



RTEMS BSP and Driver Guide

Release 5.c5749d0-modified (1st November 2019)

© 1988, 2019 RTEMS Project and contributors

CONTENTS

1	Introduction	3
2	Target Dependent Files	5
2.1	CPU Dependent	6
2.2	Board Dependent	7
2.3	Peripheral Dependent	8
2.4	Questions to Ask	9
2.5	CPU Dependent Executive Files	10
2.6	Board Support Package Structure	11
3	Linker Script	13
3.1	What is a “linkcmds” file?	14
3.2	Program Sections	15
3.3	Image of an Executable	16
3.4	Example Linker Command Script	17
3.5	Initialized Data	21
4	Miscellaneous Support Files	23
4.1	GCC Compiler Specifications File	24
4.2	README Files	25
4.3	Times	26
4.4	Tools Subdirectory	27
4.5	bsp.h Include File	28
4.6	tm27.h Include File	29
4.7	sbrk() Implementation	30
4.8	bsp_fatal_extension() - Cleanup the Hardware	31
4.9	Configuration Macros	32
4.10	set_vector() - Install an Interrupt Vector	33
4.11	Interrupt Delay Profiling	34
4.12	Programmable Interrupt Controller API	35
5	Initialization Code	37
5.1	Introduction	38
5.2	Required Global Variables	39
5.3	Board Initialization	40
5.3.1	Start Code - Assembly Language Initialization	40
5.3.2	boot_card() - Boot the Card	40
5.3.3	bsp_work_area_initialize() - BSP Specific Work Area Initialization	41

5.3.4	bsp_start() - BSP Specific Initialization	41
5.3.5	Device Driver Initialization	41
5.4	The Interrupt Vector Table	43
5.4.1	Interrupt Vector Table on the gen68340 BSP	43
5.5	Chip Select Initialization	44
5.6	Integrated Processor Registers Initialization	45
5.7	Data Section Recopy	46
5.8	The RTEMS Configuration Table	47
6	Console Driver	49
6.1	Introduction	50
6.2	Build System and Files	51
6.3	Driver Functioning Modes	52
6.4	Polled Mode	53
6.5	Interrupt Driven Mode	54
6.6	First Open	56
6.7	Last Close	57
6.8	Set Attributes	58
6.9	IO Control	59
6.10	Flow Control	60
6.11	General Initialization	61
7	Clock Driver	63
7.1	Introduction	64
7.2	Initialization	65
7.2.1	Timecounter Variant	65
7.2.2	Simple Timecounter Variant	66
7.2.3	Clock Tick Only Variant	67
7.3	Install Clock Tick Interrupt Service Routine	68
7.4	Support At Tick	69
7.5	System Shutdown Support	70
7.6	SMP Support	71
7.7	Multiple Clock Driver Ticks Per Clock Tick	72
7.8	Clock Driver Ticks Counter	73
8	Entropy Source	75
9	I2C Driver	77
10	SPI Driver	79
11	Real-Time Clock Driver	81
11.1	Introduction	82
11.2	Initialization	84
11.3	setRealTimeToRTEMS	85
11.4	setRealTimeFromRTEMS	86
11.5	getRealTime	87
11.6	setRealTime	88
11.7	checkRealTime	89
12	Networking Driver	91
12.1	Introduction	92
12.2	Learn about the network device	93

12.3	Understand the network scheduling conventions	94
12.4	Network Driver Makefile	95
12.5	Write the Driver Attach Function	96
12.6	Write the Driver Start Function.	98
12.7	Write the Driver Initialization Function.	99
12.8	Write the Driver Transmit Task	100
12.9	Write the Driver Receive Task	101
12.10	Write the Driver Interrupt Handler	102
12.11	Write the Driver IOCTL Function	103
12.12	Write the Driver Statistic-Printing Function	104
13	Frame Buffer Driver	105
13.1	Introduction	106
13.2	Driver Function Overview	107
13.2.1	Initialization	107
13.2.2	Opening the Frame Buffer Device	107
13.2.3	Closing the Frame Buffer Device	108
13.2.4	Reading from the Frame Buffer Device	108
13.2.5	Writing to the Frame Buffer Device	109
13.2.6	Frame Buffer IO Control	109
14	Ada95 Interrupt Support	111
14.1	Introduction	112
14.2	Mapping Interrupts to POSIX Signals	113
14.3	Example Ada95 Interrupt Program	114
14.4	Version Requirements	115
15	Shared Memory Support Driver	117
15.1	Shared Memory Configuration Table	118
15.2	Primitives	120
15.2.1	Convert Address	120
15.2.2	Get Configuration	120
15.2.3	Locking Primitives	120
15.2.3.1	Initializing a Shared Lock	121
15.2.3.2	Acquiring a Shared Lock	121
15.2.3.3	Releasing a Shared Lock	121
15.3	Installing the MPC1 ISR	123
16	Timer Driver	125
16.1	Benchmark Timer	126
16.1.1	benchmark_timer_initialize	126
16.1.2	Read_timer	126
16.1.3	benchmark_timer_disable_subtracting_average_overhead	126
16.2	gen68340 UART FIFO Full Mode	127
17	ATA Driver	129
17.1	Terms	130
17.2	Introduction	131
17.3	Initialization	132
17.4	ATA Driver Architecture	133
17.4.1	ATA Driver Main Internal Data Structures	133
17.4.2	Brief ATA Driver Core Overview	134

18 IDE Controller Driver	135
18.1 Introduction	136
18.2 Initialization	137
18.3 Read IDE Controller Register	138
18.4 Write IDE Controller Register	139
18.5 Read Data Block Through IDE Controller Data Register	140
18.6 Write Data Block Through IDE Controller Data Register	141
19 Command and Variable Index	143
20 Doxygen Recommendations for BSPs	145
20.1 BSP Basics	146
20.2 Common Features Found In BSPs	147
20.3 Shared Features	148
20.4 Rationale	149
20.5 The Structure of the bsp/ directory	150
20.6 Doxygen	152
20.7 Doxygen Basics	153
20.8 Doxygen Headers	154
20.9 The @defgroup Command	155
20.10The @ingroup Command	156
20.11The @brief Command	157
20.12The Two Types of Doxygen Headers	158
20.13Generating Documentation	160
20.14Doxygen in bsp/	161
20.15Group Naming Conventions	162
20.16Where to place @defgroup	163
20.17@defgroups for CPU Architectures and Shared Directories	164
20.18@defgroups for BSPs	165
20.19@defgroups for Everything Else	166
20.20Look Common Features Implemented	167
20.21Check out the Makefile	168
20.22Start with a .h, and look for files that include it	169
20.23Files with similar names	170
20.24Where to place @ingroup	171
20.25@ingroup in the first type of Doxygen Header	172
20.26@ingroup in the second type of Doxygen Header	173
20.27@ingroup for shared code	174
Index	175

Copyrights and License

© 2017 Christian Mauderer
© 2016, 2018 embedded brains GmbH
© 2016, 2018 Sebastian Huber
© 1988, 2017 On-Line Applications Research Corporation (OAR)

This document is available under the [Creative Commons Attribution-ShareAlike 4.0 International Public License](#).

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

The RTEMS Project is hosted at <https://www.rtems.org>. Any inquiries concerning RTEMS, its related support components, or its documentation should be directed to the RTEMS Project community.

RTEMS Online Resources

Home	https://www.rtems.org
Documentation	https://docs.rtems.org
Mailing Lists	https://lists.rtems.org
Bug Reporting	https://devel.rtems.org/wiki/Developer/Bug_Reporting
Git Repositories	https://git.rtems.org
Developers	https://devel.rtems.org

INTRODUCTION

This document describes how to create or modify a Board Support Package (BSP) for RTEMS, i.e. how to port RTEMS on a new microcontroller, system on chip (SoC) or board. It is strongly recommended to notify the [RTEMS development mailing](#) about any activity in this area and maybe also [add tickets](#) for specific work packages.

A basic BSP consists of the following components:

- Low-level initialization
- Console driver
- Clock driver

TARGET DEPENDENT FILES

<p>Warning: This chapter contains outdated and confusing information.</p>
--

RTEMS has a multi-layered approach to portability. This is done to maximize the amount of software that can be reused. Much of the RTEMS source code can be reused on all RTEMS platforms. Other parts of the executive are specific to hardware in some sense. RTEMS classifies target dependent code based upon its dependencies into one of the following categories.

- CPU dependent
- Board dependent
- Peripheral dependent

2.1 CPU Dependent

This class of code includes the foundation routines for the executive proper such as the context switch and the interrupt subroutine implementations. Sources for the supported processor families can be found in `cpukit/score/cpu`. A good starting point for a new family of processors is the `no_cpu` directory, which holds both prototypes and descriptions of each needed CPU dependent function.

CPU dependent code is further subcategorized if the implementation is dependent on a particular CPU model. For example, the MC68000 and MC68020 processors are both members of the m68k CPU family but there are significant differences between these CPU models which RTEMS must take into account.

The source code found in the `cpukit/score/cpu` is required to only depend upon the CPU model variations that GCC distinguishes for the purposes of multilib'ing. Multilib is the term the GNU community uses to refer to building a single library source multiple times with different compiler options so the binary code generated is compatible. As an example, from GCC's perspective, many PowerPC CPU models are just a PPC603e. Remember that GCC only cares about the CPU code itself and need not be aware of any peripherals. In the embedded community, we are exposed to thousands of CPU models which are all based upon only a relative small number of CPU cores.

Similarly for the SPARC/ERC32 BSP, the `RTEMS_CPU` is specified as `erc32` which is the name of the CPU model and BSP for this SPARC V7 system on chip. But the multilib variant used is actually `v7` which indicates the ERC32 CPU core is a SPARC V7.

2.2 Board Dependent

This class of code provides the most specific glue between RTEMS and a particular board. This code is represented by the Board Support Packages and associated Device Drivers. Sources for the BSPs included in the RTEMS distribution are located in the directory `bsps`. The BSP source directory is further subdivided based on the CPU family and BSP.

Some BSPs may support multiple board models within a single board family. This is necessary when the board supports multiple variants on a single base board. For example, the Motorola MVME162 board family has a fairly large number of variations based upon the particular CPU model and the peripherals actually placed on the board.

2.3 Peripheral Dependent

This class of code provides a reusable library of peripheral device drivers which can be tailored easily to a particular board. The libchip library is a collection of reusable software objects that correspond to standard controllers. Just as the hardware engineer chooses a standard controller when designing a board, the goal of this library is to let the software engineer do the same thing.

The source code for the reusable peripheral driver library may be found in the directory [cpukit/dev](#) or [bsps/shared/dev](#). The source code is further divided based upon the class of hardware. Example classes include serial communications controllers, real-time clocks, non-volatile memory, and network controllers.

2.4 Questions to Ask

When evaluating what is required to support RTEMS applications on a particular target board, the following questions should be asked:

- Does a BSP for this board exist?
- Does a BSP for a similar board exists?
- Is the board's CPU supported?

If there is already a BSP for the board, then things may already be ready to start developing application software. All that remains is to verify that the existing BSP provides device drivers for all the peripherals on the board that the application will be using. For example, the application in question may require that the board's Ethernet controller be used and the existing BSP may not support this.

If the BSP does not exist and the board's CPU model is supported, then examine the reusable chip library and existing BSPs for a close match. Other BSPs and libchip provide starting points for the development of a new BSP. It is often possible to copy existing components in the reusable chip library or device drivers from BSPs from different CPU families as the starting point for a new device driver. This will help reduce the development effort required.

If the board's CPU family is supported but the particular CPU model on that board is not, then the RTEMS port to that CPU family will have to be augmented. After this is done, development of the new BSP can proceed.

Otherwise both CPU dependent code and the BSP will have to be written.

This type of development often requires specialized skills and there are people in the community who provide those services. If you need help in making these modifications to RTEMS try a search in a search engine with something like "RTEMS support". The RTEMS Project encourages users to use support services however we do not endorse any providers.

2.5 CPU Dependent Executive Files

The CPU dependent files in the RTEMS executive source code are found in the `cpukit/score/cpu/${RTEMS_CPU}` directories. The `${RTEMS_CPU}` is a particular architecture, e.g. `arm`, `powerpc`, `riscv`, `sparc`, etc.

Within each CPU dependent directory inside the executive proper is a file named `cpu.h` which contains information about each of the supported CPU models within that family.

2.6 Board Support Package Structure

The BSPs are all under the `bsps` directory. The structure in this source subtree is:

- `bsps/shared`
- `bsps/${RTEMS_CPU}/shared`
- `bsps/${RTEMS_CPU}/${RTEMS_BSP_FAMILY}`

The `${RTEMS_CPU}` is a particular architecture, e.g. `arm`, `powerpc`, `riscv`, `sparc`, etc. The shared directories contain code shared by all BSPs or BSPs of a particular architecture. The `${RTEMS_BSP_FAMILY}` directories contain BSPs for a particular system on chip (SoC) or processor family.

Use the following structure under the `bsps/${RTEMS_CPU}/${RTEMS_BSP_FAMILY}`:

- `ata` - the legacy ATA/IDE driver
- `btimer` - the legacy benchmark timer driver
- `cache` - cache controller support
- `clock` - the clock driver
- `config` - build system configuration files
- `console` - the console driver
- `contrib` - imports of external sources
 - the layout of external sources should be used as is if possible
- `i2c` - the I2C driver
- `include` - public header files
- `irq` - the interrupt controller support
- `mpci` - support for heterogeneous multiprocessing (`RTEMS_MULTIPROCESSING`)
- `net` - legacy network stack drivers
- `rtc` - the RTC driver
- `spi` - the SPI driver
- `start` - everything required to run a minimal application without devices
 - `start.S` - lowest level startup code
 - `bspstart.c` - low level startup code
 - `bspsmp.c` - SMP support
 - `linkcmds` - a linker command file

LINKER SCRIPT

<p>Warning: This chapter contains outdated and confusing information.</p>
--

3.1 What is a “linkcmds” file?

The `linkcmds` file is a script which is passed to the linker at linking time. This file describes the memory configuration of the board as needed to link the program. Specifically it specifies where the code and data for the application will reside in memory.

The format of the linker script is defined by the GNU Loader `ld` which is included as a component of the GNU Binary Utilities. If you are using GNU/Linux, then you probably have the documentation installed already and are using these same tools configured for *native* use. Please visit the Binutils project <http://sourceware.org/binutils/> if you need more information.

3.2 Program Sections

An embedded systems programmer must be much more aware of the placement of their executable image in memory than the average applications programmer. A program destined to be embedded as well as the target system have some specific properties that must be taken into account. Embedded machines often mean average performances and small memory usage. It is the memory usage that concerns us when examining the linker command file.

Two types of memories have to be distinguished:

- RAM - volatile offering read and write access
- ROM - non-volatile but read only

Even though RAM and ROM can be found in every personal computer, one generally doesn't care about them. In a personal computer, a program is nearly always stored on disk and executed in RAM. Things are a bit different for embedded targets: the target will execute the program each time it is rebooted or switched on. The application program is stored in non-volatile memory such as ROM, PROM, EEPROM, or Flash. On the other hand, data processing occurs in RAM.

This leads us to the structure of an embedded program. In rough terms, an embedded program is made of sections. It is the responsibility of the application programmer to place these sections in the appropriate place in target memory. To make this clearer, if using the COFF object file format on the Motorola m68k family of microprocessors, the following sections will be present:

- code (.text) section: is the program's code and it should not be modified. This section may be placed in ROM.
- non-initialized data (.bss) section: holds uninitialized variables of the program. It can stay in RAM.
- initialized data (.data) section: holds the initialized program data which may be modified during the program's life. This means they have to be in RAM. On the other hand, these variables must be set to predefined values, and those predefined values have to be stored in ROM.

Note: Many programs and support libraries unknowingly assume that the .bss section and, possibly, the application heap are initialized to zero at program start. This is not required by the ISO/ANSI C Standard but is such a common requirement that most BSPs do this.

That brings us up to the notion of the image of an executable: it consists of the set of the sections that together constitute the application.

3.3 Image of an Executable

As a program executable has many sections (note that the user can define their own, and that compilers define theirs without any notice), one has to specify the placement of each section as well as the type of memory (RAM or ROM) the sections will be placed into. For instance, a program compiled for a Personal Computer will see all the sections to go to RAM, while a program destined to be embedded will see some of his sections going into the ROM.

The connection between a section and where that section is loaded into memory is made at link time. One has to let the linker know where the different sections are to be placed once they are in memory.

The following example shows a simple layout of program sections. With some object formats, there are many more sections but the basic layout is conceptually similar.

.text	RAM or ROM
.data	RAM
.bss	RAM

3.4 Example Linker Command Script

The GNU linker has a command language to specify the image format. This command language can be quite complicated but most of what is required can be learned by careful examination of a well-documented example. The following is a heavily commented version of the linker script used with the the gen68340 BSP This file can be found at \$BSP340_ROOT/startup/linkcmds.

```

1  /*
2  *  Specify that the output is to be coff-m68k regardless of what the
3  *  native object format is.
4  */
5  OUTPUT_FORMAT(coff-m68k)
6  /*
7  *  Set the amount of RAM on the target board.
8  *
9  *  NOTE: The default may be overridden by passing an argument to ld.
10 */
11 RamSize = DEFINED(RamSize) ? RamSize : 4M;
12 /*
13 *  Set the amount of RAM to be used for the application heap.  Objects
14 *  allocated using malloc() come from this area.  Having a tight heap
15 *  size is somewhat difficult and multiple attempts to squeeze it may
16 *  be needed reducing memory usage is important.  If all objects are
17 *  allocated from the heap at system initialization time, this eases
18 *  the sizing of the application heap.
19 *
20 *  NOTE 1: The default may be overridden by passing an argument to ld.
21 *
22 *  NOTE 2: The TCP/IP stack requires additional memory in the Heap.
23 *
24 *  NOTE 3: The GNAT/RTEMS run-time requires additional memory in
25 *  the Heap.
26 */
27 HeapSize = DEFINED(HeapSize) ? HeapSize : 0x10000;
28 /*
29 *  Set the size of the starting stack used during BSP initialization
30 *  until first task switch.  After that point, task stacks allocated
31 *  by RTEMS are used.
32 *
33 *  NOTE: The default may be overridden by passing an argument to ld.
34 */
35 StackSize = DEFINED(StackSize) ? StackSize : 0x1000;
36 /*
37 *  Starting addresses and length of RAM and ROM.
38 *
39 *  The addresses must be valid addresses on the board.  The
40 *  Chip Selects should be initialized such that the code addresses
41 *  are valid.
42 */
43 MEMORY {
44 ram : ORIGIN = 0x10000000, LENGTH = 4M
45 rom : ORIGIN = 0x01000000, LENGTH = 4M
46 }
47 /*
48 *  This is for the network driver.  See the Networking documentation
49 *  for more details.
50 */

```

```

51 ETHERNET_ADDRESS =
52 DEFINED(ETHERNET_ADDRESS) ? ETHERNET_ADDRESS : 0xDEAD12;
53 /*
54  * The following defines the order in which the sections should go.
55  * It also defines a number of variables which can be used by the
56  * application program.
57  *
58  * NOTE: Each variable appears with 1 or 2 leading underscores to
59  *       ensure that the variable is accessible from C code with a
60  *       single underscore. Some object formats automatically add
61  *       a leading underscore to all C global symbols.
62  */
63 SECTIONS {
64 /*
65  * Make the RomBase variable available to the application.
66  */
67 _RamSize = RamSize;
68 __RamSize = RamSize;
69 /*
70  * Boot PROM - Set the RomBase variable to the start of the ROM.
71  */
72 rom : {
73   _RomBase = .;
74   __RomBase = .;
75 } >rom
76 /*
77  * Dynamic RAM - set the RamBase variable to the start of the RAM.
78  */
79 ram : {
80   _RamBase = .;
81   __RamBase = .;
82 } >ram
83 /*
84  * Text (code) goes into ROM
85  */
86 .text : {
87   /*
88    * Create a symbol for each object (.o).
89    */
90   CREATE_OBJECT_SYMBOLS
91   /*
92    * Put all the object files code sections here.
93    */
94   *(.text)
95   . = ALIGN (16);      /* go to a 16-byte boundary */
96   /*
97    * C++ constructors and destructors
98    *
99    * NOTE: See the CROSSGCC mailing-list FAQ for
100    *       more details about the "[.....]".
101    */
102   __CTOR_LIST__ = .;
103   [.....]
104   __DTOR_END__ = .;
105   /*
106    * Declares where the .text section ends.
107    */

```



```

108  etext = .;
109  _etext = .;
110 } >rom
111 /*
112  * Exception Handler Frame section
113  */
114 .eh_fram : {
115     . = ALIGN (16);
116     *(.eh_fram)
117 } >ram
118 /*
119  * GCC Exception section
120  */
121 .gcc_exc : {
122     . = ALIGN (16);
123     *(.gcc_exc)
124 } >ram
125 /*
126  * Special variable to let application get to the dual-ported
127  * memory.
128  */
129 dpram : {
130     m340 = .;
131     _m340 = .;
132     . += (8 * 1024);
133 } >ram
134 /*
135  * Initialized Data section goes in RAM
136  */
137 .data : {
138     copy_start = .;
139     *(.data)
140     . = ALIGN (16);
141     _edata = .;
142     copy_end = .;
143 } >ram
144 /*
145  * Uninitialized Data section goes in ROM
146  */
147 .bss : {
148     /*
149     * M68K specific: Reserve some room for the Vector Table
150     * (256 vectors of 4 bytes).
151     */
152     M68Kvec = .;
153     _M68Kvec = .;
154     . += (256 * 4);
155     /*
156     * Start of memory to zero out at initialization time.
157     */
158     clear_start = .;
159     /*
160     * Put all the object files uninitialized data sections
161     * here.
162     */
163     *(.bss)
164     *(COMMON)

```

```
165 . = ALIGN (16);
166 _end = .;
167 /*
168  * Start of the Application Heap
169  */
170 _HeapStart = .;
171 __HeapStart = .;
172 . += HeapSize;
173 /*
174  * The Starting Stack goes after the Application Heap.
175  * M68K stack grows down so start at high address.
176  */
177 . += StackSize;
178 . = ALIGN (16);
179 stack_init = .;
180 clear_end = .;
181 /*
182  * The RTEMS Executive Workspace goes here. RTEMS
183  * allocates tasks, stacks, semaphores, etc. from this
184  * memory.
185  */
186 _WorkspaceBase = .;
187 __WorkspaceBase = .;
188 } >ram
```

3.5 Initialized Data

Now there's a problem with the initialized data: the `.data` section has to be in RAM as this data may be modified during the program execution. But how will the values be initialized at boot time?

One approach is to place the entire program image in RAM and reload the image in its entirety each time the program is run. This is fine for use in a debug environment where a high-speed connection is available between the development host computer and the target. But even in this environment, it is cumbersome.

The solution is to place a copy of the initialized data in a separate area of memory and copy it into the proper location each time the program is started. It is common practice to place a copy of the initialized `.data` section at the end of the code (`.text`) section in ROM when building a PROM image. The GNU tool `objcopy` can be used for this purpose.

The following figure illustrates the steps a linked program goes through to become a downloadable image.

.data (RAM)		.data (RAM)	
.bss (RAM)		.bss (RAM)	
.text (ROM)		.text (ROM)	.text
copy of .data (ROM)		copy of .data	
Step 1	Step 2	Step 3	

In Step 1, the program is linked together using the BSP linker script.

In Step 2, a copy is made of the `.data` section and placed after the `.text` section so it can be placed in PROM. This step is done after the linking time. There is an example of doing this in the file `$RTMEMS_ROOT/make/custom/gen68340.cfg`:

```

1 # make a PROM image using objcopy
2 m68k-rtmems-objcopy --adjust-section-vma \
3 .data=`m68k-rtmems-objdump --section-headers $(basename $@).exe | awk '[[...]]'` \
4 $(basename $@).exe

```

Note: The address of the “copy of `.data` section” is created by extracting the last address in the `.text` section with an `awk` script. The details of how this is done are not relevant.

Step 3 shows the final executable image as it logically appears in the target's non-volatile program memory. The board initialization code will copy the “copy of `.data` section” (which are stored in ROM) to their reserved location in RAM.

MISCELLANEOUS SUPPORT FILES

<p>Warning: This chapter contains outdated and confusing information.</p>
--

4.1 GCC Compiler Specifications File

The file `bsp_specs` defines the start files and libraries that are always used with this BSP. The format of this file is admittedly cryptic and this document will make no attempt to explain it completely. Below is the `bsp_specs` file from the PowerPC psim BSP:

```

1 %rename endfile old_endfile
2 %rename startfile old_startfile
3 %rename link old_link
4 *startfile:
5 %{!qrtems: %(old_startfile)} \
6 %{!nostdlib: %{qrtems: crt0.o rtems_crti.o crtbegin.o start.o}}
7 *link:
8 %{!qrtems: %(old_link)} %{qrtems: -Qy -dp -Bstatic -e _start -u __vectors}
9 *endfile:
10 %{!qrtems: %(old_endfile)} %{qrtems: crtend.o crtn.o}

```

The first section of this file renames the built-in definition of some specification variables so they can be augmented without embedded their original definition. The subsequent sections specify what behavior is expected when the `-qrtems` option is specified.

The `*startfile` section specifies that the BSP specific file `start.o` will be used instead of `crt0.o`. In addition, various EABI support files (`crti.o` etc.) will be linked in with the executable.

The `*link` section adds some arguments to the linker when it is invoked by GCC to link an application for this BSP.

The format of this file is specific to the GNU Compiler Suite. The argument used to override and extend the compiler built-in specifications is available in all recent GCC versions. The `-specs` option is present in all egcs distributions and gcc distributions starting with version 2.8.0.

4.2 README Files

Most BSPs provide one or more README files. Generally, there is a README file at the top of the BSP source. This file describes the board and its hardware configuration, provides vendor information, local configuration information, information on downloading code to the board, debugging, etc.. The intent of this file is to help someone begin to use the BSP faster.

A README file in a BSP subdirectory typically explains something about the contents of that subdirectory in greater detail. For example, it may list the documentation available for a particular peripheral controller and how to obtain that documentation. It may also explain some particularly cryptic part of the software in that directory or provide rationale on the implementation.

4.3 Times

This file contains the results of the RTEMS Timing Test Suite. It is in a standard format so that results from one BSP can be easily compared with those of another target board.

If a BSP supports multiple variants, then there may be multiple times files. Usually these are named times.VARIANTn.

4.4 Tools Subdirectory

Some BSPs provide additional tools that aid in using the target board. These tools run on the development host and are built as part of building the BSP. Most common is a script to automate running the RTEMS Test Suites on the BSP. Examples of this include:

- powerpc/psim includes scripts to ease use of the simulator
- m68k/mvme162 includes a utility to download across the VMEbus into target memory if the host is a VMEbus board in the same chassis.

4.5 bsp.h Include File

The file `include/bsp.h` contains prototypes and definitions specific to this board. Every BSP is required to provide a `bsp.h`. The best approach to writing a `bsp.h` is copying an existing one as a starting point.

Many `bsp.h` files provide prototypes of variables defined in the linker script (`linkcmds`).

4.6 tm27.h Include File

The tm27 test from the RTEMS Timing Test Suite is designed to measure the length of time required to vector to and return from an interrupt handler. This test requires some help from the BSP to know how to cause and manipulate the interrupt source used for this measurement. The following is a list of these:

- MUST_WAIT_FOR_INTERRUPT - modifies behavior of tm27.
- Install_tm27_vector - installs the interrupt service routine for the Interrupt Benchmark Test (tm27).
- Cause_tm27_intr - generates the interrupt source used in the Interrupt Benchmark Test (tm27).
- Clear_tm27_intr - clears the interrupt source used in the Interrupt Benchmark Test (tm27).
- Lower_tm27_intr - lowers the interrupt mask so the interrupt source used in the Interrupt Benchmark Test (tm27) can generate a nested interrupt.

All members of the Timing Test Suite are designed to run *WITHOUT* the Clock Device Driver installed. This increases the predictability of the tests' execution as well as avoids occasionally including the overhead of a clock tick interrupt in the time reported. Because of this it is sometimes possible to use the clock tick interrupt source as the source of this test interrupt. On other architectures, it is possible to directly force an interrupt to occur.

4.7 sbrk() Implementation

Although nearly all BSPs give all possible memory to the C Program Heap at initialization, it is possible for a BSP to configure the initial size of the heap small and let it grow on demand. If the BSP wants to dynamically extend the heap used by the C Library memory allocation routines (i.e. malloc family), then the “sbrk” routine must be functional. The following is the prototype for this routine:

```
1 void * sbrk(ptrdiff_t increment)
```

The increment amount is based upon the sbrk_amount parameter passed to the bsp_libc_init during system initialization.

If your BSP does not want to support dynamic heap extension, then you do not have to do anything special. However, if you want to support sbrk, you must provide an implementation of this method and define `CONFIGURE_MALLOC_BSP_SUPPORTS_SBRK` in `bsp.h`. This informs `rtems/confdefs.h` to configure the Malloc Family Extensions which support sbrk.

4.8 bsp_fatal_extension() - Cleanup the Hardware

The `bsp_fatal_extension()` is an optional BSP specific initial extension invoked once a fatal system state is reached. Most of the BSPs use the same shared version of `bsp_fatal_extension()` that does nothing or performs a system reset. This implementation is located in the [bsps/shared/start/bspfatal-default.c](#) file.

The `bsp_fatal_extension()` routine can be used to return to a ROM monitor, insure that interrupt sources are disabled, etc.. This routine is the last place to ensure a clean shutdown of the hardware. The fatal source, internal error indicator, and the fatal code arguments are available to evaluate the fatal condition. All of the non-fatal shutdown sequences ultimately pass their exit status to `rtems_shutdown_executive` and this is what is passed to this routine in case the fatal source is `RTEMS_FATAL_SOURCE_EXIT`.

On some BSPs, it prints a message indicating that the application completed execution and waits for the user to press a key before resetting the board. The PowerPC/gen83xx and PowerPC/gen5200 BSPs do this when they are built to support the FreeScale evaluation boards. This is convenient when using the boards in a development environment and may be disabled for production use.

4.9 Configuration Macros

Each BSP can define macros in `bsp.h` which alter some of the the default configuration parameters in `rtems/confdefs.h`. This section describes those macros:

- `CONFIGURE_MALLOC_BSP_SUPPORTS_SBRK` must be defined if the BSP has proper support for `sbrk`. This is discussed in more detail in the previous section.
- `BSP_IDLE_TASK_BODY` may be defined to the entry point of a BSP specific IDLE thread implementation. This may be overridden if the application provides its own IDLE task implementation.
- `BSP_IDLE_TASK_STACK_SIZE` may be defined to the desired default stack size for the IDLE task as recommended when using this BSP.
- `BSP_INTERRUPT_STACK_SIZE` may be defined to the desired default interrupt stack size as recommended when using this BSP. This is sometimes required when the BSP developer has knowledge of stack intensive interrupt handlers.
- `BSP_ZERO_WORKSPACE_AUTOMATICALLY` is defined when the BSP requires that RTEMS zero out the RTEMS C Program Heap at initialization. If the memory is already zeroed out by a test sequence or boot ROM, then the boot time can be reduced by not zeroing memory twice.
- `BSP_DEFAULT_UNIFIED_WORK_AREAS` is defined when the BSP recommends that the unified work areas configuration should always be used. This is desirable when the BSP is known to always have very little RAM and thus saving memory by any means is desirable.

4.10 set_vector() - Install an Interrupt Vector

On targets with Simple Vectored Interrupts, the BSP must provide an implementation of the `set_vector` routine. This routine is responsible for installing an interrupt vector. It invokes the support routines necessary to install an interrupt handler as either a “raw” or an RTEMS interrupt handler. Raw handlers bypass the RTEMS interrupt structure and are responsible for saving and restoring all their own registers. Raw handlers are useful for handling traps, debug vectors, etc.

The `set_vector` routine is a central place to perform interrupt controller manipulation and encapsulate that information. It is usually implemented as follows:

```
1 rtems_isr_entry set_vector(           /* returns old vector */
2   rtems_isr_entry handler,           /* isr routine      */
3   rtems_vector_number vector,        /* vector number    */
4   int type                           /* RTEMS or RAW intr */
5 )
6 {
7   if the type is RAW
8     install the raw vector
9   else
10    use rtems_interrupt_catch to install the vector
11    perform any interrupt controller necessary to unmask the interrupt source
12    return the previous handler
13 }
```

Note: The i386, PowerPC and ARM ports use a Programmable Interrupt Controller model which does not require the BSP to implement `set_vector`. BSPs for these architectures must provide a different set of support routines.

4.11 Interrupt Delay Profiling

The RTEMS profiling needs support by the BSP for the interrupt delay times. In case profiling is enabled via the RTEMS build configuration option `--enable-profiling` (in this case the pre-processor symbol `RTEMS_PROFILING` is defined) a BSP may provide data for the interrupt delay times. The BSP can feed interrupt delay times with the `_Profiling_Update_max_interrupt_delay()` function (`#include <rtems/score/profiling.h>`). For an example please have a look at [bsps/sparc/leon3/clock/ckinit.c](#).

4.12 Programmable Interrupt Controller API

A BSP can use the PIC API to install Interrupt Service Routines through a set of generic methods. In order to do so, the header files `<bsp/irq-generic.h>` and `<bsp/irq-info.h>` must be included by the bsp specific `irq.h` file present in the `include/` directory. The `irq.h` acts as a BSP interrupt support configuration file which is used to define some important MACROS. It contains the declarations for any required global functions like `bsp_interrupt_dispatch()`. Thus later on, every call to the PIC interface requires including `<bsp/irq.h>`

The generic interrupt handler table is initialized by invoking the `bsp_interrupt_initialize()` method from `bsp_start()` in the `bspstart.c` file which sets up this table to store the ISR addresses, whose size is based on the definition of macros, `BSP_INTERRUPT_VECTOR_MIN` and `BSP_INTERRUPT_VECTOR_MAX` in `include/bsp.h`

For the generic handler table to properly function, some bsp specific code is required, that should be present in `irq/irq.c`. The bsp-specific functions required to be written by the BSP developer are :

- `bsp_interrupt_facility_initialize()` contains bsp specific interrupt initialization code (Clear Pending interrupts by modifying registers, etc.). This method is called from `bsp_interrupt_initialize()` internally while setting up the table.
- `bsp_interrupt_handler_default()` acts as a fallback handler when no ISR address has been provided corresponding to a vector in the table.
- `bsp_interrupt_dispatch()` service the ISR by handling any bsp specific code & calling the generic method `bsp_interrupt_handler_dispatch()` which in turn services the interrupt by running the ISR after looking it up in the table. It acts as an entry to the interrupt switchboard, since the bsp branches to this function at the time of occurrence of an interrupt.
- `bsp_interrupt_vector_enable()` enables interrupts and is called in `irq-generic.c` while setting up the table.
- `bsp_interrupt_vector_disable()` disables interrupts and is called in `irq-generic.c` while setting up the table & during other important parts.

An interrupt handler is installed or removed with the help of the following functions :

```

1 rtems_status_code rtems_interrupt_handler_install( /* returns status code */
2   rtems_vector_number vector, /* interrupt vector */
3   const char *info, /* custom identification text */
4   rtems_option options, /* Type of Interrupt */
5   rtems_interrupt_handler handler, /* interrupt handler */
6   void *arg /* parameter to be passed
7               to handler at the time of
8               invocation */
9 )
10 rtems_status_code rtems_interrupt_handler_remove( /* returns status code */
11   rtems_vector_number vector, /* interrupt vector */
12   rtems_interrupt_handler handler, /* interrupt handler */
13   void *arg /* parameter to be passed to handler */
14   ↪ */
15 )

```


INITIALIZATION CODE

<p>Warning: This chapter contains outdated and confusing information.</p>
--

5.1 Introduction

The initialization code is the first piece of code executed when there's a reset/reboot. Its purpose is to initialize the board for the application. This chapter contains a narrative description of the initialization process followed by a description of each of the files and routines commonly found in the BSP related to initialization. The remainder of this chapter covers special issues which require attention such as interrupt vector table and chip select initialization.

Most of the examples in this chapter will be based on the SPARC/ERC32 and m68k/gen68340 BSP initialization code. Like most BSPs, the initialization for these BSP is contained under the start directory in the BSP source directory. The BSP source code for these BSPs is in the following directories:

```
1 bsps/m68k/gen68340
2 bsps/sparc/erc32
```

Both BSPs contain startup code written in assembly language and C. The gen68340 BSP has its early initialization start code in the start340 subdirectory and its C startup code in the startup directory. In the start340 directory are two source files. The file startfor340only.s is the simpler of these files as it only has initialization code for a MC68340 board. The file start340.s contains initialization for a 68349 based board as well.

Similarly, the ERC32 BSP has startup code written in assembly language and C. However, this BSP shares this code with other SPARC BSPs. Thus the Makefile.am explicitly references the following files for this functionality.

```
1 ../../sparc/shared/start.S
```

Note: In most BSPs, the directory named start340 in the gen68340 BSP would be simply named start or start followed by a BSP designation.

5.2 Required Global Variables

Although not strictly part of initialization, there are a few global variables assumed to exist by reusable device drivers. These global variables should only be defined by the BSP when using one of these device drivers.

The BSP author probably should be aware of the Configuration Table structure generated by `<rtems/confdefs.h>` during debug but should not explicitly reference it in the source code. There are helper routines provided by RTEMS to access individual fields.

In older RTEMS versions, the BSP included a number of required global variables. We have made every attempt to eliminate these in the interest of simplicity.

5.3 Board Initialization

This section describes the steps an application goes through from the time the first BSP code is executed until the first application task executes.

The initialization flows from assembly language start code to the shared `bootcard.c` framework then through the C Library, RTEMS, device driver initialization phases, and the context switch to the first application task. After this, the application executes until it calls `exit`, `rtems_shutdown_executive`, or some other normal termination initiating routine and a fatal system state is reached. The optional `bsp_fatal_extension` initial extension can perform BSP specific system termination.

The routines invoked during this will be discussed and their location in the RTEMS source tree pointed out as we discuss each.

5.3.1 Start Code - Assembly Language Initialization

The assembly language code in the directory `start` is the first part of the application to execute. It is responsible for initializing the processor and board enough to execute the rest of the BSP. This includes:

- initializing the stack
- zeroing out the uninitialized data section `.bss`
- disabling external interrupts
- copy the initialized data from ROM to RAM

The general rule of thumb is that the start code in assembly should do the minimum necessary to allow C code to execute to complete the initialization sequence.

The initial assembly language start code completes its execution by invoking the shared routine `boot_card()`.

The label (symbolic name) associated with the starting address of the program is typically called `start`. The start object file is the first object file linked into the program image so it is ensured that the start code is at offset 0 in the `.text` section. It is the responsibility of the linker script in conjunction with the compiler specifications file to put the start code in the correct location in the application image.

5.3.2 `boot_card()` - Boot the Card

The `boot_card()` is the first C code invoked. This file is the core component in the RTEMS BSP Initialization Framework and provides the proper sequencing of initialization steps for the BSP, RTEMS and device drivers. All BSPs use the same shared version of `boot_card()` which is located in the `bsp/shared/start/bootcard.c` file.

The `boot_card()` routine performs the following functions:

- It disables processor interrupts.
- It sets the command line argument variables for later use by the application.

- It invokes the routine `rtems_initialize_executive()` which never returns. This routine will perform the system initialization through a linker set. The important BSP-specific steps are outlined below.
- Initialization of the RTEMS Workspace and the C Program Heap. Usually the default implementation in `bsps/shared/start/bspgetworkarea-default.c` should be sufficient. Custom implementations can use `bsp_work_area_initialize_default()` or `bsp_work_area_initialize_with_table()` available as inline functions from `#include <bsp/bootcard.h>`.
- Invocation of the BSP-specific routine `bsp_start()` which is written in C and thus able to perform more advanced initialization. Often MMU, bus and interrupt controller initialization occurs here. Since the RTEMS Workspace and the C Program Heap was already initialized by `bsp_work_area_initialize()`, this routine may use `malloc()`, etc.
- Specific initialization steps can be registered via the `RTEMS_SYSINIT_ITEM()` provided by `#include <rtems/sysinit.h>`.

5.3.3 `bsp_work_area_initialize()` - BSP Specific Work Area Initialization

This is the first BSP specific C routine to execute during system initialization. It must initialize the support for allocating memory from the C Program Heap and RTEMS Workspace commonly referred to as the work areas. Many BSPs place the work areas at the end of RAM although this is certainly not a requirement. Usually the default implementation in `bsps/shared/start/bspgetworkarea-default.c` should be sufficient. Custom implementations can use `bsp_work_area_initialize_default()` or `bsp_work_area_initialize_with_table()` available as inline functions from `#include <bsp/bootcard.h>`.

5.3.4 `bsp_start()` - BSP Specific Initialization

This is the second BSP specific C routine to execute during system initialization. It is called right after `bsp_work_area_initialize()`. The `bsp_start()` routine often performs required fundamental hardware initialization such as setting bus controller registers that do not have a direct impact on whether or not C code can execute. The interrupt controllers are usually initialized here. The source code for this routine is usually found in the file `bsps/${RTEMS_CPU}/${RTEMS_BSP}/start.c`. It is not allowed to create any operating system objects, e.g. RTEMS semaphores.

After completing execution, this routine returns to the `boot_card()` routine. In case of errors, the initialization should be terminated via `bsp_fatal()`.

5.3.5 Device Driver Initialization

At this point in the initialization sequence, the initialization routines for all of the device drivers specified in the Device Driver Table are invoked. The initialization routines are invoked in the order they appear in the Device Driver Table.

The Driver Address Table is part of the RTEMS Configuration Table. It defines device drivers entry points (initialization, open, close, read, write, and control). For more information about this table, please refer to the *Configuring a System* chapter in the *RTEMS Application C User's Guide*.

The RTEMS initialization procedure calls the initialization function for every driver defined in the RTEMS Configuration Table (this allows one to include only the drivers needed by the application).

All these primitives have a major and a minor number as arguments:

- the major number refers to the driver type,
- the minor number is used to control two peripherals with the same driver (for instance, we define only one major number for the serial driver, but two minor numbers for channel A and B if there are two channels in the UART).

5.4 The Interrupt Vector Table

The Interrupt Vector Table is called different things on different processor families but the basic functionality is the same. Each entry in the Table corresponds to the handler routine for a particular interrupt source. When an interrupt from that source occurs, the specified handler routine is invoked. Some context information is saved by the processor automatically when this happens. RTEMS saves enough context information so that an interrupt service routine can be implemented in a high level language.

On some processors, the Interrupt Vector Table is at a fixed address. If this address is in RAM, then usually the BSP only has to initialize it to contain pointers to default handlers. If the table is in ROM, then the application developer will have to take special steps to fill in the table.

If the base address of the Interrupt Vector Table can be dynamically changed to an arbitrary address, then the RTEMS port to that processor family will usually allocate its own table and install it. For example, on some members of the Motorola MC68xxx family, the Vector Base Register (vbr) contains this base address.

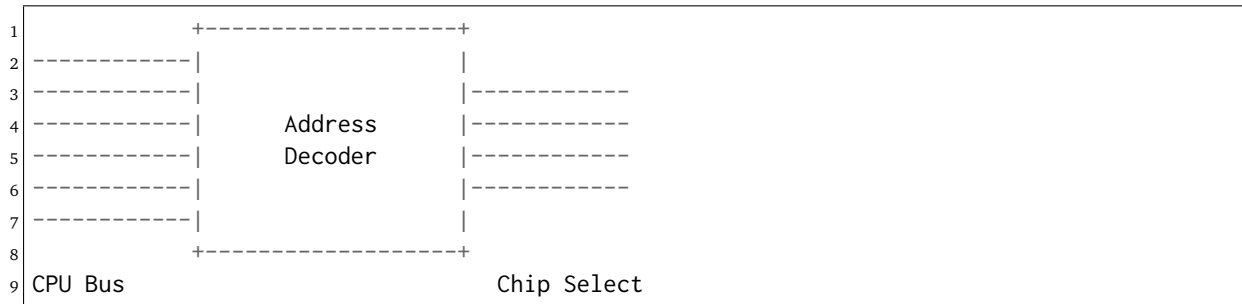
5.4.1 Interrupt Vector Table on the gen68340 BSP

The gen68340 BSP provides a default Interrupt Vector Table in the file `$BSP_ROOT/start340/start340.s`. After the entry label is the definition of space reserved for the table of interrupts vectors. This space is assigned the symbolic name of `__uhoh` in the gen68340 BSP.

At `__uhoh` label is the default interrupt handler routine. This routine is only called when an unexpected interrupts is raised. One can add their own routine there (in that case there's a call to a routine - `$BSP_ROOT/startup/dumpanic.c` - that prints which address caused the interrupt and the contents of the registers, stack, etc.), but this should not return.

5.5 Chip Select Initialization

When the microprocessor accesses a memory area, address decoding is handled by an address decoder, so that the microprocessor knows which memory chip(s) to access. The following figure illustrates this:



The Chip Select registers must be programmed such that they match the linkcmds settings. In the gen68340 BSP, ROM and RAM addresses can be found in both the linkcmds and initialization code, but this is not a great way to do this. It is better to define addresses in the linker script.

5.6 Integrated Processor Registers Initialization

The CPUs used in many embedded systems are highly complex devices with multiple peripherals on the CPU itself. For these devices, there are always some specific integrated processor registers that must be initialized. Refer to the processors' manuals for details on these registers and be VERY careful programming them.

5.7 Data Section Recopy

The next initialization part can be found in `$BSP340_ROOT/start340/init68340.c`. First the Interrupt Vector Table is copied into RAM, then the data section recopy is initiated (`_CopyDataClearBSSAndStart` in `$BSP340_ROOT/start340/startfor340only.s`).

This code performs the following actions:

- copies the `.data` section from ROM to its location reserved in RAM (see *Initialized Data* (page 21) for more details about this copy),
- clear `.bss` section (all the non-initialized data will take value 0).

5.8 The RTEMS Configuration Table

The RTEMS configuration table contains the maximum number of objects RTEMS can handle during the application (e.g. maximum number of tasks, semaphores, etc.). It's used to allocate the size for the RTEMS inner data structures.

The RTEMS configuration table is application dependent, which means that one has to provide one per application. It is usually defined by defining macros and including the header file `<rtems/confdefs.h>`. In simple applications such as the tests provided with RTEMS, it is commonly found in the main module of the application. For more complex applications, it may be in a file by itself.

The header file `<rtems/confdefs.h>` defines a constant table named `Configuration`. With RTEMS 4.8 and older, it was accepted practice for the BSP to copy this table into a modifiable copy named `BSP_Configuration`. This copy of the table was modified to define the base address of the RTEMS Executive Workspace as well as to reflect any BSP and device driver requirements not automatically handled by the application. In 4.9 and newer, we have eliminated the BSP copies of the configuration tables and are making efforts to make the configuration information generated by `<rtems/confdefs.h>` constant and read only.

For more information on the RTEMS Configuration Table, refer to the *RTEMS Application C User's Guide*.

CONSOLE DRIVER

Warning: The low-level driver API changed between RTEMS 4.10 and RTEMS 4.11. The legacy callback API is still supported, but its use is discouraged. The following functions are deprecated:

- `rtems_termios_open()`
- `rtems_termios_close()`

This manual describes the new API.

6.1 Introduction

This chapter describes the operation of a console driver using the RTEMS POSIX Termios support. Traditionally, RTEMS has referred to all serial device drivers as console drivers. **Termios** is defined by IEEE Std 1003.1-2008 (POSIX.1-2008). It supports various modes of operations at application level. This chapter focuses on the low-level serial device driver. Additional Termios information can be found in the [Linux TERMIOS\(3\)](#) manpage or the [FreeBSD TERMIOS\(4\)](#) manpage.

There are the following software layers.

Application
Termios
Low-Level Device Driver

In the default application configuration RTEMS opens during system initialization a `/dev/` console device file to create the file descriptors 0, 1 and 2 used for standard input, output and error, respectively. The corresponding device driver is usually a Termios serial device driver described here. The standard file descriptors are used by standard C library calls such as `printf()` or `scanf()` or directly via the `read()` or `write()` system calls.

6.2 Build System and Files

A new serial device driver should consist of three parts.

- A section in the BSPs Makefile.am:

```
1 [...]
2 libbsp_a_SOURCES += ../../shared/dev/serial/console-termios.c
3 libbsp_a_SOURCES += console/console.c
4 [...]
```

- A general serial device specific low-level driver providing the handler table and the device context specialization for the Termios `rtems_termios_device_install()` function. This low-level driver could be used for more than one BSP.
- A BSP-specific initialization routine `console_initialize()`, that calls `rtems_termios_device_install()` providing a low-level driver context for each installed device. This is usually defined in the file `console/console.c` relative to the BSP base directory.

The low-level driver should provide a specialization of the Termios device context. The initialization routine must provide a context for each installed device via `rtems_termios_device_install()`. Here is an example header file for a low-level serial device driver.

```
1 #ifndef MY_DRIVER_H
2 #define MY_DRIVER_H
3
4 #include <some-chip/serial.h>
5
6 #include <rtems/termiostypes.h>
7
8 /* My low-level driver specialization of Termios device context */
9 typedef struct {
10     rtems_termios_device_context base;
11     const char *device_name;
12     volatile some_chip_registers *regs;
13     /* More stuff */
14 } my_driver_context;
15
16 extern const rtems_termios_device_handler my_driver_handler_polled;
17
18 extern const rtems_termios_device_handler my_driver_handler_interrupt;
19
20 #endif /* MY_DRIVER_H */
```

6.3 Driver Functioning Modes

There are four main functioning modes for a Termios serial device driver. The mode must be set during device creation and cannot be changed afterwards.

Polled Mode (*TERMIOs_POLLED*)

In polled mode, the processor blocks on sending/receiving characters. This mode is not the most efficient way to utilize the serial device. But polled mode is usually necessary when one wants to print an error message in the event of a fatal error such as a fatal error in the BSP. This is also the simplest mode to program. Polled mode is generally preferred if the serial device is to be used primarily as a debug console. In a simple polled driver, the software will continuously check the status of the serial device when it is reading or writing to the serial device. Termios improves on this by delaying the caller for one clock tick between successive checks of the serial device on a read operation.

Interrupt Driven Mode (*TERMIOs_IRQ_DRIVEN*)

In interrupt driven mode, the processor does not block on sending/receiving characters. Data is buffered between the interrupt service routine and application code. Two buffers are used to insulate the application from the relative slowness of the serial device. One of the buffers is used for incoming characters, while the other is used for outgoing characters.

An interrupt is raised when a character is received by the serial device. The interrupt routine places the incoming character at the end of the input buffer. When an application asks for input, the characters at the front of the buffer are returned.

When the application prints to the serial device, the outgoing characters are placed at the end of the output buffer. The driver will place one or more characters in the serial device (the exact number depends on the serial device) An interrupt will be raised when all the characters have been transmitted. The interrupt service routine has to send the characters remaining in the output buffer the same way. When the transmitting side of the serial device is idle, it is typically necessary to prime the transmitter before the first interrupt will occur.

Interrupt Server Driven Mode (*TERMIOs_IRQ_SERVER_DRIVEN*)

The interrupt server driven mode is identical to the interrupt driven mode, except that a mutex is used to protect the low-level device state instead of an interrupt lock (disabled interrupts). Use this mode in case the serial device is connected via I2C or SPI and the I2C or SPI framework is used.

Task Driven Mode (*TERMIOs_TASK_DRIVEN*)

The task driven mode is similar to interrupt driven mode, but the actual data processing is done in dedicated tasks instead of interrupt routines. This mode is not available in SMP configurations. It has some implementation flaws and it is not well tested.

6.4 Polled Mode

The handler table for the polled mode should look like the following.

```

1 const rtems_termios_device_handler my_driver_handler_polled = {
2   .first_open = my_driver_first_open,
3   .last_close = my_driver_last_close,
4   .poll_read = my_driver_poll_read,
5   .write = my_driver_poll_write,
6   .set_attributes = my_driver_set_attributes,
7   .ioctl = my_driver_ioctl, /* optional, may be NULL */
8   .mode = TERMIO_POLL
9 };

```

The `my_driver_poll_write()` routine is responsible for writing `n` characters from `buf` to the serial device specified by `base`.

```

1 static void my_driver_poll_write(
2   rtems_termios_device_context *base,
3   const char *buf,
4   size_t n
5 )
6 {
7   my_driver_context *ctx;
8   size_t i;
9
10  ctx = (my_driver_context *) base;
11
12  for ( i = 0 ; i < n ; ++i ) {
13    my_driver_write_char( ctx, buf[ i ] );
14  }
15 }

```

The `my_driver_poll_read()` routine is responsible for reading a single character from the serial device specified by `base`. If no character is available, then the routine should immediately return minus one.

```

1 static int my_driver_poll_read( rtems_termios_device_context *base )
2 {
3   my_driver_context *ctx;
4
5   ctx = (my_driver_context *) base;
6
7   if ( my_driver_can_read_char( ctx ) ) {
8     /* Return the character (must be unsigned) */
9     return my_driver_read_char( ctx );
10  } else {
11    /* Return -1 to indicate that no character is available */
12    return -1;
13  }
14 }

```

6.5 Interrupt Driven Mode

The handler table for the interrupt driven mode should look like the following.

```

1 const rtems_termios_device_handler my_driver_handler_interrupt = {
2   .first_open = my_driver_first_open,
3   .last_close = my_driver_last_close,
4   .poll_read = NULL,
5   .write = my_driver_interrupt_write,
6   .set_attributes = my_driver_set_attributes,
7   .ioctl = my_driver_ioctl, /* optional, may be NULL */
8   .mode = TERMIOS_IRQ_DRIVEN
9 };

```

There is no device driver read handler to be passed to Termios. Indeed a read() call returns the contents of Termios input buffer. This buffer is filled in the driver interrupt routine.

A serial device generally generates interrupts when it is ready to accept or to emit a number of characters. In this mode, the interrupt routine is the core of the driver.

The my_driver_interrupt_handler() is responsible for processing asynchronous interrupts from the serial device. There may be multiple interrupt handlers for a single serial device. Some serial devices can generate a unique interrupt vector for each interrupt source such as a character has been received or the transmitter is ready for another character.

In the simplest case, the my_driver_interrupt_handler() will have to check the status of the serial device and determine what caused the interrupt. The following describes the operation of an my_driver_interrupt_handler() which has to do this:

```

1 static void my_driver_interrupt_handler( void *arg )
2 {
3   rtems_termios_tty *tty;
4   my_driver_context *ctx;
5   char          buf[N];
6   size_t        n;
7
8   tty = arg;
9   ctx = rtems_termios_get_device_context( tty );
10
11  /*
12   * Check if we have received something. The function reads the
13   * received characters from the device and stores them in the
14   * buffer. It returns the number of read characters.
15   */
16  n = my_driver_read_received_chars( ctx, buf, N );
17  if ( n > 0 ) {
18    /* Hand the data over to the Termios infrastructure */
19    rtems_termios_enqueue_raw_characters( tty, buf, n );
20  }
21
22  /*
23   * Check if we have something transmitted. The functions returns
24   * the number of transmitted characters since the last write to the
25   * device.
26   */
27  n = my_driver_transmitted_chars( ctx );
28  if ( n > 0 ) {

```

```

29  /*
30  * Notify Termios that we have transmitted some characters. It
31  * will call now the interrupt write function if more characters
32  * are ready for transmission.
33  */
34  rtems_termios_dequeue_characters( tty, n );
35  }
36  }

```

The `my_driver_interrupt_write()` handler is responsible for telling the device that the `n` characters at `buf` are to be transmitted. If the value `n` is zero to indicate that no more characters are to send. The driver can disable the transmit interrupts now. This routine is invoked either from task context with disabled interrupts to start a new transmission process with exactly one character in case of an idle output state or from the interrupt handler to refill the transmitter. If the routine is invoked to start the transmit process the output state will become busy and Termios starts to fill the output buffer. If the transmit interrupt arises before Termios was able to fill the transmit buffer you will end up with one interrupt per character.

```

1  static void my_driver_interrupt_write(
2  rtems_termios_device_context *base,
3  const char *buf,
4  size_t n
5  )
6  {
7  my_driver_context *ctx;
8
9  ctx = (my_driver_context *) base;
10
11  if ( n > 0 ) {
12  /*
13   * Tell the device to transmit some characters from buf (less than
14   * or equal to n). When the device is finished it should raise an
15   * interrupt. The interrupt handler will notify Termios that these
16   * characters have been transmitted and this may trigger this write
17   * function again. You may have to store the number of outstanding
18   * characters in the device data structure.
19   */
20  } else {
21  /*
22   * Termios will set n to zero to indicate that the transmitter is
23   * now inactive. The output buffer is empty in this case. The
24   * driver may disable the transmit interrupts now.
25   */
26  }
27  }

```

6.6 First Open

Upon first open of the device, the `my_driver_first_open()` handler is called by Termios. The device registered as `/dev/console` (or `CONSOLE_DEVICE_NAME`) is opened automatically during RTEMS initialization.

```

1 static bool my_driver_first_open(
2     rtems_termios_tty *tty,
3     rtems_termios_device_context *base,
4     struct termios *term,
5     rtems_libio_open_close_args_t *args
6 )
7 {
8     my_driver_context *ctx;
9     rtems_status_code sc;
10    bool ok;
11
12    ctx = (my_driver_context *) base;
13
14    /*
15     * You may add some initialization code here.
16     */
17
18    /*
19     * Sets the initial baud rate. This should be set to the value of
20     * the boot loader. This function accepts only exact Termios baud
21     * values.
22     */
23    sc = rtems_termios_set_initial_baud( tty, MY_DRIVER_BAUD_RATE );
24    if ( sc != RTEMS_SUCCESSFUL ) {
25        /* Not a valid Termios baud */
26    }
27
28    /*
29     * Alternatively you can set the best baud.
30     */
31    rtems_termios_set_best_baud( term, MY_DRIVER_BAUD_RATE );
32
33    /*
34     * To propagate the initial Termios attributes to the device use
35     * this.
36     */
37    ok = my_driver_set_attributes( base, term );
38    if ( !ok ) {
39        /* This is bad */
40    }
41
42    /*
43     * Return true to indicate a successful set attributes, and false
44     * otherwise.
45     */
46    return true;
47 }

```

6.7 Last Close

Termios will call the `my_driver_last_close()` handler if the last close happens on the device.

```
1 static void my_driver_last_close(  
2     rtems_termios_tty      *tty,  
3     rtems_termios_device_context *base,  
4     rtems_libio_open_close_args_t *args  
5 )  
6 {  
7     my_driver_context *ctx;  
8  
9     ctx = (my_driver_context *) base;  
10  
11     /*  
12      * The driver may do some cleanup here.  
13      */  
14 }
```

6.8 Set Attributes

Termios will call the `my_driver_set_attributes()` handler if a serial line configuration parameter changed, e.g. baud, character size, number of stop bits, parity, etc.

```
1 static bool my_driver_set_attributes(  
2     rtems_termios_device_context *base,  
3     const struct termios         *term  
4 )  
5 {  
6     my_driver_context *ctx;  
7  
8     ctx = (my_driver_context *) base;  
9  
10    /*  
11     * Inspect the termios data structure and configure the device  
12     * appropriately. The driver should only be concerned with the  
13     * parts of the structure that specify hardware setting for the  
14     * communications channel such as baud, character size, etc.  
15     */  
16  
17    /*  
18     * Return true to indicate a successful set attributes, and false  
19     * otherwise.  
20     */  
21    return true;  
22 }
```


6.9 IO Control

Optionally, the `my_driver_ioctl()` routine may be provided for arbitrary device-specific functions.

```
1 static int my_driver_ioctl(  
2     rtems_termios_device_context *base,  
3     ioctl_command_t              request,  
4     void                         *buffer  
5 )  
6 {  
7     my_driver_context *ctx;  
8  
9     ctx = (my_driver_context *) base;  
10  
11     switch ( request ) {  
12         case MY_DRIVER_DO_XYZ:  
13             my_driver_do_xyz(ctx, buffer);  
14             break;  
15         default:  
16             rtems_set_errno_and_return_minus_one( EINVAL );  
17     }  
18  
19     return 0;  
20 }
```

6.10 Flow Control

You can also provide handler for remote transmission control. This is not covered in this manual.

6.11 General Initialization

The BSP-specific driver initialization is called once during the RTEMS initialization process.

The `console_initialize()` function may look like this:

```

1 #include <my-driver.h>
2
3 #include <rtems/console.h>
4
5 #include <bsp.h>
6 #include <bsp/fatal.h>
7
8 static my_driver_context driver_context_table[] = {
9     { /* Some values for device 0 */ },
10    { /* Some values for device 1 */ }
11 };
12
13 rtems_device_driver console_initialize(
14     rtems_device_major_number major,
15     rtems_device_minor_number minor,
16     void *arg
17 )
18 {
19     const rtems_termios_device_handler *handler;
20     rtems_status_code sc;
21     size_t i;
22
23     #ifdef SOME_BSP_USE_INTERRUPTS
24         handler = &my_driver_handler_interrupt;
25     #else
26         handler = &my_driver_handler_polled;
27     #endif
28
29     /*
30      * Initialize the Termios infrastructure. If Termios has already
31      * been initialized by another device driver, then this call will
32      * have no effect.
33      */
34     rtems_termios_initialize();
35
36     /* Initialize each device */
37     for ( i = 0; i < RTEMS_ARRAY_SIZE( driver_context_table ) ; ++i ) {
38         my_driver_context *ctx;
39
40         ctx = &driver_context_table[ i ];
41
42         /*
43          * Install this device in the file system and Termios. In order
44          * to use the console (i.e. being able to do printf, scanf etc.
45          * on stdin, stdout and stderr), one device must be registered as
46          * "/dev/console" (CONSOLE_DEVICE_NAME).
47          */
48         sc = rtems_termios_device_install( ctx->device_name, handler, NULL, ctx );
49         if ( sc != RTEMS_SUCCESSFUL ) {
50             bsp_fatal( SOME_BSP_FATAL_CONSOLE_DEVICE_INSTALL );
51         }
52     }
53 }
```

```
52 }  
53  
54 return RTEMS_SUCCESSFUL;  
55 }
```

CLOCK DRIVER

7.1 Introduction

The purpose of the clock driver is to provide two services for the operating system.

- A steady time basis to the kernel, so that the RTEMS primitives that need a clock tick work properly. See the *Clock Manager* chapter of the *RTEMS Application C User's Guide* for more details.
- An optional **timecounter** to provide timestamps of the uptime and wall clock time with higher resolution than the clock tick.

The clock driver is usually located in the clock directory of the BSP. Clock drivers must use the *Clock Driver Shell* available via the **clockimpl.h** include file. This include file is not a normal header file and instead defines the clock driver functions declared in `#include <rtems/clockdrv.h>` which are used by RTEMS configuration file `#include <rtems/confdefs.h>`. In case the application configuration defines `#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER`, then the clock driver is registered and should provide its services to the operating system. The clock tick interval is determined by the application configuration via `#define CONFIGURE_MICROSECONDS_PER_TICK` and can be obtained via `rtems_configuration_get_microseconds_per_tick()`.

A hardware-specific clock driver must provide some functions, defines and macros for the *Clock Driver Shell* which are explained here step by step. A clock driver file looks in general like this.

```

1 /*
2  * A section with functions, defines and macros to provide hardware-specific
3  * functions for the Clock Driver Shell.
4  */
5
6 #include "../../shared/dev/clock/clockimpl.h"
```

Depending on the hardware capabilities one out of three clock driver variants must be selected.

Timecounter

The variant which provides all features needs a free running hardware counter and a periodic clock tick interrupt. This variant is mandatory in SMP configurations.

Simple Timecounter

A simple timecounter can be used if the hardware provides no free running hardware counter and only a periodic hardware counter synchronous to the clock tick interrupt is available.

Clock Tick Only

The most basic clock driver provides only a periodic clock tick interrupt. The timestamp resolution is limited to the clock tick interval.

7.2 Initialization

7.2.1 Timecounter Variant

This variant is preferred since it is the most efficient and yields the most accurate timestamps. It is also mandatory in SMP configurations to obtain valid timestamps. The hardware must provide a periodic interrupt to service the clock tick and a free running counter for the timecounter. The free running counter must have a power of two period. The `tc_counter_mask` must be initialized to the free running counter period minus one, e.g. for a 17-bit counter this is `0x0001ffff`. The `tc_get_timecount` function must return the current counter value (the counter values must increase, so if the counter counts down, a conversion is necessary). Use `RTEMS_TIMECOUNTER_QUALITY_CLOCK_DRIVER` for the `tc_quality`. Set `tc_frequency` to the frequency of the free running counter in Hz. All other fields of the struct `timecounter` must be zero initialized. Install the initialized timecounter via `rtems_timecounter_install()`.

For an example see the [QorIQ clock driver](#).

```

1 #include <rtems/timecounter.h>
2
3 static struct timecounter some_tc;
4
5 static uint32_t some_tc_get_timecount( struct timecounter *tc )
6 {
7     some.free_running_counter;
8 }
9
10 static void some_support_initialize_hardware( void )
11 {
12     uint64_t us_per_tick;
13     uint32_t counter_frequency_in_hz;
14     uint32_t counter_ticks_per_clock_tick;
15
16     us_per_tick = rtems_configuration_get_microseconds_per_tick();
17     counter_frequency_in_hz = some_tc_get_frequency();
18
19     /*
20      * The multiplication must be done in 64-bit arithmetic to avoid an integer
21      * overflow on targets with a high enough counter frequency.
22      */
23     counter_ticks_per_clock_tick =
24         (uint32_t) ( counter_frequency_in_hz * us_per_tick ) / 1000000;
25
26     /*
27      * Initialize hardware and set up a periodic interrupt for the configuration
28      * based counter ticks per clock tick.
29      */
30
31     some_tc.tc_get_timecount = some_tc_get_timecount;
32     some_tc.tc_counter_mask = 0xffffffff;
33     some_tc.tc_frequency = frequency;
34     some_tc.tc_quality = RTEMS_TIMECOUNTER_QUALITY_CLOCK_DRIVER;
35     rtems_timecounter_install( &some_tc );
36 }
37
38 #define Clock_driver_support_initialize_hardware() \
39     some_support_initialize_hardware()

```

```

40
41 #include "../../shared/dev/clock/clockimpl.h"

```

7.2.2 Simple Timecounter Variant

For an example see the [ERC32 clock driver](#).

```

1 #include <rtems/timecounter.h>
2
3 static rtems_timecounter_simple some_tc;
4
5 static uint32_t some_tc_get( rtems_timecounter_simple *tc )
6 {
7     return some.counter;
8 }
9
10 static bool some_tc_is_pending( rtems_timecounter_simple *tc )
11 {
12     return some.is_pending;
13 }
14
15 static uint32_t some_tc_get_timecount( struct timecounter *tc )
16 {
17     return rtems_timecounter_simple_downcounter_get(
18         tc,
19         some_tc_get,
20         some_tc_is_pending
21     );
22 }
23
24 static void some_tc_tick( void )
25 {
26     rtems_timecounter_simple_downcounter_tick( &some_tc, some_tc_get );
27 }
28
29 static void some_support_initialize_hardware( void )
30 {
31     uint64_t us_per_tick;
32     uint32_t counter_frequency_in_hz;
33     uint32_t counter_ticks_per_clock_tick;
34
35     us_per_tick = rtems_configuration_get_microseconds_per_tick();
36     counter_frequency_in_hz = some_tc_get_frequency();
37     counter_ticks_per_clock_tick =
38         (uint32_t) ( counter_frequency_in_hz * us_per_tick ) / 1000000;
39
40     /* Initialize hardware */
41
42     rtems_timecounter_simple_install(
43         &some_tc,
44         counter_frequency_in_hz,
45         counter_ticks_per_clock_tick,
46         some_tc_get_timecount
47     );
48 }
49

```



```
50 #define Clock_driver_support_initialize_hardware() \  
51     some_support_initialize_hardware()  
52 #define Clock_driver_timecounter_tick() \  
53     some_tc_tick()  
54  
55 #include "../../shared/dev/clock/clockimpl.h"
```

7.2.3 Clock Tick Only Variant

For an example see the [Motorola 68360 clock driver](#).

```
1  static void some_support_initialize_hardware( void )  
2  {  
3      /* Initialize hardware */  
4  }  
5  
6  #define Clock_driver_support_initialize_hardware() \  
7      some_support_initialize_hardware()  
8  
9  /* Indicate that this clock driver lacks a proper timecounter in hardware */  
10  
11 #define CLOCK_DRIVER_USE_DUMMY_TIMECOUNTER  
12  
13 #include "../../shared/dev/clock/clockimpl.h"
```

7.3 Install Clock Tick Interrupt Service Routine

The clock driver may provide a function to install the clock tick interrupt service routine via `Clock_driver_support_install_isr()`. The clock tick interrupt service routine is passed as the one and only parameter to this macro. The default implementation will do nothing.

```
1 #include <bsp/irq.h>
2 #include <bsp/fatal.h>
3
4 static void some_support_install_isr( rtems_interrupt_handler isr )
5 {
6     rtems_status_code sc;
7     sc = rtems_interrupt_handler_install(
8         SOME_IRQ,
9         "Clock",
10        RTEMS_INTERRUPT_UNIQUE,
11        isr,
12        NULL
13    );
14    if ( sc != RTEMS_SUCCESSFUL ) {
15        bsp_fatal( SOME_FATAL_IRQ_INSTALL );
16    }
17 }
18
19 #define Clock_driver_support_install_isr( isr ) \
20     some_support_install_isr( isr )
21
22 #include "../shared/dev/clock/clockimpl.h"
```

7.4 Support At Tick

The hardware-specific support at tick is specified by `Clock_driver_support_at_tick()`.

```
1 static void some_support_at_tick( void )
2 {
3     /* Clear interrupt */
4 }
5
6 #define Clock_driver_support_at_tick() \
7     some_support_at_tick()
8
9 #include "../shared/dev/clock/clockimpl.h"
```

7.5 System Shutdown Support

The clock driver system shutdown support was removed in RTEMS 5.1.

7.6 SMP Support

In SMP configurations, the clock tick service must be executed for each processor used by RTEMS. By default, the clock tick interrupt must be distributed to all processors used by RTEMS and each processor invokes the clock tick service individually. A clock driver may delegate all the work to the boot processor. It must define `CLOCK_DRIVER_USE_ONLY_BOOT_PROCESSOR` in this case.

Clock drivers must define `Clock_driver_support_set_interrupt_affinity(online_processors)` to set the interrupt affinity of the clock tick interrupt.

7.7 Multiple Clock Driver Ticks Per Clock Tick

In case the hardware needs more than one clock driver tick per clock tick (e.g. due to a limited range of the hardware timer), then this can be specified with the optional `#define CLOCK_DRIVER_ISRS_PER_TICK` and `#define CLOCK_DRIVER_ISRS_PER_TICK_VALUE` defines. This is currently used only for x86 and it hopefully remains that way.

```
1 /* Enable multiple clock driver ticks per clock tick */
2 #define CLOCK_DRIVER_ISRS_PER_TICK 1
3
4 /* Specifiy the clock driver ticks per clock tick value */
5 #define CLOCK_DRIVER_ISRS_PER_TICK_VALUE 123
6
7 #include "../../shared/dev/clock/clockimpl.h"
```

7.8 Clock Driver Ticks Counter

The *Clock Driver Shell* provide a global variable that is simply a count of the number of clock driver interrupt service routines that have occurred. This information is valuable when debugging a system. This variable is declared as follows:

```
1 volatile uint32_t Clock_driver_ticks;
```


ENTROPY SOURCE

Each BSP must provide an implementation of the `getentropy()` system call. This system call was introduced by [OpenBSD](#) and is also available in [glibc since version 2.25](#). This system call is used by the Newlib provided [ARC4RANDOM\(3\)](#) functions, which in turn are used by various cryptographic functions.

Warning: A good entropy source is critical for (nearly) all cryptographic applications. The default implementation based on the CPU counter is not suitable for such applications.

The `getentropy()` implementation must fill the specified memory region of the given size with random numbers and return 0 on success. A non-zero return may cause the `INTERNAL_ERROR_ARC4RANDOM_GETENTROPY_FAIL` internal error by one of the [ARC4RANDOM\(3\)](#) functions.

In general, for embedded systems it is not easy to get some real entropy. Normally, that can only be reached with some extra hardware support. Some microcontrollers integrate a true random number generator or something similar for cryptographic applications. That is the preferred source of entropy for most BSPs. For example the [atsam BSP uses the TRNG for its entropy source](#).

There is also a quite limited [default implementation based on the CPU counter](#). Due to the fact that it is a time based source, the values provided by `getentropy()` are quite predictable. This implementation is not appropriate for any cryptographic applications but it is good enough for some basic tasks. Use it only if you do not have any strong requirements on the entropy and if there is no better source.

I2C DRIVER

The Inter-Integrated Circuit (I2C, I²C, IIC) bus drivers should use the [I2C bus framework](#). The user API is compatible to the [Linux I2C user-space API](#).

For example I2C bus drivers see:

- [Atmel SAM I2C driver](#)
- [Cadence I2C driver](#)
- [I2C framework test](#)
- [NXP i.MX I2C driver](#)
- [NXP LPC17XX/LPC24XX/LPC40XX I2C driver](#)

For example I2C device drivers see:

- ADC
 - [TI ADS 16-Bit](#)
- [EEPROM](#)
- GPIO
 - [NXP PCA9535](#)
- Power Management
 - [NXP PCA9548A](#)
 - [TI LM25066A](#)
- Sensors
 - [NXP LM75A](#)
 - [TI TMP112](#)

SPI DRIVER

The Serial Peripheral Interface (SPI) bus drivers should use the [SPI bus framework](#). The user API is compatible to the [Linux SPI user-space API](#).

For example SPI bus drivers see:

- [Atmel SAM SPI driver](#)
- [NXP i.MX SPI driver](#)
- [NXP LPC17XX/LPC24XX/LPC40XX SSP driver](#)
- [SPI framework test](#)

REAL-TIME CLOCK DRIVER

11.1 Introduction

The Real-Time Clock (*RTC*) driver is responsible for providing an interface to an *RTC* device. The capabilities provided by this driver are:

- Set the RTC TOD to RTEMS TOD
- Set the RTEMS TOD to the RTC TOD
- Get the RTC TOD
- Set the RTC TOD to the Specified TOD
- Get the Difference Between the RTEMS and RTC TOD

Note: In this chapter, the abbreviation *TOD* is used for *Time of Day*.

The reference implementation for a real-time clock driver can be found in [bsps/shared/dev/rtc/rtc-support.c](#). This driver is based on the libchip concept and can be easily configured to work with any of the RTC chips supported by the RTC chip drivers in the directory [bsps/shared/dev/rtc](#). There is a README file in this directory for each supported RTC chip. Each of these README explains how to configure the shared libchip implementation of the RTC driver for that particular RTC chip.

The DY-4 DMV177 BSP used the shared libchip implementation of the RTC driver. There were no DMV177 specific configuration routines. A BSP could use configuration routines to dynamically determine what type of real-time clock is on a particular board. This would be useful for a BSP supporting multiple board models. The relevant parts of the DMV177's RTC_Table configuration table is below:

```

1 #include <bsp.h>
2 #include <libchip/rtc.h>
3 #include <libchip/icm7170.h>
4
5 bool dmV177_icm7170_probe(int minor);
6
7 rtc_tbl RTC_Table[] = {
8     { "/dev/rtc0",          /* sDeviceName */
9       RTC_ICM7170,         /* deviceType */
10      &icm7170_fns,         /* pDeviceFns */
11      dmV177_icm7170_probe, /* deviceProbe */
12      (void *) ICM7170_AT_1_MHZ, /* pDeviceParams */
13      DMV170_RTC_ADDRESS,    /* ulCtrlPort1 */
14      0,                     /* ulDataPort */
15      icm7170_get_register_8, /* getRegister */
16      icm7170_set_register_8, /* setRegister */
17    }
18 };
19 unsigned long RTC_Count = (sizeof(RTC_Table)/sizeof(rtc_tbl));
20 rtems_device_minor_number RTC_Minor;
21
22 bool dmV177_icm7170_probe(int minor)
23 {
24     volatile uint16_t *card_resource_reg;
25     card_resource_reg = (volatile uint16_t *) DMV170_CARD_RESOURCE_REG;
26     if ( (*card_resource_reg & DMV170_RTC_INST_MASK) == DMV170_RTC_INSTALLED )

```



```
27     return TRUE;  
28     return FALSE;  
29 }
```

11.2 Initialization

The `rtc_initialize` routine is responsible for initializing the RTC chip so it can be used. The shared libchip implementation of this driver supports multiple RTCs and bases its initialization order on the order the chips are defined in the `RTC_Table`. Each chip defined in the table may or may not be present on this particular board. It is the responsibility of the `deviceProbe` to indicate the presence of a particular RTC chip. The first RTC found to be present is considered the preferred RTC.

In the shared libchip based implementation of the driver, the following actions are performed:

```

1 rtems_device_driver rtc_initialize(
2   rtems_device_major_number  major,
3   rtems_device_minor_number  minor_arg,
4   void                       *arg
5 )
6 {
7   for each RTC configured in RTC_Table
8     if the deviceProbe for this RTC indicates it is present
9       set RTC_Minor to this device
10      set RTC_Present to TRUE
11      break out of this loop
12
13   if RTC_Present is not TRUE
14     return RTEMS_INVALID_NUMBER to indicate that no RTC is present
15
16   register this minor number as the "/dev/rtc"
17
18   perform the deviceInitialize routine for the preferred RTC chip
19
20   for RTCs past this one in the RTC_Table
21     if the deviceProbe for this RTC indicates it is present
22       perform the deviceInitialize routine for this RTC chip
23       register the configured name for this RTC
24 }
```

The `deviceProbe` routine returns TRUE if the device configured by this entry in the `RTC_Table` is present. This configuration scheme allows one to support multiple versions of the same board with a single BSP. For example, if the first generation of a board had Vendor A's RTC chip and the second generation had Vendor B's RTC chip, `RTC_Table` could contain information for both. The `deviceProbe` configured for Vendor A's RTC chip would need to return TRUE if the board was a first generation one. The `deviceProbe` routines are very board dependent and must be provided by the BSP.

11.3 setRealTimeToRTEMS

The setRealTimeToRTEMS routine sets the current RTEMS TOD to that of the preferred RTC.

```
1 void setRealTimeToRTEMS(void)
2 {
3     if no RTCs are present
4         return
5
6     invoke the deviceGetTime routine for the preferred RTC
7     set the RTEMS TOD using rtems_clock_set
8 }
```

11.4 setRealTimeFromRTEMS

The setRealTimeFromRTEMS routine sets the preferred RTC TOD to the current RTEMS TOD.

```
1 void setRealTimeFromRTEMS(void)
2 {
3     if no RTCs are present
4         return
5
6     obtain the RTEMS TOD using rtems_clock_get
7     invoke the deviceSetTime routine for the preferred RTC
8 }
```

11.5 getRealTime

The getRealTime returns the preferred RTC TOD to the caller.

```
1 void getRealTime( rtems_time_of_day *tod )
2 {
3     if no RTCs are present
4     return
5
6     invoke the deviceGetTime routine for the preferred RTC
7 }
```

11.6 setRealTime

The setRealTime routine sets the preferred RTC TOD to the TOD specified by the caller.

```
1 void setRealTime( rtems_time_of_day *tod )
2 {
3     if no RTCs are present
4         return
5
6     invoke the deviceSetTime routine for the preferred RTC
7 }
```

11.7 checkRealTime

The checkRealTime routine returns the number of seconds difference between the RTC TOD and the current RTEMS TOD.

```
1 int checkRealTime( void )
2 {
3     if no RTCs are present
4         return -1
5
6     obtain the RTEMS TOD using rtems_clock_get
7     get the TOD from the preferred RTC using the deviceGetTime routine
8     convert the RTEMS TOD to seconds
9     convert the RTC TOD to seconds
10
11     return the RTEMS TOD in seconds - RTC TOD in seconds
12 }
```


NETWORKING DRIVER

12.1 Introduction

This chapter is intended to provide an introduction to the procedure for writing RTEMS network device drivers. The example code is taken from the ‘Generic 68360’ network device driver. The source code for this driver is located in the `bsps/m68k/gen68360/net` directory in the RTEMS source code distribution. Having a copy of this driver at hand when reading the following notes will help significantly.

Legacy Networking Stack

This docuemntation is for the legacy FreeBSD networking stack in the RTEMS source tree.

12.2 Learn about the network device

Before starting to write the network driver become completely familiar with the programmer's view of the device. The following points list some of the details of the device that must be understood before a driver can be written.

- Does the device use DMA to transfer packets to and from memory or does the processor have to copy packets to and from memory on the device?
- If the device uses DMA, is it capable of forming a single outgoing packet from multiple fragments scattered in separate memory buffers?
- If the device uses DMA, is it capable of chaining multiple outgoing packets, or does each outgoing packet require intervention by the driver?
- Does the device automatically pad short frames to the minimum 64 bytes or does the driver have to supply the padding?
- Does the device automatically retry a transmission on detection of a collision?
- If the device uses DMA, is it capable of buffering multiple packets to memory, or does the receiver have to be restarted after the arrival of each packet?
- How are packets that are too short, too long, or received with CRC errors handled? Does the device automatically continue reception or does the driver have to intervene?
- How is the device Ethernet address set? How is the device programmed to accept or reject broadcast and multicast packets?
- What interrupts does the device generate? Does it generate an interrupt for each incoming packet, or only for packets received without error? Does it generate an interrupt for each packet transmitted, or only when the transmit queue is empty? What happens when a transmit error is detected?

In addition, some controllers have specific questions regarding board specific configuration. For example, the SONIC Ethernet controller has a very configurable data bus interface. It can even be configured for sixteen and thirty-two bit data buses. This type of information should be obtained from the board vendor.

12.3 Understand the network scheduling conventions

When writing code for the driver transmit and receive tasks, take care to follow the network scheduling conventions. All tasks which are associated with networking share various data structures and resources. To ensure the consistency of these structures the tasks execute only when they hold the network semaphore (`rtems_bsdnet_semaphore`). The transmit and receive tasks must abide by this protocol. Be very careful to avoid ‘deadly embraces’ with the other network tasks. A number of routines are provided to make it easier for the network driver code to conform to the network task scheduling conventions.

- `void rtems_bsdnet_semaphore_release(void)` This function releases the network semaphore. The network driver tasks must call this function immediately before making any blocking RTEMS request.
- `void rtems_bsdnet_semaphore_obtain(void)` This function obtains the network semaphore. If a network driver task has released the network semaphore to allow other network-related tasks to run while the task blocks, then this function must be called to reobtain the semaphore immediately after the return from the blocking RTEMS request.
- `rtems_bsdnet_event_receive(rtems_event_set, rtems_option, rtems_interval, rtems_event_set *)` The network driver task should call this function when it wishes to wait for an event. This function releases the network semaphore, calls `rtems_event_receive` to wait for the specified event or events and reobtains the semaphore. The value returned is the value returned by the `rtems_event_receive`.

12.4 Network Driver Makefile

Network drivers are considered part of the BSD network package and as such are to be compiled with the appropriate flags. This can be accomplished by adding `-D__INSIDE_RTEMS_BSD_TCPIP_STACK__` to the command line. If the driver is inside the RTEMS source tree or is built using the RTEMS application Makefiles, then adding the following line accomplishes this:

```
1 DEFINES += -D__INSIDE_RTEMS_BSD_TCPIP_STACK__
```

This is equivalent to the following list of definitions. Early versions of the RTEMS BSD network stack required that all of these be defined.

```
1 -D_COMPILING_BSD_KERNEL_ -DKERNEL -DINET -DNFS -DDIAGNOSTIC -DBOOTP_COMPAT
```

Defining these macros tells the network header files that the driver is to be compiled with extended visibility into the network stack. This is in sharp contrast to applications that simply use the network stack. Applications do not require this level of visibility and should stick to the portable application level API.

As a direct result of being logically internal to the network stack, network drivers use the BSD memory allocation routines. This means, for example, that `malloc` takes three arguments. See the SONIC device driver (`c/src/lib/libchip/network/sonic.c`) for an example of this. Because of this, network drivers should not include `<stdlib.h>`. Doing so will result in conflicting definitions of `malloc()`.

Application level code including network servers such as the FTP daemon are *not* part of the BSD kernel network code and should not be compiled with the BSD network flags. They should include `<stdlib.h>` and not define the network stack visibility macros.

12.5 Write the Driver Attach Function

The driver attach function is responsible for configuring the driver and making the connection between the network stack and the driver.

Driver attach functions take a pointer to an `rtems_bsdnet_ifconfig` structure as their only argument, and set the driver parameters based on the values in this structure. If an entry in the configuration structure is zero the attach function chooses an appropriate default value for that parameter.

The driver should then set up several fields in the `ifnet` structure in the device-dependent data structure supplied and maintained by the driver:

`ifp->if_softc`

Pointer to the device-dependent data. The first entry in the device-dependent data structure must be an `arpcom` structure.

`ifp->if_name`

The name of the device. The network stack uses this string and the device number for device name lookups. The device name should be obtained from the name entry in the configuration structure.

`ifp->if_unit`

The device number. The network stack uses this number and the device name for device name lookups. For example, if `ifp->if_name` is `scc` and `ifp->if_unit` is 1, the full device name would be `scc1`. The unit number should be obtained from the name entry in the configuration structure.

`ifp->if_mtu`

The maximum transmission unit for the device. For Ethernet devices this value should almost always be 1500.

`ifp->if_flags`

The device flags. Ethernet devices should set the flags to `IFF_BROADCAST|IFF_SIMPLEX`, indicating that the device can broadcast packets to multiple destinations and does not receive and transmit at the same time.

`ifp->if_snd.ifq_maxlen`

The maximum length of the queue of packets waiting to be sent to the driver. This is normally set to `ifqmaxlen`.

`ifp->if_init`

The address of the driver initialization function.

`ifp->if_start`

The address of the driver start function.

`ifp->if_ioctl`

The address of the driver `ioctl` function.

`ifp->if_output`

The address of the output function. Ethernet devices should set this to `ether_output`.

RTEMS provides a function to parse the driver name in the configuration structure into a device name and unit number.

```
1 int rtems_bsdnet_parse_driver_name (
2   const struct rtems_bsdnet_ifconfig *config,
```

```
3 char **namep  
4 );
```

The function takes two arguments; a pointer to the configuration structure and a pointer to a pointer to a character. The function parses the configuration name entry, allocates memory for the driver name, places the driver name in this memory, sets the second argument to point to the name and returns the unit number. On error, a message is printed and -1 is returned.

Once the attach function has set up the above entries it must link the driver data structure onto the list of devices by calling `if_attach`. Ethernet devices should then call `ether_ifattach`. Both functions take a pointer to the device's `ifnet` structure as their only argument.

The attach function should return a non-zero value to indicate that the driver has been successfully configured and attached.

12.6 Write the Driver Start Function.

This function is called each time the network stack wants to start the transmitter. This occurs whenever the network stack adds a packet to a device's send queue and the `IFF_OACTIVE` bit in the device's `if_flags` is not set.

For many devices this function need only set the `IFF_OACTIVE` bit in the `if_flags` and send an event to the transmit task indicating that a packet is in the driver transmit queue.

12.7 Write the Driver Initialization Function.

This function should initialize the device, attach to interrupt handler, and start the driver transmit and receive tasks. The function:

```
1 rtems_id rtems_bsdnet_newproc(  
2     char *name,  
3     int  stacksize,  
4     void (*entry)(void *),  
5     void *arg  
6 );
```

should be used to start the driver tasks.

Note that the network stack may call the driver initialization function more than once. Make sure multiple versions of the receive and transmit tasks are not accidentally started.

12.8 Write the Driver Transmit Task

This task is responsible for removing packets from the driver send queue and sending them to the device. The task should block waiting for an event from the driver start function indicating that packets are waiting to be transmitted. When the transmit task has drained the driver send queue the task should clear the `IFF_OACTIVE` bit in `if_flags` and block until another outgoing packet is queued.

12.9 Write the Driver Receive Task

This task should block until a packet arrives from the device. If the device is an Ethernet interface the function `ether_input` should be called to forward the packet to the network stack. The arguments to `ether_input` are a pointer to the interface data structure, a pointer to the ethernet header and a pointer to an mbuf containing the packet itself.

12.10 Write the Driver Interrupt Handler

A typical interrupt handler will do nothing more than the hardware manipulation required to acknowledge the interrupt and send an RTEMS event to wake up the driver receive or transmit task waiting for the event. Network interface interrupt handlers must not make any calls to other network routines.

12.11 Write the Driver IOCTL Function

This function handles ioctl requests directed at the device. The ioctl commands which must be handled are:

SIOCGIFADDR, SIOCSIFADDR

If the device is an Ethernet interface these commands should be passed on to ether_ioctl.

SIOCSIFFLAGS

This command should be used to start or stop the device, depending on the state of the interface IFF_UP and “IFF_RUNNING” bits in if_flags:

IFF_RUNNING

Stop the device.

IFF_UP

Start the device.

IFF_UP | IFF_RUNNING

Stop then start the device.

0

Do nothing.

12.12 Write the Driver Statistic-Printing Function

This function should print the values of any statistic/diagnostic counters the network driver may use. The driver `ioctl` function should call the statistic-printing function when the `ioctl` command is `SIO_RTEMS_SHOW_STATS`.

FRAME BUFFER DRIVER

In this chapter, we present the basic functionality implemented by a frame buffer driver:

- `frame_buffer_initialize()`
- `frame_buffer_open()`
- `frame_buffer_close()`
- `frame_buffer_read()`
- `frame_buffer_write()`
- `frame_buffer_control()`

13.1 Introduction

The purpose of the frame buffer driver is to provide an abstraction for graphics hardware. By using the frame buffer interface, an application can display graphics without knowing anything about the low-level details of interfacing to a particular graphics adapter. The parameters governing the mapping of memory to displayed pixels (planar or linear, bit depth, etc) is still implementation-specific, but device-independent methods are provided to determine and potentially modify these parameters.

The frame buffer driver is commonly located in the console directory of the BSP and registered by the name `/dev/fb0`. Additional frame buffers (if available) are named `/dev/fb1*`, `*/dev/fb2`, etc.

To work with the frame buffer, the following operation sequence is used: `open()`, `ioctl()` to get the frame buffer info, `read()` and/or `write()`, and `close()`.

13.2 Driver Function Overview

13.2.1 Initialization

The driver initialization is called once during the RTEMS initialization process and returns RTEMS_SUCCESSFUL when the device driver is successfully initialized. During the initialization, a name is assigned to the frame buffer device. If the graphics hardware supports console text output, as is the case with the pc386 VGA hardware, initialization into graphics mode may be deferred until the device is open() ed.

The frame_buffer_initialize() function may look like this:

```

1 rtems_device_driver frame_buffer_initialize(
2   rtems_device_major_number  major,
3   rtems_device_minor_number  minor,
4   void                       *arg)
5 {
6   rtems_status_code status;
7
8   printk( "frame buffer driver initializing..\n" );
9
10  /*
11   * Register the device
12   */
13  status = rtems_io_register_name("/dev/fb0", major, 0);
14  if (status != RTEMS_SUCCESSFUL)
15  {
16    printk("Error registering frame buffer device!\n");
17    rtems_fatal_error_occurred( status );
18  }
19
20  /*
21   * graphics hardware initialization goes here for non-console
22   * devices
23   */
24
25  return RTEMS_SUCCESSFUL;
26 }
```

13.2.2 Opening the Frame Buffer Device

The frame_buffer_open() function is called whenever a frame buffer device is opened. If the frame buffer is registered as /dev/fb0, the frame_buffer_open entry point will be called as the result of an open("/dev/fb0", mode) in the application.

Thread safety of the frame buffer driver is implementation-dependent. The VGA driver shown below uses a mutex to prevent multiple open() operations of the frame buffer device.

The frame_buffer_open() function returns RTEMS_SUCCESSFUL when the device driver is successfully opened, and RTEMS_UNSATISFIED if the device is already open:

```

1 rtems_device_driver frame_buffer_close(
2   rtems_device_major_number  major,
3   rtems_device_minor_number  minor,
4   void                       *arg)
```

```

5 )
6 {
7   if (pthread_mutex_unlock(&mutex) == 0) {
8     /* restore previous state.  for VGA this means return to text mode.
9      * leave out if graphics hardware has been initialized in
10     * frame_buffer_initialize()
11     */
12     ega_hwterm();
13     printk( "FBVGA close called.\n" );
14     return RTEMS_SUCCESSFUL;
15   }
16   return RTEMS_UNSATISFIED;
17 }

```

In the previous example, the function `ega_hwinit()` takes care of hardware-specific initialization.

13.2.3 Closing the Frame Buffer Device

The `frame_buffer_close()` is invoked when the frame buffer device is closed. It frees up any resources allocated in `frame_buffer_open()`, and should restore previous hardware state. The entry point corresponds to the device driver close entry point.

Returns `RTEMS_SUCCESSFUL` when the device driver is successfully closed:

```

1 rtems_device_driver frame_buffer_close(
2   rtems_device_major_number major,
3   rtems_device_minor_number minor,
4   void *arg)
5 {
6   pthread_mutex_unlock(&mutex);
7
8   /* TODO check mutex return value, RTEMS_UNSATISFIED if it failed.  we
9    * don't want to unconditionally call ega_hwterm()... */
10  /* restore previous state.  for VGA this means return to text mode.
11   * leave out if graphics hardware has been initialized in
12   * frame_buffer_initialize() */
13  ega_hwterm();
14  printk( "frame buffer close called.\n" );
15  return RTEMS_SUCCESSFUL;
16 }

```

13.2.4 Reading from the Frame Buffer Device

The `frame_buffer_read()` is invoked from a `read()` operation on the frame buffer device. Read functions should allow normal and partial reading at the end of frame buffer memory. This method returns `RTEMS_SUCCESSFUL` when the device is successfully read from:

```

1 rtems_device_driver frame_buffer_read(
2   rtems_device_major_number major,
3   rtems_device_minor_number minor,
4   void *arg)
5 )
6 {

```

```

7  rtems_libio_rw_args_t *rw_args = (rtems_libio_rw_args_t *)arg;
8  rw_args->bytes_moved = ((rw_args->offset + rw_args->count) > fb_fix.smem_len) ?
9                          (fb_fix.smem_len - rw_args->offset) : rw_args->count;
10 memcpy(rw_args->buffer,
11         (const void *) (fb_fix.smem_start + rw_args->offset),
12         rw_args->bytes_moved);
13 return RTEMS_SUCCESSFUL;
14 }

```

13.2.5 Writing to the Frame Buffer Device

The `frame_buffer_write()` is invoked from a `write()` operation on the frame buffer device. The frame buffer write function is similar to the read function, and should handle similar cases involving partial writes.

This method returns `RTEMS_SUCCESSFUL` when the device is successfully written to:

```

1  rtems_device_driver frame_buffer_write(
2      rtems_device_major_number major,
3      rtems_device_minor_number minor,
4      void *arg
5  )
6  {
7      rtems_libio_rw_args_t *rw_args = (rtems_libio_rw_args_t *)arg;
8      rw_args->bytes_moved = ((rw_args->offset + rw_args->count) > fb_fix.smem_len) ?
9                          (fb_fix.smem_len - rw_args->offset) : rw_args->count;
10     memcpy((void *) (fb_fix.smem_start + rw_args->offset),
11            rw_args->buffer,
12            rw_args->bytes_moved);
13     return RTEMS_SUCCESSFUL;
14 }

```

13.2.6 Frame Buffer IO Control

The frame buffer driver allows several `ioctl`s, partially compatible with the Linux kernel, to obtain information about the hardware.

All `ioctl()` operations on the frame buffer device invoke `frame_buffer_control()`.

`Ioctls supported:`

- `ioctl`s to get the frame buffer screen info (fixed and variable).
- `ioctl` to set and get palette.

```

1  rtems_device_driver frame_buffer_control(
2      rtems_device_major_number major,
3      rtems_device_minor_number minor,
4      void *arg
5  )
6  {
7      rtems_libio_ioctl_args_t *args = arg;
8
9      printk( "FBVGA ioctl called, cmd=%x\n", args->command );
10 }

```

```

11  switch( args->command ) {
12      case FBIOGET_FSCREENINFO:
13          args->iocctl_return = get_fix_screen_info( ( struct fb_fix_screeninfo * ) args->
↪buffer );
14          break;
15      case FBIOGET_VSCREENINFO:
16          args->iocctl_return = get_var_screen_info( ( struct fb_var_screeninfo * ) args->
↪buffer );
17          break;
18      case FBIOPUT_VSCREENINFO:
19          /* not implemented yet*/
20          args->iocctl_return = -1;
21          return RTEMS_UNSATISFIED;
22      case FBIOGETCMAP:
23          args->iocctl_return = get_palette( ( struct fb_cmap * ) args->buffer );
24          break;
25      case FBIOPUTCMAP:
26          args->iocctl_return = set_palette( ( struct fb_cmap * ) args->buffer );
27          break;
28      default:
29          args->iocctl_return = 0;
30          break;
31  }
32
33  return RTEMS_SUCCESSFUL;
34  }

```

See `rtems/fb.h` for more information on the list of `iocctls` and data structures they work with.

ADA95 INTERRUPT SUPPORT

14.1 Introduction

This chapter describes what is required to enable Ada interrupt and error exception handling when using GNAT over RTEMS.

The GNAT Ada95 interrupt support RTEMS was developed by Jiri Gaisler <jgais@ws.estec.esa.nl> who also wrote this chapter.

14.2 Mapping Interrupts to POSIX Signals

In Ada95, interrupts can be attached with the `interrupt_attach` pragma. For most systems, the gnat run-time will use POSIX signal to implement the interrupt handling, mapping one signal per interrupt. For interrupts to be propagated to the attached Ada handler, the corresponding signal must be raised when the interrupt occurs.

The same mechanism is used to generate Ada error exceptions. Three error exceptions are defined: program, constraint and storage error. These are generated by raising the predefined signals: SIGILL, SIGFPE and SIGSEGV. These signals should be raised when a spurious or erroneous trap occurs.

To enable gnat interrupt and error exception support for a particular BSP, the following has to be done:

- Write an interrupt/trap handler that will raise the corresponding signal depending on the interrupt/trap number.
- Install the interrupt handler for all interrupts/traps that will be handled by gnat (including spurious).
- At startup, gnat calls `__gnat_install_handler()`. The BSP must provide this function which installs the interrupt/trap handlers.

Which CPU-interrupt will generate which signal is implementation defined. There are 32 POSIX signals (1 - 32), and all except the three error signals (SIGILL, SIGFPE and SIGSEGV) can be used. I would suggest to use the upper 16 (17 - 32) which do not have an assigned POSIX name.

Note that the pragma `interrupt_attach` will only bind a signal to a particular Ada handler - it will not unmask the interrupt or do any other things to enable it. This have to be done separately, typically by writing various device register.

14.3 Example Ada95 Interrupt Program

An example program (`irq_test`) is included in the Ada examples package to show how interrupts can be handled in Ada95. Note that generation of the test interrupt (`irqforce.c`) is BSP specific and must be edited.

Note: The `irq_test` example was written for the SPARC/ERC32 BSP.

14.4 Version Requirements

With RTEMS 4.0, a patch was required to `psignal.c` in RTEMS sources (to correct a bug associated to the default action of signals 15-32). The SPARC/ERC32 RTEMS BSP includes the “gnatsupp” subdirectory that can be used as an example for other BSPs.

With GNAT 3.11p, a patch is required for `a-init.c` to invoke the BSP specific routine that installs the exception handlers.

SHARED MEMORY SUPPORT DRIVER

The Shared Memory Support Driver is responsible for providing glue routines and configuration information required by the Shared Memory Multiprocessor Communications Interface (MPCI). The Shared Memory Support Driver tailors the portable Shared Memory Driver to a particular target platform.

This driver is only required in shared memory multiprocessing systems that use the RTEMS multiprocessing support. For more information on RTEMS multiprocessing capabilities and the MPCI, refer to the *Multiprocessing Manager* chapter of the *RTEMS Application C User's Guide*.

15.1 Shared Memory Configuration Table

The Shared Memory Configuration Table is defined in the following structure:

```

1 typedef volatile uint32_t vol_u32;
2
3 typedef struct {
4     vol_u32 *address;      /* write here for interrupt */
5     vol_u32 value;         /* this value causes interrupt */
6     vol_u32 length;        /* for this length (0,1,2,4) */
7 } Shm_Interrupt_information;
8
9 struct shm_config_info {
10     vol_u32      *base;          /* base address of SHM */
11     vol_u32      length;         /* length (in bytes) of SHM */
12     vol_u32      format;         /* SHM is big or little endian */
13     vol_u32      (*convert)();   /* neutral conversion routine */
14     vol_u32      poll_intr;      /* POLLED or INTR driven mode */
15     void          (*cause_intr)( uint32_t );
16     Shm_Interrupt_information Intr; /* cause intr information */
17 };
18
19 typedef struct shm_config_info shm_config_table;

```

where the fields are defined as follows:

base

is the base address of the shared memory buffer used to pass messages between the nodes in the system.

length

is the length (in bytes) of the shared memory buffer used to pass messages between the nodes in the system.

format

is either SHM_BIG or SHM_LITTLE to indicate that the neutral format of the shared memory area is big or little endian. The format of the memory should be chosen to match most of the inter-node traffic.

convert

is the address of a routine which converts from native format to neutral format. Ideally, the neutral format is the same as the native format so this routine is quite simple.

poll_intr, cause_intr

is either INTR_MODE or POLLED_MODE to indicate how the node will be informed of incoming messages.

Intr

is the information required to cause an interrupt on a node. This structure contains the following fields:

address

is the address to write at to cause an interrupt on that node. For a polled node, this should be NULL.

value

is the value to write to cause an interrupt.

length

is the length of the entity to write on the node to cause an interrupt. This can be 0 to indicate polled operation, 1 to write a byte, 2 to write a sixteen-bit entity, and 4 to write a thirty-two bit entity.

15.2 Primitives

15.2.1 Convert Address

The `Shm_Convert_address` is responsible for converting an address of an entity in the shared memory area into the address that should be used from this node. Most targets will simply return the address passed to this routine. However, some target boards will have a special window onto the shared memory. For example, some VMEbus boards have special address windows to access addresses that are normally reserved in the CPU's address space.

```

1 void *Shm_Convert_address( void *address )
2 {
3     return the local address version of this bus address
4 }
```

15.2.2 Get Configuration

The `Shm_Get_configuration` routine is responsible for filling in the Shared Memory Configuration Table passed to it.

```

1 void Shm_Get_configuration(
2     uint32_t      localnode,
3     shm_config_table **shmcfg
4 )
5 {
6     fill in the Shared Memory Configuration Table
7 }
```

15.2.3 Locking Primitives

This is a collection of routines that are invoked by the portable part of the Shared Memory Driver to manage locks in the shared memory buffer area. Accesses to the shared memory must be atomic. Two nodes in a multiprocessor system must not be manipulating the shared data structures simultaneously. The locking primitives are used to insure this.

To avoid deadlock, local processor interrupts should be disabled the entire time the locked queue is locked.

The locking primitives operate on the lock field of the `Shm_Locked_queue_Control` data structure. This structure is defined as follows:

```

1 typedef struct {
2     vol_u32 lock; /* lock field for this queue */
3     vol_u32 front; /* first envelope on queue */
4     vol_u32 rear; /* last envelope on queue */
5     vol_u32 owner; /* receiving (i.e. owning) node */
6 } Shm_Locked_queue_Control;
```

where each field is defined as follows:

lock

is the lock field. Every node in the system must agree on how this field will be used. Many

processor families provide an atomic “test and set” instruction that is used to manage this field.

front

is the index of the first message on this locked queue.

rear

is the index of the last message on this locked queue.

owner

is the node number of the node that currently has this structure locked.

15.2.3.1 Initializing a Shared Lock

The `Shm_Initialize_lock` routine is responsible for initializing the lock field. This routines usually is implemented as follows:

```

1 void Shm_Initialize_lock(
2   Shm_Locked_queue_Control *lq_cb
3 )
4 {
5   lq_cb->lock = LQ_UNLOCKED;
6 }
```

15.2.3.2 Acquiring a Shared Lock

The `Shm_Lock` routine is responsible for acquiring the lock field. Interrupts should be disabled while that lock is acquired. If the lock is currently unavailable, then the locking routine should delay a few microseconds to allow the other node to release the lock. Doing this reduces bus contention for the lock. This routines usually is implemented as follows:

```

1 void Shm_Lock(
2   Shm_Locked_queue_Control *lq_cb
3 )
4 {
5   disable processor interrupts
6   set Shm_isrstat to previous interrupt disable level
7
8   while ( TRUE ) {
9     atomically attempt to acquire the lock
10    if the lock was acquired
11      return
12    delay some small period of time
13  }
14 }
```

15.2.3.3 Releasing a Shared Lock

The `Shm_Unlock` routine is responsible for releasing the lock field and reenabling processor interrupts. This routines usually is implemented as follows:

```
1 void Shm_Unlock(  
2   Shm_Locked_queue_Control *lq_cb  
3 )  
4 {  
5   set the lock to the unlocked value  
6   reenable processor interrupts to their level prior  
7   to the lock being acquired.  This value was saved  
8   in the global variable Shm_isrstat  
9 }
```


15.3 Installing the MPCCI ISR

The `Shm_setvec` is invoked by the portable portion of the shared memory to install the interrupt service routine that is invoked when an incoming message is announced. Some target boards support an interprocessor interrupt or mailbox scheme and this is where the ISR for that interrupt would be installed.

On an interrupt driven node, this routine would be implemented as follows:

```
1 void Shm_setvec( void )  
2 {  
3     install the interprocessor communications ISR  
4 }
```

On a polled node, this routine would be empty.

TIMER DRIVER

Warning: The Timer Driver is superfluous and should be replaced by the RTEMS counter support. Ask on the mailing list if you plan to write a Timer Driver.

The timer driver is primarily used by the RTEMS Timing Tests. This driver provides as accurate a benchmark timer as possible. It typically reports its time in microseconds, CPU cycles, or bus cycles. This information can be very useful for determining precisely what pieces of code require optimization and to measure the impact of specific minor changes.

The gen68340 BSP also uses the Timer Driver to support a high performance mode of the on-CPU UART.

16.1 Benchmark Timer

The RTEMS Timing Test Suite requires a benchmark timer. The RTEMS Timing Test Suite is very helpful for determining the performance of target hardware and comparing its performance to that of other RTEMS targets.

This section describes the routines which are assumed to exist by the RTEMS Timing Test Suite. The names used are *EXACTLY* what is used in the RTEMS Timing Test Suite so follow the naming convention.

16.1.1 benchmark_timer_initialize

Initialize the timer source.

```
1 void benchmark_timer_initialize(void)
2 {
3     initialize the benchmark timer
4 }
```

16.1.2 Read_timer

The benchmark_timer_read routine returns the number of benchmark time units (typically microseconds) that have elapsed since the last call to benchmark_timer_initialize.

```
1 benchmark_timer_t benchmark_timer_read(void)
2 {
3     stop time = read the hardware timer
4     if the subtract overhead feature is enabled
5         subtract overhead from stop time
6     return the stop time
7 }
```

Many implementations of this routine subtract the overhead required to initialize and read the benchmark timer. This makes the times reported more accurate.

Some implementations report 0 if the hardware timer value change is sufficiently small. This is intended to indicate that the execution time is below the resolution of the timer.

16.1.3 benchmark_timer_disable_subtracting_average_overhead

This routine is invoked by the “Check Timer” (tmck) test in the RTEMS Timing Test Suite. It makes the benchmark_timer_read routine NOT subtract the overhead required to initialize and read the benchmark timer. This is used by the tmoverhd test to determine the overhead required to initialize and read the timer.

```
1 void benchmark_timer_disable_subtracting_average_overhead(bool find_flag)
2 {
3     disable the subtract overhead feature
4 }
```

The benchmark_timer_find_average_overhead variable is used to indicate the state of the “subtract overhead feature”.

16.2 gen68340 UART FIFO Full Mode

The gen68340 BSP is an example of the use of the timer to support the UART input FIFO full mode (FIFO means First In First Out and roughly means buffer). This mode consists in the UART raising an interrupt when n characters have been received (n is the UART's FIFO length). It results in a lower interrupt processing time, but the problem is that a scanf primitive will block on a receipt of less than n characters. The solution is to set a timer that will check whether there are some characters waiting in the UART's input FIFO. The delay time has to be set carefully otherwise high rates will be broken:

- if no character was received last time the interrupt subroutine was entered, set a long delay,
- otherwise set the delay to the delay needed for n characters receipt.

ATA DRIVER

Warning: The ATA/IDE Drivers are out of date and should not be used for new BSPs. The preferred alternative is to port the ATA/SATA/SCSI/NVMe support from FreeBSD to RTEMS using the [libbsd](#). Ask on the mailing list if you plan to write a driver for an ATA/IDE device.

17.1 Terms

ATA device - physical device attached to an IDE controller

17.2 Introduction

ATA driver provides generic interface to an ATA device. ATA driver is hardware independent implementation of ATA standard defined in working draft “AT Attachment Interface with Extensions (ATA-2)” X3T10/0948D revision 4c, March 18, 1996. ATA Driver based on IDE Controller Driver and may be used for computer systems with single IDE controller and with multiple as well. Although current implementation has several restrictions detailed below ATA driver architecture allows easily extend the driver. Current restrictions are:

- Only mandatory (see draft p.29) and two optional (READ/WRITE MULTIPLE) commands are implemented
- Only PIO mode is supported but both poll and interrupt driven

The reference implementation for ATA driver can be found in `cpukit/libblock/src/ata.c`.

17.3 Initialization

The `ata_initialize` routine is responsible for ATA driver initialization. The main goal of the initialization is to detect and register in the system all ATA devices attached to IDE controllers successfully initialized by the IDE Controller driver.

In the implementation of the driver, the following actions are performed:

```
1 rtems_device_driver ata_initialize(  
2   rtems_device_major_number  major,  
3   rtems_device_minor_number  minor,  
4   void                       *arg  
5 )  
6 {  
7   initialize internal ATA driver data structure  
8  
9   for each IDE controller successfully initialized by the IDE Controller driver  
10    if the controller is interrupt driven  
11      set up interrupt handler  
12  
13    obtain information about ATA devices attached to the controller  
14    with help of EXECUTE DEVICE DIAGNOSTIC command  
15  
16    for each ATA device detected on the controller  
17      obtain device parameters with help of DEVICE IDENTIFY command  
18  
19    register new ATA device as new block device in the system  
20 }
```

Special processing of ATA commands is required because of absence of multitasking environment during the driver initialization.

Detected ATA devices are registered in the system as physical block devices (see libblock library description). Device names are formed based on IDE controller minor number device is attached to and device number on the controller (0 - Master, 1 - Slave). In current implementation 64 minor numbers are reserved for each ATA device which allows to support up to 63 logical partitions per device.

controller minor	device number	device name	ata device minor
0	0	hda	0
0	1	hdb	64
1	0	hdc	128
1	1	hdd	172
...

17.4 ATA Driver Architecture

17.4.1 ATA Driver Main Internal Data Structures

ATA driver works with ATA requests. ATA request is described by the following structure:

```

1 /* ATA request */
2 typedef struct ata_req_s {
3     Chain_Node    link;    /* link in requests chain */
4     char          type;    /* request type */
5     ata_registers_t regs;  /* ATA command */
6     uint32_t      cnt;     /* Number of sectors to be exchanged */
7     uint32_t      cbuf;    /* number of current buffer from breq in use */
8     uint32_t      pos;     /* current position in 'cbuf' */
9     blkdev_request *breq;   /* blkdev_request which corresponds to the ata request */
10    rtems_id       sema;    /* semaphore which is used if synchronous
11                             * processing of the ata request is required */
12    rtems_status_code status; /* status of ata request processing */
13    int            error;    /* error code */
14 } ata_req_t;

```

ATA driver supports separate ATA requests queues for each IDE controller (one queue per controller). The following structure contains information about controller's queue and devices attached to the controller:

```

1 /*
2  * This structure describes controller state, devices configuration on the
3  * controller and chain of ATA requests to the controller.
4  */
5 typedef struct ata_ide_ctrl_s {
6     bool    present;    /* controller state */
7     ata_dev_t device[2]; /* ata devices description */
8     Chain_Control reqs;  /* requests chain */
9 } ata_ide_ctrl_t;

```

Driver uses array of the structures indexed by the controllers minor number.

The following structure allows to map an ATA device to the pair (IDE controller minor number device is attached to, device number on the controller):

```

1 /*
2  * Mapping of RTEMS ATA devices to the following pairs:
3  * (IDE controller number served the device, device number on the controller)
4  */
5 typedef struct ata_ide_dev_s {
6     int ctrl_minor; /* minor number of IDE controller serves RTEMS ATA device */
7     int device;     /* device number on IDE controller (0 or 1) */
8 } ata_ide_dev_t;

```

Driver uses array of the structures indexed by the ATA devices minor number.

ATA driver defines the following internal events:

```

1 /* ATA driver events */
2 typedef enum ata_msg_type_s {
3     ATA_MSG_GEN_EVT = 1,    /* general event */

```

```
4  ATA_MSG_SUCCESS_EVT,      /* success event */
5  ATA_MSG_ERROR_EVT,       /* error event */
6  ATA_MSG_PROCESS_NEXT_EVT /* process next ata request event */
7 } ata_msg_type_t;
```

17.4.2 Brief ATA Driver Core Overview

All ATA driver functionality is available via ATA driver ioctl. Current implementation supports only two ioctls: BLKIO_REQUEST and ATAIO_SET_MULTIPLE_MODE. Each ATA driver ioctl() call generates an ATA request which is appended to the appropriate controller queue depending on ATA device the request belongs to. If appended request is single request in the controller's queue then ATA driver event is generated.

ATA driver task which manages queue of ATA driver events is core of ATA driver. In current driver version queue of ATA driver events implemented as RTEMS message queue. Each message contains event type, IDE controller minor number on which event happened and error if an error occurred. Events may be generated either by ATA driver ioctl call or by ATA driver task itself. Each time ATA driver task receives an event it gets controller minor number from event, takes first ATA request from controller queue and processes it depending on request and event types. An ATA request processing may also includes sending of several events. If ATA request processing is finished the ATA request is removed from the controller queue. Note, that in current implementation maximum one event per controller may be queued at any moment of the time.

(This part seems not very clear, hope I rewrite it soon)

IDE CONTROLLER DRIVER

Warning: The ATA/IDE Drivers are out of date and should not be used for new BSPs. The preferred alternative is to port the ATA/SATA/SCSI/NVMe support from FreeBSD to RTEMS using the [libbsd](#). Ask on the mailing list if you plan to write a driver for an ATA/IDE device.

18.1 Introduction

The IDE Controller driver is responsible for providing an interface to an IDE Controller. The capabilities provided by this driver are:

- Read IDE Controller register
- Write IDE Controller register
- Read data block through IDE Controller Data Register
- Write data block through IDE Controller Data Register

The reference implementation for an IDE Controller driver can be found in `bsps/shared/dev/ide`. This driver is based on the `libchip` concept and allows to work with any of the IDE Controller chips simply by appropriate configuration of BSP. Drivers for a particular IDE Controller chips locate in the following directories: drivers for well-known IDE Controller chips locate into `bsps/shared/dev/ide` and drivers for custom IDE Controller chips (for example, implemented on FPGA) locate into `bsps/${RTEMS_CPU}/${RTEMS_BSP}/ata`. There is a README file in these directories for each supported IDE Controller chip. Each of these README explains how to configure a BSP for that particular IDE Controller chip.

18.2 Initialization

IDE Controller chips used by a BSP are statically configured into IDE_Controller_Table. The ide_controller_initialize routine is responsible for initialization of all configured IDE controller chips. Initialization order of the chips based on the order the chips are defined in the IDE_Controller_Table.

The following actions are performed by the IDE Controller driver initialization routine:

```
1 rtems_device_driver ide_controller_initialize(  
2   rtems_device_major_number  major,  
3   rtems_device_minor_number  minor_arg,  
4   void                       *arg  
5 )  
6 {  
7   for each IDE Controller chip configured in IDE_Controller_Table  
8     if (BSP dependent probe(if exists) AND device probe for this IDE chip  
9       indicates it is present)  
10      perform initialization of the particular chip  
11      register device with configured name for this chip  
12 }
```

18.3 Read IDE Controller Register

The `ide_controller_read_register` routine reads the content of the IDE Controller chip register. IDE Controller chip is selected via the minor number. This routine is not allowed to be called from an application.

```
1 void ide_controller_read_register(  
2     rtems_device_minor_number  minor,  
3     unsigned32                 reg,  
4     unsigned32                 *value  
5 )  
6 {  
7     get IDE Controller chip configuration information from  
8     IDE_Controller_Table by minor number  
9  
10    invoke read register routine for the chip  
11 }
```


18.4 Write IDE Controller Register

The `ide_controller_write_register` routine writes IDE Controller chip register with specified value. IDE Controller chip is selected via the minor number. This routine is not allowed to be called from an application.

```
1 void ide_controller_write_register(  
2     rtems_device_minor_number minor,  
3     unsigned32                reg,  
4     unsigned32                value  
5 )  
6 {  
7     get IDE Controller chip configuration information from  
8     IDE_Controller_Table by minor number  
9  
10    invoke write register routine for the chip  
11 }
```

18.5 Read Data Block Through IDE Controller Data Register

The `ide_controller_read_data_block` provides multiple consequent read of the IDE Controller Data Register. IDE Controller chip is selected via the minor number. The same functionality may be achieved via separate multiple calls of `ide_controller_read_register` routine but `ide_controller_read_data_block` allows to escape functions call overhead. This routine is not allowed to be called from an application.

```
1 void ide_controller_read_data_block(  
2     rtems_device_minor_number minor,  
3     unsigned16                block_size,  
4     blkdev_sg_buffer          *bufs,  
5     uint32_t                  *cbuf,  
6     uint32_t                  *pos  
7 )  
8 {  
9     get IDE Controller chip configuration information from  
10    IDE_Controller_Table by minor number  
11  
12    invoke read data block routine for the chip  
13 }
```

18.6 Write Data Block Through IDE Controller Data Register

The `ide_controller_write_data_block` provides multiple consequent write into the IDE Controller Data Register. IDE Controller chip is selected via the minor number. The same functionality may be achieved via separate multiple calls of `ide_controller_write_register` routine but `ide_controller_write_data_block` allows to escape functions call overhead. This routine is not allowed to be called from an application.

```
1 void ide_controller_write_data_block(  
2     rtems_device_minor_number minor,  
3     unsigned16                block_size,  
4     blkdev_sg_buffer          *bufs,  
5     uint32_t                  *cbuf,  
6     uint32_t                  *pos  
7 )  
8 {  
9     get IDE Controller chip configuration information from  
10    IDE_Controller_Table by minor number  
11  
12    invoke write data block routine for the chip  
13 }
```


COMMAND AND VARIABLE INDEX

There are currently no Command and Variable Index entries.

DOXYGEN RECOMMENDATIONS FOR BSPS

RTEMS contains well over a hundred [Board Support Packages \(BSPs\)](#). , across over 20 different [CPU Architectures](#). . What this means is that there is a lot of hardware dependent code that gets written, and that adding Doxygen to properly document it all can be a very complicated task.

The goal of this document is to attempt to simplify this process a bit, and to get you started on adding Doxygen to the `bsps/` directory in a way that is logical and has structure. Before we move on to detailing the process of actually adding Doxygen to BSPs, you will be greatly served by having at least a basic understanding of the purpose of a Board Support Package (it always helps to know a bit about what you're documenting), as well as of the existing structure of the `bsps/` directory.

Feel free to skip around and skim parts of this.

20.1 BSP Basics

Embedded development is hard. Different CPUs have different instructions for doing the same thing, and different boards will have all sorts of different hardware that require unique drivers and interfaces. RTEMS handles this by having discrete packages, BSPs, to encapsulate code to accommodate for unique hardware. BSPs seek to implement the Hardware-Software interface. This, in a nutshell, is one of the **core purposes** of RTEMS: To abstract (as much as is possible) away from the physical hardware and provide a standards compliant real-time environment for the embedded developer. If you think about it, the operating system on your normal computer serves a very similar purpose.

20.2 Common Features Found In BSPs

Although the actual implementation code will differ between BSPs, all BSPs will share some degree of common functionality. This is because that no matter what exact hardware you have, you need some basic features implemented in order to have a real time system you can develop on. Some of the most common shared features across most boards include:

- **console**: is technically the serial driver for the BSP rather than just a console driver, it deals with the board UART (i.e. serial devices) * **clock**: support for the clock tick - a regular time basis for the kernel * **timer**: support of timer devices, used for timing tests * **rtc** or **tod**: support for the hardware real time clock * **network**: the Ethernet driver * **shmsupp**: support of shared memory driver MPCI layer in a multiprocessor system * **gnatsupp**: BSP specific support for the GNU Ada run-time * **irq**: support for how the processor handles interrupts (probably the most common module shared by all boards) * **tm27**: specific routines for the tm27 timing test * **start** and **startup**: C and assembly used to initialize the board during startups/resets/reboots

These are just some of the things you should be looking for when adding Doxygen to a BSP.

Note that there is no guarantee a particular BSP will implement all of these features, or even some of them. These are just the most common ones to look for. RTEMS follows a standardized naming convention for the BSP sub directories, so you should be able to tell in most cases what has been implemented on the BSP level and what has not.

20.3 Shared Features

Some of the RTEMS executive is hardware independent and can be abstracted so that the same piece of code can be shared across multiple CPU architectures, or across multiple boards on the same architecture. This is done so that chunks of software can be reused, as well as aiding in reducing the development and debugging time for implementing new BSPs. This greatly aids the developer, but as someone seeking to document this code, this can make your life a little bit harder. It is hard to tell by looking at the directory of a BSP which features have simply been left out and which features are being implemented by using shared code from either from the architecture (../shared) or the base bsp/ shared directory (../../shared). You may be looking at the BSP headers and notice that you have an `irq.h`, but no `irq.c` implementing it, or you might even be missing both. You know that the processor has interrupt support somehow, but where is it? The easiest way to figure this out is by looking at the `Makefile.am` for a BSP. We'll detail this process more in a bit.

20.4 Rationale

As someone adding documentation and not doing actual development work, you might think it is not necessary to know some of the in and outs of BSPs. In actuality, this information will prove to be very useful. Doxygen documentation works by grouping things and their components (i.e. functions and other definitions), and by having brief descriptions of what each group does. You can't know what to look for or know how to group it or know how to describe it without some basic knowledge of what a BSP is. For more information on any of the above or BSPs in general, check out the [BSP Development Guide](#). .

20.5 The Structure of the bsp/ directory

All BSPs are found within the bsp/ directory, which is itself very well ordered. At the first level, we find a directory for each CPU architecture RTEMS supports, as well as a directory for code shared by all implementations.

```

1 $ cd bsp
2 $ ls
3 arm  bsp.am  lm32  m68k          mips  no_cpu      README  sparc
4 avr  h8300   m32c  Makefile.am     moxie powerpc     sh      sparc64
5 bfin i386     m32r  MERGE.PROCEDURE nios2  preinstall.am shared  v850

```

If we cd into a specific architecture, we see that a similar structure is employed. bsp/arm/ contains directories for each Board Support Package for boards with an ARM cpu, along with a folder for files and .h's shared by all BSPs of that architecture.

```

1 $ cd arm
2 $ ls
3 acinclude.m4  edb7312  gumstix  Makefile.am  realview-pbx-a9  stm32f4
4 configure.ac  gba      lm3s69xx nds          rtl22xx          xilinx-zynq
5 csb336       gdbarmsim lpc24xx  preinstall.am shared
6 csb337       gp32     lpc32xx  raspberrypi  smdk2410

```

Finally, if we cd into a specific BSP, we see the files and .h's that compose the package for that particular board. You may recognize the directory names as some of the [common features] we outlined above, like “irq”, “clock”, “console”, and “startup”. These directories contain implementations of these features.

```

1 $ cd raspberrypi
2 $ ls
3 bsp_specs  configure.ac  include  make      misc      README
4 clock      console      irq      Makefile.am  preinstall.am  startup

```

Another way to get an idea of the structure of bsp/ is to navigate to a directory and execute the “tree -f” command. This outputs a nice graphic that conveys some of the hierarchical properties of a particular directory.

```

1 $ pwd
2 ~/rtems/bsp/arm/raspberrypi
3 $ tree -f
4 .
5 |-- ./bsp_specs
6 |-- ./clock
7 |   |-- ./clock/clockdrv.c
8 |-- ./configure.ac
9 |-- ./console
10 |   |-- ./console/console-config.c
11 |     |-- ./console/usart.c
12 |-- ./include
13 |   |-- ./include/bsp.h
14 |   |-- ./include/irq.h
15 |   |-- ./include/mmu.h
16 |   |-- ./include/raspberrypi.h
17 |   |-- ./include/usart.h
18 |-- ./irq
19 |   |-- ./irq/irq.c

```

```

20 |-- ./make
21 |   |-- ./make/custom
22 |       |-- ./make/custom/raspberrypi.cfg
23 |-- ./Makefile.am
24 |-- ./misc
25 |   |-- ./misc/timer.c
26 |-- ./preinstall.am
27 |-- ./README
28 |-- ./startup
29 |   |-- ./startup/bspreset.c
30 |   |-- ./startup/bspstart.c
31 |   |-- ./startup/bspstarthooks.c
32 |   |-- ./startup/linkcmds
33 |   |-- ./startup/mm_config_table.c

```

In short, BSPs will use the following directories:

- **bsps/shared** <- code used that is shared by all BSPs
- **bsps/CPU/shared** <- code used shared by all BSPs of a particular CPU architecture
- **bsps/CPU/BSP** <- code unique to this BSP

As you can see, the bsp/ directory has a very logical and easy to understand structure to it. The documentation generated by Doxygen should attempt to match this structure as closely as possible. We want an overarching parent group to serve the same purpose as the bsp/ directory. In it, we want groups for each CPU architecture and a group for the shared files. We then want groups for each BSP. Breaking our documentation up into discrete groups like this will greatly simplify the process and make the documentation much easier to go through. By learning about the existing structure of the bsp/ directory, we get an idea of how we should structure the Doxygen groups we create. More on this in the next section.

20.6 Doxygen

Now that we have covered some of the preliminaries, we can move on to what you are actually reading this wiki page for: adding Doxygen to the bsp/ directory. Let's start with some Doxygen basics. Skip this if you are already comfortable with Doxygen.

In addition to this, check out the page on **'Doxygen Recommendations <wiki:Developer/Coding/Doxygen >_.'**, which also contains a fair amount of information that will not be covered here.

20.7 Doxygen Basics

Doxygen is a documentation generator. It allows for documentation to be written right by the source code, greatly easing the pains of keeping documentation relevant and up to date. Doxygen has many commands, used for things like annotating functions with descriptions, parameter information, or return value information. You can reference other files or even other documentation.

The core component of Doxygen (that we care about right now at least) is what's called a **group**, or **module**. These are used to add structure and associate groups of files that serve a similar purpose or implement the same thing.

20.8 Doxygen Headers

Doxygen is always found in a special Doxygen comment block, known as a **Doxygen header**. In RTEMS, this block comes in the form of a multiline comment with some included Doxygen commands, which are preceded by the ‘@’ tag. Take a look at this Doxygen header that declares the arm_raspberrypi module, which houses the documentation in the BSP for the Raspberry Pi.

```
1 bsp/arm/raspberrypi/include/bsp.h:
2
3 /**
4  * @defgroup arm_raspberrypi Raspberry Pi Support
5  *
6  * @ingroup bsp_arm
7  *
8  * @brief Raspberry Pi support package
9  *
10 */
```

You see a few commands here that we’ll cover in the following sections. Briefly, the @defgroup command declares a new group, the @ingroup command nests this group as a submodule of some other group (in this case bsp_arm), and the @brief command provides a brief description of what this group is.

20.9 The @defgroup Command

The @defgroup command is used to declare new groups or modules. Think “define group”. The syntax of this command is as follows:

```
1 @defgroup <group name> <group description>
```

The group name is the name used by Doxygen elsewhere to reference this group. The group description is what is displayed when the end user navigates to this module in the resulting documentation. The group description is a couple words formatted as how it would be in a table of contents. This part is what actually shows up in the documentation, when the user navigates to this group’s module, this description will be the modules name.

Groups should only be declared (@defgroup) in .h files. This is because Doxygen is used primarily to document interfaces, which are only found in .h files. Placing @defgroups in .h files is the only real restriction. Which .h file you place the group declaration in surprisingly doesn’t matter. There is no information in the resulting documentation that indicates where the group was declared. You will see that we do have some rules for where you should place these declarations, but we also use this fact that it doesn’t matter to our advantage, in order to standardize things.

The @defgroup command is used only to define “structure”. No actual documentation is generated as a result of its use. We must @ingroup things to the group we declare in order to create documentation. Even though it does not generate visible documentation, the @defgroup command is still very important. We use it in a way that seeks to emulate the structure of the bsp/ directory itself. We do this by creating a hierarchy of groups for each CPU architecture and each BSP.

20.10 The @ingroup Command

The @ingroup command is used to add ‘things’ to already declared groups or modules. These ‘things’ can either be other groups, or files themselves. The syntax of the @ingroup command is as follows:

```
1 @ingroup <group name>
```

The group name is the actual name, not description, of the group you want to add yourself to. Remember that group name was the second argument passed to the @defgroup command.

Using the @ingroup command is how we add “meaning” to the “structure” created by using @defgroup. @ingroup associates the file it is found in and all other Doxygen found within (function annotations, prototypes, etc) with the group we declared with the @defgroup command. We add related files and headers to the same groups to create a logical and cohesive body of documentation. If the end user wanted to read documentation about how the raspberry pi handles interrupts, all they would have to do would be to navigate to the raspberry pi’s interrupt support module (which we created with a @defgroup command), and read the documentation contained within (which we added with @ingroup commands).

@ingroup is found within all Doxygen headers, along with an @brief statement. There are two types of Doxygen headers, which we will go over after we see a description of the @brief command.

20.11 The @brief Command

The @brief command is used to give either a) a brief description in the form of an entry as you would see it in a table of contents (i.e. Capitalized, only a couple of words) or b) a brief topic sentence giving a basic idea of what the group does. The reason you have two uses for the brief command is that it is used differently in the two types of Doxygen headers, as we will see shortly. The syntax of the brief command is self evident, but included for the sake of completion:

```
1 @brief <Table of Contents entry 'or' Topic Sentence>
```

20.12 The Two Types of Doxygen Headers

There are two types of Doxygen Headers. The first type is found at the beginning of a file, and contains an `@file` command. This type of header is used when `@ingroup`-ing the file into another doxygen group. The form of the `@brief` command in this case is a topic sentence, often very close to the file name or one of it's major functions. An example of this type of header, found in `bsps/arm/raspberrypi/include/bsp.h` is as follows:

```

1 Header type 1: used to add files to groups, always found at the beginning of a
  ↪ file
2 /**
3  * @file
4  *
5  * @ingroup raspberrypi
6  *
7  * @brief Global BSP definitions.
8  */
9
10 /**
11  * Copyright (c) YYYY NAME
12  *
13  * <LICENSE TERMS>
14  */

```

Notice the form and placement of this type of header. It is always found at the beginning of a file, and is in its own multiline comment block, separated by one line white space from the copyright. If you look at the header itself, you see a `@file`, `@ingroup`, and `@brief` command. Consider the `@file` and the `@ingroup` together, what this says is that we are adding this file to the `raspberrypi` group. There is actually a single argument to the `@file` command, but Doxygen can infer it, so we leave it out. Any other Doxygen, function annotations, function prototypes, `#defines`, and other code included in the file will now be visible and documented when the end user navigates to the group you added it to in the resulting documentation.

Now let's consider the second type of header. This type is syntactically very similar, but is used not to add files to groups, but to add groups to other groups. We use this type of header to define new groups and nest them within old groups. This is how we create hierarchy and structure within Doxygen. The following is found, again, in `bsps/arm/raspberrypi/include/bsp.h`:

```

1 Header type 2: Used to nest groups, found anywhere within a file
2 /**
3  * @defgroup arm_raspberrypi Raspberry Pi Support
4  *
5  * @ingroup bsp_arm
6  *
7  * @brief Raspberry Pi Support Package
8  */

```

It looks very similar to the first type of header, but notice that the `@file` command is replaced with the `@defgroup` command. You can think about it in the same way though. Here we are creating a new group, the `arm_raspberrypi` group, and nesting it within the `bsp_arm` group. The `@brief` in this case should be in the form of how you would see it in a table of contents. Words should be capitalized and there should be no period. This type of header can be found anywhere in a file, but it is typically found either in the middle before the file's main function, or at the tail end of a file. Recall that as we are using the `@defgroup` command and creating a new group in this header, the actual `.h` we place this in does not matter.

The second type of header is the **structure** header, it's how we create new groups and implement hierarchy. The first type of header was the **meaning** header, it's how we added information to the groups we created.

For more examples of Doxygen structure and syntax, refer to BSPs found within the arm architecture, the lpc32xx and raspberrypi BSPs are particularly well documented. A good way to quickly learn more is by tweaking some Doxygen in a file, then regenerating the html, and seeing what has changed.

20.13 Generating Documentation

Doxygen is a documentation generator, and as such, we must generate the actual html documentation to see the results of our work. This is a very good way to check your work, and see if the resulting structure and organization was what you had intended. The best way to do this is to simply run the `do_doxygen script`. To use the script:

Make sure Doxygen is installed. Also, the environment needs to have the root directory of RTEMS set in the variable `r` so that `$r` prints the path to RTEMS, and the script takes as argument a relative directory from there to generate the doxygen, for example to generate the doxygen for all of bsp/ you would do:

```
1 export r=~ /rtems
2 ./do_doxygen bsp
```

20.14 Doxygen in bsps/

Now that we've covered the basics of Doxygen, the basics of BSPs and the structure of the bsps/ directory, actually adding new Doxygen to bsps/ will be much easier than it was before. We will cover a set of rules and conventions that you should follow when adding Doxygen to this directory, and include some tips and tricks.

20.15 Group Naming Conventions

This is an easy one. These are in place in order for you to quickly identify some of the structure of the Doxygen groups and nested groups, without actually generating and looking at the documentation. The basic idea is this: when defining a new group (@defgroup), the form of the name should be the super group, or the name of the group you are nesting this group within, followed by an underscore, followed by the intended name of this new group. In command form:

```
1 <----- This is your group name -----> <---usual description -->
2 @defgroup <super-group name>_<name of this group> <group description>
```

Some examples of this:

- **bsp_arm**: This is the group for the arm architecture. It is a

member of the all inclusive bsp-kit group (more on this in structure conventions), so we prefix it with the “**bsp**” super group name. This is the group for the arm architecture, so the rest is just “arm”

- **arm_raspberrypi**: This is the group for the Raspberry Pi BSP. It

is is an arm board, and as such, is nested within the bsp_arm group. We prefix the group name with an “**arm**” (notice we drop the bsp prefix of the arm group - we only care about the immediate super group), and the rest is a simple “raspberrypi”, indicating this is the raspberrypi group, which is nested within the bsp_arm group.

- **raspberrypi_interrupt** This is the group for code handling

interrupts on the Raspberry Pi platform. Because this code and the group that envelops it is Raspberry Pi dependent, we prefix our name with a “**raspberrypi**”, indicating this group is nested within the raspberrypi group.

This covers where, when, and why you should place the second type of Doxygen header. Remember that our goal is to have the structure of the documentation to match the organization of the bsps/ directory as closely as possible. We accomplish this by creating groups for each cpu architecture, each BSP, and each shared directory. These groups are nested as appropriate in order to achieve a hierarchy similar to that of bsps/. The arm_raspberrypi group would be nested within the bsp_arm group, for example.

20.16 Where to place @defgroup

Remember how I said it really doesn't matter where you place the @defgroup? Well, it does and it doesn't. It would be chaotic to place these anywhere, and almost impossible to tell when you have a @defgroup and when you don't, so we do have some rules in place to guide where you should place these.

20.17 @defgroup for CPU Architectures and Shared Directories

The standardized place for these is within a special `doxygen.h` file placed within the particular architectures shared directory. This `doxygen.h` file exists solely for this purpose, to provide a standard place to house the group definitions for CPU architectures and the shared directory for that architecture. This is done because there is no single file that all architectures share, so it would be impossible to declare a standardized location for architecture declarations without the creation of a new file. This also allows others to quickly determine if the group for a particular architecture has already been defined or not. Lets look at the `doxygen.h` for the arm architecture as an example, found at `arm/shared/doxygen.h`:

```
1 /**
2  * @defgroup bsp_arm ARM
3  *
4  * @ingroup bsp_kit
5  *
6  * @brief ARM Board Support Packages
7  */
8
9 /**
10 * @defgroup arm_shared ARM Shared Modules
11 *
12 * @ingroup bsp_arm
13 *
14 * @brief ARM Shared Modules
15 */
```

The `doxygen.h` contains only 2 Doxygen headers, both of which are of the second type. One header is used to create the groups for the arm architecture **bsp_arm**, nesting it as part of the **bsp_kit** group, and the other creates an **arm_shared** group to house the code that is shared across all BSPs of this architecture. Because these are the second type of Doxygen header, where we place them does not matter. This allows us to place them in a standard `doxygen.h` file, and the end user is non the wiser. Note that this `.h` file should never be included by a `.c` file, and that the only group declarations that should be placed here are the declarations for the CPU Architecture group and the shared group.

There is also a `doxygen.h` file that exists at the root `bsps/shared` directory, to @defgroup the parent **bsp_kit** group (the only group to not be nested within any other groups) and to @defgroup the **bsp_shared** group, to serve as the holder for the `bsps/shared` directory.

If the architecture in which the BSP you are tasked with does not have one of these files already, you will need to copy the format of the file here, replacing the **arm** with whatever the CPU Architecture you are working with is. Name this file `doxygen.h`, and place it in the shared directory for that architecture.

The only groups you should ever add to this CPU group would be groups for specific BSPs and a group for the shared directory.

20.18 @defgroups for BSPs

These are much easier than placing @defgroups for CPU Architectures. The overwhelming majority of the time, the @defgroup for a BSP is found within the bsp.h file found at “bsp”/include/bsp.h. It is usually placed midway through or towards the end of the file. In the event that your board lacks a bsp.h file, include this group declaration within the most standard or commonly included header for that BSP.

The group for a BSP should **always** be nested within the group for the CPU architecture it uses. This means that the Doxygen header for defining a BSP group should always look something like this:

```
1 /**
2  * @defgroup *architecture*_BSP* *name*
3  *
4  * @ingroup bsp_*architecture*
5  *
6  * @brief *BSP* Support Package
7  */
```

20.19 @defgroups for Everything Else

Never be afraid to add more structure! Once the basic CPU and BSP group hierarchy is established, what we're left with is all the sub directories and implementation code. Whether working within a shared directory for a CPU architecture, or within a BSP directory, you should always be looking for associations you can make to group files together by. Your goal should be to avoid @ingroup-ing files directly to the `cpu_shared` group and the `cpu_bsp` group as much as possible, you want to find more groups you can nest within these groups, and then @ingroup files to those groups. Here are some things to look for:

20.20 Look Common Features Implemented

Remember that list of common features outlined in the BSP Basics section? Find the .h's that are responsible for providing the interface for these features, and @defgroup a group to @ingroup the files responsible for implementing this feature.

RTEMS has a naming convention for its BSP sub directories, so it should be a really quick and easy process to determine what features are there and what is missing.

Examples of this are found within the **arm_raspberrypi** group, which contains nested sub-groups like **raspberrypi_interrupt** to group files responsible for handling interrupts, **raspberrypi_usart** to group files responsible for implementing USART support, and many other sub-groups.

20.21 Check out the Makefile

When working within a BSP, take a look at the Makefile.am. Often times, you will find that the original developer of the code has outlined the groups nicely for you already, with comments and titles before including source files to be built. Also, this is often the only way to tell which features a BSP simply does not implement, and which features a BSP borrows from either the architecture's shared group, or the bsps/ shared group.

20.22 Start with a .h, and look for files that include it

You should end up with a `@defgroup` for “most” .h files. Some .h files are related and will not have independent groups, but most provide interfaces for different features and should have their own group defined. Declare a group for the header, then use `cscope` to find the files that include this header, and try to determine where the implementation code for prototypes are found. These are the files you should `@ingroup`.

20.23 Files with similar names

If you see that a few files have similar names, like they are all prefixed with the same characters, then these files should most likely be part of the same group.

Remember, your goal is to @defgroup as much as you can. The only files you should be @ingroup-ing directly to the BSP group or the shared group are files that don't cleanly fit into any other group.

20.24 Where to place @ingroup

The @ingroups you add should make sense.

- If you are working within an architecture's shared directory, @ingroup should be adding things either to the `*architecture*_shared` group, or some sub group of it.
- If you are working within a BSP directory, @ingroup should be adding things to either the `architecture_*`*bsp* group, or some sub group of it.

20.25 @ingroup in the first type of Doxygen Header

Remember that in the first type of Doxygen header, we are adding files to groups. This type of header should always be at the top of the file. You should be adding files that are associated in some way to the same groups. That is to say, if three different .h files provide an interface allowing interrupt support, they should be a part of the same group. Some good ways to associate files were outlined above.

20.26 @ingroup in the second type of Doxygen Header

Here we are using the @ingroup command to add groups to other groups, creating a hierarchy. The goal for bsp/ is to have one single group that holds all other groups. This root group is the **bsp_kit** group. All groups should be added either directly to this group (if you are creating an architecture group) or added to one of its sub groups.

When nesting groups, try to match the structure of bsp/ as closely as possible. For example, if a group is defined to associate all files that provide for a real time clock for the raspberrypi, nest it within the arm_raspberrypi group.

20.27 @ingroup for shared code

This is tricky. You may end up in a situation where your BSP uses code found in either the architecture shared directory, or the bspshared directory. Even though this code is logically associated with the BSP, as stated above: all files in the shared directory should be added to either the `*architecture*_shared` group, or some subgroup of it “not” the BSP group. You could make a note under the `@brief` line in the header (which shows up in the resulting documentation) that a particular BSP uses this code.

When working with shared code, you should be careful and add notes to `@brief` to indicate that it is a shared code or interface. Prefixing things with “Generic” is a good idea here. You will still be able to form groups and associate things when working on the shared level, but sometimes you will find that you have the interface (.h) to `@defgroup`, but not many files to add to the group as it may be hardware dependent. This is okay.

INDEX

B

BSP_DEFAULT_UNIFIED_WORK_AREAS, 32
BSP_IDLE_TASK_BODY, 32
BSP_IDLE_TASK_STACK_SIZE, 32
bsp_interrupt_dispatch(), 35
bsp_interrupt_facility_initialize(), 35
bsp_interrupt_handler_default(), 35
BSP_INTERRUPT_STACK_SIZE, 32
bsp_interrupt_vector_disable(), 35
bsp_interrupt_vector_enable(), 35
BSP_ZERO_WORKSPACE_AUTOMATICALLY,
32

C

CONFIGURE_MALLOC_BSP_SUPPORTS_
SBRK, 30, 32