaimed frames can be reused in new transmissions later on.

36.2.1. Driver registration

The registration of the driver is crucial for threads to be able to access the driver using standard means, such as open. The function `grtm_register` whose prototype is provided in grtm.h is used for registering the driver. It returns 0 on success and 1 on failure. A typical register call from the LEON3 Init task:

```
if ( grtm_register(&amba_conf) )
     printf("GRTM register Failed\n");
```

36.2.2. Opening the device

Opening the device enables the user to access the hardware of a certain GRTM device. The driver is used for all GRTM cores available. The cores are separated by assigning each core a unique name and a number called [minor]. The name is given during the opening of the driver. The first three names are printed out:

Table 36.1. Core number to device name conversion.

Core number	Filesystem name	Location
0	/dev/grtm0	On Chip AMBA bus
1	/dev/grtm1	On Chip AMBA bus
2	/dev/grtm2	On Chip AMBA bus
0	/dev/rastatmtc0/grtm0	GR-RASTA-TMTC PCI Target
0	/dev/rmap_fe/grtm0	SpaceWire node with destination address 0xfe.
2	/dev/rmap_1a/grtm2	SpaceWire node with destination address 0x1a.

An example of an RTEMS **open** call is shown below.

```
fd = open("/dev/grtm0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 36.1.

*Table 36.2. Open **errno** values.*

ERRNO	Description
ENODEV	Illegal device name or not available.
EBUSY	Device already opened.
ENOMEM	Driver failed to allocate necessary memory.

36.2.3. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the [grtm] driver.

36.2.4. I/O Control interface

The behaviour of the driver and hardware can be changed via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical ioctl call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly.

All supported commands and their data structures are defined in the GRTM driver's header file `grtm.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

36.2.4.1. Data structures

The `grtm_ioc_hw` data structure indicates what features the TM hardware supports and how it has been configured.

```
struct grtm_ioc_hw {
    char        cs;
    char        sp;
    char        ce;
    char        nrz;
    char        psr;
```

```
    char        te;
    unsigned char     rsdep;
    unsigned char     rs;
    char        aasm;
    char        fecf;
    char        ocf;
    char        evc;
    char        idle;
    char        fsh;
    char        mcg;
    char        iz;
    char        fhec;
    char        aos;
    unsigned short    blk_size;
    unsigned short    fifo_size;
};
```

Table 36.3. grtm_ioc_hw member descriptions.

Member	Description
cs	Indicates if Sub Carrier (SC) modulation is implemented
sp	Indicates if Split-Phase Level (SP) modulation is implemented
ce	Indicates if Convolutional Encoding (CE) is implemented
nrz	Indicates if Non-Return-to-Zero (NRZ) mark encoding is implemented
psr	Indicates if Pseudo-Randomizer (PSR) is implemented
te	Indicates if Turbo Encoder (TE) is implemented
rsdep	Reed-Solomon interleave Depth (RSDEPTH) implemented (3-bit)
rs	Indicates what Reed-Solomon encoders are implemented (0=None, 1=E16, 2=E8, 3=Both)
aasm	Indicates if Alternative ASM (AASM) implemented
fecf	Indicates if Transfer frame control field CRC is implemented
ocf	Indicates if Operational Control Field (OCF) is implemented
evc	Indicates if Extended Virtual Channel Counter is implemented
idle	Indicates if Idle Frame generation is implemented
fsh	Indicates if Frame secondary header is implemented
mcg	Indicates if Master Channel counter generation is implemented
iz	Indicates if Insert Zone (IZ) is implemented
fhec	Indicates if Frame Header Error Control (FHEC) is implemented
aos	Indicates if AOS transfer frame generation is implemented
blk_size	TM core DMA Block size in number of bytes
fifo_size	TM core FIFO size in number of bytes

The grtm_ioc_config struct is used for configuring the driver and the TM core.

```
struct grtm_ioc_config {
    unsigned char     mode;

    unsigned short    frame_length;
    unsigned short    limit;
    unsigned int      as_marker;

    /* Physical layer options */
    unsigned short    phy_subrate;
    unsigned short    phy_symbolrate;
    unsigned char     phy_opts;

    /* Coding sub-layer Options */
    unsigned char    code_rsdep;
    unsigned char    code_ce_rate;
    unsigned char    code_csel;
    unsigned int     code_opts;
```

```
    /* All Frames Generation */
    unsigned char    all_izlen;
    unsigned char    all_opts;

    /* Master Frame Generation */
    unsigned char    mf_opts;

    /* Idle frame Generation */
    unsigned short   idle_scid;
    unsigned char    idle_vcid;
    unsigned char    idle_opts;

    /* Interrupt options */
    unsigned int     enable_cnt;
    int         isr_desc_proc;
    int         blocking;
    rtems_interval   timeout;
};
```

Table 36.4. grtm_ioc_config member descriptions.

Member	Description
mode	Select mode hardware will operate in, TM=0, AOS=1
frame_length	Frame Length in bytes
limit	Number of data bytes fetched by TM DMA engine before transmission starts. Setting limit to zero will make GRTM driver to calculate the limit value from frame length and the block size of the hardware.
as_marker	Set custom Attached Synchronization Marker (ASM)
phy_subrate	Sub Carrier rate division factor - 1
phy_symbolrate	Symbol Rate division factor - 1
phy_opts	Physical layer options, mask of GRTM_IOC_PHY_XXXX
code_rsdep	Coding sub-layer Reed-Solomon interleave depth (3-bit)
code_ce_rate	Convolutional encoding rate, select one of GRTM_CERATE_00 ... GRTM_CERATE_07
code_csel	External TM clock source selection, 2-bit (application specific)
code_opts	Coding sub-layer options, mask of GRTM_IOC_CODE_XXXX
all_izlen	All frame generation FSH (TM) or Insert Zone (AOS) length in bytes
all_opts	All frame generation options, mask of GRTM_IOC_ALL_XXXX
mf_opts	Master channel frame generation, mask of GRTM_IOC_MF_XXXX
idle_scid	Idle frame spacecraft ID, 10-bit
idle_vcid	Idle frame virtual channel ID, 6-bit
idle_opts	Idle frame generation options, mask of GRTM_IOC_IDLE_XXXX
enable_cnt	Number of frames between interrupts are generated, zero disables interrupt. Allows user to fine grain interrupt generation
isr_dec_proc	Allow TM interrupt service routine (ISR) to process descriptors
blocking	Blocking mode select, GRTM_BLKMODE_POLL for polling mode or GRTM_BLMODE_BLK for blocking mode
timeout	Blocking mode time out

The grtm_frame structure is used in for transmitting TM frames and retrieving sent frames, it is the driver's representation of a TM frame. A TM frame structure can be chained together using the [next] field in grtm_frame.

```
struct grtm_frame {
    unsigned int      flags;
    struct grtm_frame *next;
    unsigned int      *payload;
};
```

Table 36.5. grtm_frame member descriptions.

Member	Description
flags	Mask indicating options, transmission state and errors for the frame. GRTM_FLAGS_XXX. See Table 36.5
next	Points to next TM frame. This field is used to make driver process multiple TM Frames at the same time, avoiding multiple ioctl calls.
payload	Points to a data area holding the complete TM frame. The area include fields such as header, payload, OCF, CRC.

Table 36.6. Frame flags descriptions.

Flag	Description
GRTM_FLAGS_SENT	Indicates whether the frame has been transmitted or not
GRTM_FLAGS_ERR	Indicates if errors has been experienced during transmission of the frame
GRTM_FLAGS_TS	Generate Time Strobe (TS) for the frame
GRTM_FLAGS_MCB	Bypass the TM core's Master Channel Counter generation
GRTM_FLAGS_FSHB	Bypass the TM core's Frame Secondary Header (FSH) generation
GRTM_FLAGS_OCFB	Bypass the TM core's Operational Control Field (OCF) generation
GRTM_FLAGS_FHECB	Bypass the TM core's Frame Header Error Control (FHEC) generation
GRTM_FLAGS_IZB	Bypass the TM core's Insert Zone (IZ) generation
GRTM_FLAGS_FECFB	Bypass the TM core's Frame Error Control Field (FECF) generation
COPY_DATA	This option has effect only on the SpaceWire version of the driver. Indicates if the TM frame payload should be copied into the assigned descriptor's buffer or not. If this option is not set then the payload address is assumed to be readable by the GRTM core and the descriptor address pointer is written with the address of the payload directly.
TRANSLATE	Translate frame payload address from CPU address to remote bus (the bus GRTM is resident on). This is useful when dealing with buffers on remote buses, for example when GRTM is on a AMBA bus accessed over PCI. This is the case for GR-RASTA-TMTC.
TRANSLATE_AND_REMEMBER	As TRANSLATE, however if the translated payload address equals the payload address the TRANSLATE_AND_REMEMBER bit is cleared and the TRANSLATE bit is set. Not used in SpaceWire version of driver.

The grtm_list structure represents a linked list, a chain of TM frames. The data structure holds the first frame and last frame in chain.

```
struct grtm_list {
     struct grtm_frame *head;
     struct grtm_frame *tail;
};
```

Table 36.7. grtm_list member descriptions.

Member	Description
head	First TM frame in chain
tail	Last TM frame in chain, last frame in list must have it's next field set to NULL

The grtm_ioc_stats structure contain statistics collected by the driver.

```
struct grtm_ioc_stats {
```

```
    unsigned long long   frames_sent;
    unsigned int         err_underrun;
};
```

Table 36.8. grtm_ioc_stats member descriptions.

Member	Description
frames_sent	Number of frames successfully sent by the TM core
err_underrun	Number of AMBA underrun errors

36.2.4.2. Configuration

The GRTM core and driver are configured using ioctl calls. Table 36.7 lists all supported **ioctl** calls. GRTM_IOC_ must be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 36.6.

An example is shown below where the statistics of the driver is copied to the user buffer **stats** by using an **ioctl** call:

```
struct grtm_ioc_stats stats;


result = ioctl(fd, GRTM_IOC_GET_STATS, &stats);
```

*Table 36.9. ERRNO values for **ioctl** calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The TM hardware is not in the correct state. Many **ioctl** calls need the TM core to be in stopped or started mode. One can switch state by calling START or STOP.
ENOMEM	Not enough memory to complete operation. This may cause other **ioctl** commands to fail.
EIO	Writing to hardware failed. Feature not available in hardware.
ENODEV	Operation aborted due to transmitter being stopped.

*Table 36.10. **ioctl** calls supported by the GRTM driver.*

Call Number	Call	Description
START	Stopped	Exit stopped mode, start the receiver.
STOP	Started	Exit started mode, enter stopped mode. Most of the settings can only be set when in stopped mode.
ISSTARTED	Both	Indicates operating status, started or stopped.
SET_BLOCKING_MODE	Both	Set blocking or non-blocking mode for RECLAIM.
SET_TIMEOUT	Both	Set time out value used in blocking mode to wake up blocked task if request takes too long time to complete.
SET_CONFIG	Stopped	Configure hardware and software driver.
GET_CONFIG	Both	Get current configuration previously set with SET_CONFIG or the driver defaults.
GET_STATS	Both	Get statistics collected by driver.
CLR_STATS	Both	Reset driver statistics.
GET_HW_IMPL	Both	Returns the features and implemented by the TM core.
GET_OCFREG	Both	Returns the address of the OCF/CLCW register, it can be used to update the transmitted OCF/CLCW.
RECLAIM	Both	Returns all TM frames sent since last call to RECLAIM, the frames are linked in a chain.

Call Number	Call	Description
SEND	Started	Add a chain of TM frames to the transmission queue of the GRTM driver.

36.2.4.2.1. START

This **ioctl** command enables the TM transmitter and changes the driver's operating status to started. Settings previously set by other **ioctl** commands are written to hardware just before starting transmission. It is necessary to enter started mode to be able to send TM frames using the **ioctl** command GRTM_IOC_SEND.

The command will fail if the transmitter is unable to be brought up, the driver or hardware configuration is invalid or if the TM core already is started. In case of failure the return code is negative and errno will be set to EIO or EINVAL, see Table 36.6.

36.2.4.2.2. STOP

This call makes the TM core leave started mode and enter stopped mode. The transmitter is stopped and no frames will be sent. After calling STOP further **ioctl** commands such as SEND, RECLAIM, ISSTARTED, STOP will behave differently or result in error.

It is necessary to enter stopped mode to change major operating parameters of the TM core and driver. See SET_CONFIG for more details.

The command will fail if the TM driver already is in stopped mode.

36.2.4.2.3. ISSTARTED

Determines if driver and hardware is in started mode. Errno will be set to EBUSY in stopped mode and return successfully in started mode.

36.2.4.2.4. SET_BLOCKING_MODE

Changes the driver's GRTM_IOC_RECLAIM command behaviour. Two modes are available blocking mode and polling mode, in polling mode the **ioctl** command RECLAIM always return directly even when no frames are available. In blocking mode the task calling RECLAIM is blocked until at least one frame can be reclaimed, it is also possible to make the blocked task time out after some time setting the **timeout** value using the SET_CONFIG or SET_TIMEOUT **ioctl** commands.

The argument is set as as described in the table below.

*Table 36.11. SET_BLOCKING_MODE **ioctl** arguments*

Bit Number	Description
GRTM_BLKMODE_POLL	Enables polling mode
GRTM_BLKMODE_BLK	Enables blocking mode

The driver's default is polling mode.

Note that the blocking mode is implemented using the DMA transmit frame interrupt, changing the `isr_desc_proc` parameter of the SET_CONFIG command effects the blocking mode characteristics. For example, enabling interrupt generation every tenth TM frame will cause the blocked task to be woken up after maximum ten frames when going into blocked mode.

This command never fail.

36.2.4.2.5. SET_TIMEOUT

Sets the blocking mode time out value, instead of blocking for eternity the task will be woken up after this time out expires. The time out value specifies the input to the RTEMS take semaphore operation

`rtems_semaphore_obtain()`. See the RTEMS documentation for more information how to set the time out value.

Note that this option has no effect in polling mode.

Note that this option is also set by SET_CONFIG.

36.2.4.2.6. SET_CONFIG

Configures the driver and core. This call updates the configuration that will be used by the driver during the START command and during operation. Enabling features not implemented by the TM core will result in EIO error when starting the TM driver. The hardware features available can be obtained by the GET_HW_IMPL command.

The input is a pointer to an initialized `grtm_ioc_config` structure described in section Section 36.2.4.1.

Note that the time out value and blocking mode can also be set with SET_TIMEOUT and SET_BLOCKING_MODE.

This call fail if the TM core is in started mode, in that case **errno** will be set to EBUSY, or if a NULL pointer is given as argument, in that case **errno** will be set to EINVAL.

36.2.4.2.7. GET_CONFIG

Returns the current configuration of the driver and hardware. The current configuration is either the driver and hardware defaults or the configuration previously set by the SET_CONFIG command.

The input to this **ioctl** command is a pointer to a data area of at least the size of a `grtm_ioc_config` structure. The data area will be updated according to the `grtm_ioc_config` data structure described in Section 36.2.4.1.

This command only fail if the pointer argument is invalid.

36.2.4.2.8. GET_STATS

This command copies the driver's internal statistics counters to a user provided data area. The format of the data written is described in the data structure subsection. See the `grtm_ioc_stats` data structure.

The call will fail if the pointer to the data is invalid.

36.2.4.2.9. CLR_STATS

This command reset the driver's internal statistics counters.

This command never fail.

36.2.4.2.10. GET_HW_IMPL

This command copies the TM core's features implemented to a user provided data area. The format of the data written is described in the data structure subsection. See the `grtm_ioc_hw` data structure.

Knowing the features supported by hardware can be used to make software run on multiple implementations of the TM core.

36.2.4.2.11. GET_OCFREG

The address of the GRTM register "GRTM Operational Control Field Register" is stored into a user provided location. The register address may be used to updated the CLCW or OCF value transmitted in TM frames to ground without using an **ioctl** command to perform the request. This address is typically used by Telecommand (TC) software to tell ground of the current FARM/COP state.

Note that OCF/ CLCW is transmitted only in started mode.

This command never fail.

36.2.4.2.12. RECLAIM

Returns processed TM frames to user. All frames returned has been provided by the user in previous calls to SEND, and need not all to have been successfully sent. RECLAIM can be configured to operate in polling mode, blocking mode and blocking mode with a time out. In polling mode the task always returns with or without processed frames, in blocking mode the task is blocked until at least one frame has been processed. See the **ioctl** command SET_CONFIG and SET_BLOCKING_MODE to change mode of the RECLAIM command.

RECLAIM stores a linked list of processed TM frames into the data area pointed to by the user argument. The format for the stored data follows the layout of the `grtm_list` structure described in Section 36.2.2. The `grtm_list` structure holds the first and last TM frame processed by the driver. The **flags** field indicates if the frame was sent or if errors were experienced during transmission of this frame. See Table 36.6 for **flags** details.

In started mode, this command enables scheduled TM frames for transmission as descriptors become free during the processing of received TM frames.

The call will fail if the pointer to the data area is invalid (EINVAL), the RECLAIM call operates in blocking mode and the time out expires (ETIMEDOUT) or the driver was stopped during the calling task was blocked (ENODEV). See table below.

*Table 36.12. ERRNO values for **RECLAIM***

ERRNO	Description
EINVAL	An invalid argument.
ETIMEDOUT	The blocked task was timed out and still no frames was transmitted.
ENODEV	The calling task was woken up from blocking mode by the transmitter being stopped. The TM driver has has entered stopped mode. Further calls to RECLAIM will retrieve sent and unsent frames.

36.2.4.2.13. SEND

Scheduling ready TM frames for transmission is done with the ioctl command SEND. The input is a linked list of TM frames to be scheduled. When all TM DMA descriptors are active, enabled and linked to a frame to transmit, the remaining frames are queued internally by the driver. The TM core is capable of generating parts of the header, the CRC and OCF/CLCW depending on the implementation and configuration of the TM core. The implemented features are selected by setting generics in the VHDL model, the implemented features can be read using the GET_HW_IMPL command. The features enabled is controlled by the SET_CONFIG command. For features available see the hardware manual for the TM core. The hardware generated parts may be overridden by setting the flags of the input TM frame structure accordingly.

Every call to SEND will trigger scheduled TM frames for transmission, calling SEND with the argument set to NULL will thus trigger previously scheduled TM frames for transmission. This might be necessary when interrupts are not used to process descriptors or when interrupt generation for TM frames are disabled, see Section 36.2.4.2.7.

The input to SEND is a pointer to a `grtm_list` data structure described in Section 36.2.4.1. The **head** and **tail** fields of the data structure points to the first and the last TM frame to be scheduled for transmission. The TM frame structure, `grtm_frame`, used is described in Section 36.2.2. The data area length pointed to by the **payload** field is assumed to be at least frame length long. The frame length is set by the **SET_CONFIG** command. The hardware generated parts may be overridden by setting the **flags** field of the TM frame structure accordingly.

Note, that the frame structure and any data pointed to by the frame scheduled for transmission must not be accessed until the frame has been reclaimed using the **ioctl** command RECLAIM.

SEND will fail if the input frame list is incorrectly set up, **errno** will be set to EINVAL in such cases.

36.2.5. Transmission

Transmitting frames are done with the ioctl call using the command SEND and RECLAIM. It is possible to send multiple frames in one call, the frames are provided to the driver using a linked list of frames. See the ioctl commands SEND and RECLAIM for more information.

37. GRCTM driver

37.1. Introduction

This section describes the GRLIB GRCTM (CCSDS Time Manager) device driver interface. The driver implements a simple interface to read and write registers of the core and interrupt handling. The driver supports the on-chip AMBA and the AMBA-over-PCI bus. It relies on the driver manager for device discovery and interrupt handling.

The GRCTM driver require the Driver Manager.

In order to use the driver interface the user must be well acquainted with GRCTM hardware, see hardware manual.

37.1.1. Examples

There is an example available that illustrates how the GRCTM driver interface can be used to configure the GRCTM core. The example application can be configured as a Time-Master or Time- Slave demonstrating both sending and receiving time over TimeWire and how it can be connected to the SPWCUC for time-codes and sending time-packets according to CCSDS Unsegmented Code Transfer Protocol using the RTEMS GRSPW driver.

Note that the example may need to be configured, see the TIME_SYNC_* options.

The example can be built by running:

```
$ cd /opt/rtems-4.10/src/samples/1553
$ make rtems-gr1553bcbm
```

37.1.2. User interface

37.1.2.1. Overview

The GRCTM software driver provides access to the GRCTM core's registers and helps with device detection, driver loading and interrupt handling.

The driver sources and interface definitions are listed in the table below, the path is given relative to the SPARC BSP sources `c/src/lib/libbsp/sparc`.

Table 37.1. GRCTM driver Source location

Filename	Description
shared/time/grctm.c	GRCTM Driver source
shared/include/grctm.h	GRCTM Driver interface declaration

37.1.2.1.1. Accessing the GRCTM core

A GRCTM core is accessed by first opening a specific GRCTM device by calling `grctm_open(INSTANCE_NUMBER)`, after successfully opening a device the returned value of `grctm_open` can be used as input other functions in the GRCTM driver interface. Registers can be accessed and interrupts enabled.

37.1.2.1.2. Interrupt service

The GRCTM core can be programmed to interrupt the CPU on certain events, see hardware manual. All interrupts causes the driver's interrupt service routine (ISR) to be called, it gathers statistics and call the optional user assigned callback. The callback is registered using the function `grctm_int_register()`.

37.1.2.2. Application Programming Interface

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

Table 37.2. function prototypes

Prototype	Description
`void *grctm_open(int minor)`	Open a GRCTM device by instance number, the number is determined by the order in which the core is found (Plug&Play order). The function returns a handle to GRCTM driver used internally, it must be given to all functions in the API.
`void grctm_close(void *grctm)`	Close a previously opened GRCTM driver.
`int spwcuc_reset(void *grctm)`	Reset the GRCTM core by writing to the GRR (Global Reset Register) register of the core.

`void grctm_int_register(`
`  void *grctm,`
`  grctm_isr_t func,`
`  void *data)`
	Register (optional) interrupt callback routine with custom argument. Called from the driver's ISR.

`void grctm_int_enable(void *grctm)`	Enable/unmask GRCTM interrupt on global interrupt controller.
`void grctm_int_disable(void *grctm)`	Disable/unmask GRCTM interrupt on global interrupt controller.

`void grctm_clear_irqs(`
`  void *grctm,`
`  int irqs)`
	Clear pending interrupts by writing to the PICR register. The input is a bit-mask of which interrupt flags to clear.

`void grctm_enable_irqs(`
`  void *grctm,`
`  int irqs)`
	Enable/unmask and/or disable/mask interrupt sources from the GRCTM core by writing the IMR register. The [irqs] argument is a bit-mask written unmodified to the register.

`void grctm_clr_stats (void *grctm)`	Clear statistics gathered by driver.

`void grctm_get_stats(`
`  void *grctm,`
`  struct grctm_stats *stats)`
	Copy driver's current statistics counters to a custom location given by **stats**.

`void grctm_enable_ext_sync (void *grctm)`	Enable external synchronisation (from SPWCUC)
`void grctm_disable_ext_sync (void *grctm)`	Disable external synchronisation (from SPWCUC)
`void grctm_enable_tw_sync (void *grctm)`	Enable TimeWire synchronisation
`void grctm_disable_tw_sync (void *grctm)`	Disable TimeWire synchronisation
`void grctm_disable_fs (void *grctm)`	Disable frequency synthesizer from driving ET
`void grctm_enable_fs (void *grctm)`	Enable frequency synthesizer to driving ET
`unsigned int grctm_get_et_coarse (void *grctm)`	Return elapsed coarse time
`unsigned int grctm_get_et_fine (void *grctm)`	Return elapsed fine time
`unsigned long long grctm_get_et (void *grctm)`	Return elapsed time (coarse and fine)
`int grctm_is_dat_latched (void *grctm, int dat)`	Return 1 if specified datation has been latched
`void grctm_set_dat_edge(void *grctm, int dat, int edge)`	Set triggering edge of datation input
`unsigned int grctm_get_dat_coarse (void *grctm, int dat)`	Return latched datation coarse time
`unsigned int grctm_get_dat_fine (void *grctm, int dat)`	Return latched datation fine time
`unsigned long long grctm_get_dat_et(void *grctm, int dat)`	Return latched datation ET
`unsigned int grctm_get_pulse_reg (void *grctm, int pulse)`	Return current pulse configuration
`void grctm_set_pulse_reg (void *grctm, int pulse,`	Set pulse register

Prototype	Description
`unsigned int val)`	
`void grctm_cfg_pulse(void *grctm, int pulse, int pp, int pw, int pl, int en)`	Configure pulse: pp = period, pw = width, pl = level, en = enable
`void grctm_enable_pulse (void *grctm, int pulse)`	Enable pulse output
`void grctm_disable_pulse (void *grctm, int pulse)`	Disable pulse output
`void grctm_register(void)`	Register the GRCTM driver to Driver Manager

37.1.2.2.1. Data structures

The `grctm_stats` data structure holds statistics gathered by the driver. It can be read by the `grctm_get_stats()` function.

```
struct grctm_stats {
     unsigned int nirqs;
     unsigned int pulse[8];
};
```

Table 37.3. grctm_status member descriptions.

Member	Description
nirqs	Total number of interrupts handled by driver
pulse	Number of interrupts generated by each pulse channel (maximum 8 channels). pulse[N] represents pulse channel N.

38. SPWCUC driver

38.1. Introduction

This section describes the GRLIB SPWCUC (SpaceWire – CCSDS Unsegmented Code Transfer Protocol) device driver interface. The driver implements a simple interface to read and write registers of the core, interrupt handling. The driver supports the on-chip AMBA and the AMBA-over-PCI bus. It relies on the driver manager for device discovery and interrupt handling.

The SPWCUC driver require the Driver Manager.

In order to use the driver interface the user must be well acquainted with SPWCUC hardware, see hardware manual.

38.1.1. Examples

There is an example available that illustrates how the SPWCUC driver interface can be used to configure the SPWCUC core and manage interrupts. The example application can be configured as a Time-Master or Time-Slave demonstrating both sending and receiving SpaceWire time-codes and sending time-packets according to CCSDS Unsegmented Code Transfer Protocol using the RTEMS GRSPW driver.

Note that the example may need to be configured, see the TIME_SYNC_* options.

The example can be built by running:

```
$ cd /opt/rtems-4.10/src/samples/1553
$ make rtems-gr1553bcbm
```

38.2. User interface

38.2.1. Overview

The SPWCUC software driver provides access to the SPWCUC core's registers and helps with device detection, driver loading and interrupt handling.

The driver sources and interface definitions are listed in the table below, the path is given relative to the SPARC BSP sources `c/src/lib/libbsp/sparc`.

Table 38.1. SPWCUC driver Source location

Filename	Description
shared/time/spwcuc.c	SPWCUC Driver source
shared/include/spwcuc.h	SPWCUC Driver interface declaration

38.2.1.1. Accessing the SPWCUC core

A SPWCUC core is accessed by first opening a specific SPWCUC device by calling `spwcuc_open(INSTANCE_NUMBER)`, after successfully opening a device the returned value of `spwcuc_open` can be used as input other functions in the SPWCUC driver interface. Registers can be accessed and interrupts can be enabled.

38.2.1.2. Interrupt service

The SPWCUC core can be programmed to interrupt the CPU on certain events, see hardware manual. All interrupts causes the driver's interrupt service routine (ISR) to be called, it gathers statistics and call the optional user assigned callback. The callback is registered using the function `spwcuc_int_register()`.

38.2.2. Application Programming Interface

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

Table 38.2. function prototypes

Prototype	Description
`void *spwcuc_open(int minor)`	Open a SPWCUC device by instance number, the number is determined by the order in which the core is found (Plug&Play order). The function returns a handle to SPWCUC driver used internally, it must be given to all functions in the API.
`void spwcuc_close(void *spwcuc)`	Close a previously opened SPWCUC driver.
`int spwcuc_reset(void *spwcuc)`	Reset the SPWCUC core by writing to the CONTROL register of the core. This function also clears pending interrupts by writing PICR.

`void spwcuc_config(`
`  void *spwcuc,`
`  struct spwcuc_cfg *cfg)`
	Configure SPWCUC registers according to [cfg] argument. See the data structure description of.

`void spwcuc_int_register(`
`  void *spwcuc,`
`  spwcuc_isr_t func,`
`  void *data)`
	Register (optional) interrupt callback routine with custom argument. Called from the driver's ISR.

`void spwcuc_int_enable(void *spwcuc)`	Enable/unmask SPWCUC interrupt on global interrupt controller
`void spwcuc_int_disable(void *spwcuc)`	Disable/mask SPWCUC interrupt on global interrupt controller

`void spwcuc_clear_irqs(`
`  void *spwcuc,`
`  int irqs)`
	Clear pending interrupts by writing to the PICR register. The input is a bit-mask of which interrupt flags to clear.

`void spwcuc_enable_irqs(`
`  void *spwcuc,`
`  int irqs)`
	Enable/unmask and/or disable/mask interrupt sources from the SPWCUC core by writing the IMR register. The [irqs] argument is a bit-mask written unmodified to the register.

`void spwcuc_clr_stats(void *spwcuc)`	Clear statistics gathered by driver.

`void spwcuc_get_stats(`
`  void *spwcuc,`
`  struct spwcuc_stats *stats)`
	Copy driver's current statistics counters to a custom location given by [stats].

`unsigned int spwcuc_get_et_coarse(` `  void *spwcuc)`	Returns 32-bit received elapsed coarse time, the value is taken from the 'T-Field Coarse Time Packet Register'.
`unsigned int spwcuc_get_et_fine(` `  void *spwcuc)`	Returns 24-bit received elapsed fine time, the value is taken from the 'T-Field Fine Time Packet Register' and shifted down 8 times.
`unsigned long long spwcuc_get_et(void *spwcuc)`	Return 56-bit received elapsed time (ET), a combination of Coarse and Fine time.
`unsigned int spwcuc_get_next_et_coarse(void *spwcuc)`	Return next 32-bit Elapsed Coarse Time.
`unsigned int spwcuc_get_next_et_fine(void *spwcuc)`	Return next 24-bit Elapsed Fine Time.
`unsigned long long spwcuc_get_next_et(void *spwcuc)`	Return next 56-bit elapsed time (combination of next Coarse and Fine Time), this time can be used when generating SpaceWire Time-Packets.
`void spwcuc_force_et(void *spwcuc, unsigned long long time)`	Force/Set the elapsed time (coarse 32-bit and fine 24-bit) by writing the T-Field Time Packet Registers and set the FORCE bit.
`unsigned int spwcuc_get_tp_et_coarse(` `  void *spwcuc)`	Return received 32-bit Elapsed Coarse Time.
`unsigned int spwcuc_get_tp_et_fine(`	Return received 24-bit Elapsed Fine Time.

Prototype	Description
`void *spwcuc)`	
`unsigned long long spwcuc_get_tp_et( void *spwcuc)`	Return received 56-bit Elapsed Time (a combination of coarse and fine).

38.2.2.1. Data structures

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

```
struct spwcuc_cfg {
      unsigned char sel_out;
      unsigned char sel_in;
      unsigned char mapping;
      unsigned char tolerance;
      unsigned char tid;
      unsigned char ctf;
      unsigned char cp;
      unsigned char txen;
      unsigned char rxen;
      unsigned char pktsyncen;
      unsigned char pktiniten;
      unsigned char pktrxen;
      unsigned char dla;
      unsigned char dla_mask;
      unsigned char pid;
      unsigned int offset;
};
```

Table 38.3. spwcuc_cfg member descriptions.

Member	Description
sel_out	Bits 3-0 enable time code transmission on respective output
sel_in	Select SpW to receive time codes on, 0-3
mapping	Define mapping of time code time info into T-field, 0-31
tolerance	Define SpaceWire time code reception tolerance, 0-31
tid	Define CUC P-Field time code identification, 1 = Level 1, 2 = Level 2
ctf	If 1 check time code flags to be all zero
cp	If 1 check P-Field time code id against tid
txen	Enable SpaceWire time code transmission
pktsyncen	Enable SpaceWire time CUC packet sync
pktiniten	Enable SpaceWire time CUC packet init
pktrxen	Enable SpaceWire time CUC packet
dla	SpaceWire destination logical address
dla_mask	SpaceWire destination logical address
pid	SpaceWire protocol ID
offset	Packet reception offset

The `spwcuc_stats` data structure holds statistics gathered by the driver. It can be read by the `spwcuc_get_stats()` function.

```
struct spwcuc_stats {
      unsigned int nirqs;
      unsigned int tick_tx;
      unsigned int tick_tx_wrap;
      unsigned int tick_rx;
      unsigned int tick_rx_wrap;
      unsigned int tick_rx_error;
      unsigned int tolerr;
      unsigned int sync;
      unsigned int syncerr;
      unsigned int wrap;
```

```
        unsigned int wraperr;
        unsigned int pkt_rx;
        unsigned int pkt_err;
        unsigned int pkt_init;
};
```

Table 38.4. spwcuc_cfg member descriptions.

Member	Description
nirqs	Total number of interrupts handled by driver
tick_tx	Number of TickTx interrupts
tick_tx_wrap	Number of TickTxWrap interrupts
tick_rx	Number of TickRx interrupts
tick_rx_wrap	Number of TickRxWrap interrupts
tick_rx_error	Number of TickRxWrap interrupts
tolerr	Number of Tolerance Error interrupts
sync	Number of Sync interrupts
syncerr	Number of Sync Error interrupts
wrap	Number of Wrap interrupts
wraperr	Number of Wrap Error interrupts
pkt_rx	Number of Packet Rx interrupts
pkt_err	Number of Packet Error interrupts
pkt_init	Number of Packet init interrupts

39. GRPWRX GRLIB PacketWire Receiver driver

39.1. Introduction

This document is intended as an aid in getting started developing with Aeroflex Gaisler GRLIB PACKETWIRE RX (GRPWRX) core using the driver described in this document. It describes accessing GRPWRX in a on-chip system and over PCI. It briefly takes the reader through some of the most important steps in using the driver such as starting GRPWRX communication, configuring the driver and receiving GRPWRX packets. The reader is assumed to be well acquainted with GRPWRX and RTEMS.

39.1.1. Software Driver

The driver provides means for threads to receive GRPWRX packets using standard I/O operations.

39.2. User interface

The RTEMS grpwrx driver supports the standard accesses to file descriptors such as **open**, **close** and **ioctl**. User applications include the grpwrx driver's header file which contains definitions of all necessary data structures and bit masks used when accessing the driver.

The driver enables the user to configure the hardware and to receive GRPWRX packets. The allocation of GRPWRX packets is handled by the user and free packets are given to the driver that processes the packets for reception in a two step process. In the first step the driver schedules packets for reception using the DMA descriptors or they are put into an internal queue when all descriptors are in use, in the second step all received packets are put into a second queue that is emptied when the user reclaims the received packets. The reclaimed packets can then be reused in new reception later on.

39.2.1. Driver registration

The registration of the driver is crucial for threads to be able to access the driver using standard means, such as open. The function `grpwrx_register_drv` whose prototype is provided in grpwrx.h is used for registering the driver:

```
grpwrx_register_drv();
```

39.2.2. Opening the device

Opening the device enables the user to access the hardware of a certain grpwrx device. The driver is used for all grpwrx cores available. The cores are separated by assigning each core a unique name and a number called [minor]. The name is given during the opening of the driver. The first three names are printed out:

Table 39.1. Core number to device name conversion.

Core number	Filesystem	Location
0	/dev/grpwrx0	On Chip AMBA bus
1	/dev/grpwrx1	On Chip AMBA bus
2	/dev/grpwrx2	On Chip AMBA bus
0	/dev/rastatmtc0/grpwrx0	GR-RASTA-TMTC PCI Target

An example of an RTEMS open call is shown below.

```
fd = open("/dev/grpwrx0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 39.1.

*Table 39.2. Open **errno** values.*

ERRNO	Description
ENODEV	Illegal device name or not available.

ERRNO	Description
EBUSY	Device already opened.
ENOMEM	Driver failed to allocate necessary memory.

39.2.3. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the `grpwrx` driver.

39.2.4. I/O Control interface

The behaviour of the driver and hardware can be changed via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

he return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly.

All supported commands and their data structures are defined in the grpwrx driver's header file `grpwrx.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

39.2.4.1. Data structures

The `grpwrx_ioc_hw` data structure indicates what features the GRPWRX hardware supports and how it has been configured.

```
 struct grpwrx_ioc_hw {
     unsigned short     fifo_size;
     unsigned short     mode;
     unsigned short     clkdivide;
};
```

Table 39.3. grpwrx_ioc_hw member descriptions.

Member	Description
fifo_size	GRPWRX core FIFO size in number of bytes
mode	GRPWRX core mode, 1=framing mode, 0 = packet mode
clkdivide	GRPWRX physical layer clock divider used

The grpwrx_ioc_config struct is used for configuring the driver and the GRPWRX core.

```
  struct grpwrx_ioc_config {

     int     framing;
     /* Physical layer options */
     unsigned short     phy_clkrise;
     unsigned short     phy_validpos;
     unsigned short     phy_readypos;
     unsigned short     phy_busypos;

     /* Interrupt options */
     unsigned int     enable_cnt;
     int          isr_desc_proc;
     int          blocking;
     rtems_interval   timeout;
};
```

Table 39.4. grpwrx_ioc_config member descriptions.

Member	Description
framing	Enable framing mode (1)

Member	Description
phy_clkrise	Rising clock edge coinciding with serial bit change
phy_validpos	Positive polarity of valid output signal
phy_readypos	Positive polarity of ready input signal
phy_busypos	Positive polarity of busy input signal
enable_cnt	Number of packets between interrupts are generated, zero disables interrupt. Allows user to fine grain interrupt generation
isr_desc_proc	Allow interrupt service routine (ISR) to process descriptors
blocking	Blocking mode select, grpwrx_BLKMODE_POLL for polling mode or grpwrx_BLMODE_BLK for blocking mode
timeout	Blocking mode time out

The grpwrx_packet structure is used in for receiving GRPWRX packets and retrieving received packets, it is the driver's representation of a GRPWRX packet. A GRPWRX packet structure can be chained together using the [next] field in grpwrx_packet.

```
  struct grpwrx_packet {
       unsigned int       flags;
       struct grpwrx_packet       *next;
       int length;
       unsigned int       *payload;
};
```

Table 39.5. grpwrx_ioc_packet member descriptions.

Member	Description
flags	Mask indicating options, transmission state and errors for the packet. GRPWRX_FLAGS_XXX. See Table Table 39.5
next	Points to next GRPWRX packet. This field is used to make driver process multiple GRPWRX packets at the same time, avoiding multiple ioctl calls.
Length	The length of the receive packet in framing mode.
payload	Points to a data area holding the complete GRPWRX packet. The area include fields such as header, payload, OCF, CRC.

Table 39.6. grpwrx_packet flags descriptions.

Flags	Description
GRPWRX_FLAGS_RECEIVED	Indicates whether the packet has been transmitted or not
GRPWRX_FLAGS_ERR	Indicates if errors has been experienced during transmission of the packet
GRPWRX_FLAGS_FHP	Indicates weather to set the First Header Pointer (FPH) flag of the GRPWRX buffer descriptor's word 0. The length of the packet should be 2 and the payload field should point to the location of the CCSDS frame's first header pointer field.
TRANSLATE	Translate packet payload address from CPU address to remote bus (the bus grpwrx is resident on). This is useful when dealing with buffers on remote buses, for example when grpwrx is on a AMBA bus accessed over PCI. This is the case for GR-RASTA-TMTC.

The grpwrx_list structure represents a linked list, a chain of GRPWRX packets. The data structure holds the first packet and last packet in chain.

```
 struct grpwrx_list {
      struct grpwrx_packet *head;
      struct grpwrx_packet *tail;
};
```

Table 39.7. grpwrx_list member descriptions.

Member	Description
head	First GRPWRX packet in chain
tail	Last GRPWRX packet in chain, last packet in list must have it's next field set to NULL

The grpwrx_ioc_stats structure contain statistics collected by the driver.

```
 struct grpwrx_ioc_stats {
      unsigned long long      packets_received;
      unsigned int            err_underrun;
};
```

Table 39.8. grpwrx_ioc_stats member descriptions.

Member	Description
packet_recieved	Number of packets successfully received by the GRPWRX core
err_underrun	Number of AMBA underrun errors

39.2.4.2. Configuration

The grpwrx core and driver are configured using **ioctl** calls. The Table 39.7 below lists all supported **ioctl** calls. grpwrx_IOC_ must be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 39.6.

An example is shown below where the statistics of the driver is copied to the user buffer stats by using an **ioctl** call:

```
struct grpwrx_ioc_stats stats;

result = ioctl(fd, grpwrx_IOC_GET_STATS, &stats);
```

*Table 39.9. ERRNO values for **ioctl** calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The GRPWRX hardware is not in the correct state. Many **ioctl** calls need the GRPWRX core to be in stopped or started mode. One can switch state by calling START or STOP.
ENOMEM	Not enough memory to complete operation. This may cause other **ioctl** commands to fail.
EIO	Writing to hardware failed. Feature not available in hardware.
ENODEV	Operation aborted due to transmitter being stopped.

*Table 39.10. **ioctl** calls supported by the grpwrx driver.*

Call Number	Call Mode	Description
START	Stopped	Exit stopped mode, start the receiver.
STOP	Started	Exit started mode, enter stopped mode. Most of the settings can only be set when in stopped mode.
ISSTARTED	Both	Indicates operating status, started or stopped.
SET_BLOCKING_MODE	Both	Set blocking or non-blocking mode for RECLAIM.
SET_TIMEOUT	Both	Set time out value used in blocking mode to wake up blocked task if request takes too long time to complete.
SET_CONFIG	Stopped	Configure hardware and software driver.
GET_CONFIG	Both	Get current configuration previously set with SET_CONFIG or the driver defaults.

Call Number	Call Mode	Description
GET_STATS	Both	Get statistics collected by driver.
CLR_STATS	Both	Reset driver statistics.
GET_HW_IMPL	Both	Returns the features and implemented by the GRPWRX core.
GET_OCFREG	Both	Returns the address of the OCF/CLCW register, it can be used to update the transmitted OCF/CLCW.
RECLAIM	Both	Returns all GRPWRX packets received since last call to RECLAIM, the packets are linked in a chain.
RECV	Started	Add a chain of GRPWRX packets to the reception queue of the grpwrx driver.

39.2.4.2.1. START

This **ioctl** command enables the GRPWRX receiver and changes the driver's operating status to started. Settings previously set by other **ioctl** commands are written to hardware just before starting reception. It is necessary to enter started mode to be able to receive GRPWRX packets using the **ioctl** command grpwrx_IOC_RECV

The command will fail if the receiver is unable to be brought up, the driver or hardware configuration is invalid or if the GRPWRX core already is started. In case of failure the return code is negative and **errno** will be set to EIO or EINVAL, see Table 39.6.

39.2.4.2.2. STOP

This call makes the GRPWRX core leave started mode and enter stopped mode. The receiver is stopped and no packets will be received. After calling STOP further **ioctl** commands such as RECV, RECLAIM, ISSTARTED, STOP will behave differently or result in error.

The command will fail if the GRPWRX driver already is in stopped mode.

39.2.4.2.3. ISSTARTED

Determines if driver and hardware is in started mode. Errno will be set to EBUSY in stopped mode and return successfully in started mode.

39.2.4.2.4. SET_BLOCKING_MODE

Changes the driver's GRPWRX_IOC_RECLAIM command behaviour. Two modes are available blocking mode and polling mode, in polling mode the **ioctl** command RECLAIM always return directly even when no packets are available. In blocking mode the task calling RECLAIM is blocked until at least one packet can be reclaimed, it is also possible to make the blocked task time out after some time setting the **timeout** value using the SET_CONFIG or SET_TIMEOUT **ioctl** commands.

The argument is set as as described in the table below.

*Table 39.11. SET_BLOCKING_MODE **ioctl** arguments*

Bit number	Description
GRPWRX_BLKMODE_POLL	Enables polling mode
GRPWRX_BLKMODE_BLK	Enables blocking mode

The driver's default is polling mode.

Note that the blocking mode is implemented using the DMA transmit packe interrupt, changing the `isr_desc_proc` parameter of the SET_CONFIG command effects the blocking mode characteristics. For example, enabling interrupt generation every tenth GRPWRX packet will cause the blocked task to be woken up after maximum ten packets when going into blocked mode.

This command never fail.

39.2.4.2.5. SET_TIMEOUT

Sets the blocking mode time out value, instead of blocking for eternity the task will be woken up after this time out expires. The time out value specifies the input to the RTEMS take semaphore operation `rtems_semaphore_obtain()`. See the RTEMS documentation for more information how to set the time out value.

Note that this option has no effect in polling mode.

Note that this option is also set by SET_CONFIG.

This command never fail.

39.2.4.2.6. SET_CONFIG

Configures the driver and core. This call updates the configuration that will be used by the driver during the START command and during operation. Enabling features not implemented by the GRPWRX core will result in EIO error when starting the GRPWRX driver. The hardware features available can be obtained by the GET_HW_IMPL command.

The input is a pointer to an initialized `grpwrx_ioc_config` structure described in Section 39.2.4.1.

Note that the time out value and blocking mode can also be set with SET_TIMEOUT and SET_BLOCKING_MODE.

This call fail if the GRPWRX core is in started mode, in that case **errno** will be set to EBUSY, or if a NULL pointer is given as argument, in that case **errno** will be set to EINVAL.

39.2.4.2.7. GET_CONFIG

Returns the current configuration of the driver and hardware. The current configuration is either the driver and hardware defaults or the configuration previously set by the SET_CONFIG command.

The input to this **ioctl** command is a pointer to a data area of at least the size of a `grpwrx_ioc_config` structure. The data area will be updated according to the `grpwrx_ioc_config` data structure described in Section 39.2.4.1

This command only fail if the pointer argument is invalid.

39.2.4.2.8. GET_STATS

This command copies the driver's internal statistics counters to a user provided data area. The format of the data written is described in the data structure subsection. See the `grpwrx_ioc_stats` data structure.

The call will fail if the pointer to the data is invalid.

39.2.4.2.9. CLR_STATS

This command reset the driver's internal statistics counters.

This command never fail.

39.2.4.2.10. GET_HW_IMPL

This command copies the GRPWRX core's features implemented to a user provided data area. The format of the data written is described in the data structure subsection. See the `grpwrx_ioc_hw` data structure.

Knowing the features supported by hardware can be used to make software run on multiple implementations of the GRPWRX core.

The call will fail if the pointer to the data is invalid.

39.2.4.2.11. RECLAIM

Returns processed GRPWRX oackets to user. All packets returned has been provided by the user in previous calls to RECV, and need not all to have been successfully received. RECLAIM can be configured to operate in polling mode, blocking mode and blocking mode with a time out. In polling mode the task always returns with or without processed packets, in blocking mode the task is blocked until at least one packet has been processed. See the **ioctl** command SET_CONFIG and SET_BLOCKING_MODE to change mode of the RECLAIM command.

RECLAIM stores a linked list of processed GRPWRX packets into the data area pointed to by the user argument. The format for the stored data follows the layout of the grpwrx_list structure described in Section 39.2.2. The grpwrx_list structure holds the first and last GRPWRX packet processed by the driver. The flags field indicates if the packet was received or if errors were experienced during transmission of this packet. See Table 39.5 for flags details.

In started mode, this command enables scheduled GRPWRX packet for transmission as descriptors become free during the processing of received GRPWRX packet.

The call will fail if the pointer to the data area is invalid (EINVAL), the RECLAIM call operates in blocking mode and the time out expires (ETIMEDOUT) or the driver was stopped during the calling task was blocked (ENODEV). See table below.

*Table 39.12. ERRNO values for **RECLAIM***

ERRNO	**Description**
EINVAL	An invalid argument.
ETIMEDOUT	The blocked task was timed out and still no packets was transmitted.
ENODEV	The calling task was woken up from blocking mode by the transmitter being stopped. The GRPWRX driver has has entered stopped mode. Further calls to RECLAIM will retrieve received packet.

39.2.4.2.12. RECV

Scheduling reception of packets is done with the **ioctl** command RECV. The input is a linked list of GRPWRX packets to be scheduled. When all GRPWRX DMA descriptors are active, enabled and linked to a packet to transmit, the remaining packets are queued internally by the driver.

Every call to RECV will trigger scheduled GRPWRX packets for reception, calling RECV with the argument set to NULL will thus trigger previously scheduled GRPWRX packets for reception. This might be necessary when interrupts are not used to process descriptors or when interrupt generation for GRPWRX packets are disabled, see SET_CONFIG.

The input to RECV is a pointer to a `grpwrx_list` data structure described in section Section 39.2.4.1. The **head** and **tail** fields of the data structure points to the first and the last GRPWRX packet to be scheduled for transmission. The GRPWRX packet structure, `grpwrx_packet`, used is described in section Section 39.2.2. The data area to store the received packet is designated by the **payload** field. In packet mode it has to be at lease 64k, in framing mode it has to be the size indicated by the **length** field.

Note, that the packet structure and any data pointed to by the packet scheduled for reception must not be accessed until the packet has been reclaimed using the **ioctl** command RECLAIM.

RECV will fail if the input packet list is incorrectly set up, **errno** will be set to EINVAL in such cases.

39.2.5. Reception

Receiving packets are done with the **ioctl** call using the command RECV and RECLAIM. It is possible to receive multiple packets in one call, the packets are provided to the driver using a linked list of packets. See the **ioctl** commands RECV and RECLAIM for more information.

40. GRAES GRLIB AES DMA driver

40.1. Introduction

This document is intended as an aid in getting started developing with Aeroflex Gaisler GRLIB AES DMA (GRAES) core using the driver described in this document. It describes accessing GRAES in a on- chip system and over PCI. It briefly takes the reader through some of the most important steps in using the driver such as starting the GRAES driver, configuring the driver and en/decrypt AES packets. The reader is assumed to be well acquainted with GRAES, AES and RTEMS.

40.1.1. Software Driver

The driver provides means for threads to receive GRAES packets using standard I/O operations.

40.2. User interface

The RTEMS graes driver supports the standard accesses to file descriptors such as **open**, **close** and **ioctl**. User applications include the **graes** driver's header file which contains definitions of all necessary data structures and bit masks used when accessing the driver.

The driver enables the user to configure the hardware and to de/encode AES packets. The allocation of AES blocks is handled by the user and blocks are given to the driver that processes the blocks in a two step process. In the first step the driver schedules blocks for de/encryption using the DMA descriptors or they are put into an internal queue when all descriptors are in use, in the second step all processed packets are put into a second queue that is emptied when the user reclaims the received blocks. The reclaimed blocks can then be reused in new processing later on.

40.2.1. Driver registration

The registration of the driver is crucial for threads to be able to access the driver using standard means, such as open. The function `graes_register_drv` whose prototype is provided in `graes.h` is used for registering the driver:

```
grpaes_register_drv();
```

40.2.2. Opening the device

Opening the device enables the user to access the hardware of a certain graes device. The driver is used for all graes cores available. The cores are separated by assigning each core a unique name and a number called [minor]. The name is given during the opening of the driver. The first three names are printed out:

Table 40.1. Core number to device name conversion

Core number	Filesystem name	Location
0	/dev/graes0	On Chip AMBA bus
1	/dev/graes1	On Chip AMBA bus
2	/dev/graes2	On Chip AMBA bus
0	/dev/rastatmtc0/graes0	GR-RASTA-GRAESTC PCI Target

An example of an RTEMS open call is shown below.

```
fd = open("/dev/graes0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 40.1.

*Table 40.2. Open **ERRNO** values.*

ERRNO	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened

ERRNO	Description
ENOMEM	Driver failed to allocate necessary memory.

40.2.3. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the `graes` driver.

40.2.4. I/O Control interface

The behaviour of the driver and hardware can be changed via the standard system call **ioctl**. Most operating systems support at least two arguments to **ioctl**, the first being an integer which selects **ioctl** function and secondly a pointer to data that may be interpreted uniquely for each function. A typical **ioctl** call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly.

The return value is 0 on success and -1 on failure and the global **errno** variable is set accordingly.

All supported commands and their data structures are defined in the graes driver's header file `graes.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

40.2.4.1. Data structures

The graes_ioc_hw data structure indicates what features the GRAES hardware supports and how it has been configured.

```
 struct graes_ioc_hw {
      unsigned short      keysize;
};
```

Table 40.3. graes_ioc_hw member descriptions.

Member	Description
keysize	GRAES core key size, fixed 256

The graes_ioc_config struct is used for configuring the driver and the GRAES core.

```
struct graes_ioc_config {

      /* Interrupt options */
      unsigned int      enable_cnt;
      int      isr_desc_proc;
      int      blocking;
      rtems_interval      timeout;
};
```

Table 40.4. graes_ioc_config member descriptions.

Member	Description
enable_cnt	Number of blocks between interrupts are generated, zero disables interrupt. Allows user to fine grain interrupt generation
isr_desc_proc	Allow GRAES interrupt service routine (ISR) to process descriptors
blocking	Blocking mode select, graes_BLKMODE_POLL for polling mode or graes_BLMODE_BLK for blocking mode
timeout	Blocking mode time out

The graes_block structure is used in for queueing GRAES blocks and retriving processed blocks, it is the driver's representation of a GRAES block. A GRAES block structure can be chained together using the **next** field in graes_block.

```
  struct graes_block {
       unsigned int       flags;
       struct graes_block       *next;

       int                          length;
       unsigned char              *key;
       unsigned char              *iv;
       unsigned char              *payload;       /* in */
       unsigned char              *out;        /* out */
};
```

Table 40.5. graes_block member descriptions.

Member	Description
flags	Mask indicating options, Processing state and errors for the block. GRAES_FLAGS_XXX. See Table 40.5
next	Points to next GRAES block. This field is used to make driver process multiple GRAES blocks at the same time, avoiding multiple ioctl calls.
length	Length of the block to de/encrypt
key	Pointer to iAES-2256 key or null
iv	Pointer to initialization vector or null
payload	Pointer to Plaintext/Ciphertext
Out	Pointer to output buffer or null

Table 40.6. graes_block flags descriptions.

Flag	Description
GRAES_BD_ED	When set encryption will be performed otherwise decryption
GRAES_FLAGS_PROCESSED	Indicates whether the block has been processed or not
GRAES_FLAGS_ERR	Indicates if errors has been experienced during processing of the block
TRANSLATE	Translate bllock payload addresses from CPU address to remote bus (the bus graes is resident on). This is useful when dealing with buffers on remote buses, for example when graes is on a AMBA bus accessed over PCI. This is the case for GR-RASTA-GRAESTC.
TRANSLATE_AND_REMEMBER	Translate bllock payload addresses from CPU address to remote bus (the bus graes is resident on). This is useful when dealing with buffers on remote buses, for example when graes is on a AMBA bus accessed over PCI. This is the case for GR-RASTA-GRAESTC.

The graes_list structure represents a linked list, a chain of GRAES blocks. The data structure holds the first block and last block in chain.

```
  struct graes_list {
       struct graes_block *head;
       struct graes_block *tail;
};
```

Table 40.7. graes_list member descriptions.

Member	Description
head	First GRAES block in chain
tail	Last GRAES block in chain, last block in list must have it's next field set to NULL

The graes_ioc_stats structure contain statistics collected by the driver.

```
  struct graes_ioc_stats {
       unsigned long long       blocks_processed;
       unsigned int             err_underrun;
};
```

Table 40.8. graes_ioc_stats member descriptions.

Member	Description
blocks_processed	Number of blocks successfully processed by the GRAES core
err_underrun	Number of AMBA underrun errors

40.2.4.2. Configuration

The graes core and driver are configured using **ioctl** calls. The Table 40.7 below lists all supported **ioctl** calls. graes_IOC_ must be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 40.6.

An example is shown below where the statistics of the driver is copied to the user buffer **stats** by using an **ioctl** call:

```
struct graes_ioc_stats stats;

result = ioctl(fd, graes_IOC_GET_STATS, &stats);
```

*Table 40.9. ERRNO values for **ioctl** calls.*

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The GRAES hardware is not in the correct state. Many **ioctl** calls need the GRAES core to be in stopped or started mode. One can switch state by calling START or STOP.
ENOMEM	Not enough memory to complete operation. This may cause other **ioctl** commands to fail.
EIO	Writing to hardware failed. Feature not available in hardware.
ENODEV	Operation aborted due to GRAES being stopped.

*Table 40.10. **ioctl** calls supported by the graes driver.*

Call Number	Call mode	Description
START	Stopped	Exit stopped mode, start the receiver.
STOP	Started	Exit started mode, enter stopped mode. Most of the settings can only be set when in stopped mode.
ISSTARTED	Both	Indicates operating status, started or stopped.
SET_BLOCKING_MODE	Both	Set blocking or non-blocking mode for RECLAIM.
SET_TIMEOUT	Both	Set time out value used in blocking mode to wake up blocked task if request takes too long time to complete.
SET_CONFIG	Stopped	Configure hardware and software driver.
GET_CONFIG	Both	Get current configuration previously set with SET_CONFIG or the driver defaults
GET_STATS	Both	Get statistics collected by driver
CLR_STATS	Both	Reset driver statistics
GET_HW_IMPL	Both	Returns the features and implemented by the GRAES core.
RECLAIM	Both	Returns all GRAES blocks processed since last call to RECLAIM, the blocks are linked in a chain.

Call Number	Call mode	Description
ENCRYPT	Started	Add a chain of GRAES blocks to the en/decryption queue of the GRAES driver.

40.2.4.2.1. START

This **ioctl** command enables the GRAES core and changes the driver's operating status to started. Settings previously set by other **ioctl** commands are written to hardware just before starting processing.

40.2.4.2.2. STOP

This call makes the GRAES core leave started mode and enter stopped mode. After calling STOP further **ioctl** commands such as ENCRYPT, RECLAIM, ISSTARTED, STOP will behave differently or result in error.

The command will fail if the GRAES driver already is in stopped mode.

40.2.4.2.3. ISSTARTED

Determines if driver and hardware is in started mode. Errno will be set to EBUSY in stopped mode and return successfully in started mode.

40.2.4.2.4. SET_BLOCKING_MODE

Changes the driver's GRAES_IOC_RECLAIM command behaviour. Two modes are available blocking mode and polling mode, in polling mode the **ioctl** command RECLAIM always return directly even when no blocks are available. In blocking mode the task calling RECLAIM is blocked until at least one block can be reclaimed, it is also possible to make the blocked task time out after some time setting the **timeout** value using the SET_CONFIG or SET_TIMEOUT **ioctl** commands.

The argument is set as as described in the table below.

*Table 40.11. SET_BLOCKING_MODE **ioctl** arguments*

Bit number	Description
GRAES_BLKMODE_POLL	Enables polling mode
GRAES_BLKMODE_BLK	Enables blocking mode

The driver's default is polling mode.

Note that the blocking mode is implemented using the DMA de/encrypt block interrupt, changing the `isr_desc_proc` parameter of the SET_CONFIG command effects the blocking mode characteristics. For example, enabling interrupt generation every tenth GRAES block will cause the blocked task to be woken up after maximum ten blocks when going into blocked mode.

This command never fail.

40.2.4.2.5. SET_TIMEOUT

Sets the blocking mode time out value, instead of blocking for eternity the task will be woken up after this time out expires. The time out value specifies the input to the RTEMS take semaphore operation `rtems_semaphore_obtain()`. See the RTEMS documentation for more information how to set the time out value.

Note that this option has no effect in polling mode.

Note that this option is also set by SET_CONFIG.

This command never fail.

40.2.4.2.6. SET_CONFIG

Configures the driver and core. This call updates the configuration that will be used by the driver during the START command and during operation. Enabling features not implemented by the GRAES core will result in EIO error when starting the GRAES driver. The hardware features available can be obtained by the GET_HW_IMPL command.

The input is a pointer to an initialized `graes_ioc_config` structure described in Section 40.2.4.2.1.

Note that the time out value and blocking mode can also be set with SET_TIMEOUT and SET_BLOCKING_MODE.

This call fail if the GRAES core is in started mode, in that case **errno** will be set to EBUSY, or if a NULL pointer is given as argument, in that case **errno** will be set to EINVAL.

40.2.4.2.7. GET_CONFIG

Returns the current configuration of the driver and hardware. The current configuration is either the driver and hardware defaults or the configuration previously set by the SET_CONFIG command.

The input to this ioctl command is a pointer to a data area of at least the size of a `graes_ioc_config` structure. The data area will be updated according to the `graes_ioc_config` data structure described in Section 40.2.4.2.1.

This command only fail if the pointer argument is invalid.

40.2.4.2.8. GET_STATS

This command copies the driver's internal statistics counters to a user provided data area. The format of the data written is described in the data structure subsection. See the `graes_ioc_stats` data structure.

The call will fail if the pointer to the data is invalid.

40.2.4.2.9. CLR_STATS

This command reset the driver's internal statistics counters.

This command never fail.

40.2.4.2.10. GET_HW_IMPL

This command copies the GRAES core's features implemented to a user provided data area. The format of the data written is described in the data structure subsection. See the `graes_ioc_hw` data structure.

Knowing the features supported by hardware can be used to make software run on multiple implementations of the GRAES core.

The call will fail if the pointer to the data is invalid.

40.2.4.2.11. RECLAIM

Returns processed GRAES block to user. All blocks returned has been provided by the user in previous calls to ENCRYPT, and need not all to have been successfully de/encrypted. RECLAIM can be configured to operate in polling mode, blocking mode and blocking mode with a time out. In polling mode the task always returns with or without processed packets, in blocking mode the task is blocked until at least one packet has been processed. See the **ioctl** command SET_CONFIG and SET_BLOCKING_MODE to change mode of the RECLAIM command.

RECLAIM stores a linked list of processed GRAES blocks into the data area pointed to by the user argument. The format for the stored data follows the layout of the `graes_list` structure described in Section 40.2.2. The `graes_list` structure holds the first and last GRAES block processed by the driver. The `flags` field indicates if the block was received or if errors were experienced during processing of this packet. See Table 40.6 for `flags` details.

In started mode, this command enables scheduled GRAES block for de/encryption as descriptors become free during the processing of GRAES blocks.

The call will fail if the pointer to the data area is invalid (EINVAL), the RECLAIM call operates in blocking mode and the time out expires (ETIMEDOUT) or the driver was stopped during the calling task was blocked (ENODEV). See table below.

*Table 40.12. ERRNO values for **RECLAIM***

ERRNO	Description
EINVAL	An invalid argument.
ETIMEDOUT	The blocked task was timed out and still no blocks was processed.
ENODEV	The calling task was woken up from blocking mode by the GRAES code being stopped. The GRAES driver has has entered stopped mode. Further calls to RECLAIM will retrieve processed packet.

40.2.4.2.12. ENCRYPT

Scheduling de/encryption of block is done with the **ioctl** command ENCRYPT. The input is a linked list of GRAES blocks to be scheduled. When all GRAES DMA descriptors are active, enabled and linked to a block, the remaining blocks are queued internally by the driver.

Every call to ENCRYPT will trigger scheduled GRAES blocks for de/encryption, calling PROCESS with the argument set to NULL will thus trigger previously scheduled GRAES blocks for de/encryption. This might be necessary when interrupts are not used to process descriptors or when interrupt generation for GRAES blocks are disabled, see SET_CONFIG.

The input to ENCRYPT is a pointer to a `graes_list` data structure described in Section 40.2.4.2.1. The **head** and **tail** fields of the data structure points to the first and the last GRAES block to be scheduled for de/encryption. The GRAES block structure, `graes_block`, used is described in Section 40.2.2, the data field corresponding to the GRAES buffer descriptor fields.

Note, that the block structure and any data pointed to by the block scheduled for de/encryption must not be accessed until the block has been reclaimed using the **ioctl** command RECLAIM.

ENCRYPT will fail if the input block list is incorrectly set up, **errno** will be set to EINVAL in such cases.

40.2.5. De/encryption

De/encrypting blocks is done with the **ioctl** call using the command ENCRYPT and RECLAIM. It is possible to de/encrypt multiple blocks in one call, the blocks are provided to the driver using a linked list of blocks. See the ioctl commands ENCRYPT and RECLAIM for more information.

Cobham Gaisler AB
Kungsgatan 12
411 19 Gothenburg
Sweden
www.cobham.com/gaisler
sales@gaisler.com
T: +46 31 7758650
F: +46 31 421407

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

Copyright © 2015 Cobham Gaisler AB