BCC User's Manual

Table of Contents

1. Introduction

1.1. Scope

BCC is a cross-compiler for LEON2, LEON3 and LEON4 processors. It is based on the GNU compiler tools, the newlib C library and a support library for programming LEON systems. The cross-compiler allows compilation of C and C++ applications.

There is also an experimental LLVM/Clang version of BCC based on the LLVM compiler framework. More information about the LLVM based toolchain can be found in Chapter 3. The GCC and LLVM/Clang versions of BCC are distributed in separate packages. The libraries in the two provided packages are compiled using the selected compiler, with the exception of libgcc which is always compiled with GCC.

BCC consists of the following packages:

- GNU GCC 7.2.0 C11/C11++ compiler with support for atomic operations
- GNU binutils 2.32
- Newlib C library 2.5.0
- `libbcc` - A user library for programming LEON systems
- `libdrv` - A user library of GRLIB peripheral drivers
- GNU GDB 8.2.1 source-level debugger

In the LLVM/Clang version, the GCC package is replaced by:

- Clang 8.0.0 C11/C11++ compiler with support for atomic operations (LLVM version)

1.2. Installation

1.2.1. Host requirements

BCC is provided for two host platforms: GNU Linux/x86_64 and Microsoft Windows. The following are the platform system requirements:

GCC Version:

Linux:	Linux-2.6.x, glibc-2.11 (or higher)
Windows:	-

LLVM Version:

Linux:	Linux-3.5.x, glibc-2.15 (or higher)
Windows:	-

In order to recompile BCC from sources, automake-1.11.1 and autoconf-2.68 is required. MSYS-DTK-1.0.1 is needed on Microsoft Windows platforms to build autoconf and automake. Sources for automake and autoconf can be found on the GNU ftp server:

- ftp://ftp.gnu.org/gnu/autoconf/
- ftp://ftp.gnu.org/gnu/automake/

MSYS and MSYS-DTK can be found at http://www.mingw.org.

1.2.2. Linux

After obtaining the compressed tar file for the binary distribution, uncompress and untar it to a suitable location. The Linux version of BCC has been prepared to reside in the `/opt/bcc-2.1.1-gcc/` directory, but can be installed in any location. The distribution can be installed with the following commands:

```
$ cd /opt
$ tar -C /opt -xf /opt/bcc-2.1.1-gcc-linux64.tar.xz
```

After the compiler is installed, add `/opt/bcc-2.1.1-gcc/bin` to the executables search path (`PATH`) and `/opt/bcc-2.1.1-gcc/man` to the manual page path (`MANPATH`).

1.2.3. Windows

BCC for Windows does not require any additional packages and can be run from a standard command prompt. The toolchain installation zip file, `/opt/bcc-2.1.1-gcc-mingw64.zip`, shall be extracted to `C:\opt` creating the directory `C:\opt\bcc-2.1.1`. The toolchain executables can be invoked from the command prompt by adding the executable directory to the `PATH` environment variable. The directory `C:\opt\bcc-2.1.1\bin` can be added to the `PATH` variable by selecting *"My Computer->Properties->Advanced->Environment Variables"*.

Development often requires some basic utilities such as **make**, but is not required to compile. On Windows platforms the MSYS Base system can be installed to get a basic UNIX like development environment (including **make**).

See http://www.mingw.org for more information on MinGW and the optional MSYS environment.

1.3. Contents of /opt/bcc-2.1.1-gcc

The binary installation of BCC contains the following sub-directories:

`bin/`	Executables
`doc/`	GNU, newlib and BCC documentation
`man/`	Manual pages for GNU tools
`sparc-gaisler-elf/`	SPARC target libraries, include files and LEON BSP
`sparc-gaisler-elf/bsp/`	Board Support Packages for LEON systems
`src/`	Various sources, examples and build scripts
`src/examples/`	BCC example applications
`src/libbcc/`	`libbcc` source code and build scripts
`src/libdrv/`	`libdrv` source code, examples and build scripts

1.4. BCC tools

The following tools are installed with BCC:

`sparc-gaisler-elf-addr2line`	Convert address to C/C++ line number
`sparc-gaisler-elf-ar`	Library archiver
`sparc-gaisler-elf-as`	Cross-assembler
`sparc-gaisler-elf-c++`	C++ cross-compiler
`sparc-gaisler-elf-c++filt`	Utility to demangle C++ symbols
`sparc-gaisler-elf-cpp`	The C preprocessor
`sparc-gaisler-elf-g++`	Same as sparc-gaisler-elf-c++
`sparc-gaisler-elf-gcc`	C/C++ cross-compiler
`sparc-gaisler-elf-gcov`	Coverage testing tool
`sparc-gaisler-elf-gdb`	GNU GDB C/C++ level Debugger
`sparc-gaisler-elf-gdb-6.8`	GNU GDB C/C++ level Debugger
`sparc-gaisler-elf-gprof`	Profiling utility
`sparc-gaisler-elf-ld`	GNU linker
`sparc-gaisler-elf-nm`	Utility to print symbol table
`sparc-gaisler-elf-objcopy`	Utility to convert between binary formats
`sparc-gaisler-elf-objdump`	Utility to dump various parts of executables
`sparc-gaisler-elf-ranlib`	Library sorter
`sparc-gaisler-elf-readelf`	ELF file information utility

`sparc-gaisler-elf-size`	Utility to display segment sizes
`sparc-gaisler-elf-strings`	Utility to dump strings from executables
`sparc-gaisler-elf-strip`	Utility to remove symbol table

1.5. Documentation

The GNU and newlib documentation is distributed together with the toolchain, located in the `doc/` directory of the installation.

GNU tools:

`as.pdf`	Using as - the GNU assembler
`binutils.pdf`	The GNU binary utilities
`cpp.pdf`	The C Preprocessor
`gdb.pdf`	Debugging with GDB
`ld.pdf`	The GNU linker
`gcc/gcc.pdf`	Using and porting GCC

Newlib C library:

`libc.pdf`	Newlib C Library
`libm.pdf`	Newlib C Math Library

BCC:

`bcc.pdf`	BCC User's Manual (this document)

All documents are all provided in PDF format, with searchable indexes.

1.6. Toolchain source code distribution

The BCC toolchain source code distribution can be used to rebuild the toolchain host binaries (compiler, Binutils) and the target C library.

NOTE: Installing the toolchain source code is *not* required for creating a new BSP or to modify an existing one. The BSP source code (`libbcc`) is installed together with the binary distribution under `src/libbcc/`.

1.6.1. BCC source code installation

The source code for the BCC 2.1.1 toolchain is distributed in an archive named `bcc-2.1.1-src.tar.bz2`, available on the Cobham Gaisler website. It contains source code for the target C library and the host compiler tools (binutils, GCC, GDB).

Installing the source code is optional but recommended when debugging applications using the C standard library. The target libraries have been built with debug information making it possible for GDB to find the sources files. It allows for example to step through the target C standard library code.

The BCC source code files are assumed to be located in `/opt/bcc-2.1.1-gcc/src/bcc-2.1.1`. The sources can be installed by extraction the source distribution archive `bcc-2.1.1-src.tar.bz2` to `/opt/bcc-2.1.1-gcc/src`. It can be done as follows for the Linux/GCC version of BCC.

```
$ cd /opt/bcc-2.1.1-gcc/src
$ tar xf bcc-2.1.1-src.tar.bz2
```

1.6.2. Building

A script named `ubuild.sh` is included in the source distribution.

To build and install the BCC compiler tools, GDB and the C library in `/tmp/bcc-2.1.1-local`, the following steps shall be performed:

```
$ cd /opt/bcc-2.1.1-gcc/src/bcc-2.1.1
$ ./ubuild.sh --destination /tmp/bcc-2.1.1-local --toolchain --gdb
```

Either of the paramters `--toolchain` or `--gdb` can be omitted. Execute `ubuild.sh --help` for more information on how to use the script.

1.7. Support

BCC is provided freely without any warranties. Technical support can be obtained from Cobham Gaisler through the purchase of technical support contract. Please contact sales@gaisler.com for more details.

2. Using BCC

This chapter gives an overview on how to develop applications using BCC 2.1.1

2.1. General development flow

Compilation and debugging of applications is typically done in the following steps:

1. Compile and link the program with GCC
2. Debug program using a simulator (GDB connected to TSIM)
3. Debug program on remote target (GDB connected to GRMON)
4. Create boot-prom for a standalone application with mkprom2

2.2. Compiler options

The GCC front-end, **sparc-gaisler-elf-gcc**, and the Clang front-end, **sparc-gaisler-elf-clang**, has been modified to support the following options specific to BCC and LEON systems:

`-qbsp=`*`bspname`*	Use target libraries, startup files and linker scripts for a specific LEON system. The parameter *bspname* corresponds to a Board Support Package (BSP). A description of the BSPs distributed with BCC is given in Chapter 7. The BSP `leon3` is used as default if the `-qbsp=` option is not given.
`-qnano`	Use a version of the newlib C library compiled for reduced foot print. The nano version implementations of the `fprintf()fscanf()` family of functions are not fully C standard compliant. Code size can decrease with up to 30 KiB when `printf()` is used.
`-qsvt`	Use the single-vector trap model described in *SPARC-V8 Supplement, SPARC-V8 Embedded (V8E) Architecture Specification*.

Useful (standard) options are:

`-g`	Generate debugging information - should be used when debugging with GDB.
`-msoft-float`	Emulate floating-point - must be used if no FPU exists in the system.
`-O2` or `-Os`	Optimize for maximum performance or minimal code size.
`-Og`	Optimize for maximum debugging experience.
`-mcpu=leon3`	Generate SPARC V8 code. Includes support for the `casa` instruction.
`-mflat`	Enable single register window model (flat). See Section 2.7.
`-mfix-gr712rc`	Enable workarounds applicable to GR712RC. `-mfix-gr712rc` enables workarounds for the following technical notes: • GRLIB-TN-0009 • GRLIB-TN-0011 • GRLIB-TN-0012 • GRLIB-TN-0013 • GRLIB-TN-0018
`-mfix-ut700`	Enable workarounds applicable to UT700 and UT699E. `-mfix-ut700` enables workarounds for the following technical notes: • GRLIB-TN-0009 • GRLIB-TN-0010 • GRLIB-TN-0013 • GRLIB-TN-0018
`-qfix-tn0018`	Enable workarounds for GRLIB technical note GRLIB-TN-0018.

2.2.1. sparc-gaisler-elf-gcc options

The following options are available in the GCC version of BCC.

`-flto`	Enable link time optimization.
`-mcpu=leon`	Generate SPARC V8 code.
`-mcpu=leon3v7`	Generate SPARC V7 code (no `mul`/`div` instructions). Includes support for `casa` instruction.
`-mfix-b2bst`	Enable workarounds for GRLIB technical note GRLIB-TN-0009.
`-mfix-tn0013`	Enable workarounds for GRLIB technical note GRLIB-TN-0013.
`-mfix-ut699`	Enable the documented workarounds for the floating-point errata and the data cache nullify errata of the UT699 processor. This option also enables workarounds for GRLIB-TN-0009, GRLIB-TN-0013 and GRLIB-TN-0018.

Other GNU GCC options are explained in the gcc manual (`doc/gcc.pdf`), see Section 1.5.

2.2.2. sparc-gaisler-elf-clang options

The following options are availble in the LLVM/Clang version of BCC.

`-Oz`	Aggressively optimize for minimal code size
`-mrex`	Enables generation of the LEON-REX SPARC instruction set extension.
`-no-integrated-as`	Use the GNU assembler instead of the LLVM integrated assembler. Note the GNU assembler does not have support for the LEON-REX extension.

Clang generates SPARC V8 code by default.

2.3. Compiling BCC applications

To compile and link a BCC application with GCC, use **sparc-gaisler-elf-gcc**:

```
$ sparc-gaisler-elf-gcc -O2 -g hello.c -o hello
```

To compile and link a BCC application with Clang, use **sparc-gaisler-elf-clang**:

```
$ sparc-gaisler-elf-clang -O2 -g hello.c -o hello
```

BCC creates executables suitable for most LEON3 systems by default. The default load address is start of RAM, i.e. `0x40000000`. Other load addresses can be specified through the use of the `-Ttext` linker option (see Section 7.1).

To generate executables customized for specific components and systems, `-qbsp=`*name* and `mcpu=`*name* options should be used during both compile and link stages. A table with recommended compiler options for LEON systems can be found in Appendix A (GCC), and Appendix B (Clang).

2.4. Floating-point considerations

If the target LEON processor has no floating-point hardware, then all applications must be compiled and linked with the `-msoft-float` option to enable floating-point emulation. When running an application compiled and linked with `-msoft-float` in the TSIM simulator, the simulator should be started with the `-nfp` option (no floating-point) to disable the FPU.

Floating-point hardware state is not automatically saved and restored when BCC dispatches an interrupt service routine (ISR). Any ISR code making use of the floating-point hardware should save and restore the context as described in Section 5.7.

To link an application which uses the C standard library math functins, the linker option `-lm` should be used. This links the application with the library file `libm.a`.

2.5. LEON SPARC V8 instructions

LEON3 processors can be configured to implement the SPARC V8 multiply and divide instructions. The GCC version of BCC does by default not issue those instructions, but emulates them trough a library. To enable gen-

eration of mul/div instruction, use the -mcpu=leon or -mcpu=leon3 option during both compilation and linking. This improves performance on compute-intensive applications and floating-point emulation.

2.6. Multiply and accumulate instructions

LEON2, LEON3 and LEON4 can support multiply and accumulate (umac/smac) instructions. The compiler will never issue those instructions but can be coded in assembly. The BCC provided assembler and utilities support this feature.

2.7. Single register window model (flat)

The BCC compilers and run-time uses the standard SPARC V8 ABI by default. GCC and Clang provides an optional ABI, enabled with the -mflat option, which does not generate any save and restore instructions. This is known as the *single register window model*, or *flat* model. Instead of switching register windows at function borders, the flat model stores registers on the stack. -mflat sets the preprocessor symbol _FLAT.

An application compiled and linked with the flat model will never generate window_overflow and window_underflow traps.

Compiling with -mflat affects code size. As an example, the Newlib C library (libc.a) text segment is 8% larger in the -mcpu=leon3 -mflat multilib compared to the -mcpu=leon3 version.

BCC run-time is compatible with the single register window model when linked with -mflat. The example below compiles and links an application with the flat model.

```
$ sparc-gaisler-elf-gcc -mflat -O2 -c main.c -o main.o
$ sparc-gaisler-elf-gcc -mflat -O2 -c somecode.c -o somecode.o
$ sparc-gaisler-elf-gcc -mflat main.o somecode.o -o myapplication.elf
```

NOTE: The current GCC 7.2.0 -mflat implementation was introduced with GCC 4.6. It is not binary compatible with the old GCC -mflat implementation which was deprecated in GCC 3.4.6.

2.8. Register usage

The compiler and run-time uses the SPARC input, local and output registers as specified by the SPARC V8 ABI. For global registers, the following applies:

%g1 ... %g4	Used by compiler and BCC run-time.
%g5	Not used by compiler. Used by BCC run-time only when -mflat is used. Can be used freely by the application if -mflat is not used.
%g6 ... %g7	Not used by compiler. Not used by BCC run-time. Can be used by the application for any purpose.

2.9. Single vector trapping

When the target hardware is configured to support single vector trapping (SVT), the -qsvt switch can be used with the linker to build an image which uses a two-level trap dispatch table rather than the standard one-level trap table. The code saving amounts to ~4KiB for the trap table and trap handling is slightly slower with single vector trapping. The number of extra instructions needed for single vector trapping dispatching is constant. The application image will try to enable SVT on boot using %asr17.

2.10. Memory organization

The resulting executables are in ELF format and have three main segments; text, data and bss. The text segment is by default at address 0x40000000 for LEON2/3/4, followed immediately by the data and bss segments.

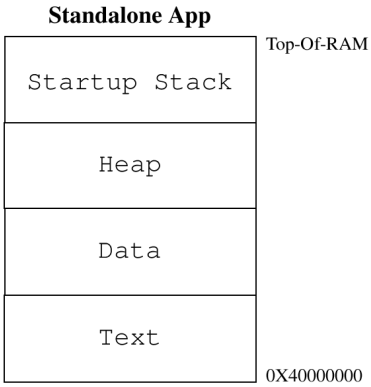

Figure 2.1. BCC RAM application memory map

NGMP based LEON4 designs such as GR740 and LEON4-N2X have RAM at `0x00000000`. This is supported by the `GR740` BSP.

The SPARC trap table is always located at the start of the `text` segment. If single vector trapping is not used, then the trap table is exactly 4 KiB. For single vector trapping, the allocated space is 380 bytes by default. The exact size depends on the user configuration.

Program stack starts at top-of-ram and extends downwards. The area between the end of `bss` and the bottom of the stack is by default used for the heap. BCC auto-detects end-of-ram by inspecting the stack pointer provided by the boot loader or GRMON at early boot. Hence the heap is sized by the boot loader by default.

Section 8.8 describes how the heap can be configured by the application.

2.11. BCC Board Support Packages

BCC uses a Board Support Package (BSP) mechanism to provide support for LEON system variations.

A BCC BSP includes the following:

- Target linker scripts.
- BCC device mapping and initialization.
- Customization of the `libbcc` user library.
- C header files with register definitions.
- Custom drivers available to the user.

BSP is selected with the `-qbsp=`*`bspname`* compiler option. This option does however not explicitly specify what code the compiler outputs. It means that the appropriate `-mcpu=`*`cpuname`* option has to be given to GCC even when a BSP is selected.

A description of the BSPs distributed with BCC is given in Chapter 7. `-qbsp=leon3` is used by default.

2.12. Peripheral driver library

BCC comes with GRLIB peripheral driver library in both object and source code. Include files are available via the BCC default include paths.

The option `-ldrv` should be given to the linker to include the library `libdrv.a`. This link library is built for each compiler multilib.

The user API is available in the Device drivers reference and examples can be found in `src/libdrv/examples/`.

2.13. Multiprocessing

BCC includes support for building Asymmetric Multiprocessing (AMP) applications: The GCC C11 compiler can generate atomic CPU instructions and the BCC AMP API described in Section 5.10 operates on LEON multiprocessor support hardware.

Symmetric Multiprocessing (SMP) is not supported by BCC.

2.14. Debugging with GDB

GDB 8.2.1 is distributed with BCC in the host executable file **sparc-gaisler-elf-gdb**. To generate debug information when compiling object files, the compiler (or assembler) option `-g` is used. Target libraries distributed with BCC are built with debug information and the related source code can be installed as described in Section 1.6.

For information on how to connect with GDB to TSIM simulator or the GRMON hardware monitor, see their respective documentation.

2.14.1. Debug information considerations

- GCC and Clang distributed with BCC 2.1.1 generates debug information in dwarf-4 format by default.
- All prebuilt target libraries distributed with BCC 2.1.1 have dwarf-4 debug information.

GDB-6.8 supports dwarf debug information version up to dwarf-2. GCC-7 and later is focused around dwarf-4. There is a dwarf-2 mode in GCC-7, but it does not work very well with GDB-6.8. Therefore, BCC 2.1.1 uses the more recent GDB-8.2.

GDB-8.2 as distributed with BCC has full support for dwarf-4. However, in modern versions of GDB (including as GDB-8.2), the semantics of the GDB `extended-remote` protocol used for communication between a GDB server and client has changed compared to GDB-6.8. TSIM and GRMON implement the server part of the GDB remote protocol.

- TSIM2 is compatible with GDB-6.8.
- TSIM3 is compatible with GDB-6.8 and GDB-8.2.
- GRMON2 is compatible with GDB-6.8.
- GRMON3 is compatible with GDB-6.8 and GDB-8.2.

This means that debugging BCC applictions in GDB is best supported with GRMON3 and TSIM3.

TSIM2 users who want to use GDB are recommended to use BCC 2.0.x which uses dwarf-2 by default and is distributed with dwarf-2 target objects.

2.15. Examples

A collection of benchmarks and examples on how to use the BCC user library can be found in the `src/examples/` directory of the BCC binary distribution. The directory also contains a `Makefile` which can be used to build the examples for different configurations (BSP:s).

To build all examples for all BSP:s, issue:

```
$ cd src/examples
$ make
sparc-gaisler-elf-gcc -g -O3 -qbsp=agga4 -mcpu=leon hello/hello.c -o bin/agga4/./hello.elf
sparc-gaisler-elf-gcc -g -O3 -qbsp=agga4 -mcpu=leon stanford/stanford.c -o bin/agga4/./stanford.elf
sparc-gaisler-elf-gcc -g -O3 -qbsp=agga4 -mcpu=leon whetstone/whetstone.c -o bin/agga4/./whetstone.elf -lm
sparc-gaisler-elf-gcc -g -O3 -qbsp=agga4 -mcpu=leon paranoia/paranoia.c -o bin/agga4/./paranoia.elf -lm
...
```

To build examples for a specific BSP, set the BSPS make variable. For example:

```
$ make BSPS="gr712rc gr716"
```

Output files are generated under `src/examples/bin/<BSP>`. The different subdirectories reflect the compiler options used.

It is also possible to build a single example by calling `make <example>`, for example:

```
$ make CFLAGS="-Os -g" ambapp.elf
sparc-gaisler-elf-gcc -Os -g -std=c99 ambapp/ambapp.c -o ambapp.elf
```

The executables will be stored in the examples root directory in thie scase. When building individual examples it is possible to control the behaviour by setting the following variables.

`CFLAGS`
: Override common compilation flags

For more information on the examples and how to build them, see the file `src/examples/README`.

2.15.1. Target specific examples

Some of the examples in `src/examples/` are adapted for specific target systems or may need customization. These shall be built from inside the respective example directory, as indicated in `src/examples/README`.

2.16. Creating a bootable ROM images

The MKPROM2 PROM image generator can be used to create boot-images for applications compiled with BCC 2.1.1. An example is provided in the BCC binary distribution directory `src/examples/mkprom-hello`. MKPROM2 is distributed with source code and is available from the Cobham Gaisler website. For more information on how to use MKPROM2, see the *MKPROM2 User's Manual*.

3. LLVM based toolchain

3.1. Introduction

With BCC 2 an LLVM based version of the toolchain is provided along side the regular GCC based toolchain. The LLVM based toolchain is currently experimental.

The LLVM compiler framework is a relatively new and modern compiler framework. It has support for a wide variety of programming languages and architectures, including SPARC. The C-family front-end of LLVM, is called Clang. Clang is the main interface to the compiler, and the binary `sparc-gaisler-elf-clang` is used to compile C and C++ programs.

The Clang interface is similar to the GCC interface, and in many cases changing the build system to use LLVM/Clang is a matter of changing the `CC` variable in a `Makefile` script from `sparc-gaisler-elf-gcc` into `sparc-gaisler-elf-clang`.

The LLVM toolchain has its own assembler which is used by default. It is also possible to switch to the GNU assembler by using a command line option. The Clang front-end has been setup to automatically use the GNU linker in a similar way to the GCC version of BCC.

All the correct libraries and header files will be used by the Clang front-end. These are selected based on the flags set by the compiler. The libraries include newlib, libbcc and libgcc. A list of recommended command line option for Clang can be found in Appendix B.

Installation, host requirements and contents of the LLVM based toolchain follows the information presented in Chapter 1. Usage instructions follows the information presented in Chapter 2.

3.2. BCC LLVM/Clang tools

The following tools are included in the LLVM version of BCC. The tools are a combination of tools from the LLVM compiler framework, the Clang C-family LLVM compiler, and GNU binutils. The tools from binutils have names prefixed with `sparc-gaisler-elf`, except `sparc-gaisler-elf-clang`, `sparc-gaisler-elf-clang++` and `sparc-gaisler-elf-cpp` which comes from Clang.

`sparc-gaisler-elf-addr2line`	Convert address to C/C++ line number
`sparc-gaisler-elf-ar`	Library archiver
`sparc-gaisler-elf-as`	GNU Cross-assembler
`sparc-gaisler-elf-c++filt`	GNU utility to demangle C++ symbols
`sparc-gaisler-elf-clang`	LLVM C language family cross compiler for SPARC
`sparc-gaisler-elf-clang++`	LLVM C++ language family cross compiler for SPARC
`sparc-gaisler-elf-cpp`	LLVM C preprocessor
`sparc-gaisler-elf-gdb`	GNU GDB C/C++ level Debugger
`sparc-gaisler-elf-gdb-6.8`	GNU GDB C/C++ level Debugger
`sparc-gaisler-elf-gprof`	GNU profiling utility
`sparc-gaisler-elf-ld`	GNU linker
`sparc-gaisler-elf-nm`	GNU utility to print symbol table
`sparc-gaisler-elf-objcopy`	GNU utility to convert between binary formats
`sparc-gaisler-elf-objdump`	GNU utility to dump various parts of executables
`sparc-gaisler-elf-ranlib`	GNU library sorter
`sparc-gaisler-elf-readelf`	GNU ELF file information utility
`sparc-gaisler-elf-size`	GNU utility to display segment sizes
`sparc-gaisler-elf-strings`	GNU utility to dump strings from executables
`sparc-gaisler-elf-strip`	GNU utility to remove symbol table

4. C standard library

BCC includes newlib 2.5.0 which is an implementation of the C standard library with full math support. Low-level functionality required by newlib is implemented in the SPARC LEON specific layer (`libbcc`).

Documentation for the newlib C library and math library is available as described in Section 1.5 Source code for newlib can be obtained as described in Section 1.6.

Most of the functionality defined by the C standard library is supported by BCC. This chapter will describe deviations and specific properties of the C library when executing on LEON systems.

4.1. File I/O

BCC newlib supports file I/O on the standard input, standard output and standard error files (`stdin`/`stdout`/`stderr`). These files are always open and are typically associated with the BCC console device driver (see Section 5.2).

NOTE: There is no support in BCC for operating on disk files. There is no file system support.

4.2. Time functions

LEON timers are used to generate the system time. The C standard library functions `time()` and `clock()` return the time elapsed in seconds and microseconds respectively. `times()` and `gettimeofday()`, defined by POSIX, are also available. The user can control how the time functions use the hardware timers as described in Section 5.3.

4.3. Dynamic memory allocation

Dynamic memory can be allocated/deallocated using for example `malloc()`, `calloc()` and `free()`. For information on customizing the memory heap, see Section 8.8.

4.4. Atomic types and operations

BCC is based on GCC version 7.2.0 which includes C11 atomic types and operations. This allows for synchronization between applications in AMP environments. Synchronization instructions such as `ldstub`, `swap` `casa`, etc. are generated by the compiler.

The C11 atomic interface is defined by `stdatomic.h`. Some of the atomic operations defined by `stdatomic.h` require hardware support not available on all LEON systems. The `ldstub` and `swap` instructions are available in all LEON processors, while `casa` is optional. All multi-core LEON based components from Cobham Gaisler have `casa`. The GCC option `-mcpu=leon3` is required for full `stdatomic.h` support.

See ISO/IEC 9899:2011 for more information on the C11 standard.

NOTE: While atomic instructions are useful for sharing memory between processors and tasks, the atomic instructions shall never be used for manipulating peripheral control registers.

4.5. Newlib nano

The nano version of newlib, selected with `-qnano`, is a compiled with options to reduce code foot print. `-qnano` has the following limitations:

- Formatted I/O lacks floating-point support. It can however be enabled as described in `newlib/newlib/README`.
- Formatted I/O lacks support for `long long`.
- Formatted I/O does not support features from the outside of C89 standard.
- Multi-byte characters are not supported.

NOTE: The option `-qnano` shall be specified both when compiling and linking.

If floating point formatted I/O is needed when using `-qnano`, then the option `-u _printf_float` can be added to the linker command line. For example via the front-end option `-Wl,-u,_printf_float`.

5. BCC library

BCC is delivered with a library, `libbcc`, containing functions for programming LEON systems. This chapter is the user documentation for the API. Later chapters will describe how the BCC run-time can be configured and customized at link time.

The library is available in the target library file `libbcc.a`. There are multiple versions of `libbcc.a`, customized for specific BSPs and compiler options (GCC multilibs). The exact versions of the library is selected based on compiler command line parameters. This also reflects that different low-level drivers are implemented for different hardware.

5.1. Usage

Functions described in this chapter have prototypes in the header file `bcc/bcc.h`. The functions are implemented in `libbcc.a` and are available per default when linking with the GCC front-end. The same user API is available independent of target LEON hardware.

5.2. Console API

The console API does not have any user functions. It can be accessed with the C standard library I/O functions (Section 4.1).

5.3. Timer API

The function `bcc_timer_get_us()` can be used to determine system time in microseconds.

Table 5.1. `bcc_timer_get_us` function declaration

Proto	`uint32_t bcc_timer_get_us(void)`
About	Get processor time
Return	uint32_t. Number of microseconds since system start.

Other time related functions which depend on the BCC run time, but are not part of the BCC user library, are available. This includes `clock()`, `time()`, `times()` and `gettimeofday()`.

5.3.1. Interrupt based timer service

By default BCC does not install any timer tick and can result in limited services provided by the C library time functions and `bcc_timer_get_us()`. The typical limitation is that time will seem to restart or stop at some point in time, due to hardware timer expiration. Exact limitations are target hardware dependent, but is typically manifested as a time wrap 2^{32} microseconds after system reset.

To overcome this limitation, a timer tick service can be enabled by calling `bcc_timer_tick_init_period()`. It will install a tick interrupt handler which is triggered periodically to maintain time integrity, ensuring that time increments. Tick period is 10 milliseconds by default.

`bcc_timer_tick_init_period()` should be called only once and at the beginning of the program. It is recommended to call it from the `__bcc_init70()` initialization hook, described in described in Section 8.6.

Table 5.2. `bcc_timer_tick_init_period` function declaration

Proto	`int bcc_timer_tick_init_period(uint32_t usec_per_tick)`
About	Enable interrupt based timer service. The function installs a tick interrupt handler which maintains local time using timer hardware. This makes C library / POSIX time functions not limited to hardware constraints anymore. The function assumes that the timer (global) scaler is set to 1 MHz.
Param	*`usec_per_tick`* [IN] Integer Requested timer tick period: number of microseconds per tick.

Return	int.	
	Value	Description
	BCC_OK	Success
	BCC_FAIL	Failed to enable interrupt based timer service, or already enabled
	BCC_NOT_AVAILABLE	Hardware or resource not available
Notes	Epoch changes to the point in time when `bcc_timer_tick_init_period()` is called.	

5.4. Cache control API

The cache control API is used to flush the local LEON processor instruction and data caches.

Functions are also provided for operating directly on the LEON cache control register (`CCR`). Bit definitions for `CCR` are available in `bcc/leon.h`.

Table 5.3. `bcc_flush_cache` function declaration

Proto	`void bcc_flush_cache(void)`
About	Flush L1 instruction and data cache.
Return	None.

Table 5.4. `bcc_flush_icache` function declaration

Proto	`void bcc_flush_icache(void)`
About	Flush L1 instruction cache.
Return	None.

Table 5.5. `bcc_flush_dcache` function declaration

Proto	`void bcc_flush_dcache(void)`
About	Flush L1 data cache.
Return	None.

Table 5.6. `bcc_set_ccr` function declaration

Proto	`void bcc_set_ccr(uint32_t data)`
About	Set Cache Control Register (`CCR`).
Param	*`data`* [IN] Integer New `CCR` value to set.
Return	None.

Table 5.7. `bcc_get_ccr` function declaration

Proto	`uint32_t bcc_get_ccr(void)`
About	Get value of Cache Control Register (`CCR`).
Return	uint32_t. `CCR`.

5.5. Bus access API

Functions are provided for loading data from memory with forced L1 cache miss.

Table 5.8. `bcc_loadnocache` function declaration

Proto	`uint32_t bcc_loadnocache(uint32_t *addr)`
About	Load 32-bit word from *`addr`* with forced cache miss.

Param	*addr* [IN] Pointer Address to load from.
Return	uint32_t. Data loaded from *addr*.

Table 5.9. bcc_loadnocache16 function declaration

Proto	`uint16_t bcc_loadnocache16(uint16_t *addr)`
About	Load 16-bit word from *addr* with forced cache miss.
Param	*addr* [IN] Pointer Address to load from.
Return	uint16_t. Data loaded from *addr*.

Table 5.10. bcc_loadnocache8 function declaration

Proto	`uint8_t bcc_loadnocache8(uint8_t *addr)`
About	Load 8-bit word from *addr* with forced cache miss.
Param	*addr* [IN] Pointer Address to load from.
Return	uint8_t. Data loaded from *addr*.

The function `bcc_dwzero()` can be used to clear a memory region using 64-bit writes with the `std` instruction.

Table 5.11. bcc_dwzero function declaration

Proto	`void bcc_dwzero(uint64_t *dst, size_t n)`
About	Set 64-bit words to zero This function sets *n* 64-bit words to zero, starting at address *dst*. All writes are performed with the SPARC V8 `std` instruction.
Param	*dst* [IN] Pointer Start address of area to set to zero. Must be aligned to a 64-bit word.
Param	*n* [IN] Integer Number of 64-bit words to set to zero.
Return	None.

5.6. IU control/status register access API

This API provides access to low-level SPARC control/status registers and controls power-down mode.

5.6.1. Processor State Register

The Processor State Register (PSR) can be read with `bcc_get_psr()` and written with `bcc_set_psr()`. Processor Interrupt Level (`PSR.PIL`) is read using `bcc_get_pil()`. `PSR.PIL` can be set with `bcc_set_pil()` which is implemented as a software trap and guarantees atomic update.

NOTE: Care must be taken when manipulating PSR using read-modify-write sequences, since the operations are interruptible. See *The SPARC Architecture Manual Version 8, section B.29*.

NOTE: It is recommended to use the safe functions described in Section 5.9.1 for manipulating `PSR.PIL`.

Table 5.12. bcc_get_psr function declaration

Proto	`uint32_t bcc_get_psr(void)`

About	Get value of Processor State Register (PSR).
Return	uint32_t. PSR.

Table 5.13. `bcc_set_psr` function declaration

Proto	`void bcc_set_psr(uint32_t psr)`
About	Set Processor State Register (PSR).
Param	*`psr`* [IN] Integer New PSR value to set.
Return	None.

Table 5.14. `bcc_get_pil` function declaration

Proto	`int bcc_get_pil(void)`
About	Get Processor Interrupt Level (`PSR.PIL`).
Return	int. Value of `PSR.PIL` (0..15) in bits 3..0.

Table 5.15. `bcc_set_pil` function declaration

Proto	`int bcc_set_pil(int newpil)`
About	Set Processor Interrupt Level atomically. This function is implemented as a software trap and guarantees atomic update of `PSR.PIL`.
Param	*`newpil`* [IN] Integer New value for `PSR.PIL` (0..15) in bits 3..0.
Return	int. Old value of `PSR.PIL` (0..15) in bits 3..0.

5.6.2. Trap Base Register

The Trap Base Register (TBR) can be read with `bcc_get_tbr()` and written with `bcc_set_tbr()`.

Table 5.16. `bcc_get_tbr` function declaration

Proto	`uint32_t bcc_get_tbr(void)`
About	Get value of Trap Base Register (TBR).
Return	uint32_t. TBR.

Table 5.17. `bcc_set_tbr` function declaration

Proto	`void bcc_set_tbr(uint32_t tbr)`
About	Set Trap Base Register (TBR).
Param	*`tbr`* [IN] Integer New TBR value to set.
Return	None.

To retrieve only the Trap Base Address (`TBR.TBA`) of TBR, the function `bcc_get_trapbase()` can be used.

Table 5.18. `bcc_get_trapbase` function declaration

Proto	`uint32_t bcc_get_trapbase(void)`
About	Get Trap Base Address (`TBR.TBA`).
Return	uint32_t. `TBR.TBA` in bits (31..12).

5.6.3. Processor power-down

The current processor is powered down by calling `bcc_power_down()`.

Table 5.19. `bcc_power_down` function declaration

Proto	`int bcc_power_down(void)`
About	Power down current processor.
Return	int. BCC_OK

5.7. FPU context API

`bcc_fpu_save()` is used to save the current state of the floating-point registers `%f0` to `%f31` and the `%fsr` register to a user-specified location. `bcc_fpu_restore()` restores an FPU context previously saved by the user. Storage for the FPU context struct bcc_fpu_state shall be allocated by the user and provided to these functions. The floating-point deferred-trap queue (`%fq`) is emptied before saving and restoring the FPU context.

These functions can be used in an interrupt service routine which performs floating-point operations.

Table 5.20. `bcc_fpu_save` function declaration

Proto	`int bcc_fpu_save(struct bcc_fpu_state *state)`
About	Save floating-point context The context shall be restored with `bcc_fpu_restore()`.
Param	*`state`* [IN] Pointer Location to save FPU context. This shall be a pointer to a preallocated struct bcc_fpu_state, aligned to 8 byte.
Return	int. BCC_OK on success

Table 5.21. `bcc_fpu_restore` function declaration

Proto	`int bcc_fpu_restore(struct bcc_fpu_state *state)`
About	Restore floating-point context The context *`state`* is FPU state previously saved with `bcc_fpu_save()`.
Param	*`state`* [IN] Pointer Location to restore FPU context from. This shall be a pointer to a preallocated struct bcc_fpu_state, aligned to 8 byte.
Return	int. BCC_OK on success

5.8. Trap API

Modifying the SPARC trap table is done using the BCC trap API. An entry can be inserted in the current trap table with `bcc_set_trap()` described in Table 5.22. The function supports both the standard SPARC trap mechanism and SPARC-V8E single vector trapping (SVT as enabled with the `-qsvt` linker option).

NOTE: After manipulating a trap table, the instruction cache may need a flush (see Section 5.4).

Below is an example on how the `window_overflow` (0x05) trap handler can be replaced with the user provided trap handler called `mynewhandler`:

```
#include <bcc/bcc.h>

extern void mynewhandler(void);
const int TT_WINDOW_OVERFLOW = 0x05;

int set_trap_example(void)
```

```
{
        int ret

        ret = bcc_set_trap(TT_WINDOW_OVERFLOW, mynewhandler);
        return ret;
}
```

Table 5.22. `bcc_set_trap` function declaration

Proto	`int bcc_set_trap(int tt, void (*handler)(void))`
About	Install trap table entry. When this function returns successfully, the current trap table has been updated such that when the trap occurs: • Execution jumps to *`handler`*. • `%l0` contains `%psr`. • `%l1` contains trapped `%pc`. • `%l2` contains trapped `%npc`. • `%l6` (0..255) contains same value as *`tt`* to `bcc_set_trap()`. The trap handler is typically written in assembly and must preserve any state it changes. It shall end with the `rett` instruction. This function operates on the current table. It supports multi vector trapping (MVT) and single vector trapping (SVT).
Param	*`tt`* [IN] Integer Trap type (0..255)
Param	*`handler`* [IN] Pointer Trap handler
Return	int.

Value	Description
BCC_OK	Success
BCC_FAIL	Trap table entry installation failed

Notes	`bcc_set_trap()` does not flush the CPU instruction cache.

5.8.1. Single vector trapping (SVT)

This section describes steps which may be required when installing custom trap handlers under the SVT trap mechanism available in some LEON systems. For the specification of SVT, see *SPARC-V8 Supplement, SPARC-V8 Embedded (V8E) Architecture Specification*. SVT is typically used in systems with small memory footprint.

The BCC approach to SVT is to look up the target trap handler routine in two levels of tables. The level 0 table contains 16 entries, each pointing to a level 1 table. A level 1 table consists of 16 entries with the location of the target trap handler routine. At trap time, `TBR.TT[7:4]` indexes into table level 0 and `TBR.TT[3:0]` indexes into table level 1. Most of the level 1 tables entries are bad trap handlers so level 1 tables can be reused to save storage.

NOTE: The BCC SVT table lookup routine executes a fixed number of instructions, independent of target trap number and independent of installed handlers.

BCC run time defines 4 of the maximum 16 level 1 tables per default when the application is linked with `-qsvt`, as illustrated in Table 5.23.

Table 5.23. Default SVT level 1 tables

Symbol name	Default trap number assignments
`__bcc_trap_table_svt_0`	`0x00..0x0F` (system trap handlers and some *bad trap* handlers)

Symbol name	Default trap number assignments
__bcc_trap_table_svt_1	0x10..0x1F (interrupt traps 1..15)
__bcc_trap_table_svt_8	0x80..0x8F (software trap 0..15)
__bcc_trap_table_svt_allbad	all other. This table contains 16 pointers to the symbol __bcc_trap_table_svt_bad which is a default handler for unexpected traps.

The single default level 0 table has symbol name __bcc_trap_table_svt_level0 and contains 16 pointers to __bcc_trap_table_svt_[0..f]. Symbols __bcc_trap_table_svt_{2,3,4,5,6,7,9,a,b,c,d,e,f} all have the same value as __bcc_trap_table_svt_allbad per default. The level 1 tables with index 0, 1 and 8 have default values according to Table 5.23.

bcc_set_trap() can be used directly on trap numbers in the ranges 0x00..0x1F and 0x80..0x8F. All other trap numbers are redirected to the common __bcc_trap_table_svt_allbad table which is never manipulated by bcc_set_trap().

It is however possible for the user to construct custom level 1 lookup tables by defining symbols with the names __bcc_trap_table_svt_x, where x is an integer value between 0 and f. The linker will pick up any the level 1 table named like this and use it instead of the *all bad* table. This is possible because all of __bcc_trap_table_svt_x are defined as weak symbols.

The following example defines a level 1 table containing one trap handler, my_trap_handler92 for tt=0x92, at link time. At run time, main() installs my_trap_handler93 as handler for tt=0x93 using bcc_set_trap(). A second call to bcc_set_trap() tries to install a handler for tt=0xa3 which will fail because the corresponding level 1 table is the default __bcc_trap_table_svt_allbad.

```
/*
 * Example for defining a custom level 1 SVT table and two trap handlers in the
 * [0x90:0x9F] range.
 *
 * NOTE: This example must linked with the -qsvt option.
 */
#include <stdio.h>
#include <bcc/bcc.h>

/* User trap handlers implemented elsewhere */
extern uint32_t my_trap_handler92;
extern uint32_t my_trap_handler93;

/* Default handler for unexpected traps */
extern uint32_t __bcc_trap_table_svt_bad;

/* Override weak symbol __bcc_trap_table_svt_9 */
uint32_t *__bcc_trap_table_svt_9[16] = {
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &my_trap_handler92,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad,
        &__bcc_trap_table_svt_bad
};

int main(void)
{
        int ret;

        ret = bcc_set_trap(0x93, &my_trap_handler93);
        printf("ret=%d (expecting 0)\n", ret);

        ret = bcc_set_trap(0xa3, &my_trap_handler93);
```

```
        printf("ret=%d (expecting non-zero)\n", ret);

        return 0;
}
```

5.9. Interrupt API

The interrupt API allows for enabling and disabling interrupt sources, interrupt remapping, attaching interrupt service routines and control of interrupt nesting.

5.9.1. Interrupt disable and enable

All maskable interrupts are disabled with `bcc_int_disable()` and enabled again with `bcc_int_enable()`. A nesting mechanism allows multiple disable operations to be performed in sequence without the corresponding enable operation inbetween. These functions provide safe manipulation of the SPARC V8 `PSR.PIL` registers. The interrupt controller is unmodified by these functions.

An integer variable is associated with every disable/enable pair which records state of the interrupt state to return to. The state is returned by `bcc_int_disable` and taken as parameter by `bcc_int_enable`. In order for the system to properly restore interrupt enable/disable state, the usage of state variables at interrupt enable operations must be in opposite order of the disable operation.

Interrupts are in the enabled state when `main()` is called.

The example below illustrates how interrupt disable operations can nest.

```
#include <bcc/bcc.h>
int int_nest_example(void)
{
        int lev0, lev1;

        /* Enter critical region 0. */
        lev0 = bcc_int_disable();
        ...
        /* Enter critical region 1A. */
        lev1 = bcc_int_disable();
        ...
        /* Leave critical region 1A.
        bcc_int_enable(lev1);
        ...
        /* Enter critical region 1B. */
        lev1 = bcc_int_disable();
        ...
        /* Leave critical region 1B.
        bcc_int_enable(lev1);
        ...
        /* Leave critical region 0. */
        bcc_int_enable(lev0);

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

Table 5.24. `bcc_int_disable` function declaration

Proto	`int bcc_int_disable(void)`
About	Disable all maskable interrupts and return the previous interrupt enable/disable state A matching `bcc_int_enable()` with the return value as parameter must be called to exit the interrupt disabled state. It is allowed to do nested calls to `bcc_int_disable()`, and if so the same number of `bcc_int_enable()` must be called. This function modifies the SPARC V8 PSR.PIL field. Interrupt controller is not touched.
Return	int. Previous interrupt level (used when calling bcc_int_enable().

Table 5.25. `bcc_int_enable` function declaration

Proto	`void bcc_int_enable(int plevel)`
About	Return to a previous interrupt enable/disable state

	The *plevel* parameter is the return value from a previous call to bcc_int_disable(). At return, interrupts may be enabled or disabled depending on *plevel*. This function modifies the SPARC V8 PSR.PIL field. Interrupt controller is not touched.
Param	*plevel* [IN] Integer The interrupt protection level to set. Must be the return value from the most recent call to bcc_int_disable().
Return	None.

5.9.2. Interrupt source masking

An interrupt source can be masked (disabled) with bcc_int_mask() and unmasked (enabled) with bcc_int_unmask(). Interrupt source masking is local to the issuing processor.

Table 5.26. bcc_int_mask function declaration

Proto	int bcc_int_mask(int source)	
About	Mask (disable) an interrupt source on the current CPU.	
Param	*source* [IN] Integer SPARC interrupt number 1-15 or extended interrupt number 16-31.	
Return	int.	
	Value	Description
	BCC_OK	Success
	BCC_NOT_AVAILABLE	Device not available

Table 5.27. bcc_int_unmask function declaration

Proto	int bcc_int_unmask(int source)	
About	Unmask (enable) an interrupt source on the current CPU.	
Param	*source* [IN] Integer SPARC interrupt number 1-15 or extended interrupt number 16-31.	
Return	int.	
	Value	Description
	BCC_OK	Success
	BCC_NOT_AVAILABLE	Device not available

5.9.3. Clear and force interrupt

Clearing an interrupt source is done with bcc_int_clear(). A SPARC interrupt level can be forced on the local processor with bcc_int_force(). An interrupt source (including extended interrupt) can be globally pended with bcc_int_pend().

Table 5.28. bcc_int_clear function declaration

Proto	int bcc_int_clear(int source)	
About	Clear an interrupt source.	
Param	*source* [IN] Integer SPARC interrupt number 1-15 or extended interrupt number 16-31.	
Return	int.	
	Value	Description
	BCC_OK	Success

	BCC_NOT_AVAILABLE	Device not available

Table 5.29. `bcc_int_force` function declaration

Proto	`int bcc_int_force(int level)`	
About	Force an interrupt *`level`* on the current processor.	
Param	*`level`* [IN] Integer SPARC interrupt request level 1..15.	
Return	int.	
	Value	Description
	BCC_OK	Success.
	BCC_NOT_AVAILABLE	Device not available.
Notes	Extended interrupts can not be forced with this function.	

Table 5.30. `bcc_int_pend` function declaration

Proto	`int bcc_int_pend(int source)`	
About	Make an interrupt source pending.	
Param	*`source`* [IN] Integer SPARC interrupt number 1-15 or extended interrupt number 16-31.	
Return	int.	
	Value	Description
	BCC_OK	Success
	BCC_NOT_AVAILABLE	Device not available

5.9.4. Interrupt remap

The IRQ(A)MP interrupt controller can optionally be implemented with functionality to allow dynamic remapping between AMBA bus interrupt lines (0..63) and interrupt controller interrupt lines (1..31). This functionality can be programmed with `bcc_int_map_set()` and `bcc_int_map_get()`.

NOTE: Interrupt remapping functionality requires hardware support available in for example GR740 and GR716.

Table 5.31. `bcc_int_map_set` function declaration

Proto	`int bcc_int_map_set(int busintline, int irqmpintline)`	
About	Set mapping from bus interrupt line to an interrupt controller interrupt line.	
Param	*`busintline`* [IN] Integer Bus interrupt line number	
Param	*`irqmpintline`* [IN] Integer Interrupt controller interrupt line	
Return	int.	
	Value	Description
	BCC_OK	Success
	BCC_NOT_AVAILABLE	Device or functionality not available

Table 5.32. `bcc_int_map_get` function declaration

Proto	`int bcc_int_map_get(int busintline)`
About	Get mapping from bus interrupt line to an interrupt controller interrupt line.

Param	*busintline* [IN] Integer Bus interrupt line number	
Return	int.	
	Value	Description
	1..31	Interrupt controller interrupt line (1..31)
	-1	Device or functionality not available

5.9.5. Interrupt service routines

BCC interrupt service routines (ISR) are convenient because they allow the user to specify C functions which are called in response to an interrupt. The API handles extended interrupts transparently.

This part of the interrupt API is a higher level mechanism compared to the trap API. Section 5.9.7 describes how the BCC trap API can be used to install low-level interrupt handlers.

Functions are provided for the user to install custom interrupt service routines. SPARC interrupts 1-15 and extended interrupts 16-31 are supported. It is possible to install multiple interrupt handlers for the same interrupt: this is referred to as *interrupt sharing*. All ISR handler dispatching is hidden from the user.

NOTE: It is not allowed to call the interrupt service routine register/unregister functions from inside an interrupt handler.

Two sets of functions are available for registering and unregistering interrupt service routines. They differ in memory allocation responsibility. Some memory is always needed when installing an ISR with the API described in this section.

5.9.5.1. Automatic memory management

`bcc_isr_register()` and `bcc_isr_unregister()` manage memory allocation automatically by using `malloc()` and `free()` internally.

Table 5.33. `bcc_isr_register` function declaration

Proto	`void *bcc_isr_register(int source, void (*handler)(void *arg, int source), void *arg)`
About	Register interrupt handler The function in parameter *handler* is registered as an interrupt handler for the given interrupt source. The handler is called with arg and source as arguments. Interrupt *source* is not enabled by this function. `bcc_int_unmask()` can be used to enable it. Multiple interrupt handlers can be registered for the same interrupt number. They are dispatched at interrupt in the same order as registered. A handler registered with this function should be unregistered with `bcc_isr_unregister()`.
Param	*source* [IN] Integer SPARC interrupt number 1-15 or extended interrupt number 16-31.
Param	*handler* [IN] Pointer Pointer to software routine to execute when the interrupt triggers.
Param	*arg* [IN] Pointer Passed as first argument to *handler*.
Return	Pointer. Status and ISR handler context
	Value / Description

	NULL	Indicates failed to install handler.
	Pointer	Pointer to ISR handler context. Should not be dereferenced by user. Used as input to `bcc_isr_unregister()`.
Notes	This function may call `malloc()`.	

Table 5.34. `bcc_isr_unregister` function declaration

Proto	`int bcc_isr_unregister(void *isr_ctx)`	
About	Unregister interrupt handler It is only allowed to unregister an interrupt handler which has previously been registered with `bcc_isr_register()`. Interrupt *`source`* is not disabled by this function. The function `bcc_int_mask()` can be used to disable it.	
Param	*`isr_ctx`* [IN] Pointer ISR handler context returned `bcc_isr_register()`.	
Return	int.	
	Value	Description
	BCC_OK	Handler successfully unregistered.
	BCC_FAIL	Failed to unregister handler.
Notes	This function may call `free()`	

Following is an example on how `bcc_isr_register()` and `bcc_isr_unregister()` can be used to install two interrupt handlers on different interrupt numbers sharing the same function but with different ISR unique data. `malloc()` and `free()` are called by the BCC library.

```
#include <bcc/bcc.h>

/* User interrupt handler */
extern void myhandler(void *arg, int source);
/* ISR uniuqe data */
extern int arg0, arg1;

static const int INTNUMA = 2;
static const int INTNUMB = 3;

int isr_reg_example(void)
{
        int ret;
        /* ISR handler contexts for using the bcc_isr_ API. */
        void *ictx0, *ictx1;

        ictx0 = bcc_isr_register(INTNUMA, myhandler, &arg0);
        if (NULL == ictx0) {
                return BCC_FAIL;
        }
        ictx1 = bcc_isr_register(INTNUMB, myhandler, &arg1);
        if (NULL == ictx1) {
                bcc_isr_unregister(ictx0);
                return BCC_FAIL;
        }
        bcc_int_unmask(INTNUMA);
        bcc_int_unmask(INTNUMB);

        ...

        bcc_int_mask(INTNUMB);
        bcc_int_mask(INTNUMA);
        ret = bcc_isr_unregister(ictx0);
        if (BCC_OK != ret) {
                return ret; /* Failure */
        }
        ret = bcc_isr_unregister(ictx1);
        if (BCC_OK != ret) {
                return ret; /* Failure */
        }
```

```
        return ret;
}
```

5.9.5.2. User memory management

`bcc_isr_register_node()` and `bcc_isr_unregister_node()` are available for cases where the user want to control all memory allocations in the application. Associated with these two functions is a type named struct bcc_isr_node. An instance of such type (ISR node) should be allocated and initialized by the user and provided to `bcc_isr_register_node()`. Node structure data provided to `bcc_isr_register_node()` must not be touched or deallocated by the user until `bcc_isr_unregister_node()` has been called with the same node. After that, the user is free to reuse or deallocat the node. The ISR node must reside in writable memory.

```
struct bcc_isr_node {
        void *__private;
        int source;
        void (*handler)(
                void *arg,
                int source
        );
        void *arg;
};
```

Table 5.35. bcc_isr_node data structure declaration

source	Interrupt source number
handler	User ISR handler
arg	Passed as parameter to handler

Table 5.36. `bcc_isr_register_node` function declaration

Proto	`int bcc_isr_register_node(struct bcc_isr_node *isr_node)`
About	Register interrupt handler, non-allocating This function is similar to `bcc_isr_register()` with the difference that the user is responsible for memory management. It will never call `malloc()`. Instead the caller has to provide a pointer to a preallocated and initialized ISR node of type struct bcc_isr_node. The memory pointed to by *`isr_node`* shall be considered owned exclusively by the run-time between the call to `bcc_isr_register_node()` and a future `bcc_isr_unregister_node()`. It means that the memory must be available for this time and must not be modified by the application. The memory pointed to by *`isr_node`* must be writable. This function should be used to install interrupt handlers in applications which want full control over memory allocation.
Param	*`isr_node`* [IN] Pointer Pointer to User initialized ISR node. The fields *`source`*, *`handler`* and optionally the *`arg`* shall be initialized by the caller.
Return	int.

Value	Description
BCC_OK	Handler installed successfully.
BCC_FAIL	Failed to install handler.

Table 5.37. `bcc_isr_unregister_node` function declaration

Proto	`int bcc_isr_unregister_node(const struct bcc_isr_node *isr_node)`
About	Unregister interrupt handler, non-allocating This function is similar to `bcc_isr_unregister()` with the difference that the user is responsible for memory management. It is only allowed to unregister an interrupt handler which has previously been registered with `bcc_isr_register_node()`.

Param	*isr_node* [IN] Pointer Same as input parameter to `bcc_isr_register_node()`.
Return	int.

Value	Description
BCC_OK	Handler successfully unregistered.
BCC_FAIL	Failed to unregister handler.

Following is an example on how `bcc_isr_register_node()` and `bcc_isr_unregister_node()` can be used to install an interrupt handler on interrupt 3. No calls to `malloc()` or `free()` are performed.

```
#include <bcc/bcc.h>

/* User interrupt handler */
extern void myhandler(void *arg, int source);
/* ISR uniuqe data */
extern int arg0;

/* ISR node allocated by user */
struct bcc_isr_node inode0;

int isr_reg_example(void)
{
        int ret;
        inode0.source = 3;
        inode0.handler = myhandler;
        inode0.arg = &arg0;

        ret = bcc_isr_register_node(&inode0);
        if (BCC_OK != ret) {
                return ret;
        }
        bcc_int_unmask(3);

        ...

        bcc_int_mask(3);
        ret = bcc_isr_unregister_node(&inode0);

        return ret;
}
```

5.9.6. Interrupt nesting

Interrupt nesting can be enabled, disabled or set to a user custom config with the interrupt nesting API. This API maintains the SPARC `PSR.PIL` field. More fine-grained masking can be done by programming the interrupt controller as described in Section 5.9.2.

Interrupt nesting is *disabled* by default in BCC, meaning that an interrupt service routine can not be preempted by any other interrupt. The function `bcc_int_enable_nesting()` enables nesting such that an ISR can be preempted by higher level processor interrupts. `bcc_int_disable_nesting()` can be used to disable nesting again.

The function `bcc_int_nestcount()` returns the interrupt nest level, starting at 0 when the function is called outside of interrupt context.

NOTE: SPARC interrupt level 15 is non-maskable.

Table 5.38. `bcc_int_nestcount` function declaration

Proto	`int bcc_int_nestcount(void)`
About	Get current interrupt nest count
Return	int.

Value	Description
0	Caller is not in interrupt context

	1	Caller is in first interrupt context level
	n	Caller is in n:th interrupt context level

Table 5.39. `bcc_int_disable_nesting` function declaration

Proto	`int bcc_int_disable_nesting(void)`
About	Disable interrupt nesting After calling this function, `PSR.PIL` will be raised to 0xf (highest) when an interrupt occurs on any level.
Return	int. BCC_OK

Table 5.40. `bcc_int_enable_nesting` function declaration

Proto	`int bcc_int_enable_nesting(void)`
About	Enable interrupt nesting After calling this function, `PSR.PIL` will be raised to the current interrupt level when an interrupt occurs.
Return	int. BCC_OK

5.9.6.1. Advanced configuration

This subsection describes custom interrupt nesting configuration. It contains advanced information which is probably not needed for most application. Standard interrupt nesting control as described in Section 5.9.6 is assumed to cover most use cases.

When a user ISR which has been registered with `bcc_isr_register()` is triggered by hardware, the BCC interrupt dispatecher routine is executed as part oft he interrupt trap handling. The dispatcher sets (raises) the SPARC register `PSR.PIL` to a new interrupt request level before reenabling traps and calling the user ISR handler. The new `PSR.PIL` level is determined by the BCC interrupt dispatcher executed as part of the interrupt trap handling. BCC maintains a private table which maps for each interrupt level, a future (raised) interrupt level to set while the ISR executes.

`bcc_int_disable_nesting()` sets the mapping from each interrupt level (1..15) to the highest interrupt level (15). `bcc_int_enable_nesting()` sets the mapping from each interrupt level (1..15) to the same interrupt level (1..15).

A custom interrupt nesting mapping can be set with the function `bcc_int_set_nesting()`. It is for example possible to program either of interrupt levels 1..7 to always raise `PIL` to 7, making the corresponding service routines mutually exclusive, while still allowing interrupts on level 8 and above. For the purpose of the example, interrupt levels 8..15 could be mapped linearly to enable normal nesting on level 8 and above. This could be utilized to setup hardware supported task switching, where each task is related to a unique interrupt request level. The following example illustrates this setup.

```
#include <bcc/bcc.h>
/*
 * Processor interrupts 1..7  set PIL=7 to lock out interrupt 1..7.
 * Processor interrupts 8..15 nest as normal.
 */
void custom_nesting(void)
{
        bcc_enable_nesting();
        for (int i = 1; i <= 7; i++) {
            bcc_set_nesting(1, 7);
        }
}
```

Table 5.41. `bcc_int_set_nesting` function declaration

Proto	`int bcc_int_set_nesting(int pil, int newpil)`

About	Configure interrupt nesting Configures in detail how the SPARC processor interrupt level is set when an interrupt occurs. After calling this function, `PSR.PIL` will be raised to *`newpil`* when an interrupt occurs on level *`pil`*.
Param	*`pil`* [IN] Integer `PSR.PIL` (0..15) level to configure.
Param	*`newpil`* [IN] Integer New value for `PSR.PIL` (0..15) during interrupt at level *`pil`*. *`newpil`* must be equal to or greater than *`pil`* parameter.
Return	int.

Value	Description
BCC_OK	Success
BCC_FAIL	Illegal parameters

5.9.7. Low-level interrupt handlers

The trap API can be used to install low-level interrupt handlers for SPARC interrupts 1-15. It is done by calling `bcc_set_trap()` with the *`tt`* parameter set to interrupt number plus `0x10`. This will disable the normal BCC ISR management for this interrupt request level. Support for interrupt sharing on the CPU interrupt level is also on the responsibility of the user when using Low-level interrupt handlers.

NOTE: It is the implementers responsibility to ensure that volatile registers are saved and restored by the trap handler. The handler should set `PSR.PIL=0xf` to avoid interrupt nesting if traps are being enabled by the handler.

The following example illustrates how a low-level interrupt handler can be installed.

```
#include <bcc/bcc.h>

/* Function for installing low-level interrupt (trap) handler */
int set_lowlevel_int_handler(int source, void (*handler)(void))
{
        if (source < 1 || 15 < source) {
                return BCC_FAIL;
        }
        return bcc_set_trap(0x10 + source, handler);
}

extern void trap_handler_for_int1(void);
int isr_lowlevel_example(void)
{
        int ret;

        ret = set_lowlevel_int_handler(1, trap_handler_for_int1);
        printf("ret=%d\n", ret);

        return ret;
}
```

5.9.8. Interrupt timestamping

The IRQ(A)MP interrupt controller can be implemented with interrupt timestamping functinality. GR740 and GR716 support this. BCC provides an API for programming the timestamping hardware.

The function `bcc_timestamp_avail()` is used to determine the number of timestamp register sets known to BCC. The timestamping API described in this section can be used only if this function retuns a value of 1 or higher.

Most functions in the timestamping API take a parameter, *`ts_no`*, which describes the timestamping register set to operate on. This parameter must zero or greater, and must be strictly less than the return value of `bcc_timestamp_avail()`.

The function `bcc_timestamp_restart()` is used to start and restart monitoring of an interrupt line. `bcc_timestamp_status()` can be used to determine if timestamping registers for assertion and acknowledge

have been latched and the functions `bcc_timestamp_get_ass()` and `bcc_timestamp_get_ack()` can be used to read out the latched values.

`bcc_timestamp_get_cnt()` is used to read the current value of the free-running timestamping counter. It can be used by the user interrupt service routine to measure the time from hardware interrupt assertion to program response.

Table 5.42. `bcc_timestamp_avail` function declaration

Proto	`int bcc_timestamp_avail(void)`
About	Return number of timestamp register sets available
Return	int. Number of timestamp register sets available

Table 5.43. `bcc_timestamp_restart` function declaration

Proto	`int bcc_timestamp_restart(int ts_no, int source)`	
About	Restart timestamping	
Param	*`ts_no`* [IN] Integer Timestamp register set to use for this stamp	
Param	*`source`* [IN] Integer Interrupt controller interrupt line to monitor	
Return	int.	
	Value	Description
	BCC_OK	Success
	BCC_NOT_AVAILABLE	Device or functionality not available

Table 5.44. `bcc_timestamp_status` function declaration

Proto	`uint32_t bcc_timestamp_status(int ts_no)`	
About	Get timestamping status The function is used to determine whether *assertion*, *acknowledge* or both have been stamped for the timestamp register set.	
Param	*`ts_no`* [IN] Integer Timestamp register set	
Return	uint32_t. A combination of the masks BCC_TIMESTAMP_ASS and BCC_TIMESTAMP_ACK	
	Value	Description
	BCC_TIMESTAMP_ASS	Assertion Stamped (S1)
	BCC_TIMESTAMP_ACK	Acknowledge Stamped (S2)

Table 5.45. `bcc_timestamp_get_ass` function declaration

Proto	`uint32_t bcc_timestamp_get_ass(int ts_no)`
About	Return value of interrupt assertion timestamp register
Param	*`ts_no`* [IN] Integer Timestamp register set
Return	uint32_t. Value of interrupt assertion timestamp register

Table 5.46. `bcc_timestamp_get_ack` function declaration

Proto	`uint32_t bcc_timestamp_get_ack(int ts_no)`
About	Return value of interrupt acknowledge timestamp register

Param	*ts_no* [IN] Integer Timestamp register set
Return	uint32_t. Value of interrupt acknowledge timestamp register

Table 5.47. `bcc_timestamp_get_cnt` function declaration

Proto	`uint32_t bcc_timestamp_get_cnt(void)`
About	Return value of interrupt timestamp counter register
Return	uint32_t. Value of interrupt timestamp counter register

Below is a code snippet which demonstrates one way of using the timestamping API. `myisr()` is an interrupt service routine registered on the interrupt line which is monitored.

The sequence of events from interrupt assertion to ISR is:

1. Peripheral asserts interrupt line. Assertion timestamp register latches.
2. CPU starts processing the interrupt trap. Acknowledge timestamp register latches.
3. Execution enters `myisr()`. Program (CPU) reads timestamp counter register.

```
#include <bcc/timestamp.h>

static volatile uint32_t mycnt;

void myisr(void *arg, int source)
{
        mycnt = bcc_timestamp_get_cnt();
        ...
}

int timestamp_example(int stampnum, int intnum)
{
        uint32_t ass, ack, cnt;
        uint32_t v;

        assert(stampnum < bcc_timestamp_avail());

        bcc_timestamp_restart(stampnum, intnum);
        /* Wait for timestamp ack to latch */
        do {
                v = bcc_timestamp_status(stampnum);
        } while (0 == (v & BCC_TIMESTAMP_ACK));

        /* Collect the numbers */
        ass = bcc_timestamp_get_ass(stampnum);
        ack = bcc_timestamp_get_ack(stampnum);
        cnt = mycnt;

        /* Do interesting stuff with ass, ack, cnt */
        ...

        return 0;
}
```

5.10. Asymmetric Multiprocessing API

This API provides basic functionality for programming AMP systems. The communication primitive is inter-processor interrupts, which can be used as a basis for shared memories and higher level services. Functions in this API typically operate using a LEON interrupt controller such as `IRQMP` or `IRQ(A)MP`.

NOTE: The functions in the AMP API are available even when running on a single-processor system. AMP services are not served in this case, but the function return values are guaranteed to be consistent (typically returning with status `BCC_NOT_AVAILABLE`).

5.10.1. Processor identification

The number of processors in the system can be retrieved with the function `bcc_get_cpu_count()` and the ID of the current processor is retrieved with `bcc_get_cpuid()`

Table 5.48. `bcc_get_cpu_count` function declaration

Proto	`int bcc_get_cpu_count(void)`
About	Get number of processor in the system.
Return	int. Number of processors in the system or -1 if unknown. 1 is returned on single-processor systems.
Notes	This function will return -1 if the run-time is not aware of the interrupt controller.

Table 5.49. `bcc_get_cpuid` function declaration

Proto	`int bcc_get_cpuid(void)`
About	Get ID of the current processor. The first processor in the system has ID 0.
Return	int. ID of the current processor. 0 is returned on single-processor systems.

5.10.2. Inter-processor control

Another processor in a multiprocessor LEON system can be started by calling `bcc_start_processor()`. Inter-processor interrupts (IPI) are sent to other processors with `bcc_send_interrupt()`.

Table 5.50. `bcc_start_processor` function declaration

Proto	`int bcc_start_processor(int cpuid)`	
About	Start a processor.	
Param	*`cpuid`* [IN] Integer The processor to start. *`cpuid`* must be in the interval from 0 to `get_cpu_count()`-1.	
Return	int.	
	Value	Description
	BCC_OK	Success.
	BCC_NOT_AVAILABLE	Processor or device not available.

Table 5.51. `bcc_send_interrupt` function declaration

Proto	`int bcc_send_interrupt(int level, int cpuid)`	
About	Force an interrupt level on a processor.	
Param	*`level`* [IN] Integer Interrupt request level (1..15).	
Param	*`cpuid`* [IN] Integer The processor to interrupt. *`cpuid`* must be in the interval from 0 to `get_cpu_count()`-1.	
Return	int.	
	Value	Description

	BCC_OK	Success.
	BCC_NOT_AVAILABLE	Processor or device not available.

5.11. Default trap handlers

Table 5.52 lists the trap handlers linked into the SPARC trap table by default in a BCC application. Individual trap handlers can be added or replaced with the trap API described in Section 5.8.

See the *SPARC V8 specification* for trap definitions.

Table 5.52. Default trap handlers for BCC 2.1.1

tt	**Description**
`0x00`	Reset. Handled by `__bcc_trap_reset_mvt` or `__bcc_trap_reset_svt`.
`0x05`	Window overflow. Handled by `__bcc_trap_window_overflow`.
`0x06`	Window underflow. Handled by `__bcc_trap_window_underflow`.
`0x11..0x1f`	Interrupt. Handled by `__bcc_trap_interrupt`.
`0x83`	Flush windows. Handled by `__bcc_trap_flush_windows`.
`0x89`	Set `PSR.PIL`. Handled by `__bcc_trap_sw_set_pil`.
others	Force processor into error mode.

5.12. API reference

This section lists all BCC library user API functions with references to the related section(s). The API is also documented in the source header files of the library, see Section 5.1.

Table 5.53. BCC library user API structure reference

Type	**Section**
struct bcc_isr_node	5.9.5.2

Table 5.54. BCC library user API function reference

Prototype	**Section**
`uint32_t bcc_timer_get_us(void)`	5.3
`int bcc_timer_tick_init_period(uint32_t usec_per_tick)`	5.3.1
`void bcc_flush_cache(void)`	5.4
`void bcc_flush_icache(void)`	5.4
`void bcc_flush_dcache(void)`	5.4
`void bcc_set_ccr(uint32_t data)`	5.4
`uint32_t bcc_get_ccr(void)`	5.4
`uint32_t bcc_loadnocache(uint32_t *addr)`	5.5
`uint16_t bcc_loadnocache16(uint16_t *addr)`	5.5
`uint8_t bcc_loadnocache8(uint8_t *addr)`	5.5
`void bcc_dwzero(uint64_t *dst, size_t n)`	5.5
`uint32_t bcc_get_psr(void)`	5.6.1
`void bcc_set_psr(uint32_t psr)`	5.6.1
`int bcc_get_pil(void)`	5.6.1
`int bcc_set_pil(int newpil)`	5.6.1
`uint32_t bcc_get_tbr(void)`	5.6.2

Prototype	Section
`void bcc_set_tbr(uint32_t tbr)`	5.6.2
`uint32_t bcc_get_trapbase(void)`	5.6.2
`int bcc_power_down(void)`	5.6.3
`int bcc_fpu_save(struct bcc_fpu_state *state)`	5.7
`int bcc_fpu_restore(struct bcc_fpu_state *state)`	5.7
`int bcc_set_trap(int tt, void (*handler)(void))`	5.8, 5.9.7
`int bcc_int_disable(void)`	5.9.1
`void bcc_int_enable(int plevel)`	5.9.1
`int bcc_int_mask(int source)`	5.9.2
`int bcc_int_unmask(int source)`	5.9.2
`int bcc_int_clear(int source)`	5.9.3
`int bcc_int_force(int level)`	5.9.3
`int bcc_int_pend(int source)`	5.9.3
`int bcc_int_map_set(int busintline, int irqmpintline)`	5.9.4
`int bcc_int_map_get(int busintline)`	5.9.4
`void *bcc_isr_register(int source, void (*handler)(void *arg, int source), void *arg)`	5.9.5.1
`int bcc_isr_unregister(void *isr_ctx)`	5.9.5.1
`int bcc_isr_register_node(struct bcc_isr_node *isr_node)`	5.9.5.2
`int bcc_isr_unregister_node(const struct bcc_isr_node *isr_node)`	5.9.5.2
`int bcc_int_nestcount(void)`	5.9.6
`int bcc_int_disable_nesting(void)`	5.9.6
`int bcc_int_enable_nesting(void)`	5.9.6
`int bcc_int_set_nesting(int pil, int newpil)`	5.9.6.1
`int bcc_timestamp_avail(void)`	5.9.8
`int bcc_timestamp_restart(int ts_no, int source)`	5.9.8
`uint32_t bcc_timestamp_status(int ts_no)`	5.9.8
`uint32_t bcc_timestamp_get_ass(int ts_no)`	5.9.8
`uint32_t bcc_timestamp_get_ack(int ts_no)`	5.9.8
`uint32_t bcc_timestamp_get_cnt(void)`	5.9.8
`int bcc_get_cpu_count(void)`	5.10.1
`int bcc_get_cpuid(void)`	5.10.1
`int bcc_start_processor(int cpuid)`	5.10.2
`int bcc_send_interrupt(int level, int cpuid)`	5.10.2

6. AMBA Plug&Play library

6.1. Introduction

This chapter describes a user library used to probe devices on systems with an on-chip GRLIB AMBA Plug&Play bus. AMBA Plug&Play is generally available on LEON3 and LEON4 systems. For more information on the AMBA Plug&Play concept, see the *GRLIB IP Library User's Manual*.

The library is used by the BCC run-time to find the console device, timer devices and the interrupt controller. Application programmers can also use the library to probe for hardware devices to pair with device drivers.

6.1.1. AMBA Plug&Play terms and names

Throughout this chapter some software terms and names are frequently used. Below is a table which summarizes some of them.

Table 6.1. AMBA Layer terms and names

Term	Description
AMBAPP, AMBA PnP	AMBA Plug&Play bus. See AHBCTRL and APBCTRL in GRLIB GRIP documentation.
device	AMBA AHB Master, AHB Slave or APB Slave interface. The `amba_ahb_info` and `amba_apb_info` structures describe any of the interfaces.
core	A AMBA IP core often consists of multiple AMBA interfaces but not more than one interface of the same type.
bus	An AMBA AHB or APB bus.
Vendor ID	A unique number assigned to a device vendor. See `include/bcc/ambapp_ids.h`
Device ID	A unique number assigned to a device by a device vendor. See `include/bcc/ambapp_ids.h`
IO area	Address to a read-only table containing Plug&Play information for all attached devices on the bus. It is typically located at address `0xFFFFF000` on LEON systems.
scanning	Process where the AMBA PnP bus is searched for all or some AMBA interfaces.
depth	Number of levels of AHB-AHB bridges from topmost AHB bus.

6.1.2. Availability

Functions described in this chapter have structure definitions and prototypes in the C header file `bcc/ambapp.h`. The functions are compiled in `libbcc.a` and are available per default when linking with the GCC front-end.

6.2. Device scanning

BCC AMBA Plug&Play API is based around a device scanning routine in the function `ambapp_visit()`. It performs recursive, depth first, scanning for devices.

The `ambapp_visit()` routine can *visit* devices during the scanning, based on a user defined device match criteria. A *visit* is performed by the routine calling a user supplied function with information on the current device as function parameters. After the user function has inspected the device information, it can decide to terminate the scanning process altogether or let the scanning routine continue with the next match. The `ambapp_visit()` function does not allocate dynamic or static memory and does not build a device tree. It stores temporary information on the stack as needed.

Example use cases for the scanning routine include:

- Count number of AMBA Plug&Play devices/buses in the system.
- Build a device tree in memory.

- Find a specific device on a user criteria.

The main scanning function ambapp_visit() is defined in Table 6.2 and the callback interface is described in Table 6.3.

Table 6.2. `ambapp_visit` function declaration

Proto	`uint32_t ambapp_visit( uint32_t ioarea, uint32_t vendor, uint32_t device, uint32_t flags, uint32_t maxdepth, uint32_t (*fn)( void *info, uint32_t vendor, uint32_t device, uint32_t type, uint32_t depth, void *arg ), void *arg )`
About	Visit AMBA Plug&Play devices A recursive AMBA Plug&Play device scanning is performed, depth first. Information records are filled in and supplied to a user function on a user match criteria. The user match criteria is defined by the parameters *`vendor`*, *`device`* and *`flags`*. When the user function (*`fn`*) returns non-zero, the device scanning is terminated and `ambapp_visit()` returns the return value of the user function. The `ambapp_visit()` function does not allocate dynamic or static memory: all state is on the stack.
Param	*`ioarea`* [IN] Address IO area of bus to start device scanning.
Param	*`vendor`* [IN] Integer Vendor ID to visit, or 0 for all vendor IDs.
Param	*`device`* [IN] Integer Device ID to visit, or 0 for all device IDs.
Param	*`flags`* [IN] Integer Mask of device types to visit (AMBAPP_VISIT_AHBMASTER, AMBAPP_VISIT_AHBSLAVE, AMBAPP_VISIT_APBSLAVE).
Param	*`maxdepth`* [IN] Integer Maximum bridge depth to visit.
Param	*`fn`* [IN] Pointer User function called when a device is matched. See separate description on how the function is called.
Param	*`fn_arg`* [IN] Pointer User argument provided with each call to fn(). `ambapp_visit()` never dereferences *`fn_arg`*.
Return	uint32_t.

Value	Description
0	*`fn()`* did never return non-zero.
non-zero	*`fn()`* returned this value.

Table 6.3. `ambapp_visit_user_fn` function declaration

Proto	`uint32_t fn(void *info, uint32_t vendor, uint32_t device, uint32_t type, uint32_t depth, void *arg)`
About	User callback called by `ambapp_visit()` when a device is matched.
Param	*`info`* [IN] Pointer Pointer to struct amba_apb_info or struct amba_ahb_info as determined by the parameter *`type`*.
Param	*`vendor`* [IN] Integer Vendor ID for matched device

Param	*device* [IN] Integer Device ID for matched device
Param	*type* [IN] Integer Type of matched device (AMBAPP_VISIT_AHBMASTER, AMBAPP_VISIT_AHBSLAVE, AMBAPP_VISIT_APBSLAVE).
Param	*depth* [IN] Integer Bridge depth of matched device. First depth is 0. The depth decrements with one for each recursed bridge.
Param	*arg* [IN] Pointer User argument which was given to `ambapp_visit()` as parameter *fn_arg*.
Return	uint32_t.

Value	Description
0	Continue scanning
non-zero	Abort scanning and propagate return value to `ambapp_visit()` for return.

6.3. User callback

6.3.1. Criteria matching

User criteria for calling the user callback function for a device is defined by the `ambapp_visit()` function parameters *vendor*, *device* and *flags*. To scan for a specific device type (AHB master, AHB slave, APB slave), the bitmasks `AMBAPP_VISIT_AHBMASTER`, `AMBAPP_VISIT_AHBSLAVE`, `AMBAPP_VISIT_APBSLAVE` shall be used. A value of 0 for *vendor* or *device* matches all vendor IDs and device IDs respectively.

Visiting all devices can thus be accomplished by the following parameter values:

```
#include <bcc/ambapp.h>
vendor = 0;
device = 0;
flags = AMBAPP_VISIT_AHBMASTER | AMBAPP_VISIT_AHBSLAVE | AMBA_VISIT_APBSLAVE;
```

6.3.2. Device information

Parameters to the user callback (Table 6.3) provides information to the user about the current device. To dereference the *info* parameter, it must first be cast to the appropriate type, based on the *type* parameter as of table Table 6.4.

Table 6.4. Data structures for device information

Value of `type`	**Type of `info`**
`AMBAPP_VISIT_AHBMASTER`	`struct amba_ahb_info *`
`AMBAPP_VISIT_AHBSLAVE`	`struct amba_ahb_info *`
`AMBAPP_VISIT_APBSLAVE`	`struct amba_apb_info *`

The device information structures contain data decoded from the AMBA AHB and APB Plug&Play records as defined in Table 6.5, Table 6.6 and Table 6.7. The raw configuration record entry is also available via the `entry` field. See the *GRLIB IP Library User's Manual* for more details on the record fields.

```
struct amba_apb_info {
        uint8_t ver;
        uint8_t irq;
        uint32_t start;
        uint32_t mask;
        const struct ambapp_apb_entry *entry;
};
```

Table 6.5. amba_apb_info data structure declaration

ver	Device version

irq	Device interrupt number
start	Device address space start
mask	Device address space mask
entry	APB Plug&Play configuration record

```
struct amba_ahb_bar {
        uint32_t start;
        uint32_t mask;
        uint8_t type;
};
```

Table 6.6. amba_ahb_bar data structure declaration

start	Device address space start	
mask	Device address space mask	
type	Bank type	
	2	AHB memory space
	3	AHB I/O space

```
struct amba_ahb_info {
        uint8_t ver;
        uint8_t irq;
        struct amba_ahb_bar bar[AMBA_AHB_NBARS];
        const struct ambapp_ahb_entry *entry;
};
```

Table 6.7. amba_ahb_info data structure declaration

ver	Device version
irq	Device interrupt number
bar	Bank Address Register
entry	AHB Plug&Play configuration record

6.4. Example

The following example extracts the base address and interrupt number of the first APBUART device in the system and then aborts the scanning by returning non-zero.

```
#include <stdio.h>
#include <bcc/ambapp.h>
#include <bcc/ambapp_ids.h>

uint32_t myarg = 0;

/* User callback which is called on devices matched with ambapp_visit(). */
uint32_t myfn(void *info, uint32_t vendor, uint32_t device, uint32_t type, uint32_t depth, void *arg)
{
        struct amba_apb_info *apbi = info;
        if (type != AMBAPP_VISIT_APBSLAVE) {
                printf("Unexpected type=%u\n", type);
                return 0;
        }

        printf("vendor=%x, device=%x, type=%x, depth=%u, arg=%p\n",
            vendor, device, type, depth, arg);
        printf("ver=%u, irq=%u, start=%08x, mask=%08x\n",
            info->ver, info->irq, info->start, info->mask);
        return apbi->start;
}

/* This function returns address of first APBUART, or 0. */
uint32_t ex0(void) {
        const uint32_t ioarea = 0xFFFFF000;
        const uint32_t maxdepth = 4;
        uint32_t ret;

        ret = ambapp_visit(
                ioarea,
```

```
                VENDOR_GAISLER,
                GAISLER_APBUART,
                AMBAPP_VISIT_APBSLAVE,
                maxdepth,
                myfn,
                &myarg
        );
        return ret;
}
```

More examples are provided with the BCC distribution.

6.5. API reference

This section lists all AMBA Plug&Play API functions with references to the related section(s). The API is also documented in the source header files of the library, `bcc/ambapp.h`.

Table 6.8. AMBA Plug&Play library data structure reference

Type	Section
struct amba_apb_info	6.3.2
struct amba_ahb_bar	6.3.2
struct amba_ahb_info	6.3.2

Table 6.9. AMBA Plug&Play library function reference

Prototype	Section
`uint32_t ambapp_visit( uint32_t ioarea, uint32_t vendor, uint32_t device, uint32_t flags, uint32_t maxdepth, uint32_t (*fn)( void *info, uint32_t vendor, uint32_t device, uint32_t type, uint32_t depth, void *arg ), void *arg )`	6.2
`uint32_t ambapp_visit_user_fn(void *info, uint32_t vendor, uint32_t device, uint32_t type, uint32_t depth, void *arg)`	6.2, 6.3

7. Board Support Packages

This chapter describes the Board Support Packages (BSP) distributed with BCC. Information on how to override the BSP behavior is available in Chapter 8. The default BSP, named LEON3, is compatible with most systems.

7.1. Overview

BSPs provide an interface between BCC and target hardware through initialization code specific to target processor and a number of device drivers. Console, timer and interrupt controller drivers are supported in all BSPs.

A BSP is selected with the GCC option `-qbsp=`*`bspname`*, where *`bspname`* specifies any of the BSPs described in this chapter. The option is typically combined with `-mcpu=`*`cpuname`* and optionally `-msoft-float` and `-qnano`.

If option `-qbsp=` is not given explicitly, then `-qbsp=leon3` is implied.

It is important that the `-qbsp=`, `-mcpu=`, `-mfix` and `-msoft-float` options are given to GCC both at the compile and link steps. `-qsvt` is only applicable to linking.

NOTE: Selecting a BSP with `-qbsp=`, does *not* automatically infer any of the `-mcpu=`, `-mfix-` or `-msoft-float` options. See Appendix A and Appendix B for recommended compiler options.

Applications are by default linked to RAM address `0x40000000` by most BSPs. This can be changed with the GCC option `-Wl,-Ttext,`*`addr`* to link anywhere in the range `0x40000000` to `0x7ffffff0`. Some BSPs have other default link addresses which is noted in the corresponding section in this chapter.

Each BSP provides memory definitions for the linker scripts to use, suitable for the target device. In some situations there is a need to link applications to non-standard locations. A special linker script named `linkcmds-any` is provided for this purpose. `linkcmds-any` is available for all BSPs. The following example links an application to address `0xABCDE000`:

```
$ sparc-gaisler-elf-gcc -T linkcmds-any -Wl,-Ttext,0xABCDE000 hello.c -o hello.elf
```

All BSPs except the LEON3 BSP have link time configuration of device base addresses needed by the BCC drivers. The LEON3 BSP uses AMBA Plug&Play to probe devices. A BCC console driver is attached to `APBUART0` by default, timer driver is attached to `GPTIMER0` and the interrupt controller driver is attached to `IRQMP`/`IRQ(A)MP`. Chapter 8 describes how device base addresses can be customized by the user.

7.2. LEON3

The LEON3 BSP is a general BSP compatible with most LEON3 based systems. This is the only BSP which uses AMBA Plug&Play to discover peripheral devices at startup.

Linking with `-qsvt` is possible if SVT is supported by the target system.

7.3. GR712RC

The GR712RC BSP is customized for the GR712RC component.

The following linker scripts are available, selectable with the GCC `-T` option.

`linkcmds` (default)	Application is linked to RAM address `0x40000000`.
`linkcmds-ahbram`	Application is linked to on-chip RAM with BCH error-correction at address `0xa0000000`.

Memory map descriptions and a linker script template for creating custom linker scripts are available in `bsp/gr712rc/linkcmds.memory` and `bsp/gr712rc/linkcmds.base`.

Linking with `-qsvt` is supported.

7.4. GR716

The GR716 BSP is customized for the GR716 component.

Partial WRPSR as described in *SPARC-V8 Supplement, SPARC-V8 Embedded (V8E) Architecture Specification* is used by BCC when possible. The interrupt remap functions described in Section 5.9.4 are available. Linking with `-qsvt` and `-qnano` is recommended to reduce code size.

Memory map descriptions and a linker script template for creating custom linker scripts are available in `bsp/gr716/linkcmds.memory` and `bsp/gr716/linkcmds.base`.

7.4.1. Supported features

Table 7.1. GR716 BSP feature support in BCC

Hardware functionality	Reference	Comments
FPU (LEON)	2.4	Supported
I/O switch matrix	7.4.4.1	Support for setting pin functionality and LVDS configuration.
Brownout	N/A	No driver support
PLL	7.4.4.3	Supported
UART	4.1, 5.3, 8.2	BCC console driver in polling mode with FIFO support. Used for C standard library `stdin`, `stdout` and `stderr`. By default uses APBUART0 which can be redirected. Supported by the BCC timer API.
UART	13	Dedicated raw data communication driver. Supports interrupt and polling mode.
On-chip Dual-port Memory with EDAC Protection	N/A	No driver support. Initialized by on-chip boot loader and GRMON.
Fault Tolerant PROM/SRAM Memory Interface	N/A	No specific support. See example in [GR716-MINI-QSG].
1553B Bus Controller	21	Supported
1553B Bus Monitor	23	Supported
1553B Remote Terminal	22	Supported
ADC	N/A	No driver support. See example in [GR716-MINI-QSG].
DAC	N/A	No driver support
CAN 2.0B	12	Supported
Clock gating unit	19	Supported
GRDMAC	N/A	No driver support
General Purpose I/O Port	17	Basic functionality supported. Pulse sequencer and sampler is not supported.
Pulse Width Modulation Generator	N/A	No driver support
PacketWire	N/A	No driver support
SpaceWire Interface and RMAP target	11	Supported
SpaceWire TDP	N/A	No driver support
General Purpose Timer Units	4.2, 8.3	BCC timer driver. Used for C standard library time related functions. Typically dedicated to GPTIMER0 subtimer0.
General Purpose Timer Units	16	Low-level driver for operating on GPTIMER. Supports all timer cores. Latch/set functionlity not supported. One timer

Hardware functionality	Reference	Comments
		instance is used by the C library time functions. Watchdog API
I2C to AHB bridge	N/A	No driver support
I2C Master	15	Supported
I2C slave	N/A	No driver support
Interrupt Controller	5.9	Supported by the BCC Interrupt API
LEON3 Statistics Unit	N/A	No driver support. Typically accessed with dedicated GRMON command.
Memory Scrubber	25	Supported
SPI to AHB bridge	N/A	No driver support
SPI Controller	14	SPI master mode is supported. SPI slave mode is not supported.
SPI for Space Slave Controller	N/A	No driver support
SPI Memory Controller	N/A	No driver support
AMBA Protection Unit	24	Supported
AHB Status Registers	18	Supported User can install hook to handle errors.
Boot ROM	N/A	No driver support. See example in [GR716-MINI-QSG].

Table 7.2. Resources

GR716-MINI-QSG	GR716-MINI Quick Start Guide [https://www.gaisler.com/index.php/products/components/gr716]

7.4.2. Boot ROM

A BCC 2 application is ready to be used with the GR716 embedded boot loader (BOOTROM). There are two main cases:

- Application is copied from persistent memory or network to RAM by the BOOTPROM. Executes from volatile RAM.
- Application executes from persistent memory (external ROM or SPI). This is also called *direct boot* or ROM resident execution.

It is also possible to disable the GR716 embedded boot loader by configuring GR716 strap signals. In this case, the application should contain its own boot loader. See Section 2.16.

The following subsections describe how to link a BCC application for use with the GR716 BOOTPROM. Information on how to load the application and configure the GR716 for *image boot* from persistent memory, *network boot* or *direct boot* from persistent memory is available in the *GR716 Data Sheet and Users's Manual*.

7.4.2.1. Executing from volatile RAM

To link an application for executing from local instruction RAM, the default linker script shall be used:

`linkcmds` (default)	Application is linked to CPU local RAM: instruction RAM at address `0x31000000` and data RAM at address `0x30000000`.

The following example links an application for storage and execution in internal RAM:

```
$ sparc-gaisler-elf-gcc -qbsp=gr716 -mcpu=leon3 -qsvt -qnano main.o -o main.elf
```

The linker option `-T linkcmds` is not required since the linker script is selected by default.

7.4.2.2. Executing from persistent memory

To link an application for executing from persistent memory such as an external ROM or SPI, use one of the following linker scripts:

`linkcmds-extprom`	Application is linked to external ROM starting at address `0x01000000`. `.data` is copied from PROM to on-chip data RAM at BCC run-time initialization. `.bss` is also put in on-chip data RAM.
`linkcmds-spi0`	Same as `linkcmds-extprom`, but for first SPI controller memory mapped at address `0x02000000`.
`linkcmds-spi1`	Same as `linkcmds-extprom`, but for second SPI controller memory mapped at address `0x04000000`.

The following example links an application for storage and execution in external ROM:

```
  $ sparc-gaisler-elf-gcc -qbsp=gr716 -mcpu=leon3 -qsvt -qnano -T linkcmds-extprom main.o -o main.elf
```

Investigation of the link output shows that `.data` is in ROM space at load time, but referenced in local data RAM at execution time. Copying of `.data` from ROM to RAM is done automatically by the BCC initialization.

```
  $ sparc-gaisler-elf-objdump -h main.elf

main.elf:     file format elf32-sparc

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000021d0  01000000  01000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000090  010021d0  010021d0  000121d0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .ext.data     00000000  40000000  40000000  000201e8  2**0
                  CONTENTS
  3 .data         000001e8  30000000  01002260  00020000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  4 .bss          000001c8  300001e8  01002448  000201e8  2**2
                  ALLOC
...
```

An example on how to build an application as described in this subsection is included with the BCC distribution in the directory `examples/gr716_romres`.

7.4.2.3. System clock

The GR716 BSP supports the full frequency operating range of GR716. A time base has to be set by the user for the BCC time functions to operate correctly when the application is started from the GR716 embedded boot loader. The supported way to do this is to define a global constant variable named `__bsp_sysfreq` initialized with the system clock frequency in MHz. This ensures a known time base for the BCC timer driver and sets the BCC console driver baud to 38400.

On a GR716 clocked at 20 MHz, the following example configures the system clock.

```
/* GR716 clocked at 20 MHz */
  const unsigned int __bsp_sysfreq = 20*1000*1000;
```

The definition can be put in any C file which is linked with the application. Note that `__bsp_sysfreq` must *not* be declared `static`.

BSP initializations related to the system clock are implemented by the custom timer and console initialization functions and can be overridden. For more details, see Section 8.2.1 and Section 8.3.1.

7.4.3. APBUART initialization

The BCC GR716 BSP default run-time initialization has support for initializing up to one APBUART controller. The purpose of this is to make the C standard library `stdin`, `stdout` and `stderr` available. This initialization, if performed, happens just before `main()` is called.

APBUART initialization in the GR716 BSP will be performed depending on the values of the application variable `__bsp_sysfreq` AND the *run-time* state of the control register, according to the following:

```
__bcc_con_init:
  IF (__bsp_sysfreq != 0) AND (APBUART debug forwarding is disabled) THEN
    Configure APBUART scaler
    Enable APBUART transmitter and receiver
    Disable debug forwarding
```

```
ELSE
  Do not touch APBUART registers
ENDIF
```

APBUART debug forwarding is disabled by hardware on GR716 power-on. APBUART debug forwarding is enabled by GRMON when the startup-option **-u** is used, or when enabled via the **forward enable** command.

This means that:

- When application is started from GRMON with debug forwarding enabled, then BCC will *not* re-initialize the APBUART.
- When application is started from embedded boot ROM or power-on reset, then BCC will initialize APBUART if `__bsp_sysfreq` is set in the application.

The APBUART in question above is APBUART0 by default. It is possible to redirect to another APBUART by defining `__bcc_con_handle` as described in Section 8.2. It is also possible to override the default behavior described above defining an application-specific version of `__bcc_con_init()` as described in Table 8.1.

Source code for the default GR716 console initialization logic is available in the file `src/libbcc/bsp/gr716/bsp_con_init.c` installed with BCC 2.1.1.

7.4.4. Chip specific API

A set of functions are provided by the BCC GR716 BSP for controlling chip specific functionality. The are included in the BSP rather than as separate peripheral drivers since the functionality is tied to the GR716 and do not generalise to other components.

7.4.4.1. Pin configuration

`gr716_set_pinfunc()` configures the GR716 I/O switch matrix. The function is available via `bcc/gr716/pin.h`

Table 7.3. `gr716_set_pinfunc` function declaration

Proto	`int gr716_set_pinfunc(unsigned int pin, unsigned int mode)`
About	Configure one IO switch matrix entry This function updates one field in `SYS.CFG.GPx` to configure the specified pin with the functionality requested by mode. Parameters *`pin`* and *`mode`* are range checked before registers are written.
Param	*`pin`* [IN] Integer GPIO pin number (0..63)
Param	*`mode`* [IN] Integer Any of `IO_MODE_*` (0..0xe)
Return	int.

Value	Description
0	Success
nonzero	Range check failure

A set of defines for `IO_MODE_*` is available:

Table 7.4. IO modes

Symbol	Description
`IO_MODE_GPIO`	GPIO
`IO_MODE_APBUART`	APBUART
`IO_MODE_MEM`	Memory controller (FTMCTRL)

Symbol	Description
IO_MODE_PW	PacketWire
IO_MODE_1553	MIL-STD-1553B
IO_MODE_CAN	GRCAN
IO_MODE_I2C	I2C
IO_MODE_SPI	SPI
IO_MODE_ADC	ADC
IO_MODE_DAC	DAC
IO_MODE_PWM	GRPWM
IO_MODE_SPW	Spacewire (rudundant interface on second port)
IO_MODE_SPI4S	SPI for Space master and slave
IO_MODE_AHBUART	AHBUART for DMA bus
IO_MODE_TDP	SpaceWire TDP (GRSPWTDP)

Example 7.1. Configure GPIO63 for CAN with `gr716_set_pinfunc()`

```
#include <bcc/gr716/pin.h>

gr716_set_pinfunc(63, IO_MODE_CAN);
```

7.4.4.2. LVDS configuration

`gr716_set_lvdsfunc()` configures the on-chip LVDS transceivers. Available via `bcc/gr716/pin.h`

Table 7.5. `gr716_set_lvdsfunc` function declaration

Proto	`int gr716_set_lvdsfunc(unsigned int mode)`
About	Configure LVDS transceivers This function updates `SYS.CFG.LVDS` to configure the on-chip LVDS transceivers with the functionality requested by *`mode`*. The *`mode`* parameter is range checked before registers are written.
Param	*`mode`* [IN] Integer Any of `LVDS_MODE_*` (0..3 or 8)
Return	int.

Value	Description
0	Success
nonzero	Range check failure

Table 7.6. LVDS modes

Symbol	Description
LVDS_MODE_SPW	Spacewire (primary interface on first port)
LVDS_MODE_SPI4S	SPI for Space master and slave
LVDS_MODE_SPIM	SPI master controller (not the SPI memory controller)
LVDS_MODE_SPIS	SPI slave controller
LVDS_DISABLE	Disable LVDS transceiver

Example 7.2. Configure LVDS for SpaceWire with `gr716_set_lvdsfunc()`

```
#include <bcc/gr716/pin.h>
```

```
gr716_set_lvdsfunc(LVDS_MODE_SPW);
```

7.4.4.3. PLL configuration

`gr716_pll_config()` configures the PLL. PLL lock can be proved with the function `gr716_pll_islocked`. `gr716_sysclk()` and `gr716_spwclk()` configures the clocks.

These functions are available via `bcc/gr716/pll.h`

Table 7.7. `gr716_pll_config` function declaration

Proto	`int gr716_pll_config(int ref, int cfg, int pd)`	
About	Configure PLL Mode #1: (*pd* = 1) • This mode powers down the PLL Mode #2: (*pd* = 0, *ref* = X and *cfg* = Y) • This mode powers on the PLL • Select the external input pin with *ref* • The PLL must be configure the correct input frequency with *cfg*.	
Param	*ref* [IN] Integer Select external input pin as PLL input. Can be any of the following values;	
	Value	Description
	`PLL_REF_SYS_CLK`	External System clock
	`PLL_REF_SPW_CLK`	External SpaceWire clock
Param	*cfg* [IN] Integer Input frequency for PLL input. Can be any of the following values;	
	Value	Description
	`PLL_FREQ_50MHZ`	50 MHz
	`PLL_FREQ_25MHZ`	25 MHz
	`PLL_FREQ_20MHZ`	20 MHz
	`PLL_FREQ_12MHZ`	12 MHz
	`PLL_FREQ_10MHZ`	10 MHz
	`PLL_FREQ_5MHZ`	5 MHz
Param	*pd* [IN] Integer PLL power down. Can be any of the following values;	
	Value	Description
	`PLL_POWER_DOWN`	Power down PLL (1)
	`PLL_POWER_ENABLE`	Enable PLL (0)
Return	int.	
	Value	Description
	0	Success
	nonzero	Failure

Table 7.8. `gr716_sysclk` function declaration

Proto	`int gr716_sysclk(int source, int div, int duty)`
About	Configure system clock Mode #1: (*source* = 0, *div* = 0 and *duty* = 0)

	• This mode bypasses the divider and uses the external clock pin as input Mode #2: (*source* = X, *div* = Y and *duty* = 0) • This mode divides the clock source with 2^Y and selects as system clock • Y can be in the range from 1 to 31 Mode #3: (*source* = X, *div* = Y and *duty* = Z) • This mode divides the clock source with Y and selects as system clock. • Clock duty cycle is set by *duty* in number of system clocks • Y can be in the range from 2 to 31 and must be at least 1 greater than X • Z can be in the range from 1 to 30
Param	*source* [IN] Integer Clock source. Can be any of the following values;

Value	Description
CLK_SOURCE_CLK	External System clock
CLK_SOURCE_SPW	PLL input
CLK_SOURCE_PLL	PLL output

Param	*div* [IN] Integer Clock divisor
Param	*duty* [IN] Integer Clock duty cycle
Return	int.

Value	Description
0	Success
nonzero	Failure

Table 7.9. gr716_spwclk function declaration

Proto	`int gr716_spwclk(int source, int div, int duty)`
About	Configure SpaceWire clock Mode #1: (*source* = 0, *div* = 0 and *duty* = 0) • This mode bypasses the divider and uses the external clock pin as input Mode #2: (*source* = X, *div* = Y and *duty* = 0) • This mode divides the clock source with 2^Y and selects as SpaceWire clock • Y can be in the range from 1 to 31 Mode #3: (*source* = X, *div* = Y and *duty* = Z) • This mode divides the clock source with Y and selects as SpaceWire clock. • Clock duty cycle is set by *duty* in number of clocks • Y can be in the range from 2 to 31 and must be at least 1 greater than X • Z can be in the range from 1 to 30
Param	*source* [IN] Integer Clock source. Can be any of the following values;

Value	Description
SPWCLK_SOURCE_CLK	PLL input
SPWCLK_SOURCE_PLL	PLL output

Param	*div* [IN] Integer

	Clock divisor	
Param	*duty* [IN] Integer Clock duty cycle	
Return	int.	
	Value	Description
	0	Success
	nonzero	Failure

Table 7.10. gr716_pll_islocked function declaration

Proto	`int gr716_pll_islocked(void)`	
About	Determine if PLL is currently locked This function returns the current value of the PLL lock output.	
Return	int.	
	Value	Description
	1	PLL is locked
	0	PLL is not locked

7.5. LEON2

The LEON2 BSP is compatible with LEON2 systems such as AT697, AT697E and AT697F.

AMBA Plug&Play configuration records are not implemented in most LEON2 systems, so the BCC AMBA Plug&Play library described in Chapter 6 may not be used. But since the hardware information is resolved by the BSP, and can be overridden as described in Chapter 8, this does not affect normal operation of BCC on LEON2 systems

`-qsvt` is not supported on LEON2.

7.6. AGGA4

The AGGA4 BSP is similar to the LEON2 BSP. It has a different console driver which is transparent to the user. Recommended compiler options for AGGA4 can be found in Appendix A.

8. Customizing BCC

The BCC run time environment is designed to fit a wide range systems and to require little user intervention to get an application up and running. In some situations however, the default behavior may need customization to fulfill specific application requirements on device discovery, console drivers, size optimization, etc. This chapter describes how the BCC run time environment can be customized.

8.1. Introduction

Three types of hardware devices are managed by the BCC run time: *console*, *timer* and *interrupt controller*. The management consists of software drivers which are embedded in the application when needed. Some of the C library functionality and the BCC user library depend on these drivers.

For most BSPs, the run time relies on hardware devices residing in predefined address spaces. For the general LEON3 BSP, the device hardware address space locations are probed with help of the AMBA Plug&Play scanning routines described in Section 6.2. Device initialization and possible probing takes place before entry to `main()` and can be overridden by the application as described later in this chapter.

Functions and variables used for user run time customization are declared in the header file `bcc/bcc_param.h`. This header file should be included in any application which overrides the default BCC behavior.

To override the default implementation of a BCC function or variable, an object file containing the same symbol name as the overridden function or variable should be linked with the application. The prototypes in `bcc/bcc.h` and `bcc/bcc_param.h` can be used for type checking. An example is provided in Section 8.5.

8.2. Console driver

The BCC console driver is used for C library input and output on `stdin`, `stdout` and `stderr`.

8.2.1. Initialization

A variable named `__bcc_con_handle` is reserved for the console driver to use. The content of this variable is console driver specific, and will typically contain an address to some hardware register space. A BSP is responsible for initializing this variable, which can be done either at compile time or run time. The function (hook) named `__bcc_con_init()` is called before `main()` as part of the BCC run time initialization. A BSP can use the hook function to initialize `__bcc_con_handle`, for example by using the AMBA Plug&Play library. Table 8.2 describes how BSPs initialize the handle.

Table 8.1. `__bcc_con_init` function declaration

Proto	`int __bcc_con_init(void)`
About	Probe and initialize the console A default implementation of this function is provided by the BSP. It can be overridden by the user.
Return	int. BCC_OK on success

Table 8.2. Implementation of `__bcc_con_init()`

BSP	**Description of `__bcc_con_init()`**
`leon3`	The AMBA Plug&Play library (Chapter 6) is used to scan for APBUART devices. `__bcc_con_handle` is assigned with the address of the register area of the first APBUART device.
others	`__bcc_con_init()` is empty. `__bcc_con_handle` is an initialized variable with value determined at link time.

8.2.2. Input and output functions

Character input is handled by the function `__bcc_con_inbyte()` and output by `__bcc_con_outbyte()`.

Table 8.3. `__bcc_con_inbyte` function declaration

Proto	`char __bcc_con_inbyte(void)`
About	Read the next character from console
Return	char. The read character

Table 8.4. `__bcc_con_outbyte` function declaration

Proto	`int __bcc_con_outbyte(char c)`
About	Write a character on the console
Param	`c` [IN] Character Character to output
Return	int. 0 on success

8.2.3. Customization

- Console redirection is performed by redefining `__bcc_con_handle`, for example in a custom, `__bcc_con_init()` hook. See Section 8.5.
- The I/O functions `__bcc_con_inbyte()` and `__bcc_con_outbyte()` can also be overridden. They shall typically make use of `__bcc_con_handle`.

8.2.4. C library I/O

All console input fed to the C library goes via `read()` and the output goes out with `write()`. An application can override these functions to get even more control on the console I/O (for example to implement terminal specific handling). See the *newlib C library* documentation for more information on how `read()` and `write()` are defined. The function call flow is illustrated below.

- `[terminal] -> __bcc_con_inbyte() -> read() -> [C library stdio]`
- `[C library stdio] -> write() -> __bcc_con_outbyte() -> [terminal]`

NOTE: Both `stdout` and `stderr` are output via `write()` and `__bcc_con_outbyte()`.

8.3. Timer driver

The BCC timer driver is used for C library time related functions such as `clock()` and `time()` (`time.h`). It is also used for `gettimeofday()` and `times()`.

8.3.1. Initialization

Initialization is similar to the console driver (Section 8.2.1). The timer handle is named `__bcc_timer_handle` and the initialization hook is named `__bcc_timer_init()`. Table 8.6 describes how BSPs initialize the handle.

Table 8.5. `__bcc_timer_init` function declaration

Proto	`int __bcc_timer_init(void)`
About	Probe timer hardware and initialize timer driver A default implementation of this function is provided by the BSP. It can be overridden by the user.
Return	int. BCC_OK on success

Table 8.6. Implementation of `__bcc_timer_init()`

BSP	**Description of `__bcc_timer_init()`**
`leon3`	The AMBA Plug&Play library (Chapter 6) is used to scan for `GPTIMER` devices. `__bcc_timer_handle` is assigned with the address of the register area of the first

BSP	Description of __bcc_timer_init()
	GPTIMER device and __bcc_timer_interrupt is assigned to the timers interrupt number.
others	__bcc_timer_init() is empty. __bcc_timer_handle and __bcc_timer_interrupt are initialized variables with values determined at link time.

8.3.2. Time access functions

Current time in microseconds is returned by the function bcc_timer_get_us() as described in Section 5.3. This function is used by the C library for time related functions (time.h).

8.3.3. Customization

The BCC timer driver initialization can be overridden by redefining the functions __bcc_timer_init() and bcc_timer_get_us().

8.4. Interrupt controller driver

The BCC interrupt controller driver is managing the BCC interrupt and AMP user API described in Section 5.9 and Section 5.10.

8.4.1. Initialization

Initialization is similar to the console driver (Section 8.2.1). The interrupt controller driver handle is named __bcc_int_handle and the initialization hook is __bcc_int_init(). Table 8.8 describes how BSPs initialize the handle.

Table 8.7. __bcc_int_init function declaration

Proto	int __bcc_int_init(void)
About	Probe interrupt controller hardware and initialize interrupt controller driver A default implementation of this function is provided by the BSP. It can be overridden by the user.
Return	int. BCC_OK on success

Table 8.8. Implementation of __bcc_int_init()

BSP	Description of __bcc_int_init()
leon3	The AMBA Plug&Play library (Chapter 6) is used to scan for IRQMP/IRQ(A)MP devices. __bcc_int_handle is assigned with the address of the register area of the first interrupt controller device. If the interrupt controller has support for multiple internal interrupt controllers (IRQ(A)MP), then __bcc_int_handle will be adjusted to match the IRQ(A)MP *Interrupt Controller Select Registers* for the executing CPU. Extended interrupt number is probed and assigned to the global variable __bcc_int_irqmp_eirq.
others	__bcc_int_init() is empty. __bcc_int_handle is an initialized variable with value determined at link time. __bcc_int_irqmp_eirq depends on if the target system supports extended interrupt.

8.4.2. Access functions

Most of the functionality of the BCC interrupt and AMP API is implemented by the interrupt controller driver in the corresponding BSP.

8.4.3. Customization

The BCC interrupt controller driver initialization can be overridden by redefining the `__bcc_int_init()` hook or `__bcc_int_handle`.

On systems which support extended interrupts (most LEON3 and LEON4 systems) the variable `__bcc_int_irqmp_eirq` can also be redefined. (Its value can be determined by reading an interrupt controller register.)

BCC interrupt and AMP services are tightly connected with the interrupt controller driver. There is no interface specified for overriding these services. Customization would typically require a re-implementation of all BCC interrupt and AMP API routines. (For details, see the source code in `libbcc/shared/interrupt/` directory of the BCC source distribution).

8.5. Initialization override example

The following example illustrates how the console, timer and interrupt controller initialization can be overridden on a GR740 system.

```
#include <stdio.h>
#include <bcc/bcc.h>
#include <bcc/bcc_param.h>

/* Forced initialization for GR740. */
int __bcc_con_init(void) {
        __bcc_con_handle = 0xff900000;
        return 0;
}

int __bcc_timer_init(void) {
        __bcc_timer_handle = 0xff908000;
        __bcc_timer_interrupt = 1;
        return 0;
}

int __bcc_int_init(void) {
        __bcc_int_handle = 0xff904000;
        __bcc_int_irqmp_eirq = 10;
        return 0;
}

int main(void) {
        puts("hello world");
        return 0;
}
```

The example can be compiled and linked by issuing the following command.

```
$ sparc-gaisler-elf-gcc -qbsp=gr740 -mcpu=leon3 example.c -o example
```

8.6. Initialization hooks

An additional set of user hooks are called during BCC initialization. They are named with numbers corresponding with execution order. A higher number means closer to `main()`. Default implementations of these hooks are empty and they can be overridden by the user.

Table 8.9. `__bcc_init40` function declaration

Proto	`void __bcc_init40(void)`
About	Called at start of reset trap before CPU initializations • Trap handling is not available. • `%sp` and `%fp` are not valid (do not save/restore) • `save` and `restore` instructions are not allowed • svt/mvt is not configured. • `.bss` section is not initialized. • This user hook should be written in assembly.

Return	None.

Table 8.10. `__bcc_init50` function declaration

Proto	`void __bcc_init50(void)`
About	Called at start of C run time initialization (`crt0`) • Trap handling is not available. • `%sp` and `%fp` are not valid (do not save/restore) • `save` and `restore` instructions are not allowed • `.bss` section is not initialized. • BCC drivers are not initialized. • This user hook should be written in assembly.
Return	None.

Table 8.11. `__bcc_init60` function declaration

Proto	`void __bcc_init60(void)`
About	Called prior to BCC driver initialization • C runtime is available. • BCC drivers are not initialized. • This user hook can be written in C. • Console API, timer API and interrupt API are not available.
Return	None.

Table 8.12. `__bcc_init70` function declaration

Proto	`void __bcc_init70(void)`
About	Called as the last step before `main()` is called. • C runtime is available. • Full BCC API is available.
Return	None.

The following example illustrates how the interrupt based timer service is activated by calling `bcc_timer_tick_init_period()` in `__bcc_init70()` before entry to `main()`. The timer tick is configured for 100 tick interrupts per second. See Section 5.3.1 for more information on `bcc_timer_tick_init_period()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <bcc/bcc.h>
#include <bcc/bcc_param.h>

void __bcc_init70(void) {
        int ret;

        /* 100 tick interrupts per second. */
        ret = bcc_timer_tick_init_period(10 * 1000);
        if (BCC_OK != ret) {
            exit(EXIT_FAILURE);
        }
}

int main(void) {
        clock_t now;
        while(1) {
                now = clock();
                printf("clock() => %09u\n", now);
        }
        return EXIT_SUCCESS;
}
```

8.7. Disable .bss section initialization

As part of its startup code, the BCC C run time initializes the .bss segment with zeroes. This initialization is disabled by defining a global variable named __bcc_cfg_skip_clear_bss. The value of __bcc_cfg_skip_clear_bss does not matter as long as the symbol address is not 0.

Disabling .bss initialization can be useful when executing an application on a simulated system where execution is slow and memory is already cleared.

NOTE: If the .bss section is not preinitialized, then disabling the initialization will result in a non-functional program.

8.7.1. Example

The following example illustrates how initialization of the .bss section can be disabled.

```
#include <bcc/bcc_param.h>

int __bcc_cfg_skip_clear_bss;

int main(void)
{
        ...
        return 0;
}
```

8.8. Heap memory configuration

By default, the application heap starts at the end of bss, and ends at the stack pointer. The heap can be relocated by the user by assigning initialization values to the variables __bcc_heap_min and __bcc_heap_max, declared in the header file bcc/bcc_param.h.

The following example configures a heap of 16 MiB starting at address 0x60000000:

```
#include <stdlib.h>
#include <stdio.h>
#include <bcc/bcc_param.h>

#define MYHEAPSIZE (16 * 1024 * 1024)
uint8_t *__bcc_heap_min = (uint8_t *) 0x60000000;
uint8_t *__bcc_heap_max = (uint8_t *) 0x60000000 + MYHEAPSIZE;

int main(void)
{
  void *p;
  p = malloc(MYHEAPSIZE / 2);
  printf("malloc(%d KiB) => %p\n", MYHEAPSIZE / 1024, p);
  free(p);
  return 0;
}
```

__bcc_heap_min and __bcc_heap_max can optionally be assigned by the application at run-time, but only before any dynamic memory functions have been called. The initialization hook __bcc_init70() is a suitable location.

To gain full control over heap allocation, the function sbrk() can be redefined by the user: see the *Newlib C library documentation*, chapter *System Calls* for more information.

8.9. Parameters to main()

BCC by defaults sets argc to 0 and argv[argc] to NULL given the main() function prototype:

```
  int main(int argc, char *argv[]);
```

The user can override this by defining the variables __bcc_argc and __bcc_argvp, declared in the header file bcc/bcc_param.h.

Before the BCC run-time initialization calls main(), the following is performed:

- `argc` is assigned to the value of `__bcc_argc`
- `argv` is loaded from `__bcc_argvp`

Below is an example on how the interface can be used.

```
#include <bcc/bcc_param.h>

char *myargs[] = { "zero", "one", "two", "three", NULL };
int __bcc_argc = 4;
char *((*__bcc_argvp)[]) = &myargs;
```

The indirection of `argv` allows for overriding the `main()` parameters after the application is loaded to memory but before it starts. See the program and script in `src/examples/mainarg/`.

8.10. API reference

This section lists API functions related to BCC customization with references to the related section(s). The API is also documented in the source header file `bcc/bcc_param.h`.

Table 8.13. BCC customization functions reference

Prototype	Section
`int __bcc_con_init(void)`	8.2.1
`char __bcc_con_inbyte(void)`	8.2.2
`int __bcc_con_outbyte(char c)`	8.2.2
`int __bcc_timer_init(void)`	8.3.1
`uint32_t bcc_timer_get_us(void)`	8.3.2, 5.3
`int __bcc_int_init(void)`	8.4.1
`void __bcc_init40(void)`	8.6
`void __bcc_init50(void)`	8.6
`void __bcc_init60(void)`	8.6
`void __bcc_init70(void)`	8.6

9. Support

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

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

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

Appendix A. Recommended GCC options for LEON systems

This appendix contains recommended GCC options for LEON systems related to code generation and linking.

NOTE: The recommendations apply to BCC version 2.1.1. Other LEON toolchains and other versions of BCC may have other recommendations.

Table A.1. Recommended GCC options for BCC 2.1.1

System	Recommended GCC options
GR740	`-qbsp=gr740 -mcpu=leon3`
GR712RC	`-qbsp=gr712rc -mcpu=leon3 -mfix-gr712rc`
GR716	`-qbsp=gr716 -mcpu=leon3 -qnano -qsvt`
UT699E, UT700	`-mcpu=leon3 -mfix-ut700`
UT699/EPICA-NEXT, SCOC3	`-mcpu=leon -mfix-ut699`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul`/`div` based on GRLIB versions up to and including build 4174.	`-mcpu=leon3 -mfix-b2bst -mfix-tn0013 -qfix-tn0018`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul`/`div` based on GRLIB version 4175 to 4206	`-mcpu=leon3 -mfix-tn0013 -qfix-tn0018`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul`/`div` based on GRLIB version 4207 to 4248.	`-mcpu=leon3 -qfix-tn0018`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul`/`div` based on GRLIB version 4249 and later.	`-mcpu=leon3`
LEON3 systems with SPARC V8 `mul`/`div` implemented without cache parity protection.	`-mcpu=leon3` For GRLIB version up to and including 4206, also add • `-mfix-tn0013`
LEON3/LEON3FT systems without SPARC V8 `mul`/`div`.	Same as above but change `-mcpu=leon3` to `-mcpu=leon3v7`.
AGGA4	`-qbsp=agga4 -mcpu=leon -mfix-at697f`
AT697	`-qbsp=leon2 -mcpu=leon -mfix-at697f`
Other LEON2 systems	`-qbsp=leon2 -mcpu=leon`

In addition to Table A.1:

- `-qnano` can always be used.
- `-msoft-float` can always be used.
- Systems which support SVT (single vector trapping) can use `-qsvt`.
- If no `-mcpu=` option is given explicitly, then SPARC V7 code will be generated.
- The BCC 2.1.1 run-time supports the GCC option `-mflat`.

The recommendations in Table A.1 apply to both compilation and linking.

Table A.2 describes the GCC `-mcpu=` options applicable to BCC 2.1.1. If no `-mcpu=` option is used, then `-mcpu=v7` is implied.

Table A.2. GCC -mcpu= options for BCC 2.1.1

Option	Description
`-mcpu=v7` (or no `-mcpu=` option)	no `mul`/`div`, no `casa`
`-mcpu=leon`	`mul`/`div`, no `casa`
`-mcpu=leon3`	`mul`/`div`, `casa`
`-mcpu=leon3v7`	no `mul`/`div`, `casa`

Appendix B. Recommended Clang options for LEON systems

This appendix contains recommended Clang options for LEON systems related to code generation and linking.

NOTE: The recommendations apply to BCC version 2.1.1. Other LEON toolchains and other versions of BCC may have other recommendations.

Table B.1. Recommended Clang options for BCC 2.1.1

System	Recommended Clang options
GR740	`-qbsp=gr740 -mcpu=gr740`
GR712RC	`-qbsp=gr712rc -mcpu=gr712rc -mfix=gr712rc`
GR716	`-qbsp=gr716 -mcpu=leon3 -qnano -qsvt`
UT699E, UT700	`-qbsp=leon3 -mcpu=leon3 -mfix=ut700`
UT699/EPICA-NEXT, SCOC3	`Unsupported`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul/div` based on GRLIB versions up to and including build 4206.	*Contact support@gaisler.com.*
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul/div` based on GRLIB version 4207 to 4248.	`-qbsp=leon3 -mcpu=leon3 -qfix-tn0018`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul/div` based on GRLIB version 4249 and later.	`-qbsp=leon3 -mcpu=leon3`
LEON3/LEON3FT systems without SPARC V8 `mul/div`.	`Unsupported`
AGGA4	`Unsupported`
AT697	`Unsupported`
Other LEON2 systems	`Unsupported`

In addition to Table B.1:

- `-qnano` can always be used.
- `-msoft-float` can always be used.
- Systems which support SVT (single vector trapping) can use `-qsvt`.
- If no `-mcpu=` option is given explicitly, then SPARC V8 code will be generated.
- Systems supporting the LEON-REX extension can use `-mrex`.
- The BCC 2.1.1 run-time supports the option `-mflat`.

The recommendations in Table B.1 apply to both compilation and linking.

Table B.2 describes the Clang `-mcpu=` options applicable to BCC 2.1.1. If no `-mcpu=` option is used, then SPARC V8 with `mul/div` is generated.

Table B.2. Clang `-mcpu=` options for BCC 2.1.1

Option	Description
no `-mcpu=` option specified	`mul/div`, no `casa`
`-mcpu=leon3, -mcpu=gr712rc, -mcpu=gr740`	`mul/div`, `casa`

Device drivers reference

The following sections describe the LEON4, LEON3 and LEON2 device drivers included in BCC 2.1.1. Each driver is described in a separate chapter.

Driver samples can be found under `src/libdrv/examples` and `src/examples` in the distribution.

10. Driver registration

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

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

Two main methods are available for registering a peripheral to a device driver:

- Automatic
- Manual

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

10.1. Automatic registration

Automatic registration is straight forward and covers many use cases. To use this method with the APBUART driver, all the user has to do is to call the function `apbuart_autoinit()`:

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

int main(void)
{
  struct apbuart_priv *dev;

  apbuart_autoinit();
  dev = apbuart_open(0);
  [...]
}
```

The user should be aware of the following behavior of *automatic registration*:

- Device private data is allocated with `malloc()`.
- AMBA Plug&Play bus scanning is performed.

In case the above behavior is not compatible with the target application, then the method described in Section 10.2 can be used instead.

10.2. Manual registration

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

Registration of a peripheral can be performed with the function

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

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

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

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

int main(void) {
  struct apbuart_priv *dev;

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

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

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

which takes a NULL terminated array as parameter:

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

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

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

int main(void) {
  struct apbuart_priv *dev;

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

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

10.3. System specific device registration tables

Device configuration tables have been prepared for the following systems:

Table 10.1. Device registration tables for manual registration

System	Header files
GR716	gr716/

11. GRSPW Packet driver

11.1. Introduction

This section describes the GRSPW packet driver for BCC.

It is an advantage to understand the SpaceWire bus/protocols, GRSPW hardware and software driver design when developing using the user interface in Section 11.3 and Section 11.4. The Section 11.2.1 describes the overall software design of the driver.

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

11.1.1. Hardware Support

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

- GRSPW, GRSPW2 and GRSPW2_DMA (router AMBA port)
- Multiple DMA channels
- Link Control
- Port Control
- RMAP Control

11.1.2. Driver sources

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

Table 11.1. Source Location

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

11.1.3. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 11.2. Driver registration functions

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

11.1.4. Examples

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

11.1.5. Known driver limitations

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

- The statistics counters are not atomic, clearing at the same the interrupt handler is called could cause invalid statistics, one must disable interrupt when reading/clearing.

11.2. Software design overview

11.2.1. Overview

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

- Device interface, see Section 11.3.
- DMA channel interface, see Section 11.4.

GRSPW device parameters that affects the GRSPW core and all DMA channels are accessed over the device API whereas DMA specific settings and buffer handling are accessed over the per DMA channel API. A GRSPW2 device may implement up to four DMA channels.

In order to access the driver the first thing is to open a GRSPW device using the device interface.

For controlling the device one must open a GRSPW device using `'id = grspw_open(dev_index)'` and call appropriate device control functions. Device operations naturally affects all DMA channels, for example when the link is disabled all DMA activity pause. However there is no connection software wise between the device functions and DMA function, except from that the `grspw_close` requires that all of its DMA channels have been closed. Closing a device fails if DMA channels are still open.

Packets are transferred using DMA channels. To open a DMA channel one calls `'dma_id = grspw_dma_open(id, dmachan_index)'` and use the appropriate transmission function with the *`dma_id`* to identify which DMA channel used.

11.2.2. Initialization

During early initialization when the operating system boots the driver performs some basic GRSPW device and software initialization. The following steps are performed or not performed:

- GRSPW device and DMA channels I/O registers are initialized to a state where most are zero.
- DMA is stopped on all channels
- Link state and settings are not changed (RMAP may be active).
- RMAP settings untouched (RMAP may be active).
- Port select untouched (RMAP may be active).
- Time Codes are disabled and TC register cleared.
- IRQ generation disabled.
- Status Register cleared.
- Node address / DMA channels node address is untouched (RMAP may be active).
- Hardware capabilities are read.
- Device index determined.

11.2.3. Link control

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

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

Function names prefix: `grspw_link_*()`.

11.2.4. Time Code support

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

Function names prefix: `grspw_tc_*()`

11.2.5. RMAP support

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

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

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

The SpaceWire node address, which also affects the RMAP target, is controlled from the address configuration routines, see Section 11.2.7.

Function names prefix: `grspw_rmap_*()`

11.2.6. Port support

The GRSPW device has optional support for two ports (two connectors), where only one port can be active at a time. The active SpaceWire port is either forced by the user or auto selected by the hardware depending on the link state of the SpaceWire ports at a certain condition.

The device interface is used to get information about the GRSPW hardware port support, current set up and to control how the active port is selected.

Function names prefix: `grspw_port_*()`

11.2.7. SpaceWire node address configuration

The GRSPW core supports assigning a SpaceWire node address or a range of addresses. The address affects the received SpaceWire Packets, both to the RMAP target and to the DMA receiver. If a received packet does not match the node address it is dropped and the GRSPW status indicates that one or more packets with invalid address was received.

The GRSPW2 and GRSPW2_DMA cores that implements multiple DMA channels use the node address as a way to determine which DMA channel a received packet shall appear at. A unique node address or range of node addresses per DMA channel must be configured in this case.

It is also possible to enable promiscuous mode to enable all node addresses to be accepted into the first DMA channel, this option does not affect the RMAP target node address decoding.

The GRSPW SpaceWire node address configuration is controlled using the device interface. A specific DMA channel's node address is thus affected by the "global" device API and not controllable using the DMA channel interface.

If supported by hardware the node address can be removed before DMA writes the packet to memory. This is a configuration option per DMA channel using the DMA channel API.

Function names prefix: `grspw_addr_*()`

11.2.8. User DMA buffer handling

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

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

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

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

Function names prefix: `grspw_dma_*()`

11.2.8.1. Buffer List help routines

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

Function names prefix: `grspw_list_*()`

11.2.9. Driver DMA buffer handling

The driver represents packets with the struct grspw_pkt packet structure, see Table 11.32. They are arranged in linked lists that are called queues by the driver. The order of the linked lists are always maintained to ensure that the packet transmission order is represented correctly.

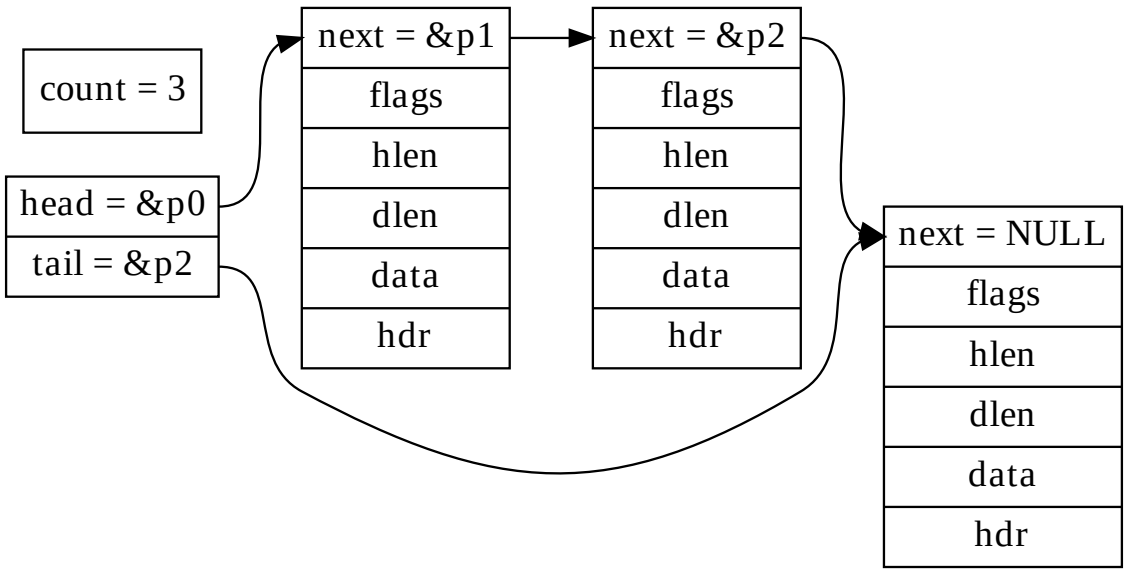

Figure 11.1. Queue example - linked list of three grspw_pkt packets

11.2.9.1. DMA Queues

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

- RX SCHED queue - packets that have been assigned a RX DMA descriptor.
- TX SCHED queue - packets that have been assigned a TX DMA descriptor.

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

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

The current number of packets in respective queue can be read by doing function calls using the DMA API, see Section 11.4.7. The user can for example use this to determine to wait or continue with packet processing.

11.2.9.2. DMA Queue operations

The user can control how the RX SCHED and TX SCHED queues are populated, by providing and removing packet buffers. The user can control how and when packets are moved from RX SCHED and TX SCHED queues into user provided queues by manually trigger the move by calling reception and transmission routines as described in Section 11.4.6 and Section 11.4.5.

For RX, the packets always flow in one direction from USER RX READY -> RX SCHED -> USER RX RECV. Likewise the TX packets flow USER TX SEND -> TX SCHED -> USER TX SENT. The procedures triggering queue packet moves are listed below and in Figure 11.2 and Figure 11.3. The interface of theses procedures are described in the DMA channel API.

- USER -> RX SCHED – `grspw_dma_rx_prepare`, Section 11.4.6.
- RX SCHED -> USER – `grspw_dma_rx_recv`, Section 11.4.6.
- USER -> TX SCHED queue – `grspw_dma_tx_send`, Section 11.4.5.
- TX SCHED -> USER – `grspw_dma_tx_reclaim`, Section 11.4.5.

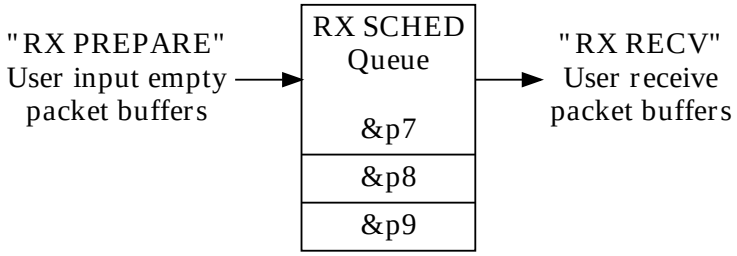

Figure 11.2. RX queue packet flow and operations

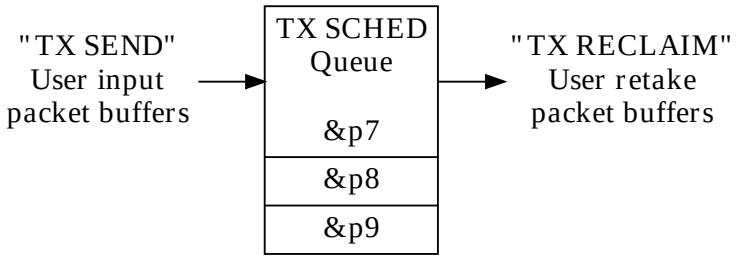

Figure 11.3. TX queue packet flow and operations

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

11.2.10. Polling mode and interrupts

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

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

11.2.11. Starting and stopping DMA

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

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

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

- Stop DMA transfers and DMA interrupts.
- Stop accepting new packets for transmission and reception. However the DMA functions will still be open for the user to retrieve sent and unsent TX packet buffers and to retrieve received and unused RX packet buffers.

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

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

11.3. Device Interface

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

11.3.1. Opening and closing device

A GRSPW device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `grspw_dev_count`. A particular device can be opened using `grspw_open` and closed `grspw_close`. The functions are described below.

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

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

- GRSPW device I/O registers are initialized to a state where most are zero.
- Descriptor tables memory for all DMA channels are allocated from the heap or from a user assigned address and cleared. The descriptor table length is always the maximum 0x400 Bytes for RX and TX.
- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the same device.

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

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

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

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

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

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

Table 11.3. `grspw_dev_count` function declaration

Proto	`int grspw_dev_count(void)`
About	Retrieve number of GRSPW devices registered to the driver.
Return	int. Number of GRSPW devices registered to driver, zero if none.
Notes	The number of GRSPW devices registered to the driver may or may not be equal to the number of devices in the system

Table 11.4. `grspw_open` function declaration

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

Table 11.5. `grspw_close` function declaration

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

11.3.2. Hardware capabilities

The features and capabilities present in hardware might not be symmetric in a system with several GRSPW devices. For example the two first GRSPW devices on the GR712RC implements RMAP whereas the others does not. The driver can read out the hardware capabilities and present it to the user. The set of functionality are determined

at design time. In some system where two or more systems are connected together it is likely to have different capabilities.

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

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

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

Table 11.6. `grspw_hw_support` function declaration

Proto	`void grspw_hw_support(void *d, struct grspw_hw_sup *hw)`
About	Get GRSPW hardware capabilities Write hardware capabilities of GRSPW device to user parameter *hw*.
Param	*d* [IN] pointer Device handle returned by `grspw_open`.
Param	*hw* [OUT] pointer Address to where the driver will write the hardware capabilities. Pointer must to memory and be valid.
Return	None.

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

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

Table 11.7. grspw_hw_sup data structure declaration

rmap	0	RMAP target functionality is not implemented in hardware.
	1	RMAP target functionality is implemented in hardware.
rmap_crc	Non-zero if RMAP CRC is available in hardware.	
rx_unalign	Non-zero if hardware can perform RX unalibned (byte boundary) DMA accesses.	
nports	Number of SpaceWire ports in hardware. Values: 1 or 2.	
ndma_chans	Number of DMA channels in hardware. Values: 1, 2, 3 or 4.	
hw_version	27..16	The 12-bits indicates GRLIB AMBA Plug & Play device ID of APB device. Indicates if GRSPW, GRSPW2 or GRSPW2_DMA.
	4..0	The 5 LSB bits indicates GRLIB AMBA Plug & Play device version of APB device. Indicates subversion of GRSPW or GRSPW2.
irq	Non-zero if SpaceWire distributed interrupt functionality is implemented in hardware.	

11.3.3. Link Control

The SpaceWire link is controlled and configured using the device API functions described below. The link control functionality is described in Section 11.2.3.

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

Table 11.8. `grspw_get_linkcfg` function declaration

Proto	`uint32_t grspw_get_linkcfg(void *d)`
About	Get link configuration The function returns the link configuration, which can be masked with the `LINKOPTS_*` defines.
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.
Return	uint32_t. Link configuration read from I/O registers

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

Table 11.9. `grspw_set_linkcfg` function declaration

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

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

Return	int. The function always returns DRV_OK.

Table 11.10. `grspw_get_clkdiv` function declaration

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

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

Table 11.11. `grspw_set_clkdiv` function declaration

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

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

Return	int. The function always returns DRV_OK.

Table 11.12. `grspw_link_state` function declaration

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

Value	Description.
ERRRST	Error reset.
ERRWAIT	Error Wait state.
READY	Error Wait state.
CONNECTING	Connecting state.
STARTED	Stated state.
RUN	Run state - link and DMA is fully operational.

Table 11.13. `grspw_get_status` function declaration

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

Table 11.14. `grspw_clear_status` function declaration

Proto	`void grspw_clear_status(void *d, uint32_t status)`
About	Clear bits in the status register

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

11.3.4. Node address configuration

This part for the device API controls the node address configuration of the RMAP target and DMA channels. The node address configuration functionality is described in Section 11.2.7. The data structures and functions involved in controlling the node address configuration are listed below.

```
struct grspw_addr_config {
 /* Ignore address field and put all received packets to first
  * DMA channel.
  */
 int8_t promiscuous;

 /* Default Node Address and Mask */
 uint8_t def_addr;
 uint8_t def_mask;
 /* DMA Channel custom Node Address and Mask */
 struct {
  int8_t node_en;    /* Enable Separate Addr */
  uint8_t node_addr; /* Node address */
  uint8_t node_mask; /* Node address mask */
 } dma_nacfg[4];
};
```

Table 11.15. grspw_addr_config data structure declaration

promiscuous	Enable (1) or disable (0) promiscous mode. The GRSPW will ignore the address field and put all received packets to first DMA channel. See hardware manual for. This field is also used to by the driver indicate if the settings should be written and read, or only read. See function description.	
def_addr	GRSPW default node address.	
def_mask	GRSPW default node address mask.	
dma_nacfg	DMA channel node address array configuration, see below field description. DMA channel N is described by *dma_nacfg[N]*.	
	Field	Description
	node_en	Enable (1) or disable (1) separate node address for DMA channel N (determined by array index).
	node_addr	Node address for DMA channel N (determined by array index).
	node_mask	Node address mask for DMA channel N (determined by array index).

Table 11.16. grspw_addr_ctrl function declaration

Proto	void grspw_addr_ctrl(void *d, const struct grspw_addr_config *cfg)
About	Set node address configuration The GRSPW device is either configured to have one single node address or a range of addresses by masking. The *cfg* input memory layout is described by the grspw_addr_config data structure in Table 11.15. When using multiple DMA channels one must assign each DMA channel a unique node address or a unique range by masking. Each DMA channel is represented by the input *dma_nacfg[N]*.
Param	*d* [IN] pointer Device handle returned by grspw_open.

Param	*cfg* [IN] pointer Address configuration to set.
Return	None.

11.3.5. Time-control codes

SpaceWire Time Code handling is controlled and configured using the device API functions described below. The Time Code functionality is described in Section 11.2.4.

Table 11.17. `grspw_get_tccfg` function declaration

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

Mask	Description
TCOPTS_EN_RX	Enable time-code receptions
TCOPTS_EN_TX	Enable time-code transmissions
TCOPTS_EN_RXIRQ	Generate interrupt when a valid time-code is received.

Table 11.18. `grspw_set_tccfg` function declaration

Proto	`void grspw_set_tccfg(void *d, uint32_t cfg)`
About	Set time-code configuration The function sets the time-code configuration in GRSPW control register.
Param	*d* [IN] pointer Device handle returned by `grspw_open`.
Param	*cfg* [IN] uint32_t Time-code configuration to write in I/O registers. The following masks can be used at configuration:

Mask	Description
TCOPTS_EN_RX	Enable time-code receptions
TCOPTS_EN_TX	Enable time-code transmissions
TCOPTS_EN_RXIRQ	Generate interrupt when a valid time-code is received.

Return	None.

Table 11.19. `grspw_get_tc` function declaration

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

	Mask	Description
	TCTRL_MASK	Time control flags of time register
	TIMECNT_MASK	Time counter of time register

11.3.6. Port Control

The SpaceWire port selection configuration, hardware support and current hardware status can be accessed using the device API functions described below. The SpaceWire port support functionality is described in Section 11.2.3.

In cases where only one SpaceWire port is implemented this part of the API can safely be ignored. The functions still deliver consistent information and error code failures when forcing Port1, however provides no real functionality.

Table 11.20. `grspw_port_ctrl` function declaration

Proto	`int grspw_port_ctrl(void *d, int *port)`	
About	Always read and optionally set port control settings of GRSPW device. The configuration determines how the hardware selects which SpaceWire port that is used. This is an optional feature in hardware to support one or two SpaceWire ports. An error is returned if operation not supported by hardware.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Param	*`port`* [IO] pointer to bit-mask The port configuration is first written if *`port`* does not point to -1. The port configuration is always read from the I/O registers and stored in the *`port`* address.	
	Value	Description
	-1	The current port configuration is read and stored into the *`port`* address.
	0	Force to use Port0.
	1	Force to use Port1.
	> 1	Hardware auto select between Port0 or Port1.
Return	Value. Description	
	0	Request successful.
	-1	Request failed. Port1 is not implemented in hardware.

Table 11.21. `grspw_port_count` function declaration

Proto	`int grspw_port_count(void *d)`	
About	Reads and returns number of ports that hardware supports.	
Param	*`d`* [IN] pointer Device identifier. Returned from `grspw_open`.	
Return	int. Number of ports implemented in hardware.	
	Value	Description
	1	One SpaceWire port is implemented in hardware. In this case `grspw_port_ctrl` function has no effect and `grspw_port_active` always returns 0.
	2	Two SpaceWire ports are implemented in hardware.

Table 11.22. `grspw_port_active` function declaration

Proto	`int grspw_port_active(void *d)`
About	Reads and returns the currently actively used SpaceWire port.
Param	*`d`* [IN] pointer

	Device identifier. Returned from `grspw_open`.	
Return	int. Currently active SpaceWire port	
	Value	Description
	0	SpaceWire port0 is active.
	1	SpaceWire port1 is active.

11.3.7. RMAP Control

The device API described below is used to configure the hardware supported RMAP target. The RMAP support is described in Section 11.2.5.

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

NOTE: When RMAP CRC is implemented in hardware it can be used to generate and append a CRC on a per packet basis. It is controlled by the DMA packet flags. Header and data CRC can be generated individually. See Table 11.32 for more information.

Table 11.23. `grspw_rmap_set_ctrl` function declaration

Proto	`int grspw_rmap_set_ctrl(void *d, uint32_t options)`	
About	Set RMAP configuration	
Param	*`d`* [IN] pointer Device handle returned by `grspw_open`.	
Param	*`options`* [IN] uint32_t RMAP control options to set in I/O registers. The following bit masks, prefixed with `RMAPOPTS_` shall be used.	
	Bit	Description
	EN_RMAP	Enable (1) or Disable (0) RMAP target handling in hardware.
	EN_BUF	Enable (0) or Disable (1) RMAP buffer. Disabling ensures that all RMAP requests are processed in the order they arrive.
Return	int. The function always returns DRV_OK.	

Table 11.24. `grspw_rmap_set_destkey` function declaration

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

11.3.8. Interrupt handling

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

The functions `grspw_dma_tx_count` and `grspw_dma_rx_count` can be used from interrupt context to determine how many TX/RX packets are (at least) available to the user. `grspw_get_status` can be used to

determine whether a new time count value (Tick Out) is available. Section 11.6 lists the API functions allowed to be called from ISR context.

11.4. DMA interface

This section covers how the driver can be interfaced to an application to send and transmit SpaceWire packets using the GRSPW hardware.

GRSPW2 and GRSPW2_DMA devices supports more than one DMA channel. The device channel zero is always present.

11.4.1. Opening and closing DMA channels

The first step before any SpaceWire packets can be transferred is to open a DMA channel to be used for transmission. As described in the device API Section 11.3.1 the GRSPW device the DMA channel belongs to must be opened and passed onto the DMA channel open routines.

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

An opened DMA channel can not be reopened unless the channel is closed first. When opening a channel the channel is marked opened by the driver. This procedure is thread-safe by protecting from other threads by using the operating system abstraction layer. Protection is used by all GRSPW devices on device opening, closing and DMA channel opening and closing.

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

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

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

```
int spw_receive_one_packet(void)
{
  void *device;
  void *dma0;
  int count;
  uint32_t linkcfg, clkdiv;
  spw_link_state_t state;
  struct grspw_list lst;

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

  /* Start Link */
  linkcfg = LINKOPTS_ENABLE | LINKOPTS_START; /* Start Link */
  grspw_set_linkcfg(device, linkcfg);
  clkdiv = (9 << 8) | 9; /* Clock Divisor factor of 10 */
  grspw_set_clkdiv(device, clkdiv);

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

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

  /* Initialize and activate DMA */
  if (DRV_OK != grspw_dma_start(dma0)) {
   grspw_dma_close(dma0);
   grspw_close(device);
   return -3;
  }
```

```
  /* ... */

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

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

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

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

  /* ... */

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

Table 11.25. `grspw_dma_open` function declaration

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

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

Notes	May block until other GRSPW device operations complete.

Table 11.26. `grspw_dma_close` function declaration

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

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

11.4.1.1. Static buffer allocation

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

Table 11.27. grspw_dma_open_userbuf function declaration

Proto	`void *grspw_dma_open_userbuf(void *d, int chan_no, struct grspw_ring *rx_ring, struct grspw_ring *tx_ring, struct grspw_rxbd *rx_bds, struct grspw_txbd *tx_bds)`
About	Opens a DMA channel of a previously opened GRSPW device. The GRSPW device is identified by its device handle *d* and the DMA channel is identified by index *chan_no*. The function requires the caller to provide buffers for the driver to use (*rx_ring* *tx_ring* *rx_bds* *tx_bds*). These memory areas shall not be referenced by the user as long as the DMA channel is opened. The areas can be reused when the channel has been closed with grspw_dma_close_userbuf. The returned value is used as input argument to all functions operating on the DMA channel.
Param	*d* [IN] pointer Device handle returned by grspw_open.
Param	*chan_no* [IN] Integer DMA channel identification number. DMA channels are indexed by 0, 1, 2 or 3. Other input values cause NULL to be returned. The index must be equal or greater than zero.
Param	*rx_ring* [IN] Pointer RX buffer ring area. Size shall be GRSPW_RXBD_NR * sizeof (struct grspw_ring), aligned to 32-bit word.
Param	*tx_ring* [IN] Pointer TX buffer ring area. Size shall be GRSPW_TXBD_NR * sizeof (struct grspw_ring), aligned to 32-bit word.
Param	*rx_bds* [IN] Pointer RX DMA buffer descriptor table area. Must be 1 KiB, and aligned to 1 KiB address boundary.
Param	*tx_bds* [IN] Pointer TX DMA buffer descriptor table area. Must be 1 KiB, and aligned to 1 KiB address boundary.
Return	Pointer. Status and driver's internal device identification.

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

Notes	May block until other GRSPW device operations complete.

Table 11.28. grspw_dma_close_userbuf function declaration

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

Return	int. Return code as indicated below.	
	Value	Description
	DRV_OK	Success.
	DRV_NOTOPEN	DMA channel `c` was not open.

11.4.2. Starting and stopping DMA operation

The start and stop operational modes are described in Section 11.2.11. The functions described below are used to change the operational mode of a DMA channels. A summary of which DMA API functions are affected are listed in Table 11.29, see function description for details on limitations.

Table 11.29. functions available in the two operational modes

Function	Stopped	Started
`grspw_dma_open`	N/A	N/A
`grspw_dma_close`	Yes	Yes
`grspw_dma_start`	Yes	No
`grspw_dma_stop`	No	Yes
`grspw_dma_rx_recv`	Yes, with limitations, see Section 11.4.6	Yes
`grspw_dma_rx_prepare`	Yes, with limitations, see Section 11.4.6	Yes
`grspw_dma_rx_flush`	Yes	No
`grspw_dma_tx_send`	Yes, with limitations, see Section 11.4.5	Yes
`grspw_dma_tx_reclaim`	Yes, with limitations, see Section 11.4.5	Yes
`grspw_dma_tx_flush`	Yes	No
`grspw_dma_config`	Yes	No
`grspw_dma_config_read`	Yes	Yes
`grspw_dma_stats_read`	Yes	Yes
`grspw_dma_stats_clr`	Yes	Yes

Table 11.30. `grspw_dma_start` function declaration

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

Param	*d* [IN] pointer Device handle returned by `grspw_open`.
Return	int. `DRV_STARTED` if channel was already started, else `DRV_OK`.

Table 11.31. `grspw_dma_stop` function declaration

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

11.4.3. Packet buffer description

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

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

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

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

Table 11.32. grspw_pkt data structure declaration

next	The packet structure can be part of a linked list. This field is used to point out the next packet in the list. Set to NULL if this packet is the last in the list or a single packet.
pkt_id	User assigned ID. This field is never touched by the driver. It can be used to store a pointer or other data to help implement the user buffer handling.
data	Data Buffer Address. DMA will read from this. The address must be 4-byte or byte aligned depending on hardware.
hdr	Header Buffer Address. DMA will read *`hlen`* bytes from this. The address must be 4-byte or byte aligned depending on hardware. This field is not used by RX operation.
dlen	Data payload lenght. The number of bytes hardware DMA read or written from/to the address indicated by the data pointer. On RX this is the complete packet data received.
flags	RX/TX transmission options and flags indicating resulting status. The bits described below is to be prefixed with TXPKT_FLAG_ or RXPKT_FLAG_ in order to match the TX or RX options definitionas declared by the driver's header file.

Bits	TX Description (prefixed TXPKT_FLAG_)

	NOCRC_MASK	Indicates to driver how many bytes shuld not be part of the header CRC calculation. 0 to 15 bytes can be omitted. Use NOCRC_LENN to select a specific lenght.
	IE	Enable (1) or Disable (0) IRQ generation on packet transmission completed.
	HCRC	Enable (1) or disable (0) Header CRC generation (if CRC is available in hardware). Header CRC will be appended (one byte at end of header).
	DCRC	Enable (1) or disable (0) Data CRC generation (if CRC is available in hardware). Data CRC will be appended (one byte at end of packet).
	TX	Is set by the driver to indicate that the packet was transmitted. This does no signal a successful transmission, but that transmission was attempted, the LINKERR bit needs to be checked for error indication.
	LINKERR	Set if a link error was exibited during transmission of this packet.
	Bits	RX Description (prefixed RXPKT_FLAG_)
	IE	Enable (1) or Disable (0) IRQ generation on packet reception completed.
	TRUNK	Set if packet was truncated.
	DCRC	Set if data CRC error detected (only valid if RMAP CRC is enabled).
	HCRC	Set if header CRC error detected (only valid if RMAP CRC is enabled).
	EEOP	Set if an End-of-Packet error occured during reception of this packet.
	RX	Marks if packet was recevied or not.
hlen	Header length. The number of bytes hardware will transfer using DMA from the address indicated by the hdr pointer. This field is not used by RX operation.	

11.4.4. Packet buffer lists

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

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

- `grspw_list_clr` initializes a list.
- `grspw_list_is_empty` determines if a list is empty.
- `grspw_list_append` appends a packet to the end of a list.
- `grspw_list_append_list` appends packets from one list to the end of another list.

11.4.5. Sending packets

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

Transmission of SpaceWire packets are described in Section 11.2.1.

In the below example Figure 11.4 three SpaceWire packets are scheduled for transmission. The *`count`* should be set to three. The second packet is programmed to generate an interrupt when transmission finished, GRSPW hardware will also generate a header CRC using the RMAP CRC algorithm resulting in a 16 bytes long SpaceWire packet.

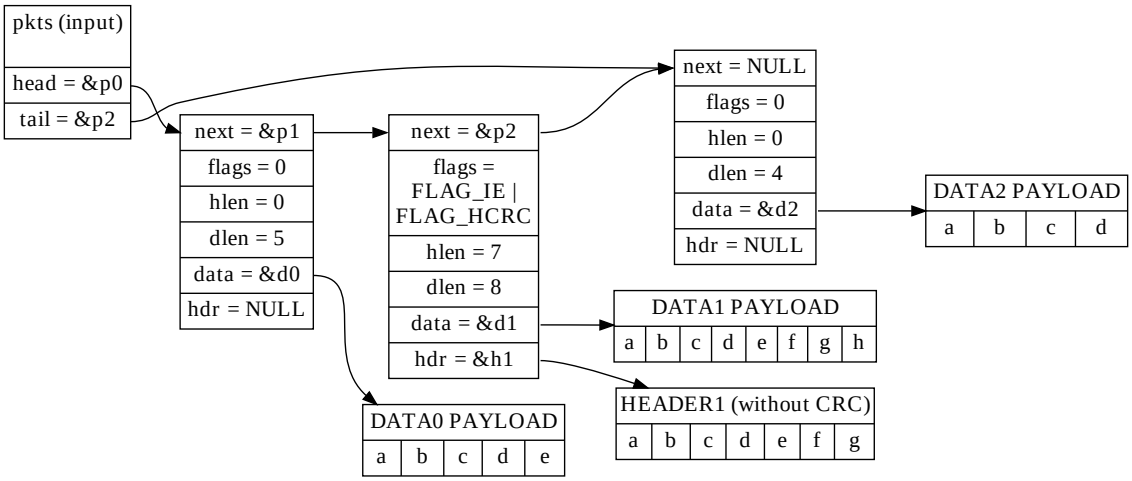

Figure 11.4. TX packet description `pkts` input to `grspw_tx_dma_send`

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

Table 11.33. `grspw_dma_tx_send` function declaration

Proto	`int grspw_dma_tx_send(void *c, struct grspw_list *pkts)`
About	Schedule list of packets for transmission at some point in future. The GRSPW transmitter is enabled when packets are added to the TX SCHED queue. (USER->SCHED) The fastest solution in retrieving sent TX packets and sending new frames is to call: 1. grspw_dma_tx_reclaim(opts=0) 2. grspw_dma_tx_send(opts=1) NOTE: the TXPKT_FLAG_TX flag must not be set in the packet structure.
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*`pkts`* [IN] pointer A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that *`head`* points to the first packet and *`tail`* points to the last. The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 11.32. Note that TXPKT_FLAG_TX of the *`flags`* field must not be set.
Return	int. See return codes below

Value	Description
-1	Error: DMA channel is not in started mode.
>=0	Successfully added pkts to TX SCHED list.

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

Table 11.34. `grspw_dma_tx_reclaim` function declaration

Proto	`int grspw_dma_tx_reclaim(void *c, struct grspw_list *pkts)`
About	Reclaim TX packet buffers that has previously been scheduled for transmission with `grspw_dma_tx_send`. The packets in the SCHED queue which have been transmitted are moved to the *`pkts`* packet list. The user *`pkts`* list is not cleared by the function. When the move has been completed the packet can

<table>
<tr><td></td><td>safely be reused again by the user. The packet structures have been updated with transmission status to indicate transfer failures of individual packets.

The typical solution for retrieving sent TX packets and sending new frames is to call:
1. grspw_dma_tx_reclaim()
2. grspw_dma_tx_send()

NOTE: the TXPKT_FLAG_TX flag indicates if the packet was transmitted.</td></tr>
<tr><td>Param</td><td>c [IN] pointer
DMA channel handle returned by grspw_dma_open.</td></tr>
<tr><td>Param</td><td>pkts [OUT] pointer
Sent TX packets will be taken from the SCHED queue and added to the pkts queue. The user queue pkts is not cleared.

The layout and content of the packet is defined by the grspw_pkt data structure is described in Table 11.32. Note that TXPKT_FLAG_TX of the flags field indicates if the packet was sent of not. In case of DMA being stopped one can use this flag to see if the packet was transmitted or not. The TXPKT_FLAG_LINKERR indicates if a link error occurred during transmission of the packet, regardless the TXPKT_FLAG_TX is set to indicate packet transmission attempt.</td></tr>
<tr><td>Return</td><td>int. See return codes below
<table><tr><td>Value</td><td>Description</td></tr><tr><td>-1</td><td>Error: DMA channel is not in started mode.</td></tr><tr><td>0</td><td>No packet reclaimed (SCHED list contains no sent packets).</td></tr><tr><td>>0</td><td>Number of packets successfully reclaimed to user list.</td></tr></table></td></tr>
<tr><td>Notes</td><td>This function can operate in stopped mode. This is useful when a link goes down and the DMA activity is stopped by user or by driver automatically.</td></tr>
</table>

11.4.6. Receiving packets

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

Reception of SpaceWire packets are described in Section 11.2.1.

In the Figure 11.5 example three SpaceWire packets are received. The *count* parameters is set to three by the driver to reflect the number of packets. The first packet exhibited an early end-of-packet during reception which also resulted in header and data CRC error. All header pointers and header lengths have been set to zero by the user since they are no used, however the values in those fields does not affect the RX operations. The RX flag is set to indicate that DMA transfer was performed.

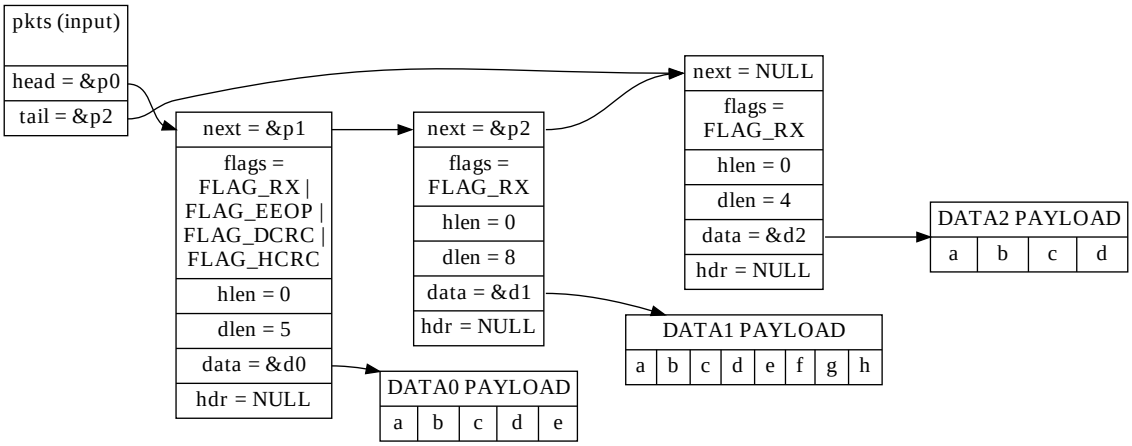

Figure 11.5. RX packet output from `grspw_dma_rx_recv`

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

Table 11.35. `grspw_dma_rx_prepare` function declaration

Proto	`int grspw_dma_rx_prepare(void *c, struct grspw_list *pkts)`
About	Add RX packet buffers for future reception. The received packets can later be read out with `grspw_dma_rx_recv`. The packets in `pkts` list are put to the SCHED queue of the driver (USER->SCHED). The typical solution for retreiving received RX packets and preparing new packet buffers for future receive, is to call: 1. grspw_dma_rx_recv(&recvlist) 2. grspw_dma_rx_prepare(&freelist) NOTE: the RXPKT_FLAG_RX flag must not be set in the packet structure.
Param	`c` [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	`pkts` [IN] pointer A linked list of initialized SpaceWire packets. The grspw_list structure must be initialized so that `head` points to the first packet and `tail` points to the last. The layout and content of the packet is defined by the grspw_pkt data structure described in Table 11.32. Note that RXPKT_FLAG_RX of the `flags` field must not be set.
Return	int. See return codes below

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

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

Table 11.36. `grspw_dma_rx_recv` function declaration

Proto	`int grspw_dma_rx_recv(void *c, struct grspw_list *pkts)`
About	Get received RX packet buffers which have previously been scheduled for reception with `grspw_dma_rx_prepare`.

	The packets in the RX SCHED queue which have been received are moved to the *pkts* packet list (SCHED->USER). When the move has been completed the packet(s) can safely be reused again by the user. The packet structures have been updated with reception status to indicate transfer failures of individual packets and received packet length. The header pointer and length fields are not touched by the driver, all data ends up in the data area. NOTE: the RXPKT_FLAG_RX flag indicates if a packet was received, thus if the data field contains new valid data or not.
Param	*c* [IN] pointer DMA channel handle returned by `grspw_dma_open`.
Param	*pkts* [OUT] pointer Received RX packets will be taken from the SCHED queue and added to the *pkts* queue. The user queue *pkts* is not cleared. The layout and content of the packet is defined by the grspw_pkt data structure described in Table 11.32. Note that RXPKT_FLAG_RX of the *flags* field indicates if the packet was received or not. In case of DMA being stopped one can use this flag to see if the packet was received or not. The TRUNK, DCRC, HCRC and EEOP flags indicates if an error occured during transmission of the packet, regardless the RXPKT_FLAG_RX is set to indicate packet reception attempt.
Return	int. See return codes below

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

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

11.4.7. Transmission queue status

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

Table 11.37. `grspw_dma_tx_count` function declaration

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

Table 11.38. `grspw_dma_rx_count` function declaration

Proto	`int grspw_dma_rx_count(void *c)`
About	Get number of packets received by hardware but not yet retrieved by the driver. This is determined by looking at the RX descriptor pointer register. The number represents how many of the prepared packets that actually have been received by hardware but not handled by the driver yet.
Param	*c* [IN] pointer

	DMA channel handle returned by `grspw_dma_open`.
Return	int. The number of packets received by hardware but not yet retrieved by the driver.
Notes	This function can be called from interrupt context.

11.4.8. Queue flushing

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

Table 11.39. `grspw_dma_tx_flush` function declaration

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

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

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

Table 11.40. `grspw_dma_rx_flush` function declaration

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

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

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

11.4.9. Statistics

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

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

Table 11.41. grspw_dma_stats data structure declaration

tx_pkts	Number of transmitted packets with link errors.
tx_err_link	Number of transmitted packets with link errors.
rx_pkts	Number of received packets.
rx_err_trunk	Number of received Truncated packets.
rx_err_endpkt	Number of received packets with bad ending.

Table 11.42. `grspw_dma_stats_read` function declaration

Proto	`void grspw_dma_stats_read(void *c, struct grspw_dma_stats *sts)`
About	Reads the current driver statistics collected from earlier events by a DMA channel and DMA channel usage. The statistics are stored to the address given by the second argument. The layout and content of the statistics are defined by the grspw_dma_stats data structure is described in Table 11.41. Note that the snapshot is taken without lock protection, as a consequence the statistics may not be synchonized with each other. This could be caused if the function is interrupted by a the GRSPW interrupt or other tasks performing driver operations on the same DMA channel.
Param	*c* [IN] pointer DMA channel identifier. Returned from `grspw_dma_open`.
Param	*sts* [OUT] pointer A snapshot of the current driver statistics are copied to this user provided buffer. The layout and content of the statistics are defined by the grspw_dma_stats data structure is described in Table 11.41.
Return	None.

Table 11.43. `grspw_dma_stats_clr` function declaration

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

11.4.10. DMA channel configuration

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

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

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

Table 11.44. grspw_dma_config data structure declaration

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

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

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

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

Table 11.45. `grspw_dma_config` function declaration

Proto	`int grspw_dma_config(void *c, struct grspw_dma_config *cfg)`	
About	Set the DMA channel configuration options as described by the input arguments. It is only possible the change the configuration on stopped DMA channels, otherwise an error code is returned. The hardware registers are not written directly. The settings take effect when the DMA channel is started calling `grspw_dma_start`.	
Param	*`c`* [IN] pointer DMA channel handle returned by `grspw_dma_open`.	
Param	*`cfg`* [IN] pointer Address to where the driver will read the DMA channel configuration from. The configuration options are described in Table 11.44.	
Return	int. Return code as indicated below.	
	Value	Description
	DRV_OK	Success.
	DRV_FAIL	Failure due to invalid input arguments or DMA has already been started.

Table 11.46. `grspw_dma_config_read` function declaration

Proto	`void grspw_dma_config_read(void *c, struct grspw_dma_config *cfg)`
About	Copies the DMA channel configuration to user defined memory area.

Param	*c* [IN] pointer DMA channel handle returned by grspw_dma_open.
Param	*sts* [OUT] pointer The driver DMA channel configuration options are copied to this user provided buffer. The layout and content of the statistics are defined by the grpsw_dma_config data structure is described in Table 11.44.
Return	int. Return code as indicated below.

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

11.4.11. DMA channel status

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

The following status information is available:

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

Table 11.47. grspw_dma_get_status function declaration

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

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

Table 11.48. grspw_dma_clear_status function declaration

Proto	`void grspw_dma_clear_status(void *c, uint32_t status)`
About	Clear DMA channel status The function clears the status bits in GRSPW DMA control register corresponding to the bits set in the *status* parameter. Current status can be retrieved with the function grspw_dma_get_status.
Param	*c* [IN] pointer

	DMA channel handle returned by `grspw_dma_open`.
Param	*`status`* [IN] uint32_t Mask of DMA channel status bits to clear in GRSPW DMA control register. The bit masks are the same as the masks for `grspw_dma_get_status` return value.
Return	None.

11.5. API reference

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

11.5.1. Data structures

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

Table 11.49. Data structures reference

Data structure name	Section
struct grspw_pkt	11.4.3
struct grspw_addr_config	11.3.4
struct grspw_hw_sup	11.3.2
struct grspw_dma_stats	11.4.9
struct grspw_dma_config	11.4.10

11.5.2. Device functions

The GRSPW device API. The functions listed in the table below operates on the GRSPW common registers and driver set up. Changes here typically affects all DMA channels and link properties .

Table 11.50. Device function reference

Prototype	Section
`int grspw_dev_count(void)`	11.3.1
`void *grspw_open(int dev_no)`	11.3.1
`int grspw_close(void *d)`	11.3.1
`void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)`	11.3.4,
`spw_link_state_t grspw_link_state(void *d)`	11.3.3,
`uint32_t grspw_get_linkcfg(void *d)`	11.3.3,
`int grspw_set_linkcfg(void *d, uint32_t cfg)`	11.3.3,
`uint32_t grspw_get_clkdiv(void *d)`	11.3.3,
`int grspw_set_clkdiv(void *d, uint32_t clkdiv)`	11.3.3,
`uint32_t grspw_get_status(void *d)`	11.3.3,
`void grspw_clear_status(void *d, uint32_t status)`	11.3.3,
`uint32_t grspw_get_tccfg(void *d)`	11.3.5,
`void grspw_set_tccfg(void *d, uint32_t cfg)`	11.3.5,
`uint32_t grspw_get_tc(void *d)`	11.3.5,

11.5.3. DMA functions

The GRSPW DMA channel API. The functions listed in the table below operates on one GRSPW DMA channel and its driver set up. This interface is used to send and receive SpaceWire packets.

GRSPW2 and GRSPW2_DMA devices supports more than one DMA channel.

Table 11.51. DMA channel function reference

Prototype	Section
`void *grspw_dma_open(void *d, int chan_no)`	11.4.1, 11.3.1
`void grspw_dma_close(void *c)`	11.4.1, 11.3.1
`void *grspw_dma_open_userbuf(void *d, int chan_no, struct grspw_ring *rx_ring, struct grspw_ring *tx_ring, struct grspw_rxbd *rx_bds, struct grspw_txbd *tx_bds)`	11.4.1, 11.3.1
`void grspw_dma_close_userbuf(void *c)`	11.4.1, 11.3.1
`int grspw_dma_start(void *c)`	11.4.2,
`void grspw_dma_stop(void *c)`	11.4.2,
`int grspw_dma_rx_recv(void *c, struct grspw_list *pkts)`	11.4.6,
`int grspw_dma_rx_prepare(void *c, struct grspw_list *pkts)`	11.4.6,
`int grspw_dma_rx_flush(void *c, struct grspw_list *pkts)`	11.4.8,
`int grspw_dma_tx_send(void *c, struct grspw_list *pkts)`	11.4.5,
`int grspw_dma_tx_reclaim(void *c, struct grspw_list *pkts)`	11.4.5,
`int grspw_dma_tx_flush(void *c, struct grspw_list *pkts)`	11.4.8,
`void grspw_dma_stats_read(void *c, struct grspw_dma_stats *sts)`	11.4.9
`void grspw_dma_stats_clear(void *c)`	11.4.9
`int grspw_dma_config(void *c, struct grspw_dma_config *cfg)`	11.4.10
`int grspw_dma_config_read(void *c, struct grspw_dma_config *cfg)`	11.4.10
`uint32_t grspw_dma_get_status(void *c)`	11.4.11
`void grspw_dma_clear_status(void *c, uint32_t status)`	11.4.11

11.6. Restrictions

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

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

12. GRCAN CAN driver

12.1. Introduction

This section describes the driver used to control the GRLIB GRCAN device for CAN DMA operation.

12.1.1. User Interface

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

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

All driver functions are non-blocking.

12.1.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 12.1. Driver registration functions

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

12.1.3. Examples

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

12.1.4. Known driver limitations

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

12.2. Opening and closing device

A GRCAN device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `grcan_dev_count`. A particular device can be opened using `grcan_open` and closed `grcan_close`. The functions are described below.

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

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

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

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

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

        count = grcan_dev_count();
```

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

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

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

Table 12.2. `grcan_dev_count` function declaration

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

Table 12.3. `grcan_open` function declaration

Proto	`struct grcan_priv *grcan_open(int dev_no)`	
About	Opens a GRCAN device. The GRCAN device is identified by index. The returned value is used as input argument to all functions operating on the device. The function allocates DMA buffers as necessary using dynamic memory allocation (`malloc()`).	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the registration order to the driver, normally dictated by the Plug & Play order. Must be equal or greater than zero, and smaller than that returned by `grcan_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device semaphore fails or device already is open.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which GRCAN device.

Table 12.4. `grcan_close` function declaration

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

12.2.1. Static buffer allocation

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

Table 12.5. `grcan_open_userbuf` function declaration

Proto	`struct grcan_priv *grcan_open_userbuf(int dev_no, void *rxbuf, int rxbuf_size, void *txbuf, int txbuf_size)`
About	Opens a GRCAN device. The GRCAN device is identified by index. The returned value is used as input argument to all functions operating on the device. The function requires the caller to provide DMA buffers for the driver to use (*`rxbuf`* and *`txbuf`*). These memory areas shall not be referenced by the user as long as the driver channel is opened. The areas can be reused when the driver has been closed with `grcan_close_userbuf`.

Param	*dev_no* [IN] Integer Device identification number. Devices are indexed by the registration order to the driver, normally dictated by the Plug & Play order. Must be equal or greater than zero, and smaller than that returned by `grcan_dev_count`.
Param	*rxbuf* [IN] Pointer RX DMA buffer address. Must be aligned to 1 KiB address boundary.
Param	*rxbuf_size* [IN] Integer RX DMA buffer size in bytes. Must be a multiple of 64.
Param	*txbuf* [IN] Pointer TX DMA buffer address. Must be aligned to 1 KiB address boundary.
Param	*txbuf_size* [IN] Integer TX DMA buffer size in bytes. Must be a multiple of 64.
Return	Pointer. Status and driver's internal device identification.
	NULL: Indicates failure to open device. Fails if device semaphore fails or device already is open.
	Pointer: Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which GRCAN device.

Table 12.6. `grcan_close_userbuf` function declaration

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

12.3. Operation mode

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

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

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

12.3.1. Starting and stopping

The `grcan_start()` function places the CAN core in STATE_STARTED mode. Configuration set by previous driver function calls are committed to hardware before started mode enters. It is necessary to enter started mode to be able to receive and transmit messages on the CAN bus. The `grcan_start()` function call will fail if receive or transmit buffers are not correctly allocated or if the CAN core is already is in started mode.

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

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

Table 12.7. grcan_get_state function declaration

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

12.4. Configuration

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

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

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

12.4.1. Channel selection

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

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

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

Table 12.8. grcan_selection member description

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

12.4.2. Timing parameters

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

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

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

Table 12.9. grcan_timing member description

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

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

12.5. Receive filters

12.5.1. Data structures

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

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

Table 12.10. grcan_filter member description

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

12.5.2. Acceptance filter

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

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

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

12.5.3. Sync filter

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

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

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

12.6. Driver statistics

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

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

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

```
struct grcan_stats {
        unsigned int rxsync_cnt;
        unsigned int txsync_cnt;
        unsigned int ahberr_cnt;
        unsigned int ints;
        unsigned int busoff_cnt;
};
```

Table 12.11. grcan_stats member description

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

12.7. Device status

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

Table 12.12. Device status word bit masks

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

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

12.8. CAN bus transfers

12.8.1. Data structures

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

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

Table 12.13. struct grcan_canmsg member description

Member	Description
extended	Indicates whether the CAN message has 29 or 11 bits ID tag. Extended or Standard frame.
rtr	Remote Transmission Request bit.
len	Length of *data*.
data	CAN message data, *data[0]* is the most significant byte – the first byte.
id	The ID field of the CAN message. An extended frame has 29 bits whereas a standard frame has only 11-bits. The most significant bits are not used.

12.8.2. Transmission

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

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

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

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

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

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

Table 12.14. `grcan_write` function declaration

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

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

12.8.3. Reception

CAN messages are received using the `grcan_read()` function. An example is shown below:

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

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

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

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

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

Table 12.15. `grcan_read` function declaration

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

	GRCAN_RET_AHBERR	Similar to BUSOFF, but was caused by AHB Error.

12.8.4. Bus-off recovery

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

12.8.5. AHB error recovery

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

12.9. Interrupt API

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

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

Table 12.16. `grcan_set_isr` function declaration

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

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

12.9.1. Interrupt generation

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

Table 12.17. `grcan_txint` function declaration

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

Table 12.18. `grcan_rxint` function declaration

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

13. UART driver

13.1. Introduction

This section describes the driver used to control the APBUART devices. The driver supports operation in interrupt or non-interrupt mode.

13.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 13.1. Driver registration functions

Registration method	**Function**
Automatic registration	`apbuart_autoinit()`
Register one device	`apbuart_register()`
Register many devices	`apbuart_init()`

13.3. Opening and closing device

An APBUART device must first be opened and configured before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `apbuart_dev_count`. A particular device can be opened using `apbuart_open` and closed using `apbuart_close`. The functions are described below.

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

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

- APBUART device I/O registers are initialized, including disabling interrupts generation and disabling transmitter and receiver.
- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the same device.

Table 13.2. `apbuart_dev_count` function declaration

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

Table 13.3. `apbuart_open` function declaration

Proto	`struct apbuart_priv *apbuart_open(int dev_no)`	
About	Open an APBUART device The APBUART device is identified by index. The returned value is used as input argument to all functions operating on the device.	
Param	*`dev_no`* [IN] Integer Device identification number. Must be equal to or greater than zero, and smaller than value returned by `apbuart_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device is already open or if *`dev_no`* is out of range.
	Pointer	APBUART device handle to use as input parameter to all device API functions for the opened device.

Table 13.4. `apbuart_close` function declaration

Proto	`int apbuart_close(struct apbuart_priv *priv)`
About	Close an APBUART device The transmitter and receiver are disabled.
Param	*`d`* [IN] pointer Device handle returned by `apbuart_open`.
Return	int.

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

13.4. Status interface

APBUART status can be read by calling the `apbuart_get_status` function. It returns a copy of the UART status register. The APBUART status register can be written with the function `apbuart_set_status`.

Table 13.5. `apbuart_get_status` function declaration

Proto	`uint32_t apbuart_get_status(struct apbuart_priv *priv)`
About	Read APBUART status register
Param	*`d`* [IN] pointer Device handler returned by `apbuart_open`.
Return	uint32_t. Copy of UART status register for device *`d`* Register definitions for the APBUART status register are available in the file `include/regs/apbuart-regs.h`. The relevant defines are prefixed with *`APBUART_STATUS_`*.

Table 13.6. `apbuart_set_status` function declaration

Proto	`void apbuart_set_status(struct apbuart_priv *priv, uint32_t status)`
About	Set APBUART status register Parameter *`status`* is written to the APBUART status register.
Param	*`d`* [IN] pointer Device handle returned by `spi_open`.
Param	*`status`* [IN] uint32_t Value to write to the status register. Register definitions for the APBUART status register are available in the file `include/regs/apbuart-regs.h`. The relevant defines are prefixed with *`APBUART_STATUS_`*.
Return	None.

13.5. Configuration interface

After opening the device, but before performing transfers, the UART device must be configured. The UART driver supports configuring baud, parity, flow control and interrupt operation mode individually for each device. UART receiver and transmitter is enabled when the device is configured with `apbuart_config`.

The baud, and parity and flow control configuration parameters are applicable to both non-interrupt and interrupt operation mode. Transmit and receive buffer configuration is only applicable when operating in interrupt mode, as described in Section 13.7.

A device can only be configured once after it is opened. If the UART device needs reconfiguration, the device must first be closed, and then opened and then configured again. An attempt to reconfigure an already configured device will result in a defined return value from `apbuart_config`

```
struct apbuart_config {
 int baud;
 int parity;
 int mode;
 int flow;
 uint8_t *txfifobuf;
 int txfifobuflen;
 uint8_t *rxfifobuf;
 int rxfifobuflen;
};
```

Table 13.7. apbuart_config data structure declaration

baud	UART baud, bits per second	
parity	Selects parity mode. Must be one of the following values:	
	Value	Description
	UART_PAR_NONE	Disable parity bit generation and checking.
	UART_PAR_EVEN	Enable even parity bit generation and checking.
	UART_PAR_ODD	Enable odd parity bit generation and checking.
mode	Selects between interrupt or non-interrupt operation mode. Must be one of the following values:	
	Value	Description
	UART_MODE_NONINT	Non-interrupt operation mode
	UART_MODE_INT	Interrupt operation mode
flow	Enables or disabled flow control. Must be one of the following values:	
	Value	Description
	0	Flow control disabled
	1	Flow control enabled
txfifobuf	Buffer area for TX SW FIFO	
txfifobuflen	Number of bytes allocated for TX SW FIFO	
rxfifobuf	Buffer area for RX SW FIFO	
rxfifobuflen	Number of bytes allocated for RX SW FIFO	

Table 13.8. `apbuart_config` function declaration

Proto	`int apbuart_config(struct apbuart_priv *priv, struct apbuart_config *cfg)`	
About	Configure APBUART device. The *`cfg`* input layout is described by the apbuart_config data structure in Table 13.7. If interrupt mode is configured, then the driver will register an ISR with help of OSAL. If the device has already been configured, this function returns *`DRV_BUSY`* and no hardware or software state is changed. Operation on the device can continue as if the function was never called.	
Param	*`d`* [IN] pointer Device handle returned by `apbuart_open`.	
Param	*`cfg`* [IN] pointer Pointer to configuration structure. (See Table 13.7.)	
Return	int.	
	Value	Description

	DRV_OK	Device configured successfully.
	DRV_BUSY	Device has already been configured.
	others	Failed to register UART ISR: device configuration aborted.
Notes	A UART device must be configured with `apbuart_config` before performing transfers using the device.	

13.6. Non-interrupt interface

One receive and one transmit function is available when operating in non-interrupt mode. Both are non-blocking and operate on one character per function call. `apbuart_outbyte` is used to transmit one byte of data and `apbuart_inbyte` is used to receive one byte. If the hardware transmit or receive FIFO is not ready then no data is transferred and the user is informed.

As the APBUART implements a hardware transmit FIFO, a successful return from `apbuart_outbyte` does not guarantee that the data has been yet been sent on the medium. The `apbuart_get_status` service can be used to determine if all scheduled transmit bytes have left the APBUART controller.

NOTE: For high performance transfers, or large transfers, the UART driver should be operated in interrupt mode.

The example below opens and configures the first APBUART device in non-interrupt mode. Then one data byte is written and one is read.

```
int apbuart_nonint_example()
{
  struct apbuart_priv *device;
  int count;
  int i;
  int data;
  struct apbuart_config cfg;

  count = apbuart_dev_count();
  printf("%d APBUART devices present\n", count);

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

  cfg.baud = 38400;
  cfg.parity = UART_PAR_NONE;
  cfg.flow = 0;
  cfg.mode = UART_MODE_NONINT;
  /* SW FIFO parameters are not used in non-interrupt mode. */
  apbuart_config(device, &cfg);

  i = 0;
  do {
   i = apbuart_outbyte(device, 'a');
  } while (1 != i);

  do {
   data = apbuart_inbyte(device);
  } while (data < 0);
  printf("Received 0x%x\n", data);

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

Table 13.9. `apbuart_outbyte` function declaration

Proto	`int apbuart_outbyte(struct apbuart_priv *priv, uint8_t data)`
About	Send one data byte The function will try to send one data byte on the UART. The operation is non-blocking and returns 0 if the transmit FIFO is full.
Param	*d* [IN] pointer Device handle returned by apbuart_open.

Param	*data* [IN] uint8_t Data byte to send	
Return	int. Number of bytes copied to transmit FIFO.	
	0	The data byte was not sent.
	1	The data byte was sent.
Notes	Transfer properties are set with the function `apbuart_config`.	

Table 13.10. `apbuart_inbyte` function declaration

Proto	`int apbuart_inbyte(struct apbuart_priv *priv)`
About	Receive one data byte The function tries to receive one byte of data from the UART receive FIFO. The operation is non-blocking and returns -1 if the transmit FIFO is empty.
Param	*d* [IN] pointer Device handle returned by `apbuart_open`.
Return	int. The received data byte, as uint8_t casted to an int. If no data byte was available then -1 is returned.
Notes	Transfer properties are set with the function `apbuart_config`.

13.7. Interrupt interface

Multiple bytes can be handled at once when transmitting and receiving with the UART driver in interrupt mode. An interrupt service routine, provided by the driver, is responsible for maintaining the hardware FIFOs.

Sending data is done by calling `apbuart_write` with a pointer to the data to be transmitted together with a count of bytes to send. The number of bytes accepted by the driver is returned by the function. The function does not block.

Receiving data is done by calling `apbuart_read` with a pointer to the destination data location and the maximum number of bytes to send. The number of bytes written to the destination is returned by the function. The function does not block.

The example below opens and configures the first APBUART device in interrupt mode. Then 4 data bytes are written and 4 are read.

```
int apbuart_int_example()
{
  static uint8_t txfifobuf[32];
  static uint8_t rxfifobuf[32];
  uint8_t userdata[4] = {'A', 'B', 'C', 'D'};
  struct apbuart_priv *device;
  int count;
  int i, j;
  struct apbuart_config cfg;

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

  cfg.baud = 38400;
  cfg.parity = UART_PAR_NONE;
  cfg.flow = 0;
  cfg.mode = UART_MODE_INT;
  cfg.txfifobuflen = 32;
  cfg.txfifobuf = txfifobuf;
  cfg.rxfifobuflen = 32;
  cfg.rxfifobuf = rxfifobuf;
  apbuart_config(device, &cfg);

  i = apbuart_write(device, userdata, 4);
  j = apbuart_read(device, userdata, 4);

  printf("Sent %i bytes, received %i bytes\n", i, j);
```

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

Table 13.11. `apbuart_write` function declaration

Proto	`int apbuart_write(struct apbuart_priv *priv, const uint8_t *buf, int count)`
About	Send zero or more data bytes This function sends up to `count` data bytes from `buf` to the UART associated with the device handle `d`. Number of bytes actually sent can be less than `count` if the hardware and software TX FIFOs become full. The operation is non-blocking.
Param	`d` [IN] pointer Device handle returned by `apbuart_open`.
Param	`buf` [IN] pointer Data bytes to send
Param	`count` [IN] Integer Number of bytes to send
Return	int. Number of bytes actually sent, which may be less than `count` if FIFOs become full.

Table 13.12. `apbuart_read` function declaration

Proto	`int apbuart_read(struct apbuart_priv *priv, uint8_t *buf, int count)`
About	Receive zero or more data bytes This function receives up to `count` bytes from the UARTs associated with the device handle `d` and stores the data at location `buf`. Number of bytes actually received can be less than `count`. The operation is non-blocking.
Param	`d` [IN] pointer Device handle returned by `apbuart_open`.
Param	`buf` [IN] pointer Receive buffer
Param	`count` [IN] Integer Number of bytes to receive
Return	int. Number of bytes actually received, which may be less than `count` if FIFOs become empty.

13.8. Restrictions

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

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

- `apbuart_dev_count`
- `apbuart_open`

As the UART driver implements its own ISR, it does not support custom user ISR:s.

14. SPI driver

14.1. Introduction

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

14.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 14.1. Driver registration functions

Registration method	**Function**
Automatic registration	`spi_autoinit()`
Register one device	`spi_register()`
Register many devices	`spi_init()`

14.3. Opening and closing device

A SPICTRL device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `spi_dev_count`. A particular device can be opened using `spi_open` and closed `spi_close`. The functions are described below.

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

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

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

Table 14.2. `spi_dev_count` function declaration

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

Table 14.3. `spi_open` function declaration

Proto	`struct spi_priv *spi_open(int dev_no)`	
About	Opens a SPICTRL device. The SPICTRL device is identified by index. The returned value is used as input argument to all functions operating on the device.	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the order registered to the driver. Must be equal to or greater than zero, and smaller than that returned by `spi_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device is already open, or invalid *`dev_no`* parameter.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, idenfifies which SPICTRL device.

Table 14.4. `spi_close` function declaration

Proto	`int spi_close(struct spi_priv *priv)`

About	Closes a previously opened device.
Param	*d* [IN] pointer Device identifier. Returned from `spi_open`.
Return	int.

Value	Description
DRV_OK	Successfully closed device.

14.4. Status service

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

Table 14.5. `spi_get_event` function declaration

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

Table 14.6. `spi_clear_event` function declaration

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

14.5. Transfer Configuration

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

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

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

```
struct spi_config {
        unsigned int freq;              /* SPI clock frequency, Hz */
```

```
        int mode;                       /* SPI mode */
        enum spi_wordlen wordlen;       /* SPI Word length */
        int intmask;                    /* SPI controller interrupt mask */
        int msb_first;                  /* If true then send MSb first, else LSb. */
        int sync;                       /* Synchronous TX/RX mode */
        uint32_t aslave;                /* Automatic slave select, active high mask */
        int clock_gap;                  /* MODE.CG */
        int tac;                        /* Toggle automatic slave select during clock gap */
        int aseldel;                    /* Automatic slave select delay */
        int igsel;                      /* Ignore SPISEL input */
};
```

Table 14.7. spi_config data structure declaration

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

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

Table 14.8. `spi_config` function declaration

Proto	`int spi_config(struct spi_priv *priv, struct spi_config *cfg)`	
About	Set transfer configuration in hardware. The *`cfg`* input layout is described by the spi_config data structure in Table 14.7.	
Param	*`d`* [IN] pointer Device identifier. Returned from `spi_open`.	
Param	*`cfg`* [IN] pointer Address to where the driver will read the transfer configuration from. (See Table 14.7.)	
Return	int.	
	Value	Description
	DRV_OK	Successfully configured device.
	DRV_FAIL	Invalid word length or frequency field in *`cfg`*. Device not configured.
	DRV_STARTED	Device is in started mode. Device not configured.

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

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

It can be used to derive default parameters.

14.6. Transfer Interface

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

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

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

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

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

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

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

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

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

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

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

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

Table 14.9. `spi_start` function declaration

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

Table 14.10. `spi_stop` function declaration

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

	DRV_OK	Success

Table 14.11. `spi_write32` function declaration

Proto	`int spi_write32(struct spi_priv *priv, const uint32_t *txbuf, int count)`
About	Write words to SPICTRL transmit queue. The function tries to write *`count`* words of the configured word length to the transmit queue. Transmission data is indicated by *`txbuf`*. Each word is represented by an uint32_t, regardless of configured word length. Words in *`txbuf`* shall be represented with its LSB at bit 0. If *`txbuf`* is `NULL` then zero valued bits will be shifted out on MOSI. The function returns as soon as the transmit queue is full or the requested number of words have been installed. This function never blocks. Transfer properties are set with the the function `spi_config`.
Param	*`d`* [IN] pointer Device handle returned by `spi_open`.
Param	*`txbuf`* [IN] pointer Transmit data. If *`txbuf`* is `NULL` then zero valued words are shifted out.
Param	*`count`* [IN] Integer Number of words to transmit
Return	int. Number of words written to transmit queue, zero if none.

Table 14.12. `spi_read32` function declaration

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

14.7. Synchronous TX/RX mode

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

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

If *cfg.sync=1*, then calls to spi_write32 will only write words to the SPI transmit queue when it is guaranteed that the receive queue will not overrun. This relaxes the restrictions on how calls to spi_write32 and spi_read32 can be combined. It means that the user does not have to maintain the number of outstanding words and the receive queue will never overrun.

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

14.8. Slave select

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

Table 14.13. spi_slave_select function declaration

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

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

14.9. Restrictions

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

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

- spi_dev_count
- spi_open

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

- spi_get_event
- spi_clear_event

15. I2C master driver

15.1. Introduction

This section describes the driver used to control the GRLIB I2CMST device for I2C master operation.

15.1.1. User Interface

This section covers how the driver can be interfaced to an application to control the I2CMST hardware.

Controlling the driver and device is done with functions provided by the driver prefixed with `i2cmst_`. All driver functions take a device handle returned by `i2cmst_open` as the first parameter. All supported commands and their data structures are defined in the driver's header file `drv/i2cmst.h`.

15.1.2. Features

- All driver functions are non-blocking.
- Optionally interrupt driven
- User supplies I2C requests to the driver by lists of of packets.
- Automatic retry operation

15.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 15.1. Driver registration functions

Registration method	Function
Automatic registration	`i2cmst_autoinit()`
Register one device	`i2cmst_register()`
Register many devices	`i2cmst_init()`

15.3. Examples

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

15.4. Opening and closing device

A I2CMST device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `i2cmst_dev_count`. A particular device can be opened using `i2cmst_open` and closed `i2cmst_close`. The functions are described below.

An opened device can not be reopened unless the device is closed first. When opening a device the device is marked opened by the driver. This procedure is thread-safe by protecting from other threads by using `osal_ldstub` from the OSAL. Protection is used by all I2CMST devices on opening and closing. It is assumed that at most one thread operates on one I2CMST device at a time.

During opening of a I2CMST device the following steps are taken:

- The device is marked opened to protect the caller from other users of the same device.
- Internal data structures are initialized.
- The driver is set to stopped operation mode.

The example below prints the number of I2CMST devices to screen then opens and closes the first I2CMST device present in the system.

```
int print_i2cmst_devices(void)
{
```

```
        struct i2cmst_priv *device;
        int count;

        count = i2cmst_dev_count();
        printf("%d I2CMST device(s) present\n", count);

        device = i2cmst_open(0);
        if (!device) {
                return -1; /* Failure */
        }

        i2cmst_close(device);
        return 0; /* success */
}
```

Table 15.2. `i2cmst_dev_count` function declaration

Proto	`int i2cmst_dev_count(void)`
About	Retrieve number of I2CMST devices registered to the driver.
Return	int. Number of I2CMST devices registered in system, zero if none.

Table 15.3. `i2cmst_open` function declaration

Proto	`struct i2cmst_priv *i2cmst_open(int dev_no)`	
About	Opens a I2CMST device. The I2CMST device is identified by index. The returned value is used as input argument to all functions operating on the device.	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the registration order to the driver, normally dictated by the Plug & Play order. Must be equal or greater than zero, and smaller than that returned by `i2cmst_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device semaphore fails or device already is open.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which I2CMST device.

Table 15.4. `i2cmst_close` function declaration

Proto	`int i2cmst_close(struct i2cmst_priv *d)`
About	Closes a previously opened device.
Param	*`d`* [IN] pointer Device identifier. Returned from `i2cmst_open`.
Return	int. DRV_OK

NOTE: No device I/O registers are modified by the open and close functions.

15.5. Operation mode

The driver always operates in one of two modes: *started* or *stopped*,

In stopped mode, bus operation is disabled and the user is allowed to configure the device and driver. Interrupts are never used in stopped mode.

In started mode, bus operations can be active. Functions for configuring the driver are not available in stopped mode.

Only the functions `i2cmst_start()` and `i2cmst_stop()` changes the operation mode while the device is open. The driver does not transfer between started and stopped by itself.

15.5.1. Starting and stopping

The `i2cmst_start()` function places the driver in started mode. Configuration set by previous driver function calls are committed to hardware and statistics are cleared before started mode enters. It is necessary to enter started mode to be able to generate transactions on the I2C bus.

The function `i2cmst_stop()` makes the driver core leave the started mode and enter stopped mode. After calling this function, further calls to `i2cmst_request()` will return `DRV_BUSY` and do not generate any side effects. It is necessary to enter stopped mode to change operating parameters of the device such as bitrate, retries, interrupt mode and address length The function will return `DRV_BUSY` if the driver is already stopped.

Function `i2cmst_isstarted()` is used to determine the driver operation mode.

Table 15.5. `i2cmst_start` function declaration

Proto	`int i2cmst_start(struct i2cmst_priv *priv)`	
About	Start driver.	
Param	*d* [IN] pointer Device handle returned by `i2cmst_open`.	
Return	int.	
	Value	Description
	DRV_OK	Device was started by the function call.
	DRV_BUSY	Device already in started mode. Nothing performed.

Table 15.6. `i2cmst_stop` function declaration

Proto	`int i2cmst_stop(struct i2cmst_priv *priv)`	
About	Stop driver.	
Param	*d* [IN] pointer Device handle returned by `i2cmst_open`.	
Return	int.	
	Value	Description
	DRV_OK	Device was stopped by the function call.
	DRV_BUSY	Device already in stopped mode. Nothing performed.

Table 15.7. `i2cmst_isstarted` function declaration

Proto	`int i2cmst_isstarted(struct i2cmst_priv *d)`	
About	Get current I2CMST software running state	
Param	*d* [IN] Pointer Device identifier. Returned by `i2cmst_open`.	
Return	int. Status	
	Value	Description
	0	Stopped
	1	Started

15.6. Configuration

The driver is configured using function calls which are available only when in stopped operation mode. Return value for most of the functions is `DRV_OK` for success and non-zero on failure.

15.6.1. Transaction retries

`i2cmst_set_retries()` configures the number of retries per transaction.

Table 15.8. `i2cmst_set_retries` function declaration

Proto	`int i2cmst_set_retries(struct i2cmst_priv *d, int retries)`
About	Set number of retries per transaction The function configures how many retry attempts to perform for each I2C transaction packet. If *`retries`* is 0, then only one try is performed and no retries. A retry is performed if arbitration is lost or on acknowledge error. For example on mult-master congestion.
Param	*`d`* [IN] pointer Device handle returned by `i2cmst_open`.
Param	*`retries`* [IN] Integer Number of retries
Return	int.

Value	Description
DRV_OK	Success
DRV_BUSY	Device in started mode. Nothing performed.
DRV_INVAL	Parameter *`retries`* is less than 0. Nothing performed.

15.6.2. Speed

The function `i2cmst_set_speed()` can be used to set the I2C bus bitrate. It takes a parameter in bit/s and calculates the appropriate scaler register parameters. Commit to registers is performed the next time started mode is entered.

Two constants are predefined in `drv/i2cmst.h`:

```
enum {
  I2CMST_SPEED_STD   = 100000,        /* Standard speed (100 kbit/s) */
  I2CMST_SPEED_FAST  = 400000,        /* Fast speed (400 kbit/s) */
};
```

Table 15.9. `i2cmst_set_speed` function declaration

Proto	`int i2cmst_set_speed(struct i2cmst_priv *d, int speed)`
About	Set I2C bus speed The function configures speed in bit/s for bus accesses.
Param	*`d`* [IN] pointer Device handle returned by `i2cmst_open`.
Param	*`speed`* [IN] Integer speed in bit/s
Return	int.

Value	Description
DRV_OK	Success
DRV_BUSY	Device in started mode. Nothing performed.
DRV_INVAL	Parameter *`speed`* is less than 0. Nothing performed.

15.6.3. Interrupt driven operation

The driver can operate in interrupt driven or non-interrupt driven mode. Both are non-blocking.

When operating in interrupt driven mode, all I2C transfer operations are triggered by interrupt requests from the I2CMST device to the processor.

When interrupt driven mode is disabled, then the I2C bus operations are triggered by the user calling `i2cmst_request()` or `i2cmst_reclaim()`.

Table 15.10. `i2cmst_set_interrupt_mode` function declaration

Proto	`int i2cmst_set_interrupt_mode(struct i2cmst_priv *d, int enable)`	
About	Configure interrupt driven operation	
Param	*`d`* [IN] pointer Device handle returned by `i2cmst_open`.	
Param	*`enable`* [IN] Integer interrupt mode	
	Value	Description
	0	Disable interrupt driven operation
	1	Enable interrupt driven operation
Return	int.	
	Value	Description
	DRV_OK	Success
	DRV_BUSY	Device in started mode. Nothing performed.
Notes	In non-interrupt driven mode, the user needs to call `i2cmst_request()` or `i2cmst_reclaim()` for I2C transactions to progress.	

15.6.4. I2C address width

I2C address width of 7 and 10 bits is supported. `i2cmst_set_ten_bit_addr()` is used to configure address width.

Table 15.11. `i2cmst_set_ten_bit_addr` function declaration

Proto	`int i2cmst_set_ten_bit_addr(struct i2cmst_priv *d, int enable)`	
About	Set I2C address width The function configures the I2C address width for all transactions.	
Param	*`d`* [IN] pointer Device handle returned by `i2cmst_open`.	
Param	*`enable`* [IN] Integer 10 bit I2C address	
	Value	Description
	0	7 bit I2C address
	1	10 bit I2C address
Return	int.	
	Value	Description
	DRV_OK	Success
	DRV_BUSY	Device in started mode. Nothing performed.

15.7. Driver statistics

The driver maintains driver statistics as described by the data structure `struct i2cmst_stats`:

```
struct i2cmst_stats {
        uint32_t packets_sent;
        uint32_t arbitration_lost;
        uint32_t packets_nack;
};
```

Table 15.12. i2cmst_stats data structure declaration

packets_sent	Number of successful I2C transactions
arbitration_lost	Number of arbitration lost events
packets_nack	Number of acknowledge error events

`i2cmst_get_stats()` copies the driver's internal statistics to a user buffer. `i2cmst_clr_stats()` clears the driver's collected statistics.

Table 15.13. `i2cmst_get_stats` function declaration

Proto	`int i2cmst_get_stats(struct i2cmst_priv *d, struct i2cmst_stats *stats)`
About	Reads the current driver statistics collected from earlier events. The statistics are stored to the address given by the second argument. The layout and content of the statistics are defined by the `i2cmst_stats` data structure. Note that the snapshot is taken without lock protection. As a consequence the statistics may not be synchonized with each other. This could be caused if the function is interrupted by a the driver interrupt. Calling this function when the driver is in stopped mode will always give consitent statistics.
Param	*`d`* [IN] pointer Device handle returned by `i2cmst_open`.
Param	*`stats`* [OUT] pointer A snapshot of the current driver statistics are copied to this user provided buffer.
Return	None.

Table 15.14. `i2cmst_clr_stats` function declaration

Proto	`int i2cmst_clr_stats(struct i2cmst_priv *d)`
About	Resets statistic counters to 0.
Param	*`d`* [IN] pointer Device handle returned by `i2cmst_open`.
Return	int. DRV_OK

15.8. I2C bus transfer

15.8.1. Data structures

15.8.1.1. Packet

`struct i2cmst_packet` is used by the application to describe I2C bus transfers. The structure is available in `drv/i2cmst.h` and describes how the drivers shall perform a transfer. See the request and reclaim sections for more information.

```
/* Driver representation of an I2C bus transfer */
struct i2cmst_packet {
        struct i2cmst_packet    *next;    /* Next packet in list */
        /* Options and status */
        uint32_t                flags;    /* Modifiers and status */
        int                     slave;    /* Slave address */
        uint32_t                addr;     /* Use with I2CMST_FLAGS_ADDR */
        int                     length;   /* Size of payload */
        uint8_t                 *payload; /* Read or write data */
};
```

Table 15.15. i2cmst_packet data structure declaration

next	Pointer to the next packet in a list. `NULL` marks end of chain.	
flags	Transfer modifiers and status	
	Bits	Description (prefixed `I2CMST_FLAGS`)

	READ	Set by user to perform a read (1) or write (0) transfer.
	ADDR	Set by user to specify that a I2C device specific location shall be addressed. This bit qualifies the addr field.
	FINISHED	Set by driver to indicate that it is done with the transfer packet
	RETRIED	Set by driver to indicate that the request was retried at least once due to acknowledge or arbitration error.
	ERR	Set by driver to indicate that the maximum number of retries was reached.
slave	I2C address	
addr	I2C device local address. Qualified by the I2CMST_FLAGS_ADDR flag.	
length	Number of bytes in payload.	
payload	Data buffer for read or write transfer.	

15.8.1.2. List

Packets are chained together on a single linked list, represented by struct i2cmst_list, when the user and driver communicates. A packet list has a head and a tail node which points to the first and last packet on the list respectively. Each packets next field points to the next packet in the list. An empty list is represented by a struct i2cmst_list where both head and tail is NULL.

```
struct i2cmst_list {
  struct i2cmst_packet    *head;    /* First packet in list */
  struct i2cmst_packet    *tail;    /* Last packet in list */
};
```

15.8.2. Request

Transfers are generated using the i2cmst_request() function. It is possible to schedule multiple I2C transfer requests in one call. An example is shown below:

```
struct i2cmst_list mylist;
struct i2cmst_packet wpkt = { 0 };
struct i2cmst_packet rpkt = { 0 };
uint8_t buf[2];

/* Write the byte value 0xa to device at I2C address 0x70 */
buf[0] = 0xa;
wpkt.slave = 0x70;
wpkt.payload = &buf[0];
wpkt.next = &rpkt;

/* Read a byte from device at I2C address 0x50 */
rpkt.flags = I2CMST_FLAGS_READ;
rpkt.slave = 0x50;
rpkt.payload = &buf[1];

mylist.head = &wpkt;
mylist.tail = &rpkt;
result = i2cmst_request(d, &mylist);
```

The transmit operation is non-blocking: i2cmst_request() will return immediately.

On success, the transfers are scheduled to the driver. The parameter *mylist* points to the list header (struct i2cmst_list) which contains the individual requests.

Each request has an individual set of parameters controlled by the struct i2cmst_packet type.

NOTE: The driver never copies user data, so the packet nodes and payload must be valid until they have been reclaimed back from the driver. Packets must not be modified by the user while they are under control of the driver.

Table 15.16. i2cmst_request function declaration

Proto	int i2cmst_request(struct i2cmst_priv *d, struct i2cmst_list *chain)
About	Request I2C transfer

	Multiple transfers can be scheduled in one call.	
Param	*d* [IN] Pointer Device identifier. Returned by `i2cmst_open`.	
Param	*chain* [IN] Pointer List of `struct packet`	
Return	int. Status	
	DRV_OK	Successfully scheduled requests
	DRV_INVAL	Invalid list
	DRV_BUSY	Driver is not in started mode. Nothing done

15.8.3. Reclaim

I2C transfer requests are brought back to the user using the `i2cmst_reclaim()` function. An example is shown below:

```
int cnt = 0;

/* Reclaim at least two packets */
while (cnt < 2) {
  struct i2cmst_list mylist = { 0 };
  struct i2cmsg_packet *pkt;
  int result;

  result = i2cmst_reclaim(d, &mylist);
  if (DRV_OK != result) {
    continue;
  ]

  pkt = mylist.head;
  while (pkt) {
    /* Do something with the pkt */
    cnt++;
    pkt = pkt->next;
  }
}
```

The actual number of transfer requests reclaimed can be calculated by iterating the list.

The reclaim operation is non-blocking: the function will return immediately with either an empty or populated list.

Table 15.17. `i2cmst_reclaim` function declaration

Proto	`int i2cmst_reclaim(struct i2cmst_priv *d, struct i2cmst_list *chain)`	
About	Reclam I2C transfer requests back from the driver Finished requests are put on *chain*. The user has ownership of the reclaimed packets of these packets and the driver will no longer reference them, including the payload.	
Param	*d* [IN] Pointer Device identifier. Returned by `i2cmst_open`.	
Param	*chain* [IN] Pointer List of `struct packet` reclaimed by the driver.	
Return	int. Status	
	DRV_OK	At least one request was reclaimed
	DRV_WOULDBLOCK	No request was reclaimed

15.9. Synchronous example

Single synchronous I2C transfers can be performed with the following example functions `i2cwrite()` and `i2cread()`.

```
/* Write buf on I2C bus and return when done */
int i2cwrite(
        struct i2cmst_priv *dev,
        int devaddr,
        uint8_t *buf,
        int length
)
{
        return dosimple(dev, 0, devaddr, 0, buf, length);
}

/* Read from I2C bus to buf and return when done */
int i2cread(
        struct i2cmst_priv *dev,
        int devaddr,
        uint8_t *buf,
        int length
)
{
        return dosimple(dev, I2CMST_FLAGS_READ, devaddr, 0, buf, length);
}

/* Self-contained synchronous request and reclaim */
int dosimple(
        struct i2cmst_priv *dev,
        int uflags,
        int devaddr,
        uint16_t memaddr,
        uint8_t *payload,
        int length
)
{
        struct i2cmst_packet pkt;
        struct i2cmst_list pkts;
        int ret;
        pkt.next = NULL;
        pkt.flags = uflags;
        pkt.slave = devaddr;
        pkt.addr = memaddr;
        pkt.length = length;
        pkt.payload = payload;

        pkts.head = &pkt;
        pkts.tail = &pkt;
        i2cmst_request(dev, &pkts);
        if (ret) {
                return 1;
        }

        struct i2cmst_packet *ptr = NULL;
        while (ptr == NULL) {
                i2cmst_reclaim(dev, &pkts);
                ptr = pkts.head;
        }

        return ptr->flags & I2CMST_FLAGS_ERR;
}
```

16. Timer driver

16.1. Introduction

This section describes the driver used to control the GPTIMER and GRTIMER devices. Each GPTIMER/GRTIMER device can host multiple subtimers. The timer driver allows for opening timer devices using the timer device API. When a timer device is open, its subtimers can be opened using the subtimer API.

16.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 16.1. Driver registration functions

Registration method	**Function**
Automatic registration	`timer_autoinit()`
Register one device	`timer_register()`
Register many devices	`timer_init()`

16.3. Device interface

The device interface handles device level operations such as scaler and latch configuration.

16.3.1. Opening and closing device

A timer must first be opened before any operations can be performed using the driver. The number of timer devices (GPTIMER/GRTIMER) registered to the driver can be retrieved using `timer_dev_count`. A particular timer device can be opened using `timer_open` and closed using `timer_close`. The functions are described below.

An opened timer device can not be reopened unless it is closed first. When opening a device the device is marked opened by the driver. This open and close operations are thread-safe by protecting from other threads by using `osal_ldstub` from the OSAL.

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

- Device I/O registers are initialized, including disabling timer latching.
- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the same device.

NOTE: The timer device open operation does not affect the state of the device subtimers.

Table 16.2. `timer_dev_count` function declaration

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

Table 16.3. `timer_open` function declaration

Proto	`struct timer_priv *timer_open(int dev_no)`
About	Open a timer device The device to open is identified by timer device index (*`dev_no`*). The returned value is a device handle used as input argument to all functions operating on the timer device. Timer device configuration and latch registers are cleared. Subtimers are not touched.
Param	*`dev_no`* [IN] Integer

	Device identification number. Must be equal or greater than zero, and smaller than that returned by `timer_dev_count`.	
Return	Pointer. Status and driver's internal subtimer identification.	
	Value	Description
	NULL	Indicates failure to open subtimer. Fails if subtimer is already open of if *`dev_no`* is out of range.
	Pointer	Device handle used to access the timer device API functions.

Table 16.4. `timer_close` function declaration

Proto	`int timer_close(struct timer_priv *priv)`
About	Close a timer device If any of the device subtimers are open, then they are closed. Timer device configuration register is cleared.
Param	*`priv`* [IN] pointer Device handle returned by `timer_open`.
Return	int. DRV_OK

16.3.2. Device control

The timer driver exports the timer device global registers to the user. Setter and getter function is defined for these registers. For information on the registers available, see the component User's Manual.

Table 16.5. `timer_set_scaler` function declaration

Proto	`void timer_set_scaler(struct timer_priv *priv, uint32_t value)`
About	Set scaler value register
Param	*`priv`* [IN] pointer Device handle returned by `timer_open`.
Param	*`value`* [IN] uint32_t Value to write to the timer Scaler value register.
Return	None.

Table 16.6. `timer_set_scaler_reload` function declaration

Proto	`void timer_set_scaler_reload(struct timer_priv *priv, uint32_t value)`
About	Set scaler_reload value register
Param	*`priv`* [IN] pointer Device handle returned by `timer_open`.
Param	*`value`* [IN] uint32_t Value to write to the timer Scaler reload value register.
Return	None.

Table 16.7. `timer_get_cfg` function declaration

Proto	`uint32_t timer_get_cfg(struct timer_priv *priv)`
About	Get configuration register
Param	*`priv`* [IN] pointer Device handle returned by `timer_open`.
Return	uint32_t.

	Value read from timer Configuration register. Register definitions for the timer Configuration register are available in the file `include/regs/gptimer-regs.h`. The relevant defines are prefixed with *`GPTIMER_CFG_`*.

Table 16.8. `timer_set_cfg` function declaration

Proto	`void timer_set_cfg(struct timer_priv *priv, uint32_t value)`
About	Set configuration register
Param	*`priv`* [IN] pointer Device handle returned by `timer_open`.
Param	*`value`* [IN] uint32_t Value to write to timer Configuration register. Register definitions for the timer Configuration register are available in the file `include/regs/gptimer-regs.h`. The relevant defines are prefixed with *`GPTIMER_CFG_`*.
Return	None.

Table 16.9. `timer_set_latch_cfg` function declaration

Proto	`void timer_set_latch_cfg(struct timer_priv *priv, uint32_t value)`
About	Set timer latch configuration register
Param	*`priv`* [IN] pointer Device handle returned by `timer_open`.
Param	*`value`* [IN] uint32_t Value to write to the Timer latch configuration register.
Return	None.

16.4. Subtimer interface

The subtimer API operates on individual subtimers of a timer device.

16.4.1. Opening and closing subtimer

A subtimer must be opened before any operations can be performed on it using the driver. The number of subtimers hosted by a timer device can be read from the timer device configuration register using `timer_get_cfg`. A subtimer is opened using `timer_sub_open` and closed using `timer_sub_close`. The functions are described below.

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

During opening of a subtimer with `timer_sub_open` the following steps are taken:

No register initialization is performed by `timer_sub_open`.

- The device is marked opened to protect the caller from other users of the same device.

Table 16.10. `timer_sub_open` function declaration

Proto	`void *timer_sub_open(struct timer_priv *priv, int sub_no)`
About	Open a subtimer Subtimer *`sub_no`* on device *`d`* is opened. No registers are affected. The returned value is used as input argument to all functions operating on the subtimer.
Param	*`priv`* [IN] pointer

	Device handle returned by `timer_open`.	
Param	*`sub_no`* [IN] Integer Subtimer identification number.	
Return	Pointer. Status or subtimer handle	
	Value	Description
	NULL	Indicates failure to open subtimer. Fails if subtimer is already open or index out of range.
	Others	Subtimer handle to be used as input parameter to all other functions of the subtimer API.

Table 16.11. `timer_sub_close` function declaration

Proto	`int timer_sub_close(struct timer_priv *priv, int sub_no)`	
About	Close a subtimer No hardware registers are affected.	
Param	*`priv`* [IN] pointer Device handle returned by `timer_open`.	
Param	*`sub_no`* [IN] Integer Subtimer identification number. Must be the same number as used in the call to `timer_sub_open`	
Return	int. Return code as indicated below.	
	Value	Description
	DRV_OK	Success.
	DRV_NOTOPEN	Subtimer *`sub_no`* was not open.
Notes	A subtimer is closed by using the *device handle* and *subtimer number*. The subtimer handle is not involved	

16.4.2. Subtimer control

The timer driver exports the subtimer register space to the user by providing setter and getter functions. Full register descriptions can be found in the component User's Manual.

The example below opens the second subtimer of the first timer device. Then it is started and then read.

```
int timer_example()
{
  const int SUBNO = 1;
  struct timer_priv *device;
  void *sub;
  int count;
  uint32_t val;

  count = timer_dev_count();
  printf("%d timer devices present\n", count);

  device = timer_open(0);
  if (NULL == device) {
   return -1; /* Failure */
  }
  sub = timer_sub_open(device, SUBNO);

  timer_set_reload(sub, 0xffff);
  timer_set_ctrl(sub, GPTIMER_CTRL_LD | GPTIMER_CTRL_RS | GPTIMER_CTRL_EN);
  val = timer_get_counter(sub);
  printf("Counter value is %u.\n", val);
  val = timer_get_counter(sub);
  printf("Counter value is %u.\n", val);

  timer_set_ctrl(sub, 0);
  timer_sub_close(device, SUBNO);
  timer_close(device);
  return 0; /* success */
}
```

Table 16.12. Subtimer getter function declarations

Proto	`uint32_t timer_get_counter(void *s)`
	`uint32_t timer_get_reload(void *s)`
	`uint32_t timer_get_ctrl(void *s)`
	`uint32_t timer_get_latch(void *s)`

About	Get subtimer counter value register. Get subtimer reload value register. Get subtimer control register. Gets subtimer latch register. The functions returns the value of the corresponding subtimer register.
Param	*s* [IN] pointer Subtimer handle returned by `timer_sub_open`.
Return	uint32_t. Value read from register. `timer_get_counter` returns subtimer counter value register. `timer_get_reload` returns subtimer reload value register. `timer_get_ctrl` returns subtimer control register. Register definitions for the subtimer control register are available in the file `include/regs/gptimer-regs.h`. The relevant defines are prefixed with *GPTIMER_CTRL_*. `timer_get_latch` returns subtimer latch register.

Table 16.13. Subtimer setter function declarations

Proto	`void timer_set_reload(void *s, uint32_t value)` `void timer_set_ctrl(void *s, uint32_t value)`
About	Set subtimer reload value register. Set subtimer control register. The function writes the corresponding subtimer register with *value*.
Param	*s* [IN] pointer Subtimer handle returned by `timer_sub_open`.
Param	*value* [IN] uint32_t Value to write to the corresponding register, according to the following: For `timer_set_reload`, the subtimer reload value register is written. For `timer_set_ctrl`, the subtimer control register is written. Register definitions for the subtimer control register are available in the file `include/regs/gptimer-regs.h`. The relevant defines are prefixed with *GPTIMER_CTRL_*.
Return	None.

16.4.3. Watchdog support

The timer driver has support functionality for operating a watchdog subtimer. A watchdog subtimer is opened and started as any other subtimer, and should always be programmed to generate interrupt on underflow.

```
int watchdog_example()
{
  const int WATCHDOG_SUB = 3;
  struct timer_priv *device;
  void *wdsub;

  device = timer_open(0);
  if (NULL == device) {
   return -1; /* Failure */
  }
  wdsub = timer_sub_open(device, WATCHDOG_SUB);

  /* Set watchdog timeout. */
  timer_set_reload(wdsub, 0xffff);
  timer_set_ctrl(wdsub, GPTIMER_CTRL_IE);
  /* Start watchdog by kicking it */
  timer_kick(wdsub);
  [...]
  /* Kick the watchdog again*/
  timer_kick(wdsub);
  [...]
  /* Temporarily disable watchdog. */
  timer_stop(wdsub);
  /* Start it again by activating interrupt and kick it. */
  timer_set_ctrl(wdsub, GPTIMER_CTRL_IE);
  timer_kick(wdsub);
  [...]

  puts("Restarting system using watchdog.");

  watchdog_system_restart(wdsub);

  /* We never return to here */

  return -1;
}
```

Table 16.14. `timer_stop` function declaration

Proto	`void timer_stop(void *s)`
About	Stop subtimer The function stops a subtimer by clearing the control register (setting it to zero value). This function can be used to temporarily stop the watchdog timer.
Param	*`s`* [IN] pointer Subtimer handle returned by `timer_sub_open`.
Return	None.

Table 16.15. `timer_kick` function declaration

Proto	`void timer_kick(void *s)`
About	Restart a subtimer The function performs the following by updating the subtimer control register: • Subtimer is loaded with value of the subtimer reload register. • Subtimer is enabled. • Interrupt pending state of subtimer is cleared. The function is typically used to kick a watchdog timer. Note that the interrupt enable (IE) bit is left unmodified by this function.
Param	*`s`* [IN] pointer Subtimer handle returned by `timer_sub_open`.
Return	None.

Table 16.16. `watchdog_system_restart` function declaration

Proto	`void watchdog_system_restart(void *s)`

About	Restart system using watchdog The system triggers the watchdog and enters an infinite loop.
Param	*s* [IN] pointer Subtimer handle returned by `timer_sub_open`. This should be the subtimer handle for the watchdog timer.
Return	None.
Notes	The function never returns.

16.5. Restrictions

The timer driver is designed to operate each opened GPTIMER/GRTIMER device in multiple tasks, with some restrictions:

- One or more devices can be opened and operated on by one task.
- Any timer device or any subtimer of any device can be operated on by any task, but only the task which opened the device may close the device and open/close its subtimers.

The following functions are allowed to be called from any task or from an ISR, provided that the associated timer device or subtimer is open:

- `timer_dev_count`
- `timer_set_scaler`
- `timer_set_scaler_reload`
- `timer_get_cfg`
- `timer_set_cfg`
- `timer_set_latch_cfg`
- `timer_get_counter`
- `timer_get_reload`
- `timer_set_reload`
- `timer_get_ctrl`
- `timer_set_ctrl`
- `timer_get_latch`
- `timer_stop`
- `watchdog_system_restart`

17. GPIO driver

17.1. Introduction

This section describes the driver used to control the GRGPIO devices available on component.

17.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 17.1. Driver registration functions

Registration method	**Function**
Automatic registration	`gpio_autoinit()`
Register one device	`gpio_register()`
Register many devices	`gpio_init()`

17.3. Opening and closing device

The driver operates on GRGPIO devices, which typically consists of multiple GPIO in/out ports. The control interface, Section 17.4, allows for setting and getting values for multiple ports at a time.

For GRGPIO devices implemented with interrupt map support, the interrupt map interface described in Section 17.5 can be used.

A GRGPIO device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `gpio_dev_count`. A particular device can be opened using `gpio_open` and closed `gpio_close`. The functions are described below.

An opened device can not be reopened unless the device is closed first. When opening a device the device is marked opened by the driver. The `gpio_open` function is thread-safe by protecting from other threads by using `osal_ldstub` from the OSAL. Protection is used by all GRGPIO devices on opening.

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

- All GPIO ports are configured as inputs and interrupts are disabled.
- Internal data structures are initialized.
- The device is marked opened to protect the caller from other users of the same device.

Table 17.2. `gpio_dev_count` function declaration

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

Table 17.3. `gpio_open` function declaration

Proto	`struct gpio_priv *gpio_open(int dev_no)`	
About	Opens a GRGPIO device. The GRGPIO device is identified by index. The returned value is used as input argument to all functions operating on the device.	
Param	*`dev_no`* [IN] Integer Device identification number. Must be equal or greater than zero, and smaller than that returned by `grgpio_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device is already open.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, idenfifies which GRGPIO device.

Table 17.4. `gpio_close` function declaration

Proto	`int gpio_close(struct gpio_priv *priv)`
About	Closes a previously opened device. All ports are configured as inputs and GPIO interrupts are disabled.
Param	*`priv`* [IN] pointer Device identifier. Returned from `gpio_open`.
Return	int. DRV_OK

17.4. Control interface

The GPIO driver exports the full GRGPIO register space to the user. One function is defined per GRGPIO register.

Before enabling GPIO interrupt by configuring the interrupt mask register with the function `gpio_intmask`, the user must register an interrupt handler on the corresponding interrupt source. This can be done by calling the function `bcc_isr_register()`.

The example below opens the last GRGPIO device. Its third port signal is configured as output and driven high.

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

int gpio_example(void)
{
        struct gpio_priv *device;
        int count;

        count = gpio_dev_count();
        printf("%d GRGPIO devices present\n", count);

        device = gpio_open(count-1);
        if (!device) {
                return -1; /* Failure */
        }

        gpio_direction(device, 1, 1<<2);
        gpio_output(device, 1, 1<<2);

        /* Outputs are disabled when the device is closed. */
        gpio_close(device);
        return 0; /* success */
}
```

Table 17.5. GPIO control function declarations

Proto	`uint32_t gpio_data(struct gpio_priv *priv)`
	
	`uint32_t gpio_output(struct gpio_priv *priv, int set, uint32_t newval)`
	
	`uint32_t gpio_direction(struct gpio_priv *priv, int set, uint32_t newval)`
	
	`uint32_t gpio_intmask(struct gpio_priv *priv, int set, uint32_t newval)`
	
	`uint32_t gpio_intpol(struct gpio_priv *priv, int set, uint32_t newval)`
	
	`uint32_t gpio_intedge(struct gpio_priv *priv, int set, uint32_t newval)`
	
	`uint32_t gpio_intflag(struct gpio_priv *priv, int set, uint32_t newval)`
	
	`uint32_t gpio_pulse(struct gpio_priv *priv, int set, uint32_t newval)`

About	Get I/O port data register. Get/set I/O port output register. Get/set I/O port direction register. Get/set interrupt mask register. Get/set interrupt polarity register. Get/set interrupt edge register. Get/set interrupt flag register. Get/set pulse register. The functions return and optionally set the value of the corresponding GRGPIO register. If *set* is 0 then nothing will be written to the register, else the register is set to the value of the *newval* parameter.
Param	*priv* [IN] pointer Device identifier. Returned from gpio_open.
Param	*set* [IN] Integer Determines if register shall be updated. 0 — Do not write register. 1 — Write value of *newval* to register.
Return	uint32_t. The register content (before *newval* value is written).

The easiest way to write and read GPIO is to use the gpio_write() and gpio_read() functions.

Table 17.6. gpio_write function declaration

Proto	static inline int gpio_write(struct gpio_priv *priv, uint32_t val)
About	Write GPIO output register
Param	*priv* [IN] pointer Device handle returned by gpio_open.
Param	*val* [OUT] uint32_t Value to write to output register.
Return	int. DRV_OK

Table 17.7. gpio_read function declaration

Proto	static inline uint32_t gpio_read(struct gpio_priv *priv)
About	Read GPIO data register
Param	*priv* [IN] pointer Device handle returned by gpio_open.
Return	uint32_t. Value read from GPIO data register.

17.4.1. Logical bit operations

The functions described in Table 17.8 perform atomic set/get operations on the GPIO registers. It allows different tasks to independently set and clear individual bits in the output, direction and interrupt mask registers.

Table 17.8. GPIO logical function declarations

Proto	int gpio_output_or(struct gpio_priv *priv, uint32_t mask)

	`int gpio_output_and(struct gpio_priv *priv, uint32_t mask)`
	`int gpio_direction_or(struct gpio_priv *priv, uint32_t mask)`
	`int gpio_direction_and(struct gpio_priv *priv, uint32_t mask)`
	`int gpio_intmask_or(struct gpio_priv *priv, uint32_t mask)`
	`int gpio_intmask_and(struct gpio_priv *priv, uint32_t mask)`

About	Logical or/and I/O port output/direction/interrupt mask register. These functions perform a logical operation on the corresponding GRGPIO register. • The `_or` functions perform logical OR with the user *`mask`* parameter. • The `_and` functions perform logical AND with the user *`mask`* parameter. **NOTE:** These functions are implemented as `static inline` functions.
Param	*`priv`* [IN] pointer Device identifier. Returned from `gpio_open`.
Param	*`mask`* [IN] uint32_t User mask for the logical operation.
Return	int. DRV_OK

17.5. Interrupt map interface

The following functions can be used if interrupt map is enabled in the GRPGIO.

Table 17.9. `gpio_intmap_set` function declaration

Proto	`int gpio_intmap_set(struct gpio_priv *priv, int i, int intline)`
About	Configure GRGPIO bit *`i`* to generate interrupt on line *`intline`*.
Param	*`priv`* [IN] pointer Device handle returned by `gpio_open`.
Param	*`i`* [IN] Integer GRGPIO bit number
Param	*`intline`* [IN] Integer Interrupt line.
Return	int. DRV_OK
Notes	This function assumes that the user parameters are valid.

Table 17.10. `gpio_intmap_get` function declaration

Proto	`int gpio_intmap_get(struct gpio_priv *priv, int i)`
About	Get interrupt line for GRGPIO bit
Param	*`priv`* [IN] pointer Device handle returned by `gpio_open`.
Param	*`i`* [IN] Integer GRGPIO bit number
Return	int. Interrupt line for GRGPIO bit *`i`*.
Notes	This function assumes that the user parameters are valid.

18. AHB Status Register driver

18.1. Introduction

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

18.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 18.1. Driver registration functions

Registration method	**Function**
Automatic registration	`ahbstat_autoinit()`
Register one device	`ahbstat_register()`
Register many devices	`ahbstat_init()`

18.3. Opening and closing device

An AHBSTAT device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `ahbstat_dev_count`. A particular device can be opened using `ahbstat_open` and closed `ahbstat_close`. The functions are described below.

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

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

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

Table 18.2. `ahbstat_dev_count` function declaration

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

Table 18.3. `ahbstat_open` function declaration

Proto	`struct ahbstat_priv *ahbstat_open(int dev_no)`	
About	Opens an AHBSTAT device. The AHBSTAT device is identified by index. The returned value is used as input argument to all functions operating on the device.	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the order registered to the driver. Must be equal or greater than zero, and smaller than that returned by `ahbstat_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device is already open, or invalid *`dev_no`* parameter.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, idenfifies the AHBSTAT device.
Notes	The AHBSTAT ISR is not installed by `ahbstat_open`.	

Table 18.4. `ahbstat_close` function declaration

Proto	`int ahbstat_close(struct ahbstat_priv *d)`
About	Closes a previously opened device. If the AHB statu register interrupt service routine has been installed, it will be uninstalled by the close operation.
Param	*d* [IN] pointer Device handle returned by `ahbstat_open`.
Return	int.

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

18.4. Register interface

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

Table 18.5. `ahbstat_get_regs` function declaration

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

18.5. Interrupt service routine

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

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

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

volatile int user_ncerr = 0;

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

```
  (*ncerr)++;

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

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

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

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

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

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

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

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

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

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

Table 18.6. `ahbstat_set_user` function declaration

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

	Device handle returned by `ahbstat_open`.
Param	*`userhandler`* [IN] pointer User callback function as described above. If *`userhandler`* is NULL then the callback is uninstalled, but the AHBSTAT ISR is still active.
Param	*`userdata`* [IN] pointer Data to pass to the user callback. It may be NULL.
Return	int. DRV_OK on success, else != DRV_OK if ISR install failed.
Notes	The AHBSTAT ISR can not be uninstalled once installed. However, the user handler can be disabled by calling `ahbstat_set_user` with *`userhandler`* set to NULL.

19. Clock gating unit driver

19.1. Introduction

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

19.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 19.1. Driver registration functions

Registration method	**Function**
Automatic registration	`clkgate_autoinit()`
Register one device	`clkgate_register()`
Register many devices	`clkgate_init()`

19.3. Opening and closing device

An device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `clkgate_dev_count`. A particular device can be opened using `clkgate_open` and closed `clkgate_close`. The functions are described below.

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

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

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

Table 19.2. `clkgate_dev_count` function declaration

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

Table 19.3. `clkgate_open` function declaration

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

Table 19.4. `clkgate_close` function declaration

Proto	`int clkgate_close(struct clkgate_priv *d)`	
About	Closes a previously opened device. This function does not change any device state.	
Param	*d* [IN] pointer Device handle returned by `clkgate_open`.	
Return	int.	
	Value	Description
	DRV_OK	Successfully closed device.
	others	Device closed, but failed to unregister interrupt handler.

19.4. Operation

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

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

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

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

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

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

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

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

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

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

Table 19.5. `clkgate_gate` function declaration

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

Param	*d* [IN] pointer Device handle returned by ahbstat_open.
Param	*coremask* [IN] uint32_t Bitmask representing the cores to operate on. (Values are CLKGATE_*.)
Return	int. DRV_OK

Table 19.6. clkgate_enable function declaration

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

19.5. Core reset

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

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

19.6. Probe clock gating status

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

Table 19.7. clkgate_status function declaration

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

19.7. CPU override

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

Table 19.8. `clkgate_override` function declaration

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

20. GR1553B Driver

20.1. Introduction

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

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

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

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

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

20.1.1. Considerations and limitations

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

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

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

20.1.2. GR1553B Hardware

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

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

20.1.3. Software driver

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

Table 20.1. Source Location

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

20.1.4. Driver Registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 20.2. Driver registration functions

Registration method	Function
Automatic registration	`gr1553b_autoinit()`
Register one device	`gr1553b_register()`
Register many devices	`gr1553b_init()`

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

21. GR1553B Bus Controller Driver

21.1. Introduction

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

21.1.1. GR1553B Bus Controller Hardware

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

The three functions (BC, BM, RT) are accessed using the same register interface, but through separate registers. In order to share hardware resources between the three GR1553B drivers, the three depends on a lower level GR1553B driver, see Chapter 20.

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

21.1.2. Software driver

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

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

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

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

The driver provides the following services:

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

The driver sources and definitions are listed in the table below, the path is given relative to the BCC toolchain.

Table 21.1. BC driver Source location

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

21.1.3. Driver registration

The driver registration is handled by the GR1553B driver, see Chapter 20.

21.2. BC Device Handling

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

21.2.1. Device API

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

Table 21.2. Device API function prototypes

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

21.2.1.1. Data Structures

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

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

Table 21.3. gr1553bc_status member descriptions

Member	Description
status	BC status register
time	BC Timer register

21.2.1.2. gr1553bc_open

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

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

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

21.2.1.3. gr1553bc_close

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

21.2.1.4. gr1553bc_start

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

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

21.2.1.5. gr1553bc_pause

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

21.2.1.6. gr1553bc_resume

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

21.2.1.7. gr1553bc_stop

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

Table 21.4. gr1553bc_stop second argument

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

21.2.1.8. gr1553bc_indication

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

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

21.2.1.9. gr1553bc_status

This function retrieves the current BC hardware status. Second argument determine where the hardware status is stored, the layout of the data stored follows the `gr1553bc_status` data structure. The data structure is described in Table 21.3.

21.2.1.10. gr1553bc_ext_trig

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

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

21.2.1.11. gr1553bc_irq_setup

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

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

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

21.3. Descriptor List Handling

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

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

21.3.1. Overview

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

A list is built up by the following building blocks:

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

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

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

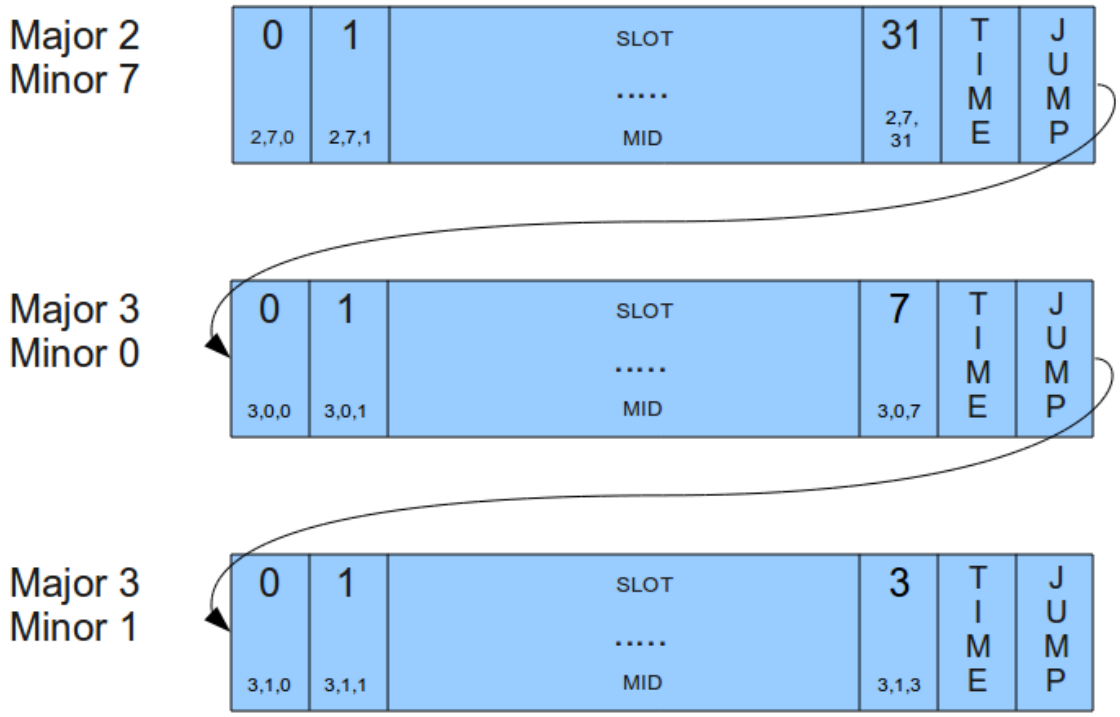

Figure 21.1. Three consecutive Minor Frames

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

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

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

Table 21.5. Macros for creating MID

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

21.3.2. Example: steps for creating a list

The typical approach when creating lists and executing it:

- gr1553bc_list_alloc(&list, MAJOR_CNT)
- gr1553bc_list_config(list, &listcfg)
- Create all Major Frames and Minor frame, for each major frame:
  1. gr1553bc_major_alloc_skel(&major, &major_minor_cfg)
  2. gr1553bc_list_set_major(list, &major, MAJOR_NUM)
- Link last and first Major Frames together:
  1. gr1553bc_list_set_major(&major7, &major0)
- gr1553bc_list_table_alloc() (Allocate Descriptor Table)

- gr1553bc_list_table_build() (Build Descriptor Table from Majors/Minors)
- Allocate and initialize Descriptors predefined before starting:
  1. gr1553bc_slot_alloc(list, &MID, TIME_REQUIRED, ..)
  2. gr1553bc_slot_transfer(MID, ..)
- START BC HARDWARE BY SCHEDULING ABOVE LIST
- Application operate on executing List

21.3.3. Major Frame

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

21.3.4. Minor Frame

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

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

Figure 21.2. Time-Managed Minor Frame of 1ms

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

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

21.3.5. Slot (Descriptor)

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

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

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

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

Transfer Descriptors (TBD) may have a time slot assigned, the BC core will wait until the time has expired before executing the next descriptor. Time slots are managed by Minor frames in the list. See Minor Frame section. A

Message Slot generating a data transmission on the 1553 bus must have a valid data pointer, pointing to a location from which the BC will read or write data.

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

Table 21.6. Slot configuration

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

Existing configured Slots can be manipulated with the following functions.

Table 21.7. Slot manipulation

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

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

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

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

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

21.3.7. Custom Memory Setup

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

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

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

21.3.8. Interrupt handling

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

Transfer Descriptors can be programmed to generate interrupt, and condition descriptors can be programmed to generate interrupt unconditionally (there exists other conditional types as well). When a Transfer descriptor causes interrupt the general ISR callback of the BC driver is called to let the user handle the interrupt. Transfers descriptor IRQ is enabled by configuring the descriptor.

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

```
void funcSetup()
{
    int MID;

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

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

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

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

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

21.3.9. List API

Table 21.8. List API function prototypes

Prototype	Description

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Prototype	Description

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

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

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

`int gr1553bc_slot_raw(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  unsigned int flags,`
`  uint32_t word0,`
`  uint32_t word1,`
`  uint32_t word2,`
`  uint32_t word3)`
	Custom descriptor initialization. Note that a bad initialization may break the BC driver.

`void gr1553bc_show_list(`
`  struct gr1553bc_list *list,`
`  int options)`
	Print information about a descriptor list to standard out. Used for debugging.

21.3.9.1. Data structures

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

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

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

Table 21.9. gr1553bc_minor_cfg member descriptions.

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

Table 21.10. gr1553bc_major_cfg member descriptions.

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

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

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

Table 21.11. gr1553bc_list_cfg member descriptions.

Member	Description
rt_timeout	Number of us timeout tolerance per RT address. The BC has a resolution of 4us.

Member	Description
bc_timeout	Number of us timeout tolerance of broadcast transfers
tropt_irq_on_err	Determines if transfer descriptors should generate IRQ on transfer errors
tropt_pause_on_err	Determines if the list should be paused on transfer error
async_list	Set to non-zero if asynchronous list

21.3.9.2. gr1553bc_list_init

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

The list is initialized according to the default configuration.

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

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

21.3.9.3. gr1553bc_list_alloc

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

The list is initialized according to the default configuration.

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

21.3.9.4. gr1553bc_list_free

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

21.3.9.5. gr1553bc_list_config

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

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

The first argument points to a list that is configure. The second argument points to the configuration description, the third argument identifies the BC device that the list will be scheduled on. The layout of the list configuration is described in Table 21.11.

21.3.9.6. gr1553bc_list_link_major

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

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

21.3.9.7. gr1553bc_list_set_major

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

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

21.3.9.8. gr1553bc_minor_table_size

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

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

21.3.9.9. gr1553bc_list_table_size

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

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

21.3.9.10. gr1553bc_list_table_init

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

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

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

21.3.9.11. gr1553bc_list_table_alloc

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

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

21.3.9.12. gr1553bc_list_table_free

Free previously allocated descriptor table memory.

21.3.9.13. gr1553bc_list_table_build

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

After this call descriptors can be initialized by user.

21.3.9.14. gr1553bc_major_init_skel

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

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

The configuration of the Major Frame is determined by the gr1553bc_major_cfg structure, described in Table 21.10.

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

21.3.9.15. gr1553bc_major_alloc_skel

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

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

The configuration of the Major Frame is determined by the gr1553bc_major_cfg structure, described in Table 21.10.

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

21.3.9.16. gr1553bc_list_freetime

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

21.3.9.17. gr1553bc_slot_alloc

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

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

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

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

21.3.9.18. gr1553bc_slot_free

Return Slot and timeslot allocated from the Minor Frame.

21.3.9.19. gr1553bc_mid_from_bd

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

21.3.9.20. gr1553bc_slot_bd

Looks up descriptor address from MID.

21.3.9.21. gr1553bc_slot_irq_prepare

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

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

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

21.3.9.22. gr1553bc_slot_irq_enable

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

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

21.3.9.23. gr1553bc_slot_irq_disable

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

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

21.3.9.24. gr1553bc_slot_jump

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

Table 21.12. gr1553bc_list_cfg member descriptions.

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

Returns zero on success.

21.3.9.25. gr1553bc_slot_exttrig

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

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

Returns zero on success.

21.3.9.26. gr1553bc_slot_transfer

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

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

The arguments are declared in the table below.

Table 21.13. gr1553bc_slot_transfer argument descriptions.

Argument	Description
list	List that the Slot is located at
mid	Slot Identification
options	Options: • Retry Mode • Number of retires

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

Returns zero on success.

21.3.9.27. gr1553bc_slot_dummy

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

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

Returns zero on success.

21.3.9.28. gr1553bc_slot_empty

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

Returns zero on success.

21.3.9.29. gr1553bc_slot_update

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

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

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

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

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

Returns zero on success.

21.3.9.30. gr1553bc_slot_raw

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

The arguments are declared in the table below.

Table 21.14. gr1553bc_slot_transfer argument descriptions.

Argument	Description
list	List that the Slot is located at

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

Returns zero on success.

21.3.9.31. gr1553bc_show_list

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

22. GR1553B Remote Terminal Driver

22.1. Introduction

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

22.1.1. GR1553B Remote Terminal Hardware

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

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

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

22.1.2. Driver registration

The driver registration is handled by the GR1553B driver, see Chapter 20.

22.2. User Interface

22.2.1. Overview

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

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

The driver sources and definitions are listed in the table below, the path is given relative to the BCC toolchain.

Table 22.1. RT driver Source location

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

22.2.1.1. Accessing an RT device

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

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

22.2.1.2. Introduction to the RT Memory areas

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

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

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

22.2.1.3. Sub Address Table

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

The SA-table is fixed size to 512 bytes.

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

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

The indication service can be used to determine the descriptor used in the next transfer, see Section 22.2.1.8.

22.2.1.4. Descriptors

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

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

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

Descriptors are managed by a data structure named `gr1553rt_list`. A List is the software representation of a chain of descriptors for a specific sub address and transfer type. Thus, 60 lists in total (two lists per SA, SA0 and SA31 are for mode codes) per RT. The List simplifies the descriptor handling for the user by introducing

descriptor numbers (`entry_no`) used when referring to descriptors rather than the descriptor address. Up to 65k descriptors are supported per List by the driver. A descriptor list is assigned to a SA and transfer type by calling `gr1553rt_list_sa()`.

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

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

22.2.1.5. Data Buffers

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

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

22.2.1.6. Event Logging

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

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

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

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

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

22.2.1.7. Interrupt service

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

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

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

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

22.2.1.8. Indication service

The indication service is typically used by the user to determine how many descriptors have been processed by the hardware for a certain SA and transfer type. The `gr1553rt_indication()` function returns the next descriptor number which will be used next transfer by the RT core. The indication function takes a sub address and an RT device as input, By remembering which descriptor was processed last the caller can determine how many and which descriptors have been accessed by the BC.

22.2.1.9. Mode Code support

The RT core a number of registers to control and interact with mode code commands. See hardware manual which mode codes are available. Each mode code can be disabled or enabled. Enabled mode codes can be logged and interrupt can be generated upon transmission events. The `gr1553rt_config_init()` function is used to configure the aforementioned mode code options. Interrupt caused by mode code transmissions can be programmed to call the user through an callback function, see the interrupt Section 22.2.1.7.

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

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

22.2.1.10. RT Time

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

22.2.2. Application Programming Interface

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

Table 22.2. Data structures

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

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

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

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

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

Protoype	Description

`int subadr,`
`int *txeno,`
`int *rxeno)`	

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

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

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

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

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

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

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

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

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

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

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

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

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

22.2.2.1. Data structures

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

```
struct gr1553rt_cfg {
    unsigned char rtaddress;
    unsigned int modecode;
    unsigned short time_res;
```

```
    void *satab_buffer;
    void *evlog_buffer;
    int evlog_size;
    int bd_count;
    void *bd_buffer;
    void *bd_sw_buffer;
};
```

Table 22.3. gr1553rt_cfg member descriptions

Member	Description
rtaddress	RT Address on 1553 bus [0..30]
modecode	Mode codes `enable/disable/IRQ/EV-Log`. Each mode code has a 2-bit configuration field. Mode Code Control Register in hardware manual
time_res	Time tag resolution in microseconds
satab_buffer	Sub Address Table (SA-table) allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of SA-table is given, the address must be aligned to 10-bit (1kB) boundary and at least 16*32 bytes.
evlog_buffer	Eventlog DMA buffer allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of eventlog is given, the address must be of `evlog_size` and aligned to `evlog_size`. See hardware manual.
evlog_size	Length in bytes of Eventlog, must be a multiple of 2. If set to zero event log is disabled, note that enabling logging in SA-table or descriptors will cause failure when eventlog is disabled.
bd_count	Number of descriptors for RT device. All descriptor lists share the descriptors. Maximum is 65K descriptors.
bd_buffer	Descriptor memory area allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of descriptors is given, the address must be aligned to 32 bytes and of *`(32 * bd_count)`* bytes size.
bd_sw_buffer	Descriptor memory area allocation for internal usage. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of descriptors is given, the area must be of *`(4 * bd_count)`* bytes size.

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

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

Table 22.4. gr1553rt_list_cfg member descriptions

Member	Description
bd_cnt	Number of descriptors in List

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

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

Table 22.5. gr1553rt_status member descriptions

Member	Description
status	Current value of RT Status Register
bus_status	Current value of RT Bus Status Register

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

22.2.2.2. gr1553rt_open

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

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

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

22.2.2.3. gr1553rt_close

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

22.2.2.4. gr1553rt_config_init

Configure memory for an RT device. The configuration parameters are stored in the location pointed to by *`cfg`*. The layout of the parameters must follow the `gr1553rt_cfg` data structure, described in Table 22.3.

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

22.2.2.5. gr1553rt_config_alloc

Configure and allocate memory for an RT device. The configuration parameters are stored in the location pointed to by *`cfg`*. The layout of the parameters must follow the `gr1553rt_cfg` data structure, described in Table 22.3.

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

22.2.2.6. gr1553bm_config_free

Free allocated memory.

22.2.2.7. gr1553rt_start

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

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

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

22.2.2.8. gr1553rt_stop

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

22.2.2.9. gr1553rt_status

Read current status of the RT core. The status is written to the location pointed to by status in the format determined by the `gr1553rt_status` structure described in Table 22.5.

22.2.2.10. gr1553rt_indication

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

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

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

22.2.2.11. gr1553rt_evlog_read

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

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

22.2.2.12. gr1553rt_set_vecword

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

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

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

22.2.2.13. gr1553rt_set_bussts

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

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

22.2.2.14. gr1553rt_sa_setopts

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

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

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

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

22.2.2.15. gr1553rt_list_sa

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

22.2.2.16. gr1553rt_sa_schedule

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

22.2.2.17. gr1553rt_irq_err

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

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

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

22.2.2.18. gr1553rt_irq_mc

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

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

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

22.2.2.19. gr1553rt_irq_sa

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

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

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

22.2.2.20. gr1553rt_list_init

Configure a list structure according to configuration given in *cfg*, see the `gr1553rt_list_cfg` data structure in Table 22.4. Assign the list to an RT device, however not to a sub address yet. The *rt* handle is stored within list.

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

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

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

22.2.2.21. gr1553rt_list_alloc

Allocate and configure a list structure according to configuration given in *cfg*, see the `gr1553rt_list_cfg` data structure in Table 22.4. Assign the list to an RT device, however not to a sub address yet. The *rt* handle is stored within list.

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

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

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

22.2.2.22. gr1553rt_bd_init

Initialize a descriptor entry in a list. This is typically done prior to scheduling the list. The descriptor and the next descriptor is given by descriptor indexes relative to the list (*entry_no and next*), see table below for options

on *next*. Set bit 30 of the argument *flags* in order to set the IRQEN bit of the descriptors Control/Status Word. The argument *dptr* is written to the descriptors Data Buffer Pointer Word.

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

Table 22.6. gr1553rt_bd_init next argument description

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

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

22.2.2.23. gr1553rt_bd_update

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

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

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

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

23. GR1553B Bus Monitor Driver

23.1. Introduction

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

23.1.1. GR1553B Remote Terminal Hardware

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

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

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

23.1.2. Driver registration

The driver registration is handled by the GR1553B driver, see Chapter 20.

23.2. User Interface

23.2.1. Overview

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

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

The driver sources and definitions are listed in the table below, the path is given relative to the BCC toolchain.

Table 23.1. BM driver Source location

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

23.2.1.1. Accessing a BM device

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

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

23.2.1.2. BM Log memory

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

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

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

23.2.1.3. Accessing the BM Log memory

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

- gr1553bm_available()
- gr1553bm_read()

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

23.2.1.4. Time

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

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

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

23.2.1.5. Filtering

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

23.2.1.6. Interrupt service

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

23.2.2. Application Programming Interface

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

Table 23.2. function prototypes

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

Prototype	Description
`int gr1553bm_config_init( void *bm, struct gr1553bm_cfg *cfg)`	Configure the BM device driver BM log DMA-memory
`int gr1553bm_config_alloc( void *bm, struct gr1553bm_cfg *cfg)`	Configure the BM device driver and allocate BM log DMA-memory
`void gr1553bm_config_free(void *bm)`	Free allocated memory
`int gr1553bm_start(void *bm)`	Start BM logging, enables Interrupts
`void gr1553bm_stop(void *bm)`	Stop BM logging, disables interrupts
`void gr1553bm_time( void *bm, uint64_t *time)`	Get 1553 64-bit Time maintained by the driver. The lowest 24-bits are taken directly from the BM timer register, the most significant 40-bits are taken from a software counter.
`int gr1553bm_available( void *bm, int *nentries)`	The current number of entries in the log is stored into `nentries`.
`int gr1553bm_read( void *bm, struct gr1553bm_entry *dst, int *max)`	Copy contents a maximum number (*`max`*) of entries from the BM log to a user provided data buffer (dst). The actual number of entries copied is stored into *`max`*.

23.2.2.1. Data structures

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

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

Table 23.3. gr1553bm_config member descriptions.

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

Member	Description
buffer_size	Size of buffer in bytes, must be aligned to 8-byte boundary.
buffer_custom	Custom BM log buffer location, must be aligned to 8-byte and be of buffer_size length. If NULL dynamic memory allocation is used.
copy_func	Custom Copy function, may be used to implement a more effective/ custom way of copying the DMA buffer. For example the DMA log may need to processed at the same time when copying.
copy_func_arg	Optional Custom Data passed onto `copy_func()`
dma_error_isr	Custom DMA error function, note that this function is called from Interrupt Context. Set to NULL to disable this callback.
dma_error_arg	COptional Custom Data passed on to `dma_error_isr()`

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

Table 23.4. gr1553bm_entry member descriptions.

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

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

23.2.2.2. gr1553bm_open

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

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

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

23.2.2.3. gr1553bm_close

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

23.2.2.4. gr1553bm_config_init

Configure the log DMA-memory for a BM device. The configuration parameters are stored in the location pointed to by *`cfg`*. The layout of the parameters must follow the `gr1553bm_config` data structure, described in Table 23.3.

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

23.2.2.5. gr1553bm_config_alloc

Configure and allocate the log DMA-memory for a BM device. The configuration parameters are stored in the location pointed to by *cfg*. The layout of the parameters must follow the gr1553bm_config data structure, described in Table 23.3.

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

23.2.2.6. gr1553bm_config_free

Free allocated memory.

23.2.2.7. gr1553bm_start

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

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

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

23.2.2.8. gr1553bm_stop

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

23.2.2.9. gr1553bm_time

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

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

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

23.2.2.10. gr1553bm_available

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

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

23.2.2.11. gr1553bm_read

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

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

24. GR716 memory protection unit driver

24.1. Introduction

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

24.1.1. User Interface

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

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

24.1.2. Features

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

24.1.3. Limitations

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

24.2. Driver registration

This driver uses the driver registration mechanism described in Chapter 10.

Table 24.1. Driver registration functions

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

24.3. Examples

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

24.4. Opening and closing device

A MEMPROT device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `memprot_dev_count`. A particular device can be opened using `memprot_open` and closed `memprot_close`. The functions are described below.

An opened device can not be reopened unless the device is closed first. When opening a device the device is marked opened by the driver. This procedure is thread-safe by protecting from other threads by using `osal_ldstub` from the OSAL. Protection is used by all MEMPROT devices on opening and closing. It is assumed that at most one thread operates on one MEMPROT device at a time.

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

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

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

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

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

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

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

Table 24.2. `memprot_dev_count` function declaration

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

Table 24.3. `memprot_open` function declaration

Proto	`struct memprot_priv *memprot_open(int dev_no)`	
About	Opens a MEMPROT device. The device is identified by index. The returned value is used as input argument to all functions operating on the device.	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the registration order to the driver, normally dictated by the Plug & Play order. Must be equal or greater than zero, and smaller than that returned by `memprot_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device semaphore fails or device already is open.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which MEMPROT device.

Table 24.4. `memprot_close` function declaration

Proto	`int memprot_close(struct memprot_priv *d)`
About	Closes a previously opened device.
Param	*`d`* [IN] pointer Device identifier. Returned from `memprot_open`.
Return	int. DRV_OK

NOTE: Memory protection configuration is not changed by the open and close functions. In particular, memory protection is not disabled by close.

24.5. Operation mode

The driver always operates in one of two modes: *started* or *stopped*,

This translates directly to whether the memory protection unit is enabled or disabled.

- *Started* is equivalent to `PCR.EN=1`. It means that the memory protection unit is enabled.
- *Stopped* is equivalent to `PCR.EN=0`. It means that the memory protection unit is disabled.

All API functions are available in both operation modes.

24.5.1. Starting and stopping

The `memprot_start()` function places the driver in started mode. The function `memprot_stop()` makes the driver core leave the started mode and enter stopped mode. `memprot_isstarted()` is used to determine the driver operation mode.

Table 24.5. `memprot_start` function declaration

Proto	`int memprot_start(struct memprot_priv *priv)`	
About	Start driver.	
Param	*d* [IN] pointer Device handle returned by `memprot_open`.	
Return	int.	
	Value	Description
	DRV_OK	Device was started by the function call.
	DRV_BUSY	Device already in started mode. Nothing performed.

Table 24.6. `memprot_stop` function declaration

Proto	`int memprot_stop(struct memprot_priv *priv)`	
About	Stop driver.	
Param	*d* [IN] pointer Device handle returned by `memprot_open`.	
Return	int.	
	Value	Description
	DRV_OK	Device was stopped by the function call.
	DRV_BUSY	Device already in stopped mode. Nothing performed.

Table 24.7. `memprot_isstarted` function declaration

Proto	`int memprot_isstarted(struct memprot_priv *d)`	
About	Get current MEMPROT driver running state	
Param	*d* [IN] Pointer Device identifier. Returned by `memprot_open`.	
Return	int. Status	
	Value	Description
	0	Stopped
	1	Started

24.6. Reset

Opening the driver does not change any of the units configuration. To reset the memory protection unit to a known accept-all state, the function `memprot_reset()` can be used.

Table 24.8. `memprot_reset` function declaration

Proto	`int memprot_reset(struct memprot_priv *d)`
About	Reset memory protection unit. This function disables the unit and disables all segment configurations.
Param	*d* [IN] Pointer

	Device identifier. Returned by `memprot_open`.
Return	int. DRV_OK

24.7. Segment configuration

24.7.1. Number of segments

The number of implemented segments can be retrieved with the function `memprot_nseg()`.

Table 24.9. `memprot_nseg` function declaration

Proto	`int memprot_nseg(struct memprot_priv *d)`
About	Retrieve number of implemented memory segments for memory protection device.
Param	*`d`* [IN] Pointer Device identifier. Returned by `memprot_open`.
Return	int. Number of memory segments supported. This is the value of the constant register field `PCR.NSGEG`.

24.7.2. Data structures

`struct memprot_seginfo` is used by the application to describe individual memory protection segments. The structure is available in `drv/memprot.h` and describes how the driver shall configure the segment.

```
/* User representation of one memory protection segment */
struct memprot_seginfo{
        uintptr_t start;
        uintptr_t end;
        uint32_t g;
        int en;
};
```

Table 24.10. memprot_seginfo data structure declaration

start	Start address	
end	End address	
g	Exclusive write grant `Gi`. This is a bit mask. See GR716-DS-UM for bit definitions of `Gi`.	
	Bit	Description
	`0`	`G0` - Grant master 0 exclusive write access.
	`1`	`G1` - Grant master 1 exclusive write access.
	`i`	`Gi` - Grant master *i* exclusive write access.
en	Disable or enable segment.	
	Value	Description
	`0`	Disable this segment.
	`1`	Enable this segment.

24.7.3. Set

An individual memory segment can be configured by calling the function `memprot_set()` with a user supplied as `struct memprot_seginfo` parameter. The following example configures segment 2.

```
        struct memprot_seginfo si;
        si.start        = 0x80004000;
        si.end          = 0x800040ff;
        si.g            = 1 << 2;
        si.en           = 1;

        memprot_reset(dev);
        memprot_set(dev, 2, &si);
        memprot_start(dev);
```

NOTE: For any segment configuration to be in effect, the device must be in started operation mode.

NOTE: Closing the driver does not cancel the configured memory protections.

Table 24.11. `memprot_set` function declaration

Proto	`int memprot_set(struct memprot_priv *d, int segment, const struct memprot_seginfo *seginfo)`
About	Configure a memory protection segment. The information contained in the *`seginfo`* is installed in the hardware registers corresponding to the *`segment`* number.
Param	*`d`* [IN] Pointer Device identifier. Returned by `memprot_open`.
Param	*`segment`* [IN] Integer Target segment number. Must be in the range 0 to `memprot_nseg()-1`.
Param	*`seginfo`* [IN] Pointer User representation of segment configuration.
Return	int. DRV_OK

24.7.4. Get

Memory protection segments can be read back from hardware into a `struct memprot_seginfo` record with the function `memprot_get()`. Everything in the record is qualified with the `en` field.

Protection segments are not affected when opening the driver which means that the previous configuration can be read out.

Table 24.12. `memprot_get` function declaration

Proto	`int memprot_get(struct memprot_priv *d, int segment, struct memprot_seginfo *seginfo)`
About	Read back memory protection segment configuration from hardware. The configuration contained in the hardware registers corresponding segment indexed by *`segment`* is read back and written to the *`seginfo`*.
Param	*`d`* [IN] Pointer Device identifier. Returned by `memprot_open`.
Param	*`segment`* [IN] Integer Target segment number. Must be in the range 0 to `memprot_nseg()-1`.
Param	*`seginfo`* [OUT] Pointer User representation of segment configuration.
Return	int. DRV_OK

24.7.4.1. Example

The following example function `printall()` prints information on all memory protection segment of a particular device. In addition to the `en` field, `isstarted()` can be used as a global qualifier to determine if a segment is in effect.

```
static void printsi(const struct memprot_seginfo *si)
{
        printf("  start = %08x\n", (unsigned) si->start);
        printf("  end   = %08x\n", (unsigned) si->end);
        printf("  g     = %08x\n", (unsigned) si->g);
        printf("  en    = %d (%s)\n", si->en, si->en ? "enabled" : "disabled");
}

void printall(struct memprot_priv *dev)
{
        const int nseg = memprot_nseg(dev);
        for (int i = 0; i < nseg; i++) {
                struct memprot_seginfo si;
                printf("SEGMENT %d\n", i);
                memprot_get(dev, i, &si);
                printsi(&si);
                puts("");
        }
}
```

25. Memory scrubber

25.1. Introduction

This section describes the Memory Scrubber (MEMSCRUB) driver for SPARC/LEON processors.

25.1.1. Hardware Support

The MEMSCRUB core hardware interface is documented in the GRIP Core User's manual. The MEMSCRUB core is used to monitor the memory AHB bus and can be programmed to scrub a memory area.

25.1.2. Driver sources

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

Table 25.1. MEMSCRUB driver source location

Location	Description
`src/include/drv/memscrub.h`	MEMSCRUB user interface definition
`src/memscrub`	MEMSCRUB driver implementation

25.1.3. Examples

There is an example available that uses the MEMSCRUB driver to scrub a memory area and log the different events. The example is part of the driver distribution, it can be found under `examples/memscrub`.

25.2. Software design overview

The driver provides a function interface, an API, to the user.

The API is not designed for multi-threadding, i.e. multiple threads operating on the driver independently. The driver does not contain any lock or protection for SMP environments. Changing the MEMSCRUB configuration is not intended to be done extensively at runtime or independently of the rest of the system, since it usually has a system-level impact. Therefore the user must take care of any impact that the different actions might have on other parts of the system (such as threads, CPUs, DMAs, ...).

25.2.1. Driver usage

The driver provides a set of functions that allow to start and stop the scrubber in different modes. The first step is to setup the memory range (or memory ranges) in which the scrubber is going to act (see Section 25.3.3).

After setting up the range we can start the scrubber in one of the three modes available (see Section 25.3.4):

- Init mode: Initialize the memory area.
- Scrub mode: Scrub the memory area.
- Regen mode: Regenerate the memory area. Similar to scrub mode, but has an optimized access pattern for correcting many errors.

Note that scrub and regen mode can be changed on the fly.

The driver provides functions to check if the scrubber is active and to stop it (see Section 25.3.4).

When dealing with errors, the drivers provides two different interfaces:

- Interrupts (see Section 25.3.6): Allows the user to install an Interrupt Service Routine (ISR) that will be executed whenever an error exceeds its corresponding threshold. Also the MEMSCRUB core allows to generate an interrupt when its done.
- Polling (see Section 25.3.7): Allows the user to poll the error status to check if an error have occurred.

Only one of these interfaces can be used at a given time.

The different errors that the MEMSCRUB can report are:

- AHB correctable error.
- AHB uncorrectable error.
- Scrubber run count errors.
- Scrubber block count errors.

There are functions that allow to configure the error count thresholds for each type of error individually (see Section 25.3.5). When the error count for a certain type exceeds the threshold, the error status is updated and an interrupt is generated. If a threshold is disabled, the error status is not updated and no interrupt is generated.

25.3. Memory scrubber user interface

25.3.1. Return values

```
MEMSCRUB_ERR_OK
MEMSCRUB_ERR_EINVAL
MEMSCRUB_ERR_ERROR
```

All the driver function calls return the following values when an error occurred:

- MEMSCRUB_ERR_OK - Successful execution.
- MEMSCRUB_ERR_EINVAL - Invalid input parameter. One of the input values checks failed.
- MEMSCRUB_ERR_ERROR - Internal error. Can have different causes.

25.3.2. Opening and closing device

A MEMSCRUB device must first be opened before any operations can be performed using the driver. The number of devices registered to the driver can be retrieved using `memscrub_dev_count`. A particular device can be opened using `memscrub_open` and closed `memscrub_close`. The functions are described below.

An opened device can not be reopened unless the device is closed first. When opening a device the device is marked opened by the driver. This procedure is thread-safe by protecting from other threads by using `osal_ldstub` from the OSAL. Protection is used by all MEMSCRUB devices on opening and closing. It is assumed that at most one thread operates on one MEMSCRUB device at a time.

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

- The device is marked opened to protect the caller from other users of the same device.
- Internal data structures are initialized.
- Error and interrupt status is cleared.

The example below prints the number of MEMSCRUB devices to standard output. It then opens and closes the first MEMSCRUB device present in the system.

```
int print_memscrub_devices(void)
{
        struct memscrub_priv *device;
        int count;

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

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

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

Table 25.2. `memscrub_dev_count` function declaration

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

Table 25.3. `memscrub_open` function declaration

Proto	`struct memscrub_priv *memscrub_open(int dev_no)`	
About	Opens a MEMSCRUB device. The device is identified by index. The returned value is used as input argument to all functions operating on the device.	
Param	*`dev_no`* [IN] Integer Device identification number. Devices are indexed by the registration order to the driver, normally dictated by the Plug & Play order. Must be equal or greater than zero, and smaller than that returned by `memscrub_dev_count`.	
Return	Pointer. Status and driver's internal device identification.	
	NULL	Indicates failure to open device. Fails if device semaphore fails or device already is open.
	Pointer	Pointer to internal driver structure. Should not be dereferenced by user. Input to all device API functions, identifies which MEMPROT device.

Table 25.4. `memscrub_close` function declaration

Proto	`int memscrub_close(struct memscrub_priv *d)`
About	Closes a previously opened device.
Param	*`d`* [IN] pointer Device identifier. Returned from `memscrub_open`.
Return	int. MEMSCRUB_ERR_OK

NOTE: Hardware configuration is not changed by the `memscrub_open()` function, apart from clearing the error and interrupt status at open. `memscrub_close()` does not change the current hardware configuration.

25.3.3. Configuring the memory range

```
int memscrub_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )
int memscrub_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )
int memscrub_secondary_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )
int memscrub_secondary_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )
int memscrub_scrub_position( struct memscrub_priv *priv, uint32_t * position )
```

The driver uses these functions to setup the primary and secondary memory ranges of the MEMSCRUB core. The scrubber will act on the range from address *`start`* to *`end`*, both inclusive.

The position function shows the actual position of the MEMSCRUB whithin the memory range.

These functions return a negative value if something went wrong, as explained in Section 25.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 25.5. `memscrub_range_set` function declaration

Proto	`int memscrub_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )`
About	Set the primary memory range for the MEMSCRUB core. The range is defined by the memory addresses *`start`* and *`end`*, both inclusive. See Section 25.3.3.
Param	*`start`* [IN] Integer 32-bit start address. The address bits below the burst size alignment are constant ‘0’.
Param	*`end`* [IN] Integer 32-bit end address. The address bits below the burst size alignment are constant ‘1’.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

Table 25.6. `memscrub_range_get` function declaration

Proto	`int memscrub_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )`
About	Get the primary memory range for the MEMSCRUB core. The range is defined by the memory addresses *`start`* and *`end`*, both inclusive. See Section 25.3.3.
Param	*`start`* [IN] Pointer Pointer to the 32-bit start address. The address bits below the burst size alignment are constant '0'.
Param	*`end`* [IN] Pointer Pointer to the 32-bit end address. The address bits below the burst size alignment are constant '1'.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

Table 25.7. `memscrub_secondary_range_set` function declaration

Proto	`int memscrub_secondary_range_set( struct memscrub_priv *priv, uint32_t start, uint32_t end )`
About	Set the primary memory range for the MEMSCRUB core. The range is defined by the memory addresses *`start`* and *`end`*, both inclusive. See Section 25.3.3.
Param	*`start`* [IN] Integer 32-bit start address. The address bits below the burst size alignment are constant '0'.
Param	*`end`* [IN] Integer 32-bit end address. The address bits below the burst size alignment are constant '1'.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

Table 25.8. `memscrub_secondary_range_get` function declaration

Proto	`int memscrub_secondary_range_get( struct memscrub_priv *priv, uint32_t * start, uint32_t * end )`
About	Get the secondary memory range for the MEMSCRUB core. The range is defined by the memory addresses *`start`* and *`end`*, both inclusive. See Section 25.3.3.
Param	*`start`* [IN] Pointer Pointer to the 32-bit start address. The address bits below the burst size alignment are constant '0'.
Param	*`end`* [IN] Pointer Pointer to the 32-bit end address. The address bits below the burst size alignment are constant '1'.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

Table 25.9. `memscrub_scrub_position` function declaration

Proto	`int memscrub_scrub_position( struct memscrub_priv *priv, uint32_t * position )`
About	Get the position of the scrubber within the memory range. See Section 25.3.3.
Param	*`position`* [IN] Pointer Pointer to the 32-bit position address.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

25.3.4. Starting/stoping different modes.

```
int memscrub_init_start( struct memscrub_priv *priv, uint32_t value, uint8_t delay, int options )
```

```
int memscrub_scrub_start( struct memscrub_priv *priv, uint8_t delay, int options )
int memscrub_regen_start( struct memscrub_priv *priv, uint8_t delay, int options )
int memscrub_stop( struct memscrub_priv *priv )
int memscrub_active( struct memscrub_priv *priv )
```

The driver uses these functions to start or stop the different modes of the MEMSCRUB core:

- Init mode: Initialize the memory area.
- Scrub mode: Scrub the memory area.
- Regen mode: Regenerate the memory area. Similar to scrub mode, but has an optimized access pattern for correcting many errors.

All the modes act on the configured memory range (see Section 25.3.3).

The active functions checks if the scrubber is currently running.

These functions return a negative value if something went wrong, as explained in Section 25.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 25.10. `memscrub_init_start` function declaration

Proto	`int memscrub_init_start( struct memscrub_priv *priv, uint32_t value, uint8_t delay, int options )`	
About	Start the initialization mode of the scrubber. See Section 25.3.4.	
Param	*`value`* [IN] Integer 32-bit value to be written into each memory position.	
Param	*`delay`* [IN] Integer 8-bit delay value. Processor cycles delay time between processed blocks.	
Param	*`options`* [IN] Integer Options.	
	Value	Description
	MEMSCRUB_OPTIONS_INTERRUPTDONE_ENABLE	Enable interrupt when done.
	MEMSCRUB_OPTIONS_INTERRUPTDONE_DISABLE	Disable interrupt when done (default).
	MEMSCRUB_OPTIONS_EXTERNALSTART_ENABLE	Enable external start.
	MEMSCRUB_OPTIONS_EXTERNALSTART_DISABLE	Disable external start (default).
	MEMSCRUB_OPTIONS_LOOPMODE_ENABLE	Enable loop mode.
	MEMSCRUB_OPTIONS_LOOPMODE_DISABLE	Disable loop mode (default).
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_ENABLE	Enable secondary memory range.
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_DISABLE	Disable secondary memory range (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.	

Table 25.11. `memscrub_scrub_start` function declaration

Proto	`int memscrub_scrub_start( struct memscrub_priv *priv, uint8_t delay, int options )`
About	Start the scrubbing mode of the scrubber. See Section 25.3.4.
Param	*`delay`* [IN] Integer 8-bit delay value. Processor cycles delay time between processed blocks.
Param	*`options`* [IN] Integer

	Options.	
	Value	Description
	MEMSCRUB_OPTIONS_INTERRUPTDONE_ENABLE	Enable interrupt when done.
	MEMSCRUB_OPTIONS_INTERRUPTDONE_DISABLE	Disable interrupt when done (default).
	MEMSCRUB_OPTIONS_EXTERNALSTART_ENABLE	Enable external start.
	MEMSCRUB_OPTIONS_EXTERNALSTART_DISABLE	Disable external start (default).
	MEMSCRUB_OPTIONS_LOOPMODE_ENABLE	Enable loop mode.
	MEMSCRUB_OPTIONS_LOOPMODE_DISABLE	Disable loop mode (default).
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_ENABLE	Enable secondary memory range.
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_DISABLE	Disable secondary memory range (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.	

Table 25.12. `memscrub_regen_start` function declaration

Proto	`int memscrub_regen_start( struct memscrub_priv *priv, uint8_t delay, int options )`	
About	Start the regeneration mode of the scrubber. See Section 25.3.4.	
Param	*`delay`* [IN] Integer 8-bit delay value. Processor cycles delay time between processed blocks.	
Param	*`options`* [IN] Integer Options.	
	Value	Description
	MEMSCRUB_OPTIONS_INTERRUPTDONE_ENABLE	Enable interrupt when done.
	MEMSCRUB_OPTIONS_INTERRUPTDONE_DISABLE	Disable interrupt when done (default).
	MEMSCRUB_OPTIONS_EXTERNALSTART_ENABLE	Enable external start.
	MEMSCRUB_OPTIONS_EXTERNALSTART_DISABLE	Disable external start (default).
	MEMSCRUB_OPTIONS_LOOPMODE_ENABLE	Enable loop mode.
	MEMSCRUB_OPTIONS_LOOPMODE_DISABLE	Disable loop mode (default).
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_ENABLE	Enable secondary memory range.
	MEMSCRUB_OPTIONS_SECONDARY_MEMRANGE_DISABLE	Disable secondary memory range (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.	

Table 25.13. `memscrub_stop` function declaration

Proto	`int memscrub_stop( struct memscrub_priv *priv )`
About	Stop the scrubber. See Section 25.3.4.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

Table 25.14. `memscrub_active` function declaration

Proto	`int memscrub_active( struct memscrub_priv *priv )`
About	Returns the active status of the scrubber. When the scrubber is active, it returns a non-zero positive value. When the scrubber is stopped, it returns zero. See Section 25.3.4.
Return	int. Positive value when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

25.3.5. Setting up error thresholds

```
int memscrub_ahberror_setup(struct memscrub_priv *priv, int uethres, int cethres, int options)
int memscrub_scruberror_setup(struct memscrub_priv *priv, int blkthres, int runthres, int options)
```

The driver uses these functions to setup the thresholds for ahb and scrub errors respectively. The following thresholds can be enabled or disabled:

- AHB correctable error.
- AHB uncorrectable error.
- Scrubber run count errors.
- Scrubber block count errors.

If a threshold is disabled, no error status or interrupt will be generated for that type of error. If a threshold is enabled, the error status or interrupt will be triggered when the error count exceeds the threshold value.

These functions return a negative value if something went wrong, as explained in Section 25.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 25.15. `memscrub_ahberror_setup` function declaration

Proto	`int memscrub_ahberror_setup( struct memscrub_priv *priv, int uethres, int cethres, int options )`	
About	Setup the AHB correctable and uncorrectable error thresholds for the MEMSCRUB core. See Section 25.3.5.	
Param	*`uethres`* [IN] Integer AHB uncorrectable error threshold value (only 8 LSB used).	
Param	*`cethres`* [IN] Integer AHB correctable error threshold value (only 10 LSB used).	
Param	*`options`* [IN] Integer Options.	
	Value	Description
	MEMSCRUB_OPTIONS_AHBERROR_CORTHRES_ENABLE	Enable AHB correctable error threshold.
	MEMSCRUB_OPTIONS_AHBERROR_CORTHRES_DISABLE	Disable AHB correctable error threshold (default).
	MEMSCRUB_OPTIONS_AHBERROR_UNCORTHRES_ENABLE	Enable AHB uncorrectable error threshold.
	MEMSCRUB_OPTIONS_AHBERROR_UNCORTHRES_DISABLE	Disable AHB uncorrectable error threshold (default).
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.	

Table 25.16. `memscrub_scruberror_setup` function declaration

Proto	`int memscrub_scruberror_setup( struct memscrub_priv *priv, int blkthres, int runthres, int options )`

About	Setup the scrubber run and block count error thresholds for the MEMSCRUB core. See Section 25.3.5.
Param	*blkthres* [IN] Integer Block count error threshold value (only 8 LSB used).
Param	*runthres* [IN] Integer Run count error threshold value (only 10 LSB used).
Param	*options* [IN] Integer Options.

Value	Description
MEMSCRUB_OPTIONS_SCRUBERROR_RUNTHRES_ENABLE	Enable run count error threshold.
MEMSCRUB_OPTIONS_SCRUBERROR_RUNTHRES_DISABLE	Disable run count error threshold (default).
MEMSCRUB_OPTIONS_SCRUBERROR_BLOCKTHRES_ENABLE	Enable block count error threshold.
MEMSCRUB_OPTIONS_SCRUBERROR_BLOCKTHRES_DISABLE	Disable block count error threshold (default).

Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

25.3.6. Registering an ISR

```
typedef void (*memscrub_isr_t) (
        void *arg,
        uint32_t ahbaccess,
        uint32_t ahbstatus,
        uint32_t scrubstatus
)
int memscrub_isr_register( struct memscrub_priv *priv, memscrub_isr_t isr, void * data )
int memscrub_isr_unregister( struct memscrub_priv *priv )
```

The driver uses these functions to register and unregister an ISR for error interrupts. When registering an ISR, interrupts are enabled. To set the error thresholds that trigger interrupts use the functions described in Section 25.3.5.

These functions return a negative value if something went wrong, as explained in Section 25.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 25.17. `memscrub_isr_register` function declaration

Proto	`int memscrub_isr_register( struct memscrub_priv *priv, memscrub_isr_t isr, void * arg )`
About	Registers an ISR for the MEMSCRUB core. See Section 25.3.6.
Param	*isr* [IN] Pointer The ISR function pointer.
Param	*arg* [IN] Pointer The ISR argument pointer.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

Table 25.18. `memscrub_isr_unregister` function declaration

Proto	`int memscrub_isr_unregister( struct memscrub_priv *priv )`
About	Unregisters an ISR for the MEMSCRUB core. See Section 25.3.6.

Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

25.3.7. Polling the error status

```
int memscrub_error_status( struct memscrub_priv *priv, uint32_t * ahbaccess, uint32_t * ahbstatus, uint32_t * scrubstatus )
```

The driver uses this function to poll the error status and clear the error status in case an error is found. To set the error thresholds that trigger error status use the functions described in Section 25.3.5.

This function returns a negative value if something went wrong, as explained in Section 25.3.1. Otherwise, the function returns MEMSCRUB_ERR_OK when successful.

Table 25.19. `memscrub_error_status` function declaration

Proto	`int memscrub_error_status( struct memscrub_priv *priv, uint32_t * ahbaccess, uint32_t * ahbstatus, uint32_t * scrubstatus )`
About	Poll the state of the error status registers. Returns the status registers and the AHB failing access register. If a error has been detected the function automatically clears the status in order to catch new errors. See Section 25.3.7.
Param	*`ahbaccess`* [OUT] Pointer The value pointed will be updated with the AHB failing access.
Param	*`ahbstatus`* [OUT] Pointer The value pointed will be updated with the AHB error status register content.
Param	*`scrubstatus`* [OUT] Pointer The value pointed will be updated with the scrub error status register content.
Return	int. MEMSCRUB_ERR_OK when successful. Otherwise, returns a negative value if something went wrong, as explained in Section 25.3.1.

25.4. API reference

This section lists all functions part of the MEMSCRUB driver API, and in which section(s) they are described. The API is also documented in the source header file of the driver, see Section 25.1.2.

Table 25.20. MEMSCRUB function reference

Prototype	Section
`int memscrub_range_get(struct memscrub_priv *priv, uint32_t *start, uint32_t *end)`	25.3.3
`int memscrub_range_set(struct memscrub_priv *priv, uint32_t start, uint32_t end)`	25.3.3
`int memscrub_secondary_range_get(struct memscrub_priv *priv, uint32_t *start, uint32_t *end)`	25.3.3
`int memscrub_secondary_range_set(struct memscrub_priv *priv, uint32_t start, uint32_t end)`	25.3.3
`int memscrub_scrub_position(struct memscrub_priv *priv, uint32_t *position)`	25.3.3
`int memscrub_init_start(struct memscrub_priv *priv, uint32_t value, uint8_t delay, int options)`	25.3.4
`int memscrub_scrub_start(struct memscrub_priv *priv, uint8_t delay, int options)`	25.3.4
`int memscrub_regen_start(struct memscrub_priv *priv, uint8_t delay, int options)`	25.3.4

Prototype	Section
`int memscrub_stop(struct memscrub_priv *priv)`	25.3.4
`int memscrub_active(struct memscrub_priv *priv)`	25.3.4
`int memscrub_ahberror_setup(struct memscrub_priv *priv, int uethres, int cethres, int options)`	25.3.5
`int memscrub_scruberror_setup(struct memscrub_priv *priv, int blkthres, int runthres, int options)`	25.3.5
`int memscrub_isr_register(struct memscrub_priv *priv, memscrub_isr_t isr, void * data)`	25.3.6
`int memscrub_isr_unregister(struct memscrub_priv *priv)`	25.3.6
`int memscrub_error_status(struct memscrub_priv *priv, uint32_t *ahbaccess, uint32_t *ahbstatus, uint32_t *scrubstatus)`	25.3.7

Cobham Gaisler AB
Kungsgatan 12
411 19 Gothenburg
Sweden
www.cobham.com/gaisler
sales@gaisler.com
T: +46 31 7758650
F: +46 31 421407