Gaisler RTEMS driver documentation

Software Drivers for Aeroflex Gaisler RTEMS distribution

GR-RTEMS-DRIVER
Version 1.2.1.0
March 2012

Written by Daniel Hellström, Kristoffer Glembo, Marko Isomäki

Kungsgatan 12	tel	+46 31 7758650
411 19 Göteborg	fax	+46 31 421407
Sweden		

www.aeroflex.com/gaisler

Table of Contents

1 Drivers documentation introduction....12

2 GRLIB AMBA Plug & Play....13
2.1 Introduction....13
2.1.1 AMBA Plug&Play terms and names....13
2.1.2 Sources....13
2.2 Overview....14
2.3 Initialization....14
2.4 Finding AMBAPP devices by Plug&Play....15
2.5 Allocating a device....16
2.6 Name database....16
2.7 Frequency of a device....17

3 Driver Manager....18
3.1 Introduction....18
3.1.1 Driver manager terms and names....18
3.1.2 Sources....18
3.2 Overview....19
3.2.1 Bus and bus driver....20
3.2.1.1 Bus specific device information....22
3.2.2 Root driver....22
3.2.3 Device driver....22
3.2.4 Device....23
3.2.5 Driver resources....24
3.2.6 Driver interface....25
3.3 Configuration....25
3.3.1 Available LEON drivers....27
3.4 Initialization....29
3.4.1 LEON3 BSP....29
3.5 Interrupt....29
3.6 Address translation....30
3.7 Function interface....31

4 RMAP Stack....32
4.1 Introduction....32
4.1.1 Examples....32
4.2 Driver interface....32
4.3 Logical and Path Addressing....32
4.4 Zero-Copy implementation....32
4.5 RMAP GRSPW Driver....33
4.6 Thread-safe....33
4.7 User interface....33
4.7.1 Data structures....33
4.7.2 Function interface description....36

5 SpaceWire Network model....38
5.1 Introduction....38
5.2 Overview....38
5.3 Requirements....38
5.4 Node description....38
5.4.1 The Node ID....39
5.5 Read and Write operation....39
5.6 Interrupt handling....39
5.7 Using the SpaceWire Bus Driver....39

6 AMBA over SpaceWire....40

6.1 Introduction........40
6.2 Overview........40
6.3 Requirements........40
6.4 Interrupt handling........40
6.5 Memory allocation on Target........40
6.6 Differences between on-chip AMBA drivers........41

7 SPARC/LEON PCI DRIVERS........42
7.1 INTRODUCTION........42
7.1.1 Examples........42
7.2 Sources........42
7.3 Configuration........42
7.3.1 GRPCI........43
7.3.2 GRPCI2........43
7.3.3 AT697........43
7.4 User interface........44
7.4.1 PCI Address space........44
7.4.2 PCI Interrupt........44
7.4.3 PCI Endianess........45

8 GR-RASTA-ADCDAC PCI TARGET........46

9 GR-RASTA-IO PCI TARGET........47

10 GR-RASTA-TMTC PCI TARGET........48

11 GR-RASTA-SPW_ROUTER PCI TARGET........49

12 Gaisler SpaceWire (GRSPW)........50
12.1 Introduction........50
12.1.1 Software driver........50
12.1.2 Examples........50
12.1.3 Support........50
12.2 User interface........50
12.2.1 Driver registration........50
12.2.2 Driver resource configuration........51
12.2.2.1 Custom DMA area parameters........51
12.2.3 Opening the device........51
12.2.4 Closing the device........52
12.2.5 I/O Control interface........52
12.2.5.1 Data structures........53
12.2.5.2 Configuration........56
12.2.6 Transmission........63
12.2.7 Reception........64
12.3 Receiver example........65

13 SpaceWire ROUTER........66
13.1 Introduction........66
13.1.1 SpaceWire Router register driver........66
13.1.2 AMBA port driver........66

14 SpaceWire ROUTER Register Driver........67
14.1 Introduction........67
14.2 User interface........67
14.2.1 Driver registration........67
14.2.2 Driver resource configuration........67
14.2.3 Opening the device........67
14.2.4 Closing the device........68
14.2.5 I/O Control interface........68
14.2.5.1 HWINFO........70
14.2.5.2 CFG_SET........71

14.2.5.3 CFG_GET ... 72
14.2.5.4 ROUTES_SET ... 72
14.2.5.5 ROUTES_GET ... 72
14.2.5.6 PS_SET ... 72
14.2.5.7 PS_GET ... 73
14.2.5.8 WE_SET ... 73
14.2.5.9 PORT ... 73
14.2.5.10 CFGSTS_SET ... 74
14.2.5.11 CFGSTS_GET ... 74
14.2.5.12 TC_GET ... 74

15 GR1553B DRIVER ... 75
15.1 INTRODUCTION ... 75
15.1.1 GR1553B Hardware ... 75
15.1.2 Software Driver ... 75
15.1.3 Driver Registration ... 75

16 GR1553B REMOTE TERMINAL DRIVER ... 76
16.1 INTRODUCTION ... 76
16.1.1 GR1553B Remote Terminal Hardware ... 76
16.1.2 Examples ... 76
16.2 User Interface ... 76
16.2.1 Overview ... 76
16.2.1.1 Accessing an RT device ... 77
16.2.1.2 Introduction to the RT Memory areas ... 77
16.2.1.3 Sub Address Table ... 78
16.2.1.4 Descriptors ... 78
16.2.1.5 Data Buffers ... 79
16.2.1.6 Event Logging ... 79
16.2.1.7 Interrupt service ... 79
16.2.1.8 Indication service ... 80
16.2.1.9 Mode Code support ... 80
16.2.1.10 RT Time ... 80
16.2.2 Application Programming Interface ... 80
16.2.2.1 Data structures ... 82
16.2.2.2 gr1553rt_open ... 84
16.2.2.3 gr1553rt_close ... 84
16.2.2.4 gr1553rt_config ... 85
16.2.2.5 gr1553rt_start ... 85
16.2.2.6 gr1553rt_stop ... 85
16.2.2.7 gr1553rt_status ... 85
16.2.2.8 gr1553rt_indication ... 85
16.2.2.9 gr1553rt_evlog_read ... 85
16.2.2.10 gr1553rt_set_vecword ... 85
16.2.2.11 gr1553rt_set_bussts ... 86
16.2.2.12 gr1553rt_sa_setopts ... 86
16.2.2.13 gr1553rt_list_sa ... 86
16.2.2.14 gr1553rt_sa_schedule ... 86
16.2.2.15 gr1553rt_irq_err ... 86
16.2.2.16 gr1553rt_irq_mc ... 86
16.2.2.17 gr1553rt_irq_sa ... 87
16.2.2.18 gr1553rt_list_init ... 87
16.2.2.19 gr1553rt_bd_init ... 87
16.2.2.20 gr1553rt_bd_update ... 88

17 GR1553B BUS MONITOR DRIVER ... 89
17.1 INTRODUCTION ... 89
17.1.1 GR1553B Remote Terminal Hardware ... 89
17.1.2 Examples ... 89

17.2 User Interface....89
17.2.1 Overview....89
17.2.1.1 Accessing a BM device....90
17.2.1.2 BM Log memory....90
17.2.1.3 Accessing the BM Log memory....90
17.2.1.4 Time....91
17.2.1.5 Filtering....91
17.2.1.6 Interrupt service....91
17.2.2 Application Programming Interface....91
17.2.2.1 Data structures....92
17.2.2.2 gr1553bm_open....95
17.2.2.3 gr1553bm_close....95
17.2.2.4 gr1553bm_config....95
17.2.2.5 gr1553bm_start....95
17.2.2.6 gr1553bm_stop....96
17.2.2.7 gr1553bm_time....96
17.2.2.8 gr1553bm_available....96
17.2.2.9 gr1553bm_read....96

18 GR1553B Bus Controller DRIVER....97
18.1 INTRODUCTION....97
18.1.1 GR1553B Bus Controller Hardware....97
18.1.2 Software Driver....97
18.1.3 Examples....98
18.2 BC Device Handling....98
18.2.1 Device API....98
18.2.1.1 Data Structures....99
18.2.1.2 gr1553bc_open....99
18.2.1.3 gr1553bc_close....100
18.2.1.4 gr1553bc_start....100
18.2.1.5 gr1553bc_pause....100
18.2.1.6 gr1553bc_resume....100
18.2.1.7 gr1553bc_stop....100
18.2.1.8 gr1553bc_indication....100
18.2.1.9 gr1553bc_status....100
18.2.1.10 gr1553bc_ext_trig....101
18.2.1.11 gr1553bc_irq_setup....101
18.3 Descriptor List Handling....101
18.3.1 Overview....101
18.3.2 Example: steps for creating a list....102
18.3.3 Major Frame....103
18.3.4 Minor Frame....103
18.3.5 Slot (Descriptor)....103
18.3.6 Changing a scheduled BC list (during BC-runtime)....104
18.3.7 Custom Memory Setup....105
18.3.8 Interrupt handling....105
18.3.9 List API....106
18.3.9.1 Data structures....109
18.3.9.2 gr1553bc_list_alloc....110
18.3.9.3 gr1553bc_list_free....110
18.3.9.4 gr1553bc_list_config....110
18.3.9.5 gr1553bc_list_link_major....111
18.3.9.6 gr1553bc_list_set_major....111
18.3.9.7 gr1553bc_minor_table_size....111
18.3.9.8 gr1553bc_list_table_size....111
18.3.9.9 gr1553bc_list_table_alloc....111
18.3.9.10 gr1553bc_list_table_free....111
18.3.9.11 gr1553bc_list_table_build....111

18.3.9.12 gr1553bc_major_alloc_skel....112
18.3.9.13 gr1553bc_list_freetime....112
18.3.9.14 gr1553bc_slot_alloc....112
18.3.9.15 gr1553bc_slot_free....112
18.3.9.16 gr1553bc_mid_from_bd....112
18.3.9.17 gr1553bc_slot_bd....112
18.3.9.18 gr1553bc_slot_irq_prepare....112
18.3.9.19 gr1553bc_slot_irq_enable....113
18.3.9.20 gr1553bc_slot_irq_disable....113
18.3.9.21 gr1553bc_slot_jump....113
18.3.9.22 gr1553bc_slot_exttrig....113
18.3.9.23 gr1553bc_slot_transfer....113
18.3.9.24 gr1553bc_slot_dummy....114
18.3.9.25 gr1553bc_slot_empty....114
18.3.9.26 gr1553bc_slot_update....114
18.3.9.27 gr1553bc_slot_raw....115
18.3.9.28 gr1553bc_show_list....115

19 Gaisler B1553BRM DRIVER (BRM)....116
19.1 INTRODUCTION....116
19.1.1 BRM Hardware....116
19.1.2 Software Driver....116
19.1.3 Supported OS....116
19.1.4 Examples....116
19.2 User interface....116
19.2.1 Driver registration....117
19.2.2 Driver resource configuration....117
19.2.2.1 Custom DMA area parameter....117
19.2.3 Opening the device....117
19.2.4 Closing the device....118
19.2.5 I/O Control interface....118
19.2.5.1 Data structures....119
19.2.6 Configuration....122
19.2.6.1 SET_MODE....123
19.2.6.2 SET_BUS....124
19.2.6.3 SET_MSGTO....124
19.2.6.4 SET_RT_ADDR....124
19.2.6.5 BRM_SET_STD....124
19.2.6.6 BRM_SET_BCE....124
19.2.6.7 BRM_TX_BLOCK....124
19.2.6.8 BRM_RX_BLOCK....124
19.2.6.9 BRM_CLR_STATUS....124
19.2.6.10 BRM_GET_STATUS....124
19.2.6.11 BRM_SET_EVENTID....125
19.2.7 Remote Terminal operation....125
19.2.8 Bus Controller operation....126
19.2.9 Bus monitor operation....127

20 Gaisler B1553RT DRIVER (RT)....128
20.1 INTRODUCTION....128
20.1.1 RT Hardware....128
20.1.2 Examples....128
20.2 User interface....128
20.2.1 Driver registration....128
20.2.2 Driver resource configuration....128
20.2.2.1 Custom DMA area parameter....129
20.2.3 Opening the device....129
20.2.4 Closing the device....130
20.2.5 I/O Control interface....130

20.2.5.1 Data structures....130
20.2.6 Configuration....132
20.2.6.1 RT_SET_ADDR....133
20.2.6.2 RT_SET_BCE....133
20.2.6.3 RT_SET_VECTORW....133
20.2.6.4 RT_RX_BLOCK....133
20.2.6.5 RT_SET_EXTMDATA....133
20.2.6.6 RT_CLR_STATUS....133
20.2.6.7 RT_GET_STATUS....133
20.2.6.8 RT_SET_EVENTID....133
20.2.7 Remote Terminal operation....134

21 CAN DRIVER INTERFACE (GRCAN)....135
21.1 User interface....135
21.1.1 Driver registration....135
21.1.2 Driver resource configuration....135
21.1.2.1 Custom DMA area parameters....135
21.1.3 Opening the device....135
21.1.4 Closing the device....136
21.1.5 I/O Control interface....136
21.1.5.1 Data structures....136
21.1.5.2 Configuration....139
21.1.6 Transmission....143
21.1.7 Reception....144

22 Gaisler Opencores CAN driver (OC_CAN)....146
22.1 INTRODUCTION....146
22.1.1 CAN Hardware....146
22.1.2 Software Driver....146
22.1.3 Supported OS....146
22.1.4 Examples....146
22.1.5 Support....146
22.2 User interface....146
22.2.1 Driver registration....147
22.2.2 Driver resource configuration....147
22.2.3 Opening the device....147
22.2.4 Closing the device....147
22.2.5 I/O Control interface....148
22.2.5.1 Data structures....148
22.2.5.2 Configuration....151
22.2.6 Transmission....154
22.2.7 Reception....154

23 Gaisler SatCAN FPGA driver (SatCAN)....156
23.1 INTRODUCTION....156
23.1.1 SatCAN Hardware Wrapper....156
23.1.2 Software Driver....156
23.1.3 Supported OS....156
23.1.4 Examples....156
23.1.5 Support....156
23.2 User interface....156
23.2.1 Driver registration....156
23.2.2 Opening the device....158
23.2.3 Closing the device....158
23.2.4 Reading from the device....158
23.2.5 Writing to the device....159
23.2.6 I/O Control interface....160
23.2.6.1 Data structures....160
23.2.6.2 Configuration....161

24 Gaisler CAN_MUX driver (CAN_MUX)....168
24.1 INTRODUCTION....168
24.1.1 CAN_MUX Hardware....168
24.1.2 Software Driver....168
24.1.3 Supported OS....168
24.1.4 Examples....168
24.1.5 Support....168
24.2 User interface....168
24.2.1 Driver registration....168
24.2.2 Opening the device....168
24.2.3 Closing the device....169
24.2.4 I/O Control interface....169
24.2.4.1 Configuration....169

25 Gaisler ASCS (GRASCS)....171
25.1 Introduction....171
25.1.1 Software driver....171
25.1.2 Examples....171
25.1.3 Support....171
25.2 User interface....171
25.2.1 ASCS_init....173
25.2.2 ASCS_input_select....173
25.2.3 ASCS_etr_select....174
25.2.4 ASCS_start....174
25.2.5 ASCS_stop....174
25.2.6 ASCS_iface_status....174
25.2.7 ASCS_TC_send....175
25.2.8 ASCS_TC_send_block....175
25.2.9 ASCS_TC_sync_start....175
25.2.10 ASCS_TC_sync_stop....177
25.2.11 ASCS_TM_recv....177
25.2.12 ASCS_TM_recv_block....177
25.3 Example code....178

26 RAW UART DRIVER INTERFACE (APBUART)....179
26.1 User interface....179
26.1.1 Driver registration....179
26.1.2 Driver resource configuration....179
26.1.3 Opening the device....179
26.1.4 Closing the device....180
26.1.5 I/O Control interface....180
26.1.5.1 Configuration....180
26.1.6 Transmission....183
26.1.7 Reception....183

27 Gaisler SPICTRL SPI DRIVER (SPICTRL)....185
27.1 INTRODUCTION....185
27.1.1 SPI Hardware....185
27.1.2 Examples....185
27.2 User interface....185
27.2.1 Driver registration....185
27.2.2 Accessing the SPI bus....185
27.2.3 Extensions to the standard RTEMS interface....186
27.2.3.1 PERIOD_START....187
27.2.3.2 PERIOD_STOP....187
27.2.3.3 CONFIG....187
27.2.3.4 STATUS....188
27.2.3.5 PERIOD_WRITE....188
27.2.3.6 PERIOD_READ....189

28 Gaisler i2C Master DRIVER (I2CMST)....190
28.1 INTRODUCTION....190
28.1.1 I2C Hardware....190
28.1.2 Examples....190
28.2 User interface....190
28.2.1 Driver registration....190
28.2.2 Accessing the I2C bus....190

29 GPIO Library....192
29.1 INTRODUCTION....192
29.1.1 Examples....192
29.2 Driver interface....192
29.3 User interface....192
29.3.1 Accessing a GPIO port....192
29.3.2 Interrupt handler registration....193
29.3.3 Data structures....193
29.3.4 Function prototype description....194
29.3.4.1 GPIO Library functions....194

30 Gaisler GPIO DRIVER (GRGPIO)....196
30.1 INTRODUCTION....196
30.1.1 GPIO Hardware....196
30.1.2 Examples....196
30.2 User interface....196
30.2.1 Driver registration....196
30.2.2 Driver resource configuration....196
30.2.3 Accessing GPIO ports....197

31 Gaisler ADC/DAC DRIVER (GRADCDAC)....200
31.1 INTRODUCTION....200
31.1.1 ADC/DAC Hardware....200
31.1.2 Examples....200
31.2 User interface....200
31.2.1 Driver registration....200
31.2.2 Driver resource configuration....200
31.2.3 Accessing ADC/DAC....201
31.2.4 Interrupt handler registration....201
31.2.5 Data structures....202
31.2.6 Function prototype description....205
31.2.6.1 General ADC/DAC functions....205
31.2.6.2 Status interpretation help function....207
31.2.6.3 ADC functions....207
31.2.6.4 DAC functions....208

32 Gaisler TC driver (GRTC)....209
32.1 INTRODUCTION....209
32.1.1 TC Hardware....209
32.1.2 Software Driver....209
32.1.2.1 GRTC over SpaceWire....209
32.1.3 Support....209
32.2 User interface....209
32.2.1 Driver registration....210
32.2.2 Opening the device....210
32.2.3 Closing the device....210
32.2.4 I/O Control interface....211
32.2.4.1 Data structures....211
32.2.4.2 Configuration....215
32.2.5 Operating mode....221
32.2.5.1 Driver frame pools....221

32.2.6 Reception in FRAME mode......222
32.2.7 Reception using RAW mode......222

33 Gaisler TM driver (GRTM)......223
33.1 INTRODUCTION......223
33.1.1 TM Hardware......223
33.1.2 Software Driver......223
33.1.2.1 GRTM over SpaceWire......223
33.1.3 Support......223
33.2 User interface......223
33.2.1 Driver registration......224
33.2.2 Opening the device......224
33.2.3 Closing the device......225
33.2.4 I/O Control interface......225
33.2.4.1 Data structures......225
33.2.4.2 Configuration......230
33.2.5 Transmission......235

34 GRCTM DRIVER......236
34.1 INTRODUCTION......236
34.1.1 Examples......236
34.2 User Interface......236
34.2.1 Overview......236
34.2.1.1 Accessing the GRCTM core......236
34.2.1.2 Interrupt service......237
34.2.2 Application Programming Interface......237
34.2.2.1 Data structures......239

35 SPWCUC DRIVER......240
35.1 INTRODUCTION......240
35.1.1 Examples......240
35.2 User Interface......240
35.2.1 Overview......240
35.2.1.1 Accessing the SPWCUC core......240
35.2.1.2 Interrupt service......241
35.2.2 Application Programming Interface......241
35.2.2.1 Data structures......243

36 Gaisler Packetwire RX driver (GRPWRX)......246
36.1 INTRODUCTION......246
36.1.1 Software Driver......246
36.1.2 Support......246
36.2 User interface......246
36.2.1 Driver registration......246
36.2.2 Opening the device......246
36.2.3 Closing the device......247
36.2.4 I/O Control interface......247
36.2.4.1 Data structures......247
36.2.4.2 Configuration......250
36.2.5 Reception......255

37 Gaisler AES DMA driver (GRAES)......256
37.1 INTRODUCTION......256
37.1.1 Software Driver......256
37.1.2 Support......256
37.2 User interface......256
37.2.1 Driver registration......256
37.2.2 Opening the device......256
37.2.3 Closing the device......257
37.2.4 I/O Control interface......257

37.2.4.1 Data structures....257
37.2.4.2 Configuration....260
37.2.5 De/encryption....264

38 Support....265

1 Drivers documentation 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/examples/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

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 Aeroflex Gaisler RTEMS 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.

Term	Description
AMBAPP, AMBA PnP	AMBA Plug&Play
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

Table 1: AMBA Layer terms and names

2.1.2 Sources

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

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_depth.c	Function to get bus depth of a device
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

Table 2: AMBAPP Layer Sources

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

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 Aeroflex Gaisler RTEMS-4.10 SPARC distribution. It is located in *cpukit/libdrvmgr* in the 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.

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 driver	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

Table 3: Driver Manager terms and names

3.1.2 Sources

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

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

Table 4: Driver Manager Sources

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 component 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.

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
int_unmask	Manual unmask (enable) interrupt source at interrupt controller

Table 5: Interrupt backend interface of driver manager

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.

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.

Table 6: Root device driver entry points for LEON systems

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_.

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_GR1553RT
GR1553B BM	AMBAPP_GAISLER_GR1553BM
GR1553B BC	AMBAPP_GAISLER_GR1553BC
B1553BRM	AMBAPP_GAISLER_B1553BRM
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 target	PCI_GR_RASTA_IO
GR-RASTA-TMTC PCI target	PCI_GR_RASTA_TMTC
GR-701 PCI target	PCI_GR_701
GR-TMTC-1553 PCI target	PCI_GR_TMTC_1553
GR-RASTA-SPW-ROUTER PCI Target	PCI_GR_RASTA_SPW_ROUTER

Table 7: LEON device drivers available

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 BSP

In the Aeroflex Gaisler RTEMS 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.

Operation	Description
drvmgr_interrupt_register	Register an interrupt service routine (ISR) and unmask(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

Table 8: Driver interrupt interface

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 device to set up address maps in at least one direction. If a bus does not support DMA for example, only the CPU 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 -1 if no map matches the translation requested, or 0 is successful. If a bridge has no map, the addresses are translated 1:1 (not changed).

```
extern int drvmgr_translate(
     struct drvmgr_dev *dev,
     int cpu_addresses,
     int upstream,
     void *src_address,
     void **dst_address);
```

Access method	**Translate direction**	**Example usage**	**Arguments**
Downstream, CPU access	From hardware address to CPU bus address	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	dev=PCI-dev cpu_address=1 upsteam=0
Downstream, CPU access	From CPU bus address to hardware address.	Translate an address the CPU access to get to a PCI device into the actual PCI address	dev=PCI-dev cpu_address=1 upsteam=1
Upstream, DMA access	From hardware address to CPU bus address	The CPU reads out the the DMA address from a descriptor, it can then translate it into the memory address the CPU can access since the memory is located at the CPU bus	dev=PCI-dev cpu_address=0 upsteam=1
Upstream, DMA access	From CPU bus address to hardware address.	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	dev=PCI-dev cpu_address=0 upsteam=0

Table 9: Translate options to *drvmgr_translate()*

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.

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 1. All paths are given relative the RTEMS kernel source root.

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

Table 10: RMAP stack source location

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;
}
```

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.
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.

Table 11: rmap_config members

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;
}
```

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.

Table 12: rmap_command members

4.7.2 Function interface description

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

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

Table 13: RMAP stack function prototypes

4.7.2.1.1 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.

4.7.2.1.2 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.

4.7.2.1.3 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.

4.7.2.1.4 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.

4.7.2.1.5 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 on the SpaceWire network and a RMAP stack handle used to communicate with the target nodes.

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 SPARC/LEON PCI DRIVERS

7.1 INTRODUCTION

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

- GRPCI2
- GRPCI
- PCIF
- AT697 PCI

The PCI drivers require the Driver Manager and PCI Library available in the Aeroflex Gaisler RTEMS distribution. The PCI Library documentation is available in the *doc/user* directory in the Aeroflex Gaisler RTEMS 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 Aeroflex Gaisler RTEMS 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.

Location	Description
.../libbsp/sparc/leon2/pci/at697_pci.c	AT697 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

Table 14: PCI driver source location

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.

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#

Table 15: PCI Host driver parameter description

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	PTR	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
tgtbar1	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 7.3.

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 TARGET

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

The GR-RASTA-ADCDAC driver require the RTEMS 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 TARGET

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

The GR-RASTA-IO driver require the RTEMS 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 TARGET

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

The GR-RASTA-TMTC driver require the RTEMS 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 TARGET

This section describes the GR-RASTA-SPW_ROUTER PCI target 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 Gaisler SpaceWire (GRSPW)

12.1 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.

12.1.1 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.

12.1.2 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.

12.1.3 Support

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

12.2 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.

12.2.1 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.

12.2.2 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.

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.
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.

Table 16: GRSPW driver parameter description

12.2.2.1 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.

12.2.3 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.

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

Table 17: Device number to device name conversion.

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 16.

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

Table 18: Open errno values.

12.2.4 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.

12.2.5 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.

12.2.5.1 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;
```

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.

Table 19: spw_ioctl_packetsize member descriptions.

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;
```

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.

Table 20: spw_ioctl_pkt_send member descriptions.

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;
```

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.

Table 21: spw_stats member descriptions.

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;
```

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

Table 22: spw_config member descriptions.

12.2.5.2 Configuration

The GRSPW core and driver are configured using ioctl calls. Table 18 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 17.

An example is shown below where the node address of a device previously opened with *open* is set to 254 by using an ioctl call:

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

Operating system specific calls are described last in this section.

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 23: ERRNO values for ioctl calls.

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.
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)

Table 24: ioctl calls supported by the GRSPW driver.

12.2.5.2.1 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.

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.

Table 25: START argument description

12.2.5.2.2 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.

12.2.5.2.3 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.

12.2.5.2.4 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.

12.2.5.2.5 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.

12.2.5.2.6 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.

12.2.5.2.7 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.

12.2.5.2.8 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.

12.2.5.2.9 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.

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.

Table 26: SET_COREFREQ argument description

12.2.5.2.10 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.

12.2.5.2.11 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.

12.2.5.2.12 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.

12.2.5.2.13 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.

12.2.5.2.14 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.

12.2.5.2.15 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.

12.2.5.2.16 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.

12.2.5.2.17 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.

12.2.5.2.18 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.

12.2.5.2.19 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.

12.2.5.2.20 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.

12.2.5.2.21 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.

12.2.5.2.22 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.

12.2.5.2.23 CLR_STATISTICS

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

12.2.5.2.24 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.

12.2.5.2.25 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.

12.2.5.2.26 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.

12.2.5.2.27 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.

12.2.5.2.28 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);
```

12.2.5.2.29 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.

12.2.5.2.30 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.

12.2.6 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 data from 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.

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).

Table 27: ERRNO values for write and ioctl send.

12.2.7 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.

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.

Table 28: ERRNO values for read calls.

12.3 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 */
    ...
}
```

13 SpaceWire ROUTER

13.1 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.

13.1.1 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.

13.1.2 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.

14 SpaceWire ROUTER Register Driver

14.1 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.

14.2 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.

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 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.

14.2.2 Driver resource configuration

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

14.2.3 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.

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]

Table 29: Device number to device name conversion.

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 29.

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

Table 30: Open errno values.

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

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.

Call Number	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	Input/Output	struct router_port *	Write and/or Read (in that order) the port control and port status registers of one port of the SpaceWire router. The `flag` field determines which operations should be performed. See ROUTER_PORTFLG_*. The `port` field selects which port is to be written/read.
CFGSTS_SET	Argument	*unsigned int **	Writes the Config/Status register.
CFGSTS_GET	Output	*unsigned int **	Copies the current value of the Config/Status register to the user provided buffer.
TC_GET	Output	*unsigned int **	Copies the current value of the Time-code register to the user provided buffer.

Table 31: ioctl calls supported by the SpaceWire Router driver.

Note that no detailed descriptions of the hardware register fields are given here. They are found in the GRSPWROUTER 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.

14.2.5.1 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;
};
```

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	Minor version number of the router
ver_patch	Patch number of the router
iid	Instance ID of the router

Table 32: router_hw_info member descriptions.

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.

14.2.5.2 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];
};
```

Member	Descriptiion
flags	Flags that determine which of the configuration parameters should be written
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 initialization divisor 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 33: router_config member descriptions

Flag	Descriptiion
ROUTER_FLG_IID	Write iid field
ROUTER_FLG_IDIV	Write Idiv field
ROUTER_FLG_CFG	Write config field
ROUTER_FLG_TPRES	Write timer_prescaler field
ROUTER_FLG_TRLD	Write all timer_reload fields (one per port)

Table 34: router_config flag s descriptions

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.

14.2.5.3 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.

14.2.5.4 ROUTES_SET

```
struct router_routes {

    unsigned int route[224];

};
```

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

Table 35: router_routes member descriptions

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.

14.2.5.5 ROUTES_GET

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

14.2.5.6 PS_SET

```
struct router_ps {

    unsigned int ps[31];

    unsigned int ps_logical[224];

};
```

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

Table 36: router_ps member descriptions

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.

14.2.5.7 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.

14.2.5.8 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).

14.2.5.9 PORT

```
struct router_port {

    unsigned int flag;

    int port;

    unsigned int ctrl;

    unsigned int sts;

};
```

Member	Descriptiion
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 37: router_port member descriptions

Flag	Descriptiion
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.

Table 38: router_port flag descriptions

14.2.5.10 CFGSTS_SET

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

14.2.5.11 CFGSTS_GET

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

14.2.5.12 TC_GET

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

15 GR1553B DRIVER

15.1 INTRODUCTION

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 up 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.

The GR1553B driver require the Driver Manager.

15.1.1 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.

15.1.2 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 SPARC BSP sources *c/src/lib/libbsp/sparc*.

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

Table 39: Source location

15.1.3 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.

16 GR1553B REMOTE TERMINAL DRIVER

16.1 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.

The GR1553B RT driver require the Driver Manager.

16.1.1 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.

16.1.2 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
```

16.2 USER INTERFACE

16.2.1 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 interface definitions are listed in the table below, the path is given relative to the SPARC BSP sources *c/src/lib/libbsp/sparc*.

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

Table 40: RT driver Source location

16.2.1.1 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()*.

16.2.1.2 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.

16.2.1.3 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 16.2.1.8.

16.2.1.4 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.

16.2.1.5 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.

16.2.1.6 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.

16.2.1.7 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.

16.2.1.8 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.

16.2.1.9 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 16.2.1.7.

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()*.

16.2.1.10 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.

16.2.2 Application Programming Interface

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

Prototype	**Description**
void *gr1553rt_open(int minor)	Open an RT device by instance number. Returns a handle identifying the specific RT device. The handle is given as input in most functions of the API
void gr1553rt_close(void *rt)	Close a previously opened RT device
int gr1553rt_config(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,	Assign a Data Transfer Interrupt handler callback routine and custom argument to a certain sub address

int subadr, int tx, gr1553rt_irq_t func, void *data)	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, unsigned short next)	Initialize a Descriptor in a List identified by number.
int gr1553rt_bd_update(struct gr1553rt_list *list, int entry_no, unsigned int *status, uint16_t **dptr)	Update the status and/or the data buffer pointer of a descriptor.

Table 41: function prototypes

16.2.2.1 Data structures

The *gr1553rt_cfg* data structure is used to configure an RT device. The configuration paramters 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;
};
```

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, must be a multiple of 2. If set to zero event log is disabled, note that enabling logging in SA-table or descriptors will cause failure when eventlog is disabled.
bd_count	Number of descriptors for RT device. All descriptor lists share the descriptors. Maximum is 65K descriptors.
bd_buffer	Descriptor memory area allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of descriptors is given, the address must be aligned to 32 bytes and of (32 * *bd_count)* bytes size.

Table 42: gr1553rt_cfg member descriptions.

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

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

Member	Description
bd_cnt	Number of descriptors in List

Table 43: gr1553rt_list_cfg member descriptions.

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;
};
```

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.

Table 44: gr1553rt_status member descriptions.

16.2.2.2 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.

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

16.2.2.3 gr1553rt_close

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

16.2.2.4 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 42.

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

16.2.2.5 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.

16.2.2.6 gr1553rt_stop

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

16.2.2.7 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 41.

16.2.2.8 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).

16.2.2.9 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.

16.2.2.10 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*.

16.2.2.11 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)
```

16.2.2.12 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.

16.2.2.13 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).

16.2.2.14 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.

16.2.2.15 gr1553rt_irq_err

This 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.

16.2.2.16 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.

16.2.2.17 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.

16.2.2.18 gr1553rt_list_init

Allocate and configure a list structure according to configuration given in *cfg*, see the *gr1553rt_list_cfg* data structure in table 40. 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.

16.2.2.19 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.

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

Table 45: gr1553rt_bd_init next argument description

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

16.2.2.20 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.

17 GR1553B BUS MONITOR DRIVER

17.1 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.

The GR1553B RT driver require the Driver Manager.

17.1.1 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.

17.1.2 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
```

17.2 USER INTERFACE

17.2.1 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 SPARC BSP sources *c/src/lib/libbsp/sparc*.

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

- **Table 46: BM driver Source location**

17.2.1.1 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()*.

17.2.1.2 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.

17.2.1.3 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.

17.2.1.4 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.

17.2.1.5 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()*.

17.2.1.6 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.

17.2.2 Application Programming Interface

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

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*.

Table 47: function prototypes

17.2.2.1 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;
};
```

Member	**Description**
time_resolution	8-bit time resolution, the BM will update the time according to this setting. 0 will make the time tag be of highest resolution (no division), 1 will make the BM increment the time tag once for two time ticks (div with 2), etc.
time_ovf_irq	Enable Time Overflow IRQ handling. Setting this to 1 makes the driver to update the 64-bit time by it self, it will use time overflow IRQ to detect when the 64-bit time counter must be incremented. If set to zero, the driver expect the user to call *gr1553bm_time()* regularly, it must be called more often than the time overflows to avoid an incorrect time.
filt_error_options	Bus error log options: bit0,4-31 = reserved, set to zero Bit1 = Enables logging of Invalid mode code errors Bit2 = Enables logging of Unexpected Data errors Bit3 = Enables logging of Manchester/parity errors
filt_rtadr	RT Address filtering bit mask. Each bit enables (if set) logging of a certain RT sub address. Bit 31 enables logging of broadcast messages.
filt_subadr	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	Optional Custom Data passed on to *dma_error_isr()*

Table 48: gr1553bm_config member descriptions.

The *gr1553bm_entry* data structure represent one entry in the BM log.

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

<table>
<tr><th>Member</th><th colspan="2">Description</th></tr>
<tr><td>time</td><td colspan="2">Time of word transfer entry. Bit31=1, bit 30..24=0, bit 23..0=time</td></tr>
<tr><td rowspan="8">data</td><td colspan="2">Transfer status and data word</td></tr>
<tr><td>Bits</td><td>Description</td></tr>
<tr><td>31</td><td>Zero</td></tr>
<tr><td>30..20</td><td>Zero</td></tr>
<tr><td>19</td><td>0=BusA, 1=BusB</td></tr>
<tr><td>18..17</td><td>Word Status: 00=Ok, 01=Manchester error, 10=Parity error</td></tr>
<tr><td>16</td><td>Word type: 0=Data, 1=Command/Status</td></tr>
<tr><td>15..0</td><td>16-bit Data on detected on bus</td></tr>
</table>

Table 49: gr1553bm_entry member descriptions.

17.2.2.2 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.

17.2.2.3 gr1553bm_close

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

17.2.2.4 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 46.

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.

17.2.2.5 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.

17.2.2.6 gr1553bm_stop

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

17.2.2.7 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*.

17.2.2.8 gr1553bm_available

This function stores the current number of entries stored in the BM Log into the address pointed to by *nentries*.

This function cannot be called in stopped mode, it will fail trying to do so.

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

17.2.2.9 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.

18 GR1553B Bus Controller DRIVER

18.1 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.

The GR1553B BC driver require the Driver Manager.

18.1.1 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 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 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 SPARC BSP sources *c/src/lib/libbsp/sparc*.

Filname	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

Table 50: BC driver Source location

18.1.3 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
```

18.2 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 18.3. In this section services related to direct access of BC hardware registers and Interrupt are described. The function API is declared in *gr1553bc.h*.

18.2.1 Device API

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

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.

Table 51: Device API function prototypes

18.2.1.1 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;
};
```

Member	Description
status	BC status register
time	BC Timer register

Table 52: gr1553bc_status member descriptions.

18.2.1.2 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.

18.2.1.3 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.

18.2.1.4 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.

18.2.1.5 gr1553bc_pause

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

18.2.1.6 gr1553bc_resume

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

18.2.1.7 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.

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

Table 53: gr1553bc_stop second argument

18.2.1.8 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.

18.2.1.9 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 52.

18.2.1.10 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.

18.2.1.11 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.

18.3 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.

18.3.1 Overview

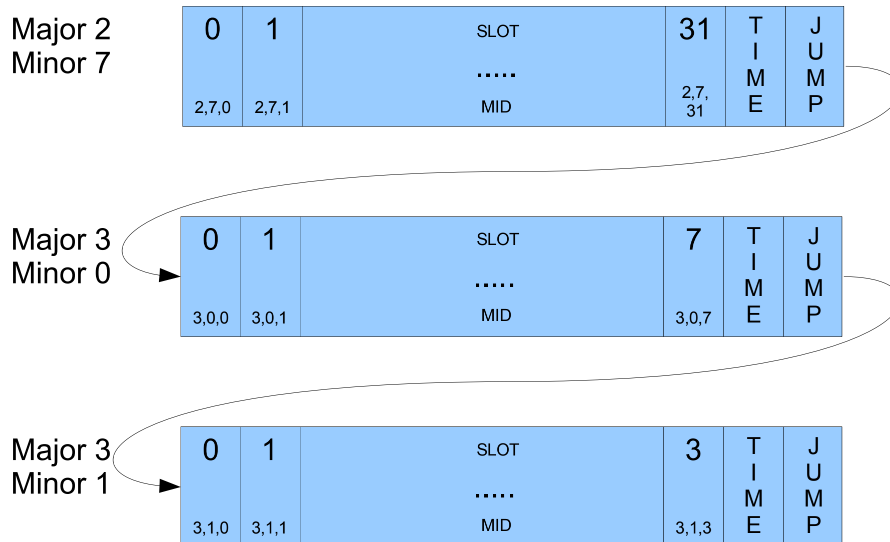
This section describes the Descriptor List Application Programming Interface (API). It provides functionality to create and manage BC descriptor lists.

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.



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.

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)

Table 54: Macros for creating MID

18.3.2 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:
  - gr1553bc_major_alloc_skel(&major, &major_minor_cfg)
  - gr1553bc_list_set_major(list, &major, MAJOR_NUM)
- Link last and first Major Frames together:
  - 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:
  - gr1553bc_slot_alloc(list, &MID, TIME_REQUIRED, ..)
  - gr1553bc_slot_transfer(MID, ...)
- START BC HARDWARE BY SHCDULING ABOVE LIST

- Application operate on executing List

18.3.3 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.

18.3.4 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.

Major 3 Minor 0	0	1	2	3	4	5	6	7	TIME	JUMP
	200 us	60 us	0 us	220 us	120 us	free 0us	120 us	free 0us	DUMMY 280us	

Illustration 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).

18.3.5 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.

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

Table 55: Slot configuration

Existing configured Slots can be manipulated with the following functions.

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

Table 56: Slot manipulation

18.3.6 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.

18.3.7 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.

18.3.8 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);
}
```

18.3.9 List API

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

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.
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, unsigned int flags, uint32_t word0, uint32_t word1, uint32_t word2, uint32_t word3)	Custom descriptor initialization. Note that a bad initialization may break the BC driver.
void gr1553bc_show_list(struct gr1553bc_list *list, int options)	Print information about a descriptor list to standard out. Used for debugging.

Table 57: List API function prototypes

18.3.9.1 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];
};
```

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

Table 58: gr1553bc_minor_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.

Table 59: gr1553bc_major_cfg member descriptions.

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;
};
```

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

Table 60: gr1553bc_list_cfg member descriptions.

18.3.9.2 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.

18.3.9.3 gr1553bc_list_free

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

18.3.9.4 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 51.

18.3.9.5 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.

18.3.9.6 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.

18.3.9.7 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.

18.3.9.8 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.

18.3.9.9 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.

18.3.9.10 gr1553bc_list_table_free

Free previously allocated descriptor table memory.

18.3.9.11 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.

18.3.9.12 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 50.

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

18.3.9.13 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.

18.3.9.14 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.

18.3.9.15 gr1553bc_slot_free

Return Slot and timeslot allocated from the Minor Frame.

18.3.9.16 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.

18.3.9.17 gr1553bc_slot_bd

Looks up descriptor address from MID.

18.3.9.18 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.

18.3.9.19 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.

18.3.9.20 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.

18.3.9.21 gr1553bc_slot_jump

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

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.

Table 61: gr1553bc_slot_jump argument descriptions.

Returns zero on success.

18.3.9.22 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.

18.3.9.23 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.

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)

Table 62: gr1553bc_slot_transfer argument descriptions.

Returns zero on success.

18.3.9.24 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.

18.3.9.25 gr1553bc_slot_empty

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

Returns zero on success.

18.3.9.26 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.

18.3.9.27 gr1553bc_slot_raw

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

The arguments are declared in the table below.

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

Table 63: gr1553bc_slot_transfer argument descriptions.

Returns zero on success.

18.3.9.28 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.

19 Gaisler B1553BRM DRIVER (BRM)

19.1 INTRODUCTION

This document is intended as an aid in getting started developing with Gaisler 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.

19.1.1 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.

19.1.2 Software Driver

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

The driver supports three different operating modes:

- Bus Controller
- Remote Terminal
- Bus monitor

19.1.3 Supported OS

Currently the driver is available for RTEMS.

19.1.4 Examples

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
```

19.2 USER INTERFACE

The RTEMS MIL-STD-1553 B 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.

19.2.1 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.

19.2.2 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.

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.

Table 64: B1553BRM driver parameter description

19.2.2.1 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.

19.2.3 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.

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

Table 65: Device number to device name conversion.

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 64.

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

Table 66: Open *errno* values.

19.2.4 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.

19.2.5 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.

19.2.5.1 Data structures

19.2.5.1.1 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;
};
```

Member	Description	
miw	Message Information Word.	
	Bit(s)	**Description**
	15-11	Word count / mode code - For subaddresses 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	RTRT - 1 if message is part of an RT to RT transfer
	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	TO - If set, the number of received words was less than the amount specified by the word count.
	2	OVR - If set, the number of received words was more than amount specified by the word count.
	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.	

Table 67: *rt_msg* member descriptions.

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:

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

Table 68: Descriptor table

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.

19.2.5.1.2 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>
<tr><th>Member</th><th colspan="2">Description</th></tr>
<tr><td>rtaddr</td><td colspan="2">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.</td></tr>
<tr><td>subaddr</td><td colspan="2">The subaddr array works in the same manner as rtaddr but for the subaddresses.</td></tr>
<tr><td>wc</td><td colspan="2">Word Count - Specifies the word count, or mode code if subaddress is 0 or 31.</td></tr>
<tr><td rowspan="10">ctrl</td><td>Bit(s)</td><td>Description</td></tr>
<tr><td>15</td><td>Message Error. Set by BRM while traversing list if protocol error is detected.</td></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 rowspan="2">1</td><td>1 RT to RT</td></tr>
<tr><td>0 normal</td></tr>
<tr><td rowspan="2">0</td><td>0 RT Transmit</td></tr>
<tr><td>1 RT receive (ignored for RT to RT)</td></tr>
<tr><td>tsw</td><td colspan="2">Status words</td></tr>
<tr><td>data</td><td colspan="2">Data in message, not used for RT receive (ctrl.0 = 1).</td></tr>
</table>

Table 69: struct bc_msg member description

19.2.5.1.3 Bus Monitor operating 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>
<tr><th>Member</th><th colspan="2">Description</th></tr>
<tr><td rowspan="14">miw</td><td>Bit(s)</td><td>Description</td></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>
<tr><td colspan="2"></td></tr>
<tr><td>cw1</td><td colspan="2">1553 Command word 1</td></tr>
<tr><td>cw2</td><td colspan="2">1553 Command word 2, only used for RT to RT transfers and then holds the transmit command.</td></tr>
<tr><td>sw1</td><td colspan="2">1553 Status word 1</td></tr>
<tr><td>sw2</td><td colspan="2">1553 Status word 2, is only used for RT to RT transfers and then holds the status word from the transmitting RT.</td></tr>
<tr><td>time</td><td colspan="2">Time tag (time) Contains the value of the internal timer register when the message was captured.</td></tr>
<tr><td>data</td><td colspan="2">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.</td></tr>
</table>

Table 70: struct bm_msg member description

19.2.6 Configuration

The BRM core and driver are configured using *ioctl* calls. The table 67 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 66.

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);
```

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 71: ERRNO values for *ioctl* calls.

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

Table 72: *ioctl* calls supported by the BRM driver.

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.

19.2.6.1 SET_MODE

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

19.2.6.2 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.

19.2.6.3 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.

19.2.6.4 SET_RT_ADDR

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

19.2.6.5 BRM_SET_STD

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

19.2.6.6 BRM_SET_BCE

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

19.2.6.7 BRM_TX_BLOCK

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

19.2.6.8 BRM_RX_BLOCK

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

19.2.6.9 BRM_CLR_STATUS

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

19.2.6.10 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.

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

Table 73: Status bit mask

19.2.6.11 BRM_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 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 65 for the description of the notification argument.

19.2.7 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 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.

19.2.8 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.

19.2.9 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.

20 Gaisler B1553RT DRIVER (RT)

20.1 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.

20.1.1 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.

20.1.2 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.

20.2 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.

20.2.1 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.

20.2.2 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.

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.

Table 74: B1553RT driver parameter description

20.2.2.1 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.

20.2.3 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.

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

Table 75: Device number to device name conversion.

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 74.

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

Table 76: Open *errno* values.

20.2.4 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.

20.2.5 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*.

20.2.5.1 Data structures

20.2.5.1.1 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;
};
```

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.	

Table 77: *rt_msg* member descriptions.

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:

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

Table 78: Descriptor table

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.

20.2.6 Configuration

The RT core and driver are configured using *ioctl* calls. The table 77 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 76.

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

```
unsigned int mode = 1;
result = ioctl(fd, RT_SET_ADDR, &mode);
```

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

Table 79: 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	

Table 80: *ioctl* calls supported by the RT driver.

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.

20.2.6.1 RT_SET_ADDR

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

20.2.6.2 RT_SET_BCE

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

20.2.6.3 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.

20.2.6.4 RT_RX_BLOCK

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

20.2.6.5 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.

20.2.6.6 RT_CLR_STATUS

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

20.2.6.7 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.

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.
RT_MERR_IRQ	Message Error
RT_ILLCMD_IRQ	Illegal Command

Table 81: Status bit mask

20.2.6.8 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 75 for the description of the notification argument.

20.2.7 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.

21 CAN DRIVER INTERFACE (GRCAN)

21.1 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.

21.1.1 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.

21.1.2 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.

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.

Table 82: GRCAN driver parameter description

21.1.2.1 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.

21.1.3 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.

Device number	Filesystem name	Location
0	/dev/grcan0	On-Chip Bus
1	/dev/grcan1	On-Chip Bus
2	/dev/grcan2	On-Chip Bus
Depends on system configuration	/dev/rastaio0/grcan0	GR-RASTA-IO

Table 83: Core number to device name conversion.

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 82.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 84: Open *errno* values.

21.1.4 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.

21.1.5 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.

21.1.5.1 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;
};
```

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.

Table 85: grcan_filter member description

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;
```

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.

Table 86: CANMsg member description

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;
```

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.

Table 87: grcan_stats member description

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>
<tr><th>Member</th><th colspan="2">Description</th></tr>
<tr><td>scaler</td><td colspan="2">Prescaler</td></tr>
<tr><td>ps1</td><td colspan="2">Phase segment 1</td></tr>
<tr><td>ps2</td><td colspan="2">Phase segment 2</td></tr>
<tr><td>rsj</td><td colspan="2">Resynchronization jumps, 1..4</td></tr>
<tr><td rowspan="5">bpr</td><td>Value</td><td>Baud rate</td></tr>
<tr><td>0</td><td>system clock / (scaler+1) / 1</td></tr>
<tr><td>1</td><td>system clock / (scaler+1) / 2</td></tr>
<tr><td>2</td><td>system clock / (scaler+1) / 4</td></tr>
<tr><td>3</td><td>system clock / (scaler+1) / 8</td></tr>
</table>

Table 88: grcan_timing member description

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;
};
```

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

Table 89: grcan_selection member description

21.1.5.2 Configuration

The CAN core and driver are configured using *ioctl* calls. The table 85 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 84.

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);
```

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.

Table 90: ERRNO values for *ioctl* calls

Call Number	**Call Mode**	**Description**
START	Stopped	Exit paused mode, brings up the link. Enables read and write. Called after *bus-off* or *open*.
STOP	Running	Exit operating mode, enter reset mode. Most of the settings can only be set when in reset mode.
ISSTARTED	Don't Care	Return error status when not running and success when driver is in running mode.
FLUSH	Running	Wait until all messages are transmitted.
SET_SILENT	Stopped	Enable/disable silent mode, it is possible to read messages but not write messages to the CAN bus.
SET_ABORT	Don't Care	Stop or continue on AMBA AHB transaction error.
SET_SELECTION	Stopped	Redundant channel selection. Pass a pointer to a *grcan_selection* data structure when calling this command.
SET_SPEED	Stopped	Not implemented. (Set baud rate from frequency)
SET_BTRS	Stopped	Sets timing parameters which control the baud rate using the *grcan_timing* data structure.
SET_RXBLOCK	Don't Care	Set read blocking/non-blocking mode
SET_TXBLOCK	Don't Care	Set write blocking/non-blocking mode
SET_TXCOMPLETE	Don't Care	Set option to complete the write request, making write returning after all data has been written to buffer. Note: Has an effect only in blocking mode.
SET_RXCOMPLETE	Don't Care	Set option to complete the read request, making read returning after requested data has been read to buffer. Note: Has an effect only in blocking mode.
GET_STATS	Don't care	Get current statistics collected by driver.
CLR_STATS	Don't Care	Clear statistics collected by driver.
SET_AFILTER	Don't Care	Set acceptance filter. Let the second argument to the ioctl command point to a grcan_filter data structure.
SET_SFILTER	Don't Care	Set Rx/Tx SYNC filter. Let the second argument to the ioctl command point to a grcan_filter data structure.
GET_STATUS	Don't care	Get the status register. Bus off among others can be read out.

Table 91: *ioctl* calls supported by the CAN driver.

21.1.5.2.1 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.

21.1.5.2.2 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.

21.1.5.2.3 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.

21.1.5.2.4 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.

21.1.5.2.5 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.

21.1.5.2.6 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.

21.1.5.2.7 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*.

21.1.5.2.8 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.

21.1.5.2.9 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.

21.1.5.2.10 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.

21.1.5.2.11 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.

Note: This option is only relevant in TX blocking mode.

This call never fail.

21.1.5.2.12 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.

Note: This option is only relevant in RX blocking mode.

This call never fail.

21.1.5.2.13 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 gr*can_stats* data structure.

The call will fail if the pointer to the data is invalid.

21.1.5.2.14 CLR_STATS

Clears the driver's collected statistics.

This call never fail.

21.1.5.2.15 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.

21.1.5.2.16 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.

21.1.5.2.17 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.

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

Table 92: Status bit mask

This call never fail.

21.1.6 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.

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.

Table 93: ERRNO values for *write*

Each Message has an individual set of options controlled in the CANMsg structure. See the data structure subsection for structure member descriptions.

21.1.7 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.

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 operating mode by issuing a START *ioctl* command.
ETIMEDOUT	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.

Table 94: ERRNO values for *read* calls.

22 Gaisler Opencores CAN driver (OC_CAN)

22.1 INTRODUCTION

This document is intended as an aid in getting started developing with Gaisler 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.

22.1.1 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.

22.1.2 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.

22.1.3 Supported OS

Currently the driver is available for RTEMS.

22.1.4 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.

22.1.5 Support

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

22.2 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.

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 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.

22.2.2 Driver resource configuration

This driver does not have any configurable resources. All configuration can be made though the *ioctl* interface.

22.2.3 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:

Device number	Filesystem name
0	/dev/occan0
1	/dev/occan1
2	/dev/occan2

Table 95: Device number to device name conversion.

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 95.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 96: Open *errno* values.

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 *occan* 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 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.

22.2.5.1 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;
};
```

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.
single_mode	Set to none-zero for a single filter - single filter mode, zero selects dual filter mode.

Table 97: occan_afilter member descriptions.

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;
```

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.

Table 98: CANMsg member descriptions.

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;
```

Member	Description
rx_msgs	Number of CAN messages received.
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.

Table 99: occan_stats member descriptions.

22.2.5.2 Configuration

The OC_CAN core and driver are configured using *ioctl* calls. The table 98 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 97.

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);
```

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 100: ERRNO values for *ioctl* calls.

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.
SET_SPEED	Reset	Set baud rate
SET_BLK_MODE	Don't Care	Set blocking or non-blocking mode for read and write.
SET_BUFLEN	Reset	Set receive and transmit buffer length.
SET_BTRS	Reset	Set timing registers manually.

Table 101: *ioctl* calls supported by the OC_CAN driver.

22.2.5.2.1 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.

22.2.5.2.2 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.

22.2.5.2.3 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.

22.2.5.2.4 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 96above.

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)

Table 102: Status bit mask

This call never fail.

22.2.5.2.5 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.

22.2.5.2.6 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.

22.2.5.2.7 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.

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.

Table 103: SET_BLK_MODE *ioctl* arguments

This call never fail.

22.2.5.2.8 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.

Mask	Description
0x0000ffff	Receive buffer length in number of *CANMsg* structures.
0xffff0000	Transmit buffer length in number of *CANMsg* structures.

Table 104: SET_BUF_LEN *ioctl* argument

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.

22.2.6 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 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.

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
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.

Table 105: ERRNO values for *write*

Each Message has an individual set of options controlled in the CANMsg structure. See the data structure subsection for structure member descriptions.

22.2.7 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.

ERRNO	**Description**
EINVAL	A 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.
ETIMEDOUT	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.

Table 106: ERRNO values for *read* calls.

23 Gaisler SatCAN FPGA driver (SatCAN)

23.1 INTRODUCTION

This document is intended as an aid in getting started developing with the Gaisler GRLIB wrapper for the SatCAN FPGA 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.

23.1.1 SatCAN Hardware Wrapper

See the SatCAN wrapper manual.

23.1.2 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.

23.1.3 Supported OS

Currently the driver is available for RTEMS.

23.1.4 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.

23.1.5 Support

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

23.2 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.

23.2.1 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 ( occan_register(&amba_conf, &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;
```

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.

Table 107: Members in satcan_config structure

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.

23.2.2 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 107.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 108: Open *errno* values.

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.

23.2.3 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.

23.2.4 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;
```

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_PAYLOAD_SIZE is 8.

Table 109: Members in satcan_msg structure

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 structre 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.

ERRNO	Description
EINVAL	A NULL pointer was passed as the data pointer or the length was illegal.

Table 110: ERRNO values for *read* calls.

23.2.5 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");
```

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.

Table 111: ERRNO values for *write*

23.2.6 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*.

23.2.6.1 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;
```

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.

Table 112: Members in satcan_regmod structure

23.2.6.2 Configuration

The SatCAN core and driver are configured using *ioctl* calls. The table 109 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 108.

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.
ENOMEM	Not enough memory to complete operation. This may cause other *ioctl* commands to fail.

Table 113: ERRNO values for *ioctl* calls.

Call Number	Description
DMA_2K	Instructs driver and core to use 2K DMA mode. Default setting.
DMA_8K	Instructs driver and core to use 8K DMA mode.
GET_REG	Provides direct read access to all core registers.
SET_REG	Provides direct write 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_TX2_DIS_TX1	Changes internal driver state to disable DMA TX channel 1, Enable 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

Table 114: *ioctl* calls supported by the SatCAN FPGA driver.

23.2.6.2.1 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.

23.2.6.2.2 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");
}
```

23.2.6.2.3 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.

Register constant	Register name
SATCAN_SWRES	Software reset
SATCAN_INT_EN	Interrupt enable
SATCAN_FIFO	FIFO read
SATCAN_FIFO_RES	FIFO reset
SATCAN_TSTAMP	Current time stamp
SATCAN_CMD0	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 current 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_WAHBADDR	Wrapper AHB address register

Table 115: Values used together with GET_REG and SET_REG

23.2.6.2.4 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"):
```

23.2.6.2.5 OR_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");
```

23.2.6.2.6 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.

23.2.6.2.7 EN_TX1_DIS_TX2

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");
}
```

23.2.6.2.8 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.

23.2.6.2.9 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.

23.2.6.2.10 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");
```

23.2.6.2.11 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");
```

23.2.6.2.12 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");
```

23.2.6.2.13 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.

23.2.6.2.14 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.

23.2.6.2.15 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.

23.2.6.2.16 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.

24 Gaisler CAN_MUX driver (CAN_MUX)

24.1 INTRODUCTION

This document is intended as an aid in getting started developing with Gaisler 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.

24.1.1 CAN_MUX Hardware

See the CAN_MUX core manual.

24.1.2 Software Driver

The driver provides means for setting the CAN_MUX MUX control register.

24.1.3 Supported OS

Currently the driver is available for RTEMS.

24.1.4 Examples

The rtems-satcan example uses the CAN_MUX driver.

24.1.5 Support

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

24.2 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.

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 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");
```

24.2.2 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 116.

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

Table 116: Open *errno* values.

24.2.3 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.

24.2.4 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.

24.2.4.1 Configuration

The CAN_MUX core and driver is controlled using *ioctl* calls. The table 118 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 117.

An example is shown below where CAN bus A is routed to the SatCAN core.

```
result = ioctl(fd, CANMUX_IOC_BUSA_SATCAN);
```

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.

Table 117: ERRNO values for *ioctl* calls.

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

Table 118: *ioctl* calls supported by the CAN_MUX driver.

25 Gaisler ASCS (GRASCS)

25.1 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.

25.1.1 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 1.2 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.

25.1.2 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.

25.1.3 Support

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

25.2 USER INTERFACE

In table 1.1 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.

Function name	Described in section	Description
ASCS_init	1.2.1	Initializes driver and GRASCS core
ASCS_input_select	1.2.2	Selects slave
ASCS_etr_select	1.2.3	Select source for synchronization pulse
ASCS_start	1.2.4	Starts serial interface
ASCS_stop	1.2.5	Stops serial interface
ASCS_iface_status	1.2.6	Report status of serial and synchronization interfaces
ASCS_TC_send	1.2.7	Performs a data write (TC)
ASCS_TC_send_block	1.2.8	Performs a number of TCs
ASCS_TC_sync_start	1.2.9	Starts synchronization interface
ASCS_TC_sync_stop	1.2.10	Stops synchronization interface
ASCS_TM_recv	1.2.11	Performs a data read (TM)
ASCS_TM_recv_block	1.2.12	Performs a number of TMs

Table 119: GRASCS driver interface

25.2.1 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.

25.2.2 ASCS_input_select

Prototype: int ASCS_input_select(int slave)

Argument	Description
slave	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).

25.2.3 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.

25.2.4 ASCS_start

Prototype: void 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.

25.2.5 ASCS_stop

Prototype: void 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.

25.2.6 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.

25.2.7 ASCS_TC_send

Prototype: int ASCS_TC_send(int *word)

Argument	Description
word	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.
ntrans	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.

25.2.8 ASCS_TC_send_block

Prototype: int ASCS_TC_send_block(int *block, int ntrans)

Argument	Description
block	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.
ntrans	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.

25.2.9 ASCS_TC_sync_start

Prototype: void 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.

25.2.10 ASCS_TC_sync_stop

Prototype: void 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.

25.2.11 ASCS_TM_recv

Prototype: int ASCS_TM_recv(int *word)

Argument	Description
word	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.

25.2.12 ASCS_TM_recv_block

Prototype: int ASCS_TM_recv(int *block, in ntrans)

Argument	Description
block	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.
ntrans	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.

25.3 EXAMPLE 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");

  }
```

26 RAW UART DRIVER INTERFACE (APBUART)

26.1 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.

26.1.1 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 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.

26.1.2 Driver resource configuration

This driver does not have any configurable resources. All configuration can be made though the *ioctl* interface.

26.1.3 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.

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

Table 120: Device number to device name conversion.

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 120.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 121: Open *errno* values.

26.1.4 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 *apbuart* driver.

26.1.5 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 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.

26.1.5.1 Configuration

The UART core and driver are configured using *ioctl* calls. The table 122 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 121.

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);
```

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 122: ERRNO values for *ioctl* calls

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.
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 into 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	Set/unset ASCII mode. When ASCII mode is enabled a new line is replaced with a new line and a carriage return. '\n' => '\n\r'

Table 123: *ioctl* calls supported by the APBUART driver.

26.1.5.1.1 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.

26.1.5.1.2 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.

The command will fail if the driver already is in stopped mode.

26.1.5.1.3 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.

26.1.5.1.4 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.

26.1.5.1.5 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.

26.1.5.1.6 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.

26.1.5.1.7 SET_BLOCKING

Sets receive, transmit or transmit flush blocking mode. The argument to SET_BLOCKING is a bitmask as described in the table below.

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.

Table 124: SET_BLOCKING Argument Bit Mask

26.1.5.1.8 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.

26.1.5.1.9 CLR_STATS

Resets drivers statistics counters.

26.1.5.1.10 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.

26.1.6 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.

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.

Table 125: ERRNO values for *write*

26.1.7 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.

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.

Table 126: ERRNO values for *read* calls.

27 Gaisler SPICTRL SPI DRIVER (SPICTRL)

27.1 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.

27.1.1 SPI Hardware

The SPICTRL core is documented in the GR-IP core's manual. The driver supports multiple SPI cores.

27.1.2 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.

27.2 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.

27.2.1 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.

27.2.2 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 127. All paths are given relative the RTEMS kernel source root.

Source description	Location
I2C Library	cpukit/libi2c
High level drivers	c/src/libchip/i2c
SPICTRL driver	c/src/lib/libbsp/sparc/shared/spi

Table 127: SPI source location

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*.

27.2.3 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.

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 transmit 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

Table 128: Additional *ioctl* commands

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()

27.2.3.1 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.

27.2.3.2 PERIOD_STOP

Stops any ongoing period transfer by writing zero to the AM configuration register.

27.2.3.3 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;
};
```

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.

Table 129: spictrl_ioctl_config field description

27.2.3.4 STATUS

Copies the Event register of the SPICTRL core to a user provided buffer.

27.2.3.5 PERIOD_WRITE

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;
};
```

Field	Description		
options	Selects operation performed by command		
	READ	BIT0	1=Read Mask registers into masks[].
		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.		

Table 130: spictrl_period_io field description

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.

27.2.3.6 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 128.

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).

28 Gaisler i2C Master DRIVER (I2CMST)

28.1 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.

28.1.1 I2C Hardware

The I2CMST core is documented in the GR-IP core's manual. The driver supports multiple I2C cores.

28.1.2 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.

28.2 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.

28.2.1 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.

28.2.2 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 131. All paths are given relative the RTEMS kernel source root.

Source description	Location
I2C Library	cpukit/libi2c
High level drivers	c/src/libchip/i2c
I2CMST driver	c/src/lib/libbsp/sparc/shared/i2c

Table 131: I2C source location

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*.

29 GPIO Library

29.1 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.

29.1.1 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.

29.2 DRIVER INTERFACE

The driver interface is not described in this document.

29.3 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 132. All paths are given relative the RTEMS kernel source root.

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

Table 132: GPIOLIB source location

29.3.1 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.

29.3.2 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 133. 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.

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.

Table 133: gpiolib_irq_register argument description

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.

29.3.3 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;
}
```

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

Table 134: gpiolib_config members

29.3.4 Function prototype description

29.3.4.1 GPIO Library functions

A short summary to the functions are presented in the prototype lists below.

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)

Table 135: GPIO per port functions

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.

29.3.4.1.1 grpiolib_set_config

Configures one GPIO port according to the the *gpiolib_config* data structure.

The *gpiolib_config* structure is described in table 134.

29.3.4.1.2 gpiolib_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.

29.3.4.1.3 gpiolib_get

Get the input value of a GPIO port. The value is stored into the address indicated by the argument *inval*.

29.3.4.1.4 gpiolib_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.

29.3.4.1.5 gpiolib_irq_force

Force an interrupt by writing to the interrupt controller that the GPIO port is attached to.

29.3.4.1.6 gpiolib_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.

29.3.4.1.7 gpiolib_irq_disable

Mask GPIO port interrupt on the interrupt controller the GPIO port is attached to. This disable interrupt generation at the interrupt controller.

29.3.4.1.8 gpiolib_irq_register

Attaches a interrupt service routine to a GPIO port. Described separately above.

30 Gaisler GPIO DRIVER (GRGPIO)

30.1 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 not documented here.

The GRGPIO driver require the RTEMS Driver Manager.

30.1.1 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.

30.1.2 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.

30.2 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.

30.2.1 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.

30.2.2 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.

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.

Table 136: GRGPIO driver parameter description

30.2.3 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.

GRGPIO Core	GRGPIO I/O port	Registration order
0	PIO[0]	0
	PIO[1]	1
	PIO[2]	2
	PIO[3]	3
	PIO[4]	4
	PIO[5]	5
	PIO[6]	6
	PIO[7]	7
1	PIO[0]	8
	PIO[1]	9
	PIO[2]	10
	PIO[3]	11
	PIO[4]	12
	PIO[5]	13
	PIO[6]	14
	PIO[7]	15
2	PIO[0]	16
	PIO[1]	17
	PIO[2]	18
	PIO[3]	19
	PIO[4]	20
	PIO[5]	21
	PIO[6]	22
	PIO[7]	23

Table 137: GRGPIO registration order

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]"
```

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

Table 138: GRGPIO port naming

The location of the GRGPIO drivers and the GPIO Library is indicated in table 136. All paths are given relative the RTEMS kernel source root.

Source description	**Location**
GPIO Library	c/src/lib/libbsp/sparc/shared/gpio/gpiolib.c
GRGPIO driver	c/src/lib/libbsp/sparc/shared/gpio/grgpio.c

Table 139: GPIO source location

31 Gaisler ADC/DAC DRIVER (GRADCDAC)

31.1 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.

31.1.1 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.

31.1.2 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.

31.2 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 140. All paths are given relative the RTEMS kernel source root.

Source description	**Location**
GRADCDAC driver	c/src/lib/libbsp/sparc/shared/analog/gradcdac.c
Driver Interface	c/src/lib/libbsp/sparc/shared/include/gradcdac.h

Table 140: GRADCDAC source location

31.2.1 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.

31.2.2 Driver resource configuration

The driver does not support configurable resource parameters.

31.2.3 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]"
```

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

Table 141: GRADCDAC core naming

31.2.4 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 142. 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>
<tr><th>Name</th><th colspan="2">Description</th></tr>
<tr><td>cookie</td><td colspan="2">Handle used internally by the function interface, it is returned by the open function.</td></tr>
<tr><td rowspan="4">adc</td><td>Value</td><td>Function</td></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>
<tr><td>isr</td><td colspan="2">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 colspan="2">Argument passed to the isr function when called as the second argument.</td></tr>
</table>

Table 142: gradcdac_install_irq_handler argument description

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.

31.2.5 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;
};
```

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 ADC 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
			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

Table 143: gradcdac_config member and ADCONF reg definition

31.2.6 Function prototype description

31.2.6.1 General ADC/DAC functions

A short summary to the functions are presented in the prototype lists below.

Prototype Name
void gradcdac_set_config(void *cookie, struct gradcdac_config *cfg)
void gradcdac_get_config(void *cookie, struct gradcdac_config *cfg)
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)

Table 144: General ADC/DAC functions

All functions takes a handle to the ADC/DAC core by the argument cookie. The handle is returned by the *gradcdac_open* function.

31.2.6.1.1 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 143.

31.2.6.1.2 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 143.

31.2.6.1.3 gradcdac_set_cfg

Sets the configuration register directly.

The bits of the ADCONF configuration register are described in table 143.

31.2.6.1.4 gradcdac_get_cfg

Returns the current configuration register value as it is.

The bits of the ADCONF configuration register are described in table 143.

31.2.6.1.5 gradcdac_get_status

Returns the current ADC / DAC controller's status register value.

31.2.6.1.6 gradcdac_get_adrinput

Returns the current address input register value.

31.2.6.1.7 gradcdac_get_adroutput

Returns the current address output register value.

31.2.6.1.8 gradcdac_set_adroutput

Sets the controller's address output register to the argument *output*.

31.2.6.1.9 gradcdac_get_adrdir

Returns the current address direction register value.

31.2.6.1.10 gradcdac_set_adrdir

Sets the controller's address direction register to the argument *dir*.

31.2.6.1.11 gradcdac_get_datainput

Returns the current data input register value.

31.2.6.1.12 gradcdac_get_dataoutput

Returns the current data output register value.

31.2.6.1.13 gradcdac_set_dataoutput

Sets the controller's data output register to the argument *output*.

31.2.6.1.14 gradcdac_get_datadir

Returns the current data direction register value.

31.2.6.1.15 gradcdac_set_datadir

Sets the controller's data direction register to the argument *dir*.

31.2.6.2 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 144. The functions does not actually read or write any ADC/DAC register therefore the handle (cookie) is omitted.

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

Table 145: Status interpretation help functions

31.2.6.3 ADC functions

A short summary to the functions are presented in the prototype lists below.

Operating on all ports
void gradcdac_adc_convert_start(void)
int gradcdac_adc_convert_try(unsigned short *digital_value)
int gradcdac_adc_convert(unsigned short *digital_value)

Table 146: ADC functions

31.2.6.3.1 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*.

31.2.6.3.2 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.

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

Table 147: gradcdac_adc_convert_try return code

31.2.6.3.3 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 141 for a description of the return values.

31.2.6.4 DAC functions

A short summary to the functions are presented in the prototype lists below.

Operates on a single port
int gradcdac_dac_convert_try(unsigned short digital_value)
void gradcdac_dac_convert(unsigned short digital_value)

Table 148: DAC functions

For a more detailed description see each function's respective sub section.

31.2.6.4.1 gradcdac_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.

31.2.6.4.2 gradcdac_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.

32 Gaisler TC driver (GRTC)

32.1 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.

32.1.1 TC Hardware

See the GRTC core manual. When the GRTC core is accessed over SpaceWire RMAP is used.

32.1.2 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.

32.1.2.1 GRTC over SpaceWire

The SpaceWire capable GRTC driver introduces some limitations listed below:

- RAW mode is not supported (the read call)
- 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.
- 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.

32.1.3 Support

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

32.2 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.

32.2.1 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");
```

32.2.2 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:

Core number	Filesystem name
0	/dev/grtc0
1	/dev/grtc1
2	/dev/grtc2
0	/dev/rastatmtc0/grtc0
0	/dev/rmap_fe/grtc0

Table 149: Core number to device name conversion.

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 149.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 150: Open *errno* values.

32.2.3 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.

32.2.4 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.

32.2.4.1 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;
};
```

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 alignment 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.

Table 151: grtc_ioc_buf_params member descriptions.

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;
};
```

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

Table 152: grtc_ioc_config member descriptions.

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;
};
```

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

Table 153: grtc_ioc_hw_status member descriptions.

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];
};
```

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

Table 154: grtc_frame member descriptions.

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;
};
```

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

Table 155: grtc_list member descriptions.

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];
};
```

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.

Table 156: grtc_ioc_pools_setup member descriptions.

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;
};
```

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

Table 157: grtc_ioc_assign_frm_pool member descriptions.

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;
};
```

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.

Table 158: grtc_ioc_stats member descriptions.

32.2.4.2 Configuration

The TC core and driver are configured using *ioctl* calls. The table 152 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 151.

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);
```

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.
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 159: ERRNO values for *ioctl* calls.

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_HW_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.

Table 160: *ioctl* calls supported by the GRTC driver.

32.2.4.2.1 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 151.

32.2.4.2.2 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.

32.2.4.2.3 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.

32.2.4.2.4 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.

Bit number	Description
GRTC_BLKMODE_POLL	Enables polling mode
GRTC_BLKMODE_BLK	Enables blocking mode

Table 161: SET_BLOCKING_MODE *ioctl* arguments

The driver's default is polling mode.

Note that the blocking mode is implemented using the CLTU stored interrupt.

This command never fail.

32.2.4.2.5 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.

32.2.4.2.6 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.

32.2.4.2.7 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()*.

32.2.4.2.8 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 32.2.2.

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.

32.2.4.2.9 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 32.2.2.

This command only fail if the pointer argument is invalid.

32.2.4.2.10 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.

32.2.4.2.11 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.

32.2.4.2.12 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.

32.2.4.2.13 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.

32.2.4.2.14 CLR_STATS

This command reset the driver's internal statistics counters.

This command never fail.

32.2.4.2.15 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.

32.2.4.2.16 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.

32.2.4.2.17 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.

32.2.4.2.18 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.

Note that the DMA area will not be processed in stopped mode, the RECV command will only return already processed frames.

The call will fail if the pointer to the frame list is invalid or if in RAW mode.

32.2.5 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.

32.2.5.1 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.

32.2.6 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.

32.2.7 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.

ERRNO	**Description**
EINVAL	A NULL pointer was passed as the data pointer or the length was illegal.
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.

Table 162: ERRNO values for *read* calls.

33 Gaisler TM driver (GRTM)

33.1 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.

33.1.1 TM Hardware

See the GRTM core manual. When the GRTM core is accessed over SpaceWire RMAP is used.

33.1.2 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.

33.1.2.1 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.

33.1.3 Support

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

33.2 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.

33.2.1 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");
```

33.2.2 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:

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.

Table 163: Core number to device name conversion.

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 163.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 164: Open *errno* values.

33.2.3 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.

33.2.4 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.

33.2.4.1 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;
};
```

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

Table 165: grtm_ioc_hw member descriptions.

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;
};
```

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_desc_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

Table 166: grtm_ioc_config member descriptions.

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;
};
```

Member	Description
flags	Mask indicating options, transmission state and errors for the frame. GRTM_FLAGS_XXX. See Table 167
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 167: grtm_frame member 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. Not used in SpaceWire version of driver.
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.

Table 168: Frame flags description.

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;
};
```

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

Table 169: grtm_list member descriptions.

The grtm_ioc_stats structure contain statistics collected by the driver.

```
struct grtm_ioc_stats {
     unsigned long long      frames_sent;
     unsigned int            err_underrun;
};
```

Member	Description
frames_sent	Number of frames successfully sent by the TM core
err_underrun	Number of AMBA underrun errors

Table 170: grtm_ioc_stats member descriptions.

33.2.4.2 Configuration

The GRTM core and driver are configured using *ioctl* calls. The table 169 below 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 168.

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);
```

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 171: ERRNO values for *ioctl* calls.

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 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.
SEND	Started	Add a chain of TM frames to the transmission queue of the GRTM driver.

Table 172: *ioctl* calls supported by the GRTM driver.

33.2.4.2.1 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 168.

33.2.4.2.2 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.

33.2.4.2.3 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.

33.2.4.2.4 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.

Bit number	Description
GRTM_BLKMODE_POLL	Enables polling mode
GRTM_BLKMODE_BLK	Enables blocking mode

Table 173: SET_BLOCKING_MODE *ioctl* arguments

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.

33.2.4.2.5 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.

33.2.4.2.6 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 33.2.4.1.

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.

33.2.4.2.7 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 33.2.4.1.

This command only fail if the pointer argument is invalid.

33.2.4.2.8 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.

33.2.4.2.9 CLR_STATS

This command reset the driver's internal statistics counters.

This command never fail.

33.2.4.2.10 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.

The call will fail if the pointer to the data is invalid.

33.2.4.2.11 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.

33.2.4.2.12 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 33.2.2. 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 167 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.

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.

Table 174: ERRNO values for *RECLAIM*

33.2.4.2.13 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 SET_CONFIG.

The input to SEND is a pointer to a *grtm_list* data structure described in section 33.2.4.1. 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 33.2.2. 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.

33.2.5 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.

34 GRCTM DRIVER

34.1 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.

34.1.1 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
```

34.2 USER INTERFACE

34.2.1 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*.

Filname	Description
shared/time/grctm.c	GRCTM Driver source
shared/include/grctm.h	GRCTM Driver interface declaration

Table 175: GRCTM driver Source location

34.2.1.1 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.

34.2.1.2 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().*

34.2.2 Application Programming Interface

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

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/mask 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 drive 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, unsigned int val)	Set pulse register
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

Table 176: function prototypes

34.2.2.1 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];
};
```

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.

Table 177: grctm_status member descriptions.

35 SPWCUC DRIVER

35.1 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.

35.1.1 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
```

35.2 USER INTERFACE

35.2.1 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*.

Filname	Description
shared/time/spwcuc.c	SPWCUC Driver source
shared/include/spwcuc.h	SPWCUC Driver interface declaration

Table 178: SPWCUC driver Source location

35.2.1.1 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.

35.2.1.2 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().*

35.2.2 Application Programming Interface

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

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(void *spwcuc)	Return received 24-bit Elapsed Fine Time.
unsigned long long spwcuc_get_tp_et(void *spwcuc)	Return received 56-bit Elapsed Time (a combination of coarse and fine).

Table 179: function prototypes

35.2.2.1 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;
};
```

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
rxen	Enable SpaceWire time code reception
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

Table 180: spwcuc_cfg member descriptions.

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;
};
```

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

Table 181: spwcuc_status member descriptions.

36 Gaisler Packetwire RX driver (GRPWRX)

36.1 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.

36.1.1 Software Driver

The driver provides means for threads to receive GRPWRX packets using standard I/O operations.

36.1.2 Support

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

36.2 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.

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 *grpwrx_register_drv* whose prototype is provided in grpwrx.h is used for registering the driver:

```
grpwrx_register_drv();
```

36.2.2 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:

Core number	Filesystem name	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

Table 182: Core number to device name conversion.

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 182.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 183: Open *errno* values.

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 *grpwrx* 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 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.

36.2.4.1 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;
};
```

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

Table 184: grpwrx_ioc_hw member descriptions.

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;
};
```

Member	Description
framing	Enable framing mode (1)
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

Table 185: grpwrx_ioc_config member descriptions.

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;
};
```

Member	Description
flags	Mask indicating options, transmission state and errors for the packet. GRPWRX_FLAGS_XXX. See Table 186
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 186: grpwrx_packet member descriptions.

Flag	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.
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.

Table 187: grpwrx_packet flags description.

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;
};
```

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

Table 188: grpwrx_list member descriptions.

The grpwrx_ioc_stats structure contain statistics collected by the driver.

```
struct grpwrx_ioc_stats {
     unsigned long long      packets_received;
     unsigned int            err_underrun;
};
```

Member	Description
packets_receved	Number of packets successfully received by the GRPWRX core
err_underrun	Number of AMBA underrun errors

Table 189: grpwrx_ioc_stats member descriptions.

36.2.4.2 Configuration

The grpwrx core and driver are configured using *ioctl* calls. The table 188 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 187.

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);
```

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 190: ERRNO values for *ioctl* calls.

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 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.

Table 191: *ioctl* calls supported by the grpwrx driver.

36.2.4.2.1 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 187.

36.2.4.2.2 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.

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 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.

Bit number	**Description**
GRPWRX_BLKMODE_POLL	Enables polling mode
GRPWRX_BLKMODE_BLK	Enables blocking mode

Table 192: SET_BLOCKING_MODE *ioctl* arguments

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.

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.

This command never fail.

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 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 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 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.

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

grpwrx_ioc_config structure. The data area will be updated according to the *grpwrx_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 *grpwrx_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 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.

36.2.4.2.11 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 36.2.2. 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 186 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.

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.

Table 193: ERRNO values for *RECLAIM*

36.2.4.2.12 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 36.2.4.1. 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 36.2.2. 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.

36.2.5 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.

37 Gaisler AES DMA driver (GRAES)

37.1 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.

37.1.1 Software Driver

The driver provides means for threads to receive GRAES packets using standard I/O operations.

37.1.2 Support

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

37.2 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.

37.2.1 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();
```

37.2.2 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:

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

Table 194: Core number to device name conversion.

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 194.

Errno	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory.

Table 195: Open *errno* values.

37.2.3 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.

37.2.4 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 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.

37.2.4.1 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;
};
```

Member	Description
keysize	GRAES core key size, fixed 256

Table 196: graes_ioc_hw member descriptions.

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;
};
```

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

Table 197: graes_ioc_config member descriptions.

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 */
};
```

Member	Description
flags	Mask indicating options, Processing state and errors for the block. GRAES_FLAGS_XXX. See Table 198
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 198: graes_block member 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	As TRANSLATE, however if the translated payload addresses equals the payload address the TRANSLATE_AND_REMEMBER bit is cleared and the TRANSLATE bit is set.

Table 199: graes_block flags description.

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;
};
```

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

Table 200: graes_list member descriptions.

The graes_ioc_stats structure contain statistics collected by the driver.

```
struct graes_ioc_stats {
     unsigned long long     blocks_processed;
     unsigned int           err_underrun;
};
```

Member	Description
blocks_processed	Number of blocks successfully processed by the GRAES core
err_underrun	Number of AMBA underrun errors

Table 201: graes_ioc_stats member descriptions.

37.2.4.2 Configuration

The graes core and driver are configured using *ioctl* calls. The table 200 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 199.

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);
```

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 202: ERRNO values for *ioctl* calls.

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.
ENCRYPT	Started	Add a chain of GRAES blocks to the en/decryption queue of the GRAES driver.

Table 203: *ioctl* calls supported by the graes driver.

37.2.4.2.1 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.

37.2.4.2.2 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.

37.2.4.2.3 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.

37.2.4.2.4 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.

Bit number	Description
GRAES_BLKMODE_POLL	Enables polling mode
GRAES_BLKMODE_BLK	Enables blocking mode

Table 204: SET_BLOCKING_MODE *ioctl* arguments

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.

37.2.4.2.5 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.

37.2.4.2.6 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 37.2.4.1.

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 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.

37.2.4.2.7 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 37.2.4.1.

This command only fail if the pointer argument is invalid.

37.2.4.2.8 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.

37.2.4.2.9 CLR_STATS

This command reset the driver's internal statistics counters.

This command never fail.

37.2.4.2.10 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.

37.2.4.2.11 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 37.2.2. 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 198 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.

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.

Table 205: ERRNO values for *RECLAIM*

37.2.4.2.12 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 37.2.4.1. 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 37.2.2, 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.

37.2.5 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.

38 Support

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