Gaisler LEON3 RTEMS driver documentation

Software Drivers for Gaisler RTEMS distribution

GR-RTEMS-DRIVER
Version 1.0.2

Written by Daniel Hellström, Kristoffer Glembo , Marko Isomäki

October 2007

Första Långgatan 19
413 27 Göteborg
Sweden

tel +46 317758650
fax +46 31 421407
homepage: www.gaisler.com

Table of Contents

1 Drivers documentation introduction

This document contain a compilation of documents describing some of the LEON3 drivers included in the Gaisler RTEMS distribution. Each driver is described in a separate chapter. Generally, the drivers are designed to have common interface for multiple operating systems all supported by Gaisler. Naturally, some of the drivers share their documentation between operating systems, parts specific to other operating systems than RTEMS may be skipped.

Gaisler RTEMS samples and a common makefile can be found under /opt/rtems-4.6/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 Gaisler SpaceWire (GRSPW)

2.1 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.

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

2.1.1 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.

2.1.2 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.

2.1.3 Support

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

2.2 USER INTERFACE

The RTEMS GRSPW driver supports the standard accesses 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.

2.2.1 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 registration is performed by a call to *grspw_register* with appropriate arguments. The function *grspw_register* whose prototype is provided in *grspw.h* returns 0 on success and 1 on failure. The function takes one argument, a pointer to a data structure describing the AMBA PnP bus that the driver should expect the GRSPW core to be found. For LEON3 systems this is almost always *&amba_conf*. However, if the GRSPW is attached to an external AMBA bus for example on another chip accessed over PCI one need to create another data structure describing that bus. The function *amba_scan* is used to scan AMBA buses, it is used in the RASTA and the Companion Chip drivers for example.

2.2.2 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. The first 3 names are printed out:

Device number	Filesystem name
0	/dev/grspw0
1	/dev/grspw1
2	/dev/grspw2

Table 1: 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 2.

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

Table 2: Open errno values.

2.2.3 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 SpaceWire driver.

2.2.4 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.

2.2.4.1 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 3: 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 4: 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 5: 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 6: spw_config member descriptions.

2.2.4.2 Configuration

The GRSPW core and driver are configured using ioctl calls. Table 8 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 7.

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 7: 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.

Table 8: ioctl calls supported by the GRSPW driver.

2.2.4.2.1 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 9: START argument description

2.2.4.2.2 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.

2.2.4.2.3 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.

2.2.4.2.4 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.

2.2.4.2.5 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.

2.2.4.2.6 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.

2.2.4.2.7 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.

2.2.4.2.8 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.

2.2.4.2.9 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 10: SET_COREFREQ argument description

2.2.4.2.10 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.

2.2.4.2.11 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.

2.2.4.2.12 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.

2.2.4.2.13 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.

2.2.4.2.14 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.

2.2.4.2.15 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.

2.2.4.2.16 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.

2.2.4.2.17 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.

2.2.4.2.18 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.

2.2.4.2.19 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.

2.2.4.2.20 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.

2.2.4.2.21 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.

2.2.4.2.22 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.

2.2.4.2.23 CLR_STATISTICS

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

2.2.4.2.24 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.

2.2.4.2.25 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.

2.2.4.2.26 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.

2.2.4.2.27 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.

2.2.5 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 11: ERRNO values for write and ioctl send.

2.2.6 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 12: ERRNO values for read calls.

2.3 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 */
    ...
}
```

3 Gaisler B1553BRM DRIVER (BRM)

3.1 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.

3.1.1 BRMHardware

See the B1553BRM Hardware manual available at www.gaisler.com.

3.1.2 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

3.1.3 Supported OS

Currently the driver is available for RTEMS.

3.1.4 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-4.6/src/examples/samples/rtems-brm.c, brm_lib.c and brm_lib.h.

The example can be built by running:

```
cd /opt/rtems-4.6/src/examples/samples
make clean rtems-brm_rt rtems-brm_bc rtems-brm_bm
```

3.2 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.

3.2.1 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 *brm_register* whose prototype is provided in *brm.h* is used for registering the driver. It returns 0 on success and 1 on failure. A typical register call from a LEON3 Init task:

```
if ( brm_register(&amba_conf,2,0,3) )
     printf("BRM register Failed\n");
```

The first argument is a pointer to the AMBA bus, the rest of the parameters are used to control the clocking of the BRM device. See table below for a description.

Arg No	Arg Name	Description
1	bus	pointer to a scanned AMBA bus, always &amba_conf.
2	clksel	Selects clock source (input value to the clock MUX)
3	clkdiv	Selects clock prescaler, may not be available for all clock sources
4	brm_freq	The input clock frequency to the BRM core. 0 = 12MHz, 1 = 16MHz, 2= 20MHz, 3 = 24MHz.

Table 13: *brm_register* argument description

During the registration process all BRM devices in the system will be registered as indicated in table 14. The registration function is called once.

3.2.2 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. The first 3 names are printed out:

Device number	**Filesystem name**
0	/dev/brm0
1	/dev/brm1
2	/dev/brm2

Table 14: Device number to device name conversion.

An example of an RTEMS *open* call is shown below.

```
fd = open("/dev/brm0", O_RDWR)
```

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

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

Table 15: Open *errno* values.

3.2.3 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 *brm* driver.

3.2.4 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.

3.2.4.1 Data structures

3.2.4.1.1 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 16: *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 17: 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.

3.2.4.1.2 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];
};
```

Member	**Description**	
rtaddr	Remote terminal address - For non RT to RT message only rtaddr[0] is used. It specifies the address of the remote terminal to which the message should be sent. For RT to RT messages rtaddr[0] specifies the receive address and rtaddr[1] the transmit address.	
subaddr	The subaddr array works in the same manner as rtaddr but for the subaddresses.	
wc	Word Count - Specifies the word count, or mode code if subaddress is 0 or 31.	
ctrl	**Bit(s)**	**Description**
	15	Message Error. Set by BRM while traversing list if protocol error is detected.
	14-6	-
	5	END. Indicates end of list
	4-3	Retry, Number of retries, 0=4, 1=1, 2=2, 3=3. BC will alternate buses during retries.
	2	AB, 1 – Bus B, 0 - Bus A
	1	1 RT to RT
		0 normal
	0	0 RT Transmit
		1 RT receive (ignored for RT to RT)
tsw	Status words	
data	Data in message, not used for RT receive (ctrl.0 = 1).	

Table 18: struct bc_msg member description

3.2.4.1.3 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];
};
```

Member	**Description**	
miw	**Bit(s)**	**Description**
	15	Overrun- Indicates that the monitor message queue has been overrun.
	14-10	-
	9	Channel A/B -1 if message captured on bus A, 0 if captured on bus B.
	8	RT to RT transfer - 1 if message is part of an RT to RT transfer
	7	Message Error - 1 if an error was encountered during message processing. Bit 4-0 gives the details of the error.
	6	Mode code without data - 1 if a mode code without data word was captured.
	5	Broadcast - 1 if a broadcast message was captured.
	4	-
	3	Time out - If set, the number of captured data words was less than the amount specified by the word count.
	2	Overrun -If set, the number of captured data words was more than amount specified by the word count.
	1	Parity- 1 if the BM detected a parity error in the received data.
	0	Manchester error - 1 if a Manchester decoding error was detected during data reception.
cw1	1553 Command word 1	
cw2	1553 Command word 2, only used for RT to RT transfers and then holds the transmit command.	
sw1	1553 Status word 1	
sw2	1553 Status word 2, is only used for RT to RT transfers and then holds the status word from the transmitting RT.	
time	Time tag (time) Contains the value of the internal timer register when the message was captured.	
data	An array of 32 16 bit words. The command word specifies how many data words that are valid. For receive mode codes with data the first data word is valid.	

Table 19: struct bm_msg member description

3.2.5 Configuration

The BRM core and driver are configured using *ioctl* calls. The table 15 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 16.

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 20: 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 21: *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.

3.2.5.1 SET_MODE

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

3.2.5.2 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.

3.2.5.3 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.

3.2.5.4 SET_RT_ADDR

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

3.2.5.5 BRM_SET_STD

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

3.2.5.6 BRM_SET_BCE

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

3.2.5.7 BRM_TX_BLOCK

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

3.2.5.8 BRM_RX_BLOCK

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

3.2.5.9 BRM_CLR_STATUS

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

3.2.5.10 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 22: Status bit mask

3.2.5.11 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 13 for the description of the notification argument.

3.2.6 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.

3.2.7 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.

3.2.8 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.

4 CAN DRIVER INTERFACE (GRCAN)

4.1 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.

4.1.1 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 function *grcan_register* whose prototype is provided in grcan.h is used for registering the driver. It returns 0 on success and 1 on failure. A typical register call from the initialization routines from a LEON2 *Init* task:

```
amba_confarea_type amba_bus;

/* Scan AMBA bus Plug&Play */
amba_scan(&amba_bus,0xfff00000,NULL);

if ( grcan_register(&amba_bus) )
      printf("GRCAN register Failed\n");
```

For LEON3 targets it is enough to call *grcan_register* directly because the AMBA bus that the processor is attached to is already scanned by the LEON3 BSP:

```
if ( grcan_register(&amba_conf) )
      printf("GRCAN register Failed\n");
```

4.1.2 Opening the device

Opening the device enables the user to access the hardware of a certain CAN device. The driver is used for all GRCAN 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. The first 3 names are printed out:

Device number	**Filesystem name**
0	/dev/grcan0
1	/dev/grcan1
2	/dev/grcan2

Table 23: Device 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 23.

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

Table 24: Open *errno* values.

4.1.3 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 *grcan* driver.

4.1.4 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.

4.1.4.1 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 25: 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 26: 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 27: 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;
};
```

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

Table 28: 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 29: grcan_selection member description

4.1.4.2 Configuration

The CAN core and driver are configured using *ioctl* calls. The table 26 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 25.

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 30: 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 31: *ioctl* calls supported by the CAN driver.

4.1.4.2.1 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.

4.1.4.2.2 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.

4.1.4.2.3 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.

4.1.4.2.4 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.

4.1.4.2.5 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.

4.1.4.2.6 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.

4.1.4.2.7 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*.

4.1.4.2.8 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.

4.1.4.2.9 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.

4.1.4.2.10 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.

4.1.4.2.11 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.

4.1.4.2.12 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.

4.1.4.2.13 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.

4.1.4.2.14 CLR_STATS

Clears the driver's collected statistics.

This call never fail.

4.1.4.2.15 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.

4.1.4.2.16 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.

4.1.4.2.17 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 32: Status bit mask

This call never fail.

4.1.5 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 33: 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.

4.1.6 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. 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 34: ERRNO values for *read* calls.

5 Gaisler Opencores CAN driver (OC_CAN)

5.1 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.

5.1.1 CAN Hardware

See the OC_CAN Hardware manual available at www.gaisler.com.

5.1.2 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.

5.1.3 Supported OS

Currently the driver is available for RTEMS.

5.1.4 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-4.6/src/examples/samples/rtems-occan.c, occan_lib.c and occan_lib.h.

The example can be built by running:

```
cd /opt/rtems-4.6/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.

5.1.5 Support

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

5.2 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.

5.2.1 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 *occan_register* whose prototype is provided in occan.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) )
     printf("OCCAN register Failed\n");
```

5.2.2 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 35: 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 35.

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

Table 36: Open *errno* values.

5.2.3 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 *occan* driver.

5.2.4 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.

5.2.4.1 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 37: occan_afilter member descriptions.

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. 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 38: 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 39: occan_stats member descriptions.

5.2.4.2 Configuration

The OC_CAN core and driver are configured using *ioctl* calls. The table 38 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 37.

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 40: 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 41: *ioctl* calls supported by the OC_CAN driver.

5.2.4.2.1 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.

5.2.4.2.2 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.

5.2.4.2.3 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.

5.2.4.2.4 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 36above.

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 42: Status bit mask

This call never fail.

5.2.4.2.5 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.

5.2.4.2.6 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.

5.2.4.2.7 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 43: SET_BLK_MODE *ioctl* arguments

This call never fail.

5.2.4.2.8 SET_BUF_LEN

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 44: 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.

5.2.5 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 45: 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.

5.2.6 Reception

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

```
CANMsg rx_msgs[5];

len = read(fd, rx_msgs, sizeof(rx_msgs));
```

The requested number of bytes to be read is given in the third argument. The messages will be stored in rx_msgs. The actual number of received bytes (a multiple of sizeof(CANMsg)) is returned by the function on success and -1 on failure. In the latter case *errno* is also set.

The *CANMsg* data structure is described in the data structure subsection.

The call will fail if a null pointer is passed, invalid buffer length, the CAN core is in reset mode or due to a bus off error in blocking mode.

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

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 46: ERRNO values for *read* calls.

6 RAW UART DRIVER INTERFACE (APBUART)

6.1 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 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.

6.1.1 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 function *apbuart_register* whose prototype is provided in *apbuart.h* is used for registering the driver. It returns 0 on success and 1 on failure. A typical register call from the initialization routines from a LEON2 *Init* task:

```
amba_confarea_type amba_bus;

/* Scan AMBA bus Plug&Play */
amba_scan(&amba_bus,0xfff00000,NULL);

if ( apbuart_register(&amba_bus) )
      printf("APBUART register Failed\n");
```

For LEON3 targets it is enough to call *apbuart_register* directly because the AMBA bus that the processor is attached to is already scanned by the LEON3 BSP:

```
if ( apbuart_register(&amba_conf) )
      printf("APBUART register Failed\n");
```

6.1.2 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. The first 3 names are printed out:

Device number	Filesystem name
0	/dev/apbuart0
1	/dev/apbuart1
2	/dev/apbuart2

Table 47: 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 47.

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

Table 48: Open *errno* values.

6.1.3 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 *apbuart* driver.

6.1.4 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.

6.1.4.1 Configuration

The UART core and driver are configured using *ioctl* calls. The table 49 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 48.

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 49: 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 50: *ioctl* calls supported by the APBUART driver.

6.1.4.1.1 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.

6.1.4.1.2 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.

6.1.4.1.3 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.

6.1.4.1.4 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.

6.1.4.1.5 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.

6.1.4.1.6 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.

6.1.4.1.7 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 51: SET_BLOCKING Argument Bit Mask

6.1.4.1.8 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.

6.1.4.1.9 CLR_STATS

Resets drivers statistics counters.

6.1.4.1.10 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.

6.1.5 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 52: ERRNO values for *write*

6.1.6 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 53: ERRNO values for *read* calls.

7 Support

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