COBHAM

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 2.25.51
- Newlib C library 2.5.0
- `libbcc` - A user library for programming LEON systems
- GNU GDB 6.8 source-level debugger

In the LLVM/Clang version, the GCC package is replaced by:
- Clang 4.0.0 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-3.10.x, glibc-2.19, libstdc++.so.6.0.19 (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-2.0.2-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-2.0.2-gcc-linux64.tar.xz
```

After the compiler is installed, add `/opt/bcc-2.0.2-gcc/bin` to the executables search path (`PATH`) and `/opt/bcc-2.0.2-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-2.0.2-gcc-mingw64.zip`, shall be extracted to `C:\opt` creating the directory `C:\opt\bcc-2.0.2`. 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-2.0.2\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-2.0.2-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 make scripts
`src/examples/`	BCC example applications
`src/libbcc/`	`libbcc` source code and make 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-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 2.0.2 toolchain is distributed in an archive named `bcc-2.0.2-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-2.0.2-gcc/src/bcc-2.0.2`. The sources can be installed by extraction the source distribution archive `bcc-2.0.2-src.tar.bz2` to `/opt/bcc-2.0.2-gcc/src`. It can be done as follows for the Linux/GCC version of BCC.

```
$ cd /opt/bcc-2.0.2-gcc/src
$ tar xf bcc-2.0.2-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-2.0.2-local`, the following steps shall be performed:

```
$ cd /opt/bcc-2.0.2-gcc/src/bcc-2.0.2
$ ./ubuild.sh --destination /tmp/bcc-2.0.2-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 2.0.2

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.

2.2.1. sparc-gaisler-elf-gcc options

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

`-mflat`	Enable single register window model (flat). See Section 2.7.
`-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-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
`-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

-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 and GRLIB-TN-0013.

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 are options only 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.

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 generation 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.

The LLVM/Clang version of BCC generates SPARC V8 by default and can therefore not be used with LEON3 processors that do not implement the SPARC V8 multiply and divide instructions.

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 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.

Standalone App

```
                          Top-Of-RAM
  +-------------------+
  |                   |
  |  Startup Stack    |
  |                   |
  +-------------------+
  |                   |
  |       Heap        |
  |                   |
  +-------------------+
  |                   |
  |       Data        |
  |                   |
  +-------------------+
  |                   |
  |       Text        |
  |                   |
  +-------------------+
                          0X40000000
```
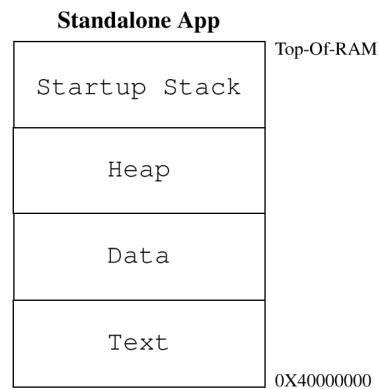
Figure 2.1. BCC RAM application memory map

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. 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.

2.13. Debugging with GDB

GDB 6.8 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. 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`.

2.14.1. 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`.

2.15. Creating a bootable ROM images

The MKPROM2 PROM image generator can be used to create boot-images for applications compiled with BCC 2.0.2. 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.

`clang-format`	A tool to format C/C++/Java/JavaScript/Objective-C/Protobuf code
`git-clang-format`	`clang-format` for git patches
`llvm-objdump`	LLVM object file dumper (similar to GNU objdump). Should be used instead of the binutils provided objdump if using REX
`scan-build`	scan-build is a command line utility that enables a user to run the Clang static analyzer over their code base as part of performing a regular build
`scan-view`	The clang static analyzer results viewer
`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-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.

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

---

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()`. 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()` 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` function declaration

Proto	`int bcc_timer_tick_init(void)`		
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.		
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()` 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;
}
```

5.10. 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 5.42. `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 5.43. `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 5.44. `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 5.45. `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 5.46 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 5.46. Default trap handlers for BCC 2.0.2

`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 5.47. BCC library user API structure reference

Type	**Section**
struct bcc_isr_node	5.9.5.2

Table 5.48. BCC library user API function reference

Prototype	**Section**
`uint32_t bcc_timer_get_us(void)`	5.3
`int bcc_timer_tick_init(void)`	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_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 depth, uint32_t (*fn)( void *in-fo, 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, it uses 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	*depth* [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 the same as the depth parameter to ambapp_visit(). 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 and defined as in Table 6.5, Table 6.6 and Table 6.7. 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;
};
```

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

```
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];
};
```

Table 6.7. amba_ahb_info data structure declaration

ver	Device version
irq	Device interrupt number
bar	Bank Address Register

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 depth = 4;
        uint32_t ret;

        ret = ambapp_visit(
                ioarea,
                VENDOR_GAISLER,
                GAISLER_APBUART,
                AMBAPP_VISIT_APBSLAVE,
                depth,
                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 ambapp_apb_info	6.3.2
struct ambapp_ahb_bar	6.3.2
struct ambapp_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 depth, 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. It also describes how custom BSPs can be created and used.

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`. It is important that the `-qbsp=`, `-mcpu=`, `-mfix` and `-msoft-float` options are given to GCC both at the compile and link steps. If option `-qbsp=` is not given explicitly, then `-qbsp=leon3` is implied. `-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.

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 for reduced 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. 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*.

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 2.15.

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*.

7.4.1.1. 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.

7.4.1.2. 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`.

7.4.1.3. 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 19200.

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.

7.5. 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 `AP-BUART` 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()` in `__bcc_init70()` before entry to `main()`. See Section 5.3.1 for more information on `bcc_timer_tick_init()`.

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

void __bcc_init70(void) {
        int ret;

        ret = bcc_init_ticks();
        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. 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

Prototype	Section
`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 2.0.2. Other LEON toolchains and other versions of BCC may have other recommendations.

Table A.1. Recommended GCC options for BCC 2.0.2

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`
LEON3FT and LEON3FT-RTAX systems without SPARC V8 `mul/div` based on GRLIB versions up to and including build 4174.	`-mcpu=leon3v7 -mfix-b2bst -mfix-tn0013`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul/div` based on GRLIB versions later than build 4174. 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`
LEON3FT and LEON3FT-RTAX systems without SPARC V8 `mul/div` based on GRLIB versions later than build 4174. LEON3 systems without SPARC V8 `mul/div` implemented without cache parity protection.	`-mcpu=leon3v7` For GRLIB version up to and including 4206, also add • `-mfix-tn0013`
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 2.0.2 run-time supports the GCC option `-mflat`.

Table A.2 describes the GCC `-mcpu=` options applicable to BCC 2.0.2. If no `-mcpu=` option is used, then `-mcpu=v7` is implied.

Table A.2. GCC `-mcpu=` options for BCC 2.0.2

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 2.0.2. Other LEON toolchains and other versions of BCC may have other recommendations.

Table B.1. Recommended Clang options for BCC 2.0.2

System	Recommended Clang options
GR740	`-qbsp=gr740 -mcpu=gr740`
GR712RC [1]	`-qbsp=gr712rc -mcpu=gr712rc`
GR716	`-qbsp=gr716 -mcpu=leon3 -qnano -qsvt`
UT699E, UT700 [1]	`-qbsp=leon3 -mcpu=leon3`
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. [1]	`-qbsp=leon3 -mcpu=leon3`
LEON3FT and LEON3FT-RTAX systems with SPARC V8 `mul/div` based on GRLIB versions later than build 4206.	`-qbsp=leon3 -mcpu=leon3`
LEON3/LEON3FT systems without SPARC V8 `mul/div`.	`Unsupported`
AGGA4	`Unsupported`
AT697	`Unsupported`
Other LEON2 systems	`Unsupported`

[1] The LLVM/Clang toolchain does not implement workarounds for the errata described in GRLIB-TN-0009, GRLIB-TN-0010, GR-LIB-TN-0011, GRLIB-TN-0012 and GRLIB-TN-0013. It is recommended to use the GCC based toolchain on devices affected by one or more of these errata.

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`.

Table B.2 describes the Clang `-mcpu=` options applicable to BCC 2.0.2. If no `-mcpu=` option is used, then SPARC V8 with `mul/div` is generated.

Table B.2. Clang `-mcpu=` options for BCC 2.0.2

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