RTEMS BSP and Driver Guide

Release 7.0-rc1 (24th January 2026)

CONTENTS

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://gitlab.rtems.org
- Git Repositories https://gitlab.rtems.org
- Developers https://gitlab.rtems.org

CHAPTER

ONE

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 open an issue at for specific work packages.

A basic BSP consists of the following components:

- Low-level initialization
- Console driver
- Clock driver

CHAPTER

TWO

TARGET DEPENDENT FILES

Warning

This chapter contains outdated and confusing information.

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 SPARC LEON3 board family has a fairly large number of variations based upon the particular CPU model and the peripherals actually placed on the SoC.

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

CHAPTER

THREE

LINKER SCRIPT

> **Warning**
>
> This chapter contains outdated and confusing information.

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.

```
/*
 *  Specify that the output is to be coff-m68k regardless of what the
 *  native object format is.
 */
OUTPUT_FORMAT(coff-m68k)
/*
 *  Set the amount of RAM on the target board.
 *
 *  NOTE: The default may be overridden by passing an argument to ld.
 */
RamSize = DEFINED(RamSize) ? RamSize : 4M;
/*
 *  Set the amount of RAM to be used for the application heap.  Objects
 *  allocated using malloc() come from this area.  Having a tight heap
 *  size is somewhat difficult and multiple attempts to squeeze it may
 *  be needed reducing memory usage is important.  If all objects are
 *  allocated from the heap at system initialization time, this eases
 *  the sizing of the application heap.
 *
 *  NOTE 1: The default may be overridden by passing an argument to ld.
 *
 *  NOTE 2: The TCP/IP stack requires additional memory in the Heap.
 *
 *  NOTE 3: The GNAT/RTEMS run-time requires additional memory in
 *  the Heap.
 */
HeapSize = DEFINED(HeapSize) ? HeapSize : 0x10000;
/*
 *  Set the size of the starting stack used during BSP initialization
 *  until first task switch.  After that point, task stacks allocated
 *  by RTEMS are used.
 *
 *  NOTE: The default may be overridden by passing an argument to ld.
 */
StackSize = DEFINED(StackSize) ? StackSize : 0x1000;
/*
 *  Starting addresses and length of RAM and ROM.
 *
 *  The addresses must be valid addresses on the board.  The
 *  Chip Selects should be initialized such that the code addresses
 *  are valid.
 */
MEMORY {
ram : ORIGIN = 0x10000000, LENGTH = 4M
```

(continues on next page)

(continued from previous page)

```
rom : ORIGIN = 0x01000000, LENGTH = 4M
}
/*
 *  This is for the network driver.  See the Networking documentation
 *  for more details.
 */
ETHERNET_ADDRESS =
DEFINED(ETHERNET_ADDRESS) ? ETHERNET_ADDRESS : 0xDEAD12;
/*
 *  The following defines the order in which the sections should go.
 *  It also defines a number of variables which can be used by the
 *  application program.
 *
 *  NOTE: Each variable appears with 1 or 2 leading underscores to
 *        ensure that the variable is accessible from C code with a
 *        single underscore.  Some object formats automatically add
 *        a leading underscore to all C global symbols.
 */
SECTIONS {
/*
 *  Make the RomBase variable available to the application.
 */
_RamSize = RamSize;
__RamSize = RamSize;
/*
 *  Boot PROM  - Set the RomBase variable to the start of the ROM.
 */
rom : {
  _RomBase = .;
  __RomBase = .;
} >rom
/*
 * Dynamic RAM - set the RamBase variable to the start of the RAM.
 */
ram : {
  _RamBase = .;
  __RamBase = .;
} >ram
/*
 *  Text (code) goes into ROM
 */
.text : {
  /*
   *  Create a symbol for each object (.o).
   */
  CREATE_OBJECT_SYMBOLS
  /*
   *  Put all the object files code sections here.
   */
```

(continues on next page)

(continued from previous page)

```
  *(.text)
  . = ALIGN (16);      /*  go to a 16-byte boundary */
  /*
   *  C++ constructors and destructors
   *
   *  NOTE:  See the CROSSGCC mailing-list FAQ for
   *         more details about the "\[......]".
   */
  __CTOR_LIST__ = .;
   [......]
  __DTOR_END__ = .;
  /*
   *  Declares where the .text section ends.
   */
  etext = .;
 _etext = .;
} >rom
/*
 *  Exception Handler Frame section
 */
.eh_fram : {
  . = ALIGN (16);
  *(.eh_fram)
} >ram
/*
 *  GCC Exception section
 */
.gcc_exc : {
  . = ALIGN (16);
  *(.gcc_exc)
} >ram
/*
 *  Special variable to let application get to the dual-ported
 *  memory.
 */
dpram : {
  m340 = .;
  _m340 = .;
  . += (8 * 1024);
} >ram
/*
 *  Initialized Data section goes in RAM
 */
.data : {
  copy_start = .;
  *(.data)
  . = ALIGN (16);
  _edata = .;
  copy_end = .;
```

(continues on next page)

(continued from previous page)

```
} >ram
/*
 *  Uninitialized Data section goes in ROM
 */
.bss : {
  /*
  *  M68K specific: Reserve some room for the Vector Table
  *  (256 vectors of 4 bytes).
  */
  M68Kvec = .;
  _M68Kvec = .;
  . += (256 * 4);
  /*
  *  Start of memory to zero out at initialization time.
  */
  clear_start = .;
  /*
   *  Put all the object files uninitialized data sections
   *  here.
   */
  *(.bss)
  *(COMMON)
  . = ALIGN (16);
  _end = .;
  /*
   *  Start of the Application Heap
   */
  _HeapStart = .;
  __HeapStart = .;
  . += HeapSize;
  /*
  *  The Starting Stack goes after the Application Heap.
  *  M68K stack grows down so start at high address.
  */
  . += StackSize;
  . = ALIGN (16);
  stack_init = .;
  clear_end = .;
  /*
  *  The RTEMS Executive Workspace goes here.  RTEMS
  *  allocates tasks, stacks, semaphores, etc. from this
  *  memory.
  */
  _WorkspaceBase = .;
  __WorkspaceBase = .;
} >ram
```

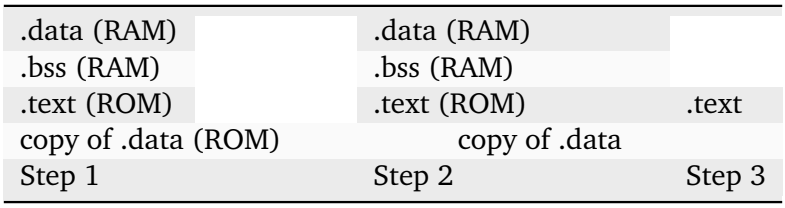
3.5 Initialized Data

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.



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 $RTEMS_ROOT/make/custom/gen68340.cfg:

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

CHAPTER

FOUR

MISCELLANEOUS SUPPORT FILES

> **Warning**
>
> This chapter contains outdated and confusing information.

4.1 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.

4.2 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`.

4.3 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`).

4.4 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 occassionally 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.

4.5 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:

```
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`.

4.6 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.

4.7 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_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.

4.8 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:

```
rtems_isr_entry set_vector(                 /* returns old vector */
  rtems_isr_entry handler,                  /* isr routine        */
  rtems_vector_number vector,               /* vector number      */
  int                 type                  /* RTEMS or RAW intr  */
)
{
  if the type is RAW
    install the raw vector
  else
    use rtems_interrupt_catch to install the vector
  perform any interrupt controller necessary to unmask the interrupt source
  return the previous handler
}
```

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.

4.9 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 `RTEMS_PROFILING` being set to True. 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.

4.10 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 intitalized 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 writen 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()` services 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 :

```
rtems_status_code rtems_interrupt_handler_install(   /* returns status code */
  rtems_vector_number     vector,                    /* interrupt vector */
  const char             *info,                      /* custom identification␣
↪text */
  rtems_option            options,                   /* Type of Interrupt */
  rtems_interrupt_handler handler,                   /* interrupt handler */
  void                   *arg                        /* parameter to be passed
                                                        to handler at the time of
                                                        invocation */
)
rtems_status_code rtems_interrupt_handler_remove(   /* returns status code */
  rtems_vector_number     vector,                   /* interrupt vector */
  rtems_interrupt_handler handler,                  /* interrupt handler */
  void                   *arg                       /* parameter to be passed to␣
↪handler */
)
```

CHAPTER

FIVE

SYSTEM INITIALIZATION

5.1 Introduction

The system initialization consists of a low-level initialization performed by the start code in the start file (`start.o`) and a high-level initialization carried out by `boot_card()`. The final step of a successful high-level initialization is to switch to the initialization task and change into the normal system mode with multi-threading enabled. Errors during system initialization are fatal and end up in a call to `_Terminate()`.

5.2 Low-Level Initialization via Start Code in the Start File (start.o)

The start code in the start file (`start.o`) must be provided by the BSP. It is the first file presented to the linker and starts the process to link an executable (application image). It should contain the entry symbol of the executable. It is the responsibility of the linker script in conjunction with the compiler specifications file or compiler options to put the start code in the correct location in the executable. The start code is typically written in assembly language since it will tinker with the stack pointer. The general rule of thumb is that the start code in assembly language should do the minimum necessary to allow C code to execute to complete the initialization sequence.

The low-level system initialization may depend on a platform initialization carried out by a boot loader. The low-level system initialization may perform the following steps:

- Initialize the initialization stack. The initialization stack should use the ISR stack area. The symbols `_ISR_Stack_area_begin`, `_ISR_Stack_area_end`, and `_ISR_Stack_size` should be used to do this.
- Initialize processor registers and modes.
- Initialize pins.
- Initialize clocks (PLLs).
- Initialize memory controllers.
- Initialize instruction, data, and unified caches.
- Initialize memory management or protection units (MMU).
- Initialize processor exceptions.
- Copy the data sections from a read-only section to the runtime location.
- Set the BSS (`.bss`) section to zero.
- Initialize the C runtime environment.
- Call `boot_card()` to hand over to the high-level initialization.

For examples of start file codes see:

- bsps/arm/shared/start/start.S
- bsps/riscv/shared/start/start.S

5.3 High-Level Initialization via boot_card()

The high-level initialization is carried out by `boot_card()`. For the high-level initialization steps see the `Initialization Manager` chapter in the RTEMS Classic API Guide. There are several system initialization steps which must be implemented by the BSP.

5.3.1 Early BSP Initialization

The BSP may provide a system initialization handler (order `RTEMS_SYSINIT_BSP_EARLY`) to perform an early BSP initialization. This handler is invoked before the memory information and high-level dynamic memory services (workspace and C program heap) are initialized.

5.3.2 Memory Information

The BSP must provide the memory information to the system with an implementation of the `_Memory_Get()` function. The BSP should use the default implementation in bsps/shared/shared/start/bspgetworkarea-default.c. The memory information is used by low-level memory consumers such as the per-CPU data, the workspace, and the C program heap. The BSP may use a system initialization handler (order `RTEMS_SYSINIT_MEMORY`) to set up the infrastructure used by `_Memory_Get()`.

5.3.3 BSP Initialization

The BSP must provide an implementation of the `bsp_start()` function. This function is registered as a system initialization handler (order `RTEMS_SYSINIT_BSP_START`) in the module implementing `boot_card()`. The `bsp_start()` function should perform a general platform initialization. The interrupt controllers are usually initialized here. The C program heap may be used in this handler. It is not allowed to create any operating system objects, e.g. RTEMS semaphores or tasks. The BSP may register additional system initialization handlers in the module implementing `bsp_start()`.

5.4 Error Handling

Errors during system initialization are fatal and end up in a call to `_Terminate()`. See also the `Fatal Error Manager` chapter in the RTEMS Classic API Guide.

The BSP may use BSP-specific fatal error codes, see <bsp/fatal.h>.

The BSP should provide an initial extension which implements a fatal error handler. It should use the default implementation provided by <bsp/default-initial-extension.h> and bspfatal-default.c. If the default implementation is used, the BSP must implement a `bsp_reset()` function which should reset the platform.

CHAPTER

SIX

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:

```
[...]
libbsp_a_SOURCES += ../../shared/dev/serial/console-termios.c
libbsp_a_SOURCES += console/console.c
[...]
```

- 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.

```
#ifndef MY_DRIVER_H
#define MY_DRIVER_H

#include <some-chip/serial.h>

#include <rtems/termiostypes.h>

/* My low-level driver specialization of Termios device context */
typedef struct {
  rtems_termios_device_context base;
  const char *device_name;
  volatile some_chip_registers *regs;
  /* More stuff */
} my_driver_context;

extern const rtems_termios_device_handler my_driver_handler_polled;

extern const rtems_termios_device_handler my_driver_handler_interrupt;

#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.

```
const rtems_termios_device_handler my_driver_handler_polled = {
  .first_open = my_driver_first_open,
  .last_close = my_driver_last_close,
  .poll_read = my_driver_poll_read,
  .write = my_driver_poll_write,
  .set_attributes = my_driver_set_attributes,
  .ioctl = my_driver_ioctl, /* optional, may be NULL */
  .mode = TERMIOS_POLLED
};
```

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

```
static void my_driver_poll_write(
  rtems_termios_device_context *base,
  const char                   *buf,
  size_t                        n
)
{
  my_driver_context *ctx;
  size_t             i;

  ctx = (my_driver_context *) base;

  for ( i = 0 ; i < n ; ++i ) {
    my_driver_write_char( ctx, buf[ i ] );
  }
}
```

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.

```
static int my_driver_poll_read( rtems_termios_device_context *base )
{
  my_driver_context *ctx;

  ctx = (my_driver_context *) base;

  if ( my_driver_can_read_char( ctx ) ) {
    /* Return the character (must be unsigned) */
    return my_driver_read_char( ctx );
  } else {
    /* Return -1 to indicate that no character is available */
    return -1;
  }
}
```

6.5 Interrupt Driven Mode

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

```
const rtems_termios_device_handler my_driver_handler_interrupt = {
  .first_open = my_driver_first_open,
  .last_close = my_driver_last_close,
  .poll_read = NULL,
  .write = my_driver_interrupt_write,
  .set_attributes = my_driver_set_attributes,
  .ioctl = my_driver_ioctl, /* optional, may be NULL */
  .mode = TERMIOS_IRQ_DRIVEN
};
```

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:

```
static void my_driver_interrupt_handler( void *arg )
{
  rtems_termios_tty *tty;
  my_driver_context *ctx;
  char               buf[N];
  size_t             n;

  tty = arg;
  ctx = rtems_termios_get_device_context( tty );

  /*
   * Check if we have received something.  The function reads the
   * received characters from the device and stores them in the
   * buffer.  It returns the number of read characters.
   */
  n = my_driver_read_received_chars( ctx, buf, N );
  if ( n > 0 ) {
    /* Hand the data over to the Termios infrastructure */
    rtems_termios_enqueue_raw_characters( tty, buf, n );
  }

  /*
   * Check if we have something transmitted.  The functions returns
   * the number of transmitted characters since the last write to the
```

(continues on next page)

(continued from previous page)

```
   * device.
   */
  n = my_driver_transmitted_chars( ctx );
  if ( n > 0 ) {
    /*
     * Notify Termios that we have transmitted some characters.  It
     * will call now the interrupt write function if more characters
     * are ready for transmission.
     */
    rtems_termios_dequeue_characters( tty, n );
  }
}
```

The `my_driver_interrupt_write()` handler is responsible for telling the device that the `n` characters at `buf` are to be transmitted. It 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.

```
static void my_driver_interrupt_write(
  rtems_termios_device_context  *base,
  const char                    *buf,
  size_t                         n
)
{
  my_driver_context *ctx;

  ctx = (my_driver_context *) base;

  if ( n > 0 ) {
    /*
     * Tell the device to transmit some characters from buf (less than
     * or equal to n).  When the device is finished it should raise an
     * interrupt.  The interrupt handler will notify Termios that these
     * characters have been transmitted and this may trigger this write
     * function again.  You may have to store the number of outstanding
     * characters in the device data structure.
     */
  } else {
    /*
     * Termios will set n to zero to indicate that the transmitter is
     * now inactive.  The output buffer is empty in this case.  The
     * driver may disable the transmit interrupts now.
     */
  }
}
```

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.

```
static bool my_driver_first_open(
  rtems_termios_tty             *tty,
  rtems_termios_device_context  *base,
  struct termios                *term,
  rtems_libio_open_close_args_t *args
)
{
  my_driver_context *ctx;
  rtems_status_code  sc;
  bool               ok;

  ctx = (my_driver_context *) base;

  /*
   * You may add some initialization code here.
   */

  /*
   * Sets the initial baud rate.  This should be set to the value of
   * the boot loader.  This function accepts only exact Termios baud
   * values.
   */
  sc = rtems_termios_set_initial_baud( tty, MY_DRIVER_BAUD_RATE );
  if ( sc != RTEMS_SUCCESSFUL ) {
    /* Not a valid Termios baud */
  }

  /*
   * Alternatively you can set the best baud.
   */
  rtems_termios_set_best_baud( term, MY_DRIVER_BAUD_RATE );

  /*
   * To propagate the initial Termios attributes to the device use
   * this.
  */
  ok = my_driver_set_attributes( base, term );
  if ( !ok ) {
    /* This is bad */
  }

  /*
   * Return true to indicate a successful set attributes, and false
   * otherwise.
   */
```

(continues on next page)

(continued from previous page)

```
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.

```
static void my_driver_last_close(
  rtems_termios_tty             *tty,
  rtems_termios_device_context  *base,
  rtems_libio_open_close_args_t *args
)
{
  my_driver_context *ctx;

  ctx = (my_driver_context *) base;

  /*
   * The driver may do some cleanup here.
   */
}
```

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.

```
static bool my_driver_set_attributes(
  rtems_termios_device_context *base,
  const struct termios         *term
)
{
  my_driver_context *ctx;

  ctx = (my_driver_context *) base;

  /*
   * Inspect the termios data structure and configure the device
   * appropriately.  The driver should only be concerned with the
   * parts of the structure that specify hardware setting for the
   * communications channel such as baud, character size, etc.
   */

  /*
   * Return true to indicate a successful set attributes, and false
   * otherwise.
   */
  return true;
}
```

6.9 IO Control

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

```
static int my_driver_ioctl(
  rtems_termios_device_context *base,
  ioctl_command_t               request,
  void                         *buffer
)
{
  my_driver_context *ctx;

  ctx = (my_driver_context *) base;

  switch ( request ) {
    case MY_DRIVER_DO_XYZ:
      my_driver_do_xyz(ctx, buffer);
      break;
    default:
      rtems_set_errno_and_return_minus_one( EINVAL );
  }

  return 0;
}
```

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:

```
#include <my-driver.h>

#include <rtems/console.h>

#include <bsp.h>
#include <bsp/fatal.h>

static my_driver_context driver_context_table[] = {
  { /* Some values for device 0 */ },
  { /* Some values for device 1 */ }
};

rtems_device_driver console_initialize(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                       *arg
)
{
  const rtems_termios_device_handler *handler;
  rtems_status_code                   sc;
  size_t                              i;

  #ifdef SOME_BSP_USE_INTERRUPTS
    handler = &my_driver_handler_interrupt;
  #else
    handler = &my_driver_handler_polled;
  #endif

  /*
   * Initialize the Termios infrastructure.  If Termios has already
   * been initialized by another device driver, then this call will
   * have no effect.
   */
  rtems_termios_initialize();

  /* Initialize each device */
  for ( i = 0; i < RTEMS_ARRAY_SIZE( driver_context_table ) ; ++i ) {
    my_driver_context *ctx;

    ctx = &driver_context_table[ i ];

    /*
     * Install this device in the file system and Termios.  In order
     * to use the console (i.e. being able to do printf, scanf etc.
     * on stdin, stdout and stderr), one device must be registered as
```

(continues on next page)

(continued from previous page)

```
     * "/dev/console" (CONSOLE_DEVICE_NAME).
     */
    sc = rtems_termios_device_install( ctx->device_name, handler, NULL, ctx );
    if ( sc != RTEMS_SUCCESSFUL ) {
      bsp_fatal( SOME_BSP_FATAL_CONSOLE_DEVICE_INSTALL );
    }
  }

  return RTEMS_SUCCESSFUL;
}
```

CHAPTER

SEVEN

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.

```
/*
 * A section with functions, defines and macros to provide hardware-specific
 * functions for the Clock Driver Shell.
 */

#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

The clock driver is initialized by the `_Clock_Initialize()` system initialization handler if requested by the application configuration option `CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER`. The clock driver does not use the legacy IO driver framework.

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.

```
#include <rtems/timecounter.h>

static struct timecounter some_tc;

static uint32_t some_tc_get_timecount( struct timecounter *tc )
{
  some.free_running_counter;
}

static void some_support_initialize_hardware( void )
{
  uint64_t us_per_tick;
  uint32_t counter_frequency_in_hz;
  uint32_t counter_ticks_per_clock_tick;

  us_per_tick = rtems_configuration_get_microseconds_per_tick();
  counter_frequency_in_hz = some_tc_get_frequency();

  /*
   * The multiplication must be done in 64-bit arithmetic to avoid an integer
   * overflow on targets with a high enough counter frequency.
   */
  counter_ticks_per_clock_tick =
    (uint32_t) ( counter_frequency_in_hz * us_per_tick ) / 1000000;

  /*
   * Initialize hardware and set up a periodic interrupt for the configuration
   * based counter ticks per clock tick.
   */

  some_tc.tc_get_timecount = some_tc_get_timecount;
```

(continues on next page)

(continued from previous page)

```
  some_tc.tc_counter_mask = 0xffffffff;
  some_tc.tc_frequency = frequency;
  some_tc.tc_quality = RTEMS_TIMECOUNTER_QUALITY_CLOCK_DRIVER;
  rtems_timecounter_install( &some_tc );
}

#define Clock_driver_support_initialize_hardware() \
  some_support_initialize_hardware()

#include "../../../shared/dev/clock/clockimpl.h"
```

7.2.2 Simple Timecounter Variant

For an example see the ERC32 clock driver. The argument parameter of `Clock_driver_timecounter_tick( arg )` is the argument used to install the clock interrupt handler. Device drivers may use this argument to access their control state.

```
#include <rtems/timecounter.h>

static rtems_timecounter_simple some_tc;

static uint32_t some_tc_get( rtems_timecounter_simple *tc )
{
  return some.counter;
}

static bool some_tc_is_pending( rtems_timecounter_simple *tc )
{
  return some.is_pending;
}

static uint32_t some_tc_get_timecount( struct timecounter *tc )
{
  return rtems_timecounter_simple_downcounter_get(
    tc,
    some_tc_get,
    some_tc_is_pending
  );
}

static void some_tc_tick( rtems_timecounter_simple *tc )
{
  rtems_timecounter_simple_downcounter_tick( tc, some_tc_get );
}

static void some_support_initialize_hardware( void )
{
  uint64_t us_per_tick;
  uint32_t counter_frequency_in_hz;
```

(continues on next page)

(continued from previous page)

```
  uint32_t counter_ticks_per_clock_tick;

  us_per_tick = rtems_configuration_get_microseconds_per_tick();
  counter_frequency_in_hz = some_tc_get_frequency();
  counter_ticks_per_clock_tick =
    (uint32_t) ( counter_frequency_in_hz * us_per_tick ) / 1000000;

  /* Initialize hardware */

  rtems_timecounter_simple_install(
    &some_tc,
    counter_frequency_in_hz,
    counter_ticks_per_clock_tick,
    some_tc_get_timecount
  );
}

#define Clock_driver_support_initialize_hardware() \
  some_support_initialize_hardware()
#define Clock_driver_timecounter_tick( arg ) \
  some_tc_tick( arg )

#include "../../../shared/dev/clock/clockimpl.h"
```

7.2.3 Clock Tick Only Variant

For an example see the Motrola 68360 clock driver.

```
static void some_support_initialize_hardware( void )
{
  /* Initialize hardware */
}

#define Clock_driver_support_initialize_hardware() \
  some_support_initialize_hardware()

/* Indicate that this clock driver lacks a proper timecounter in hardware */

#define CLOCK_DRIVER_USE_DUMMY_TIMECOUNTER

#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( isr )`. The clock tick interrupt service routine is passed as the one and only parameter to this macro. The default implementation will do nothing. The argument parameter (in the code below `&some_instance`) for the installed interrupt handler is available in the `Clock_driver_support_at_tick( arg )` and `Clock_driver_support_initialize_hardware( arg )` customization macros.

```
#include <bsp/irq.h>
#include <bsp/fatal.h>

static some_control some_instance;

static void some_support_install_isr( rtems_interrupt_handler isr )
{
  rtems_status_code sc;
  sc = rtems_interrupt_handler_install(
    SOME_IRQ,
    "Clock",
    RTEMS_INTERRUPT_UNIQUE,
    isr,
    &some_instance
  );
  if ( sc != RTEMS_SUCCESSFUL ) {
    bsp_fatal( SOME_FATAL_IRQ_INSTALL );
  }
}

#define Clock_driver_support_install_isr( isr ) \
  some_support_install_isr( isr )

#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( arg )`. The `arg` is the argument used to install the clock interrupt handler. Device drivers may use this argument to access their control state.

```
static void some_support_at_tick( some_control *arg )
{
  /* Clear interrupt */
}

#define Clock_driver_support_at_tick( arg ) \
  some_support_at_tick( arg )

#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.

```
/* Enable multiple clock driver ticks per clock tick */
#define CLOCK_DRIVER_ISRS_PER_TICK 1

/* Specifiy the clock driver ticks per clock tick value */
#define CLOCK_DRIVER_ISRS_PER_TICK_VALUE 123

#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:

```
volatile uint32_t Clock_driver_ticks;
```

CHAPTER

EIGHT

TARGET HASH

Each BSP must provide an implementation of the `rtems_get_target_hash()` directive. The default implementation is based on the CPU counter frequency. A BSP-specific implementation may be provided which covers also for example the device tree, settings of the memory controller, processor and bus frequencies, a serial number of a chip, etc. For a BSP-specific implementation start with the default implementation and add more values to the target hash using the functions `_Hash_Add_data()` and `_Hash_Add_string()`. The target hash can be used to distinguish test suite results obtained from different target systems.

CHAPTER

NINE

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.

CHAPTER

TEN

CAN DRIVER

RTEMS provides fully featured CAN/CAN FD stack. The API to the driver is provided in the form of the POSIX character driver with each CAN controller (chip) registered as node into "/dev" namespace by name (i.e. "can0", "can1", ...). The stack utilizes FIFO queues (also called edges) organized into oriented edges between controller side and application side. Edges, responsible for message transfers from an application to controller and vice versa, can have different priorities and function as priority classes. The naming of the edges and functions using these edges is taken from the FIFO's point of view. Therefore, outgoing edge is the edge that passes the frames to the FIFO and incoming edge retrieves the frames from FIFO. Note that these can be used both from application and device driver point of view as both sides may use the same FIFO related functions.

Controller takes frames from FIFOs according their priority class and transmits them to the network. Successfully sent frames are echoed through queues back to open file instances except the sending one (that filtering is configurable). Received frames a filtered to all queues to applications ends of the queues which filter matches the CAN identifier and frame type.

The stack provides run time configuration options to create new queues with desired priority, direction and filter, making it suitable for various applications requirements. There is also a possibility to configure controller's characteristics (bit rate, mode, chip specific ioctl calls).

The device can be opened in both blocking and nonblocking mode and one device can be opened from multiple applications. Read and write operations wait on binary semaphore indefinitely if blocking mode is used and frame can not be passed to the framework immediately (full FIFO queue for example) or there is no received message to process.

10.1 Include Headers

To use the infrastructure, an application, BSP initiator or controller device driver has to include few header files that provides related structures, definitions and function declarations.

10.1.1 Application

```
#include <dev/can/can.h>
```

The only required include for standard application using the structure through POSIX character device API is <dev/can/can.h>. This provides all defines, ioctl calls, and structures required to operate with the infrastructure from application layer. These are in detail described in RTEMS CAN API section.

The mentioned header also includes several other headers. These are still strictly API interface, but separated to multiple headers for clearer code organization. This contains the header defining CAN frame structure, header introducing CAN bit timing structures, filters for FIFO queues or CAN RX/TX statistics tracking. The application does not have to bother include all of these headers as they are already included through <dev/can/can.h>.

10.1.2 BSP Registration

```
#include <dev/can/can-bus.h>
#include <dev/can/controller-dependent.h>
```

It is expected the controller will be initialized and registered from board support package during board initialization, but this includes will work anywhere else (even from an application if required by the user). The header <dev/can/can-bus.h> provides definition of `rtems_can_bus` structure and declarations of `rtems_can_bus_register()` that registers the controller to standard /dev namespace. The usage of these functions is described in Registering CAN Bus section.

The source code will most likely have to include a controller dependent header that declares the initialization function for the specific controller.

10.1.3 Device Driver

```
#include <dev/can/can.h>
#include <dev/can/can-devcommon.h>
```

The device driver (i.e. the file that implements the controller) has to include <dev/can/can.h> because of CAN frame definition and other defines/structures with which it has to interact. Functions and structures providing the interface with the infrastructure (obtaining frames from FIFO, pushing frames to FIFO, slot abort and so on) are included through <dev/can/can-devcommon.h> header. The usage of these functions is described in Driver Interface section.

It is expected <dev/can/can-devcommon.h> will be used primarily from a controller device driver (i.e. from RTEMS kernel code), but there is a possibility to include this header even from an application if the user has a special needs surpassing the API.

10.2 RTEMS CAN API

```
#include <dev/can/can.h>
```

It is necessary to include the header above to operate with CAN infrastructure through the described application interface.

Application Interface is provided with standard POSIX calls `open()`, `write()`, `read()`. `close()` and `ioctl()`. Functions `write()` and `read()` are used with `can_frame` structure representing one CAN frame (message).

10.2.1 Opening Device and Configuration

Device is registered as a node into "/dev" namespace and can be opened with POSIX `open()` call. A single chip can be opened multiple times with each instance creating its own queues between the controller and application. The frames received by the controller are filtered to all connected queues if they match the filter set by the user. Therefore, the applications do not race for the received frames.

CALLING SEQUENCE:

```
int open( const char* pathname, int flags );
```

DESCRIPTION:

Opens CAN device at path `pathname` with mode defined in `flags` argument. Modes are defined according to POSIX standard.

Both infrastructure resources and controller itself can be configured once the device is opened. The configuration is provided via ioctl calls. Some of these configuration are available only if the controller is stopped.

10.2.1.1 Managing Queues

One RX and one TX queue is created by default during `open()` operation. These queues have the lowest priority and default filter and size settings. If needed, more queues can be created with `RTEMS_CAN_CREATE_QUEUE` call.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_CREATE_QUEUE, &queue );
```

DESCRIPTION:

Creates new queue with characteristics defined in "queue" field provided as `rtems_can_queue_param` structure.

```
struct rtems_can_queue_param {
  uint8_t direction;
  uint8_t priority;
  uint8_t dlen_max;
  uint8_t buffer_size;
  struct rtems_can_filter filter;
};
```

Field `dlen_max` set maximum CAN frame data length that can be sent through the queue. This allows the user to limit the size of allocated memory if only shorter frames are sent to the network. If set to zero, default value (64 bytes for CAN FD capable controllers, 8 bytes otherwise) is used. It is not possible to set an invalid value (less than zero or greater than 64). Field `buffer_size` configures number of slots (frames) that fits in the FIFO.

```
struct rtems_can_filter {
  uint32_t id;
  uint32_t id_mask;
  uint32_t flags;
  uint32_t flags_mask;
};
```

Structure `rtems_can_filter` is used to set queue's filter. It holds the CAN frame identifier and flag filters, ensuring only frames matching this filter are passed to the queue's ends. Fields `id` and `flags` hold identifier bits and frames' flags, respectively, required to be present in a CAN frame to assign it to the corresponding FIFO queue. In other words, it specifies that only specific identifiers and/or flags shall be assigned to the queue. Members with `_mask` postfix are used to mask out identifiers or flags that are forbidden for a given FIFO queue. Refer to CAN frame description for possible flags.

The filter can be used to create queues that process only defined subset of CAN frames. This may be used to create priority classes based on frame identifier range or special queues for certain type of frames (echo, error etc.). Setting all fields of `rtems_can_filter` to zero means all frames are passed through the queue.

Default queues created during `open()` operation allows all identifiers and filters out error and echo frames.

Queues can be removed (discarded) with RTEMS_CAN_DISCARD_QUEUES command. It is not possible to discard one specific queue, just all RX or/and all TX queues for given opened instance (file descriptor) at once. Direction can be defined by RTEMS_CAN_QUEUE_TX and RTEMS_CAN_QUEUE_RX defines. Terms TX and RX are used from the application's point of view: TX meaning queues transferring messages from an application to a controller, RX from a controller to an application.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_DISCARD_QUEUES, type );
```

DESCRIPTION:

Discard TX and/or RX queues based on integer "type" argument. Defines RTEMS_CAN_QUEUE_TX and RTEMS_CAN_QUEUE_RX can be used to specify queues for deletion.

Queues can also be flushed with RTEMS_CAN_FLUSH_QUEUES command.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_FLUSH_QUEUES, type );
```

DESCRIPTION:

Flushes TX and/or RX queues based on integer "type" argument. Defines RTEMS_CAN_QUEUE_TX and RTEMS_CAN_QUEUE_RX can be used to specify queues for deletion. The operation flushes all RX or/and all TX queues even if multiple queues are used.

10.2.1.2 Setting Bit Timing

There are two ways to set CAN bit timing. Either the user can pass desired bit rate value and let the infrastructure calculate bit timing, or precomputed bit timing values can be passed directly. ioctl call `RTEMS_CAN_SET_BITRATE` is used for this purpose.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_SET_BITRATE, &set_bittiming );
```

DESCRIPTION:

Sets bit timing based on "set_bittiming" parameter passed as a pointer to `can_set_bittiming` structure.

```
struct rtems_can_bittiming {
  uint32_t bitrate;
  uint32_t sample_point;
  uint32_t tq;
  uint32_t prop_seg;
  uint32_t phase_seg1;
  uint32_t phase_seg2;
  uint32_t sjw;
  uint32_t brp;
};

struct rtems_can_set_bittiming {
  uint16_t type;
  uint16_t from;
  struct rtems_can_bittiming bittiming;
};
```

Field `type` determines the bit timing type to be set (`CAN_BITTIME_TYPE_NOMINAL` or `CAN_BITTIME_TYPE_DATA`), field `from` determines the source of the bit timing values (`CAN_BITTIME_FROM_BITRATE` or `CAN_BITTIME_FROM_PRECOMPUTED`).

Actual bit timing values and controller's bit timing constants can be retrieved with RTEMS_CAN_GET_BITTIMING.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_GET_BITTIMING, &get_bittiming );
```

DESCRIPTION:

Retrieves currently set bit timing values and controller's bit timing constants.

```
struct rtems_can_bittiming_const {
  char name[32];
  uint32_t tseg1_min;
  uint32_t tseg1_max;
  uint32_t tseg2_min;
  uint32_t tseg2_max;
  uint32_t sjw_max;
  uint32_t brp_min;
  uint32_t brp_max;
  uint32_t brp_inc;
};

struct rtems_can_get_bittiming {
  uint16_t type;
  struct rtems_can_bittiming bittiming;
  struct rtems_can_bittiming_const bittiming_const;
};
```

Field type determines bit timing to be set (CAN_BITTIME_TYPE_NOMINAL or CAN_BITTIME_TYPE_DATA).

10.2.1.3 Setting Mode

Different modes of the chip can be enabled/disabled. ioctl call `RTEMS_CAN_CHIP_SET_MODE` is used to set the mode as a 32-bit large unsigned integer mask.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_SET_MODE, mode );
```

DESCRIPTION:

Argument `mode` is a 32-bit large unsigned integer with modes to be set. Available modes are

- `CAN_CTRLMODE_LOOPBACK`,
- `CAN_CTRLMODE_LISTENONLY`,
- `CAN_CTRLMODE_3_SAMPLES`,
- `CAN_CTRLMODE_ONE_SHOT`,
- `CAN_CTRLMODE_BERR_REPORTING`,
- `CAN_CTRLMODE_FD`,
- `CAN_CTRLMODE_PRESUME_ACK`,
- `CAN_CTRLMODE_FD_NON_ISO`,
- `CAN_CTRLMODE_CC_LEN8_DLC`,
- `CAN_CTRLMODE_TDC_AUTO`, and
- `CAN_CTRLMODE_TDC_MANUAL`.

The modes are implemented to be compatible with GNU/Linux's SocketCAN stack and possibly with other operating systems as well. It is possible to set multiple modes during one ioctl call. The controller should be implemented in such a way that not setting particular mode in this ioctl call disables this mode. Therefore, the same ioctl call may be used for both enable and disable operation.

Every controller should know its supported mode. An attempt to set a mode not supported by the controller leads to the ioctl call returning an error. It is also possible to change controller's modes only if the controller is stopped, otherwise error is returned.

10.2.1.4 Starting Chip

Opening the device does not automatically start the chip, this operation has to be handled by specific ioctl call `RTEMS_CAN_CHIP_START`.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_START );
```

DESCRIPTION:

Starts the chip (enables write/read). Repeated calls on already started chip do not have any effect.

It is also possible to start the chip with a function call `rtems_can_chip_start()`. This way the controller may be started even before the first open, for example from board support package right after its initialization.

CALLING SEQUENCE:

```
#include <dev/can/can-devcommon.h>

int rtems_can_chip_start( struct rtems_can_chip *chip )
```

DESCRIPTION:

Starts the chip (enables write/read). Repeated calls on already started chip do not have any effect.

10.2.1.5 Stopping Chip

Similarly to the chip start operation, chip stop is performed with `RTEMS_CAN_CHIP_STOP` ioctl call.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_STOP, &timeout );
```

DESCRIPTION:

Stops the chip (disables write/read). Repeated calls on already stopped chip do not have any effect. The call is nonblocking if `timeout` parameter is set to `NULL`, otherwise the calling thread is blocked for a timeout specified as a relative timeout with `timespec` structure.

This gives the controller the time to abort the frames already present in its buffers and to return these frames and the frames from FIFO queues back to the applications that opened it as TX error frames. This way the applications can get the information their frames were not transmitted because the controller was stopped. If timed out before all frames are returned as error frames, the queues are flushed and the frames are lost. In any way, it is ensured the queues are empty when/if the chip is started again. Therefore, the minimal implementation should always at least flush the FIFO queues from the application to the controller.

NOTES.

It is important to check the number of users (applications) using the chip before turning it off as there can be more than one user per chip. The infrastructure allows turning off the controller even if there are other users using it. Read and write calls from other applications return error in that case.

10.2.1.6 Controller Related Information

An ioctl call `RTEMS_CAN_CHIP_GET_INFO` can be used to obtain some information about the device driver (controller).

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_GET_INFO, info_type );
```

DESCRIPTION:

Obtains information about the chip. The information to be obtained is defined as integer argument info_type. Following parameters can be obtained.

```
RTEMS_CAN_CHIP_BITRATE,
RTEMS_CAN_CHIP_DBITRATE,
RTEMS_CAN_CHIP_NUSERS,
RTEMS_CAN_CHIP_FLAGS,
RTEMS_CAN_CHIP_MODE, and
RTEMS_CAN_CHIP_MODE_SUPPORTED.
```

The defines listed above may be used to obtain information from the controller. It is possible to obtain only one information for one ioctl call. `RTEMS_CAN_CHIP_MODE` and `RTEMS_CAN_CHIP_MODE_SUPPORTED` are used to obtain currently set controller modes and all modes supported by the controller, respectively. Stop command described previously may benefit from `RTEMS_CAN_CHIP_NUSERS` providing number of users currently using the controller. Controller's flags obtained by `RTEMS_CAN_CHIP_FLAGS` provide various information including FD capability of the controller, status of the chip (configured, running), and so on. Refer to source code documentation for possible chip's status defines.

10.2.1.7 Controller Statistics

The controller can keep track of its statistics as number of received/transmitted frames, number of received/transmitted bytes, number of errors and so on. These statistics are represented in `can_stats` structure and can be obtained with `RTEMS_CAN_CHIP_STATISTICS` ioctl call.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_CHIP_STATISTICS, &statistics );
```

DESCRIPTION:

Obtains controller's statistics provided in with argument `statistics` as a pointer to the `rtems_can_stats` structure.

```
enum can_state {
  CAN_STATE_ERROR_ACTIVE = 0,
  CAN_STATE_ERROR_WARNING,
  CAN_STATE_ERROR_PASSIVE,
  CAN_STATE_BUS_OFF,
  CAN_STATE_STOPPED,
  CAN_STATE_SLEEPING,
  CAN_STATE_STOPPING,
  CAN_STATE_MAX
};

struct rtems_can_stats {
  unsigned long tx_done;
  unsigned long rx_done;
  unsigned long tx_bytes;
  unsigned long rx_bytes;
  unsigned long tx_error;
```

(continues on next page)

(continued from previous page)

```
  unsigned long rx_error;
  unsigned long rx_overflows;
  int chip_state;
};
```

10.2.2 CAN Frame Representation

The represenation of one CAN frame is defined statically with a header separated in its own structure `can_frame_header`. It has an 8 bytes long timestamp, 4 bytes long CAN identifier, 2 bytes long flag field and 2 bytes long field with information about data length. Data field itself is a 64 byte long array with byte access.

Only first 11 bits of the identifier are valid (29 if extended identifier format is used). Having any of the upper three bits set to one indicates an invalid CAN frame format. If these are set, the user should check frame's flags to get information if this is not an error frame generated by the controller.

```
struct can_frame_header {
  uint64_t timestamp;
  uint32_t can_id;
  uint16_t flags;
  uint16_t len;
};

struct can_frame {
    struct can_frame_header header;
    uint8_t data[CAN_FRAME_MAX_DLEN];
};
```

Flags are used to distinguish frame formats (extended identifier, CAN FD format, remote request and so on). Following defines can be used.

- `CAN_FRAME_IDE`,
- `CAN_FRAME_RTR`,
- `CAN_FRAME_ECHO`,
- `CAN_FRAME_LOCAL`,
- `CAN_FRAME_TXERR`,
- `CAN_FRAME_ERR`.
- `CAN_FRAME_FIFO_OVERFLOW`.
- `CAN_FRAME_FDF`,
- `CAN_FRAME_BRS`, and
- `CAN_FRAME_ESI`.

Extended frame format (`CAN_FRAME_IDE`) is forced automatically if identifier exceeds 11 bits. Flags `CAN_FRAME_FDF` and `CAN_FRAME_BRS` (if bit rate switch between arbitration and data phase is intended) should be set for CAN FD frame transmission.

Some of these flags are automatically masked for the first queues created during the instance open operation. These include `CAN_FRAME_ECHO` and both error flags `CAN_FRAME_TXERR` and `CAN_FRAME_ERR`. Flag `CAN_FRAME_FIFO_OVERFLOW` is set automatically by the stack for RX frames and can not be filtered out. It indicates FIFO overflow occurred, and some frames on the receiver side have been discarded. More specifically, it informs the user there are discarded frames between the frame with `CAN_FRAME_FIFO_OVERFLOW` flag and a previous correctly received frame.

10.2.3 Frame Transmission

Frame is transmitted to the CAN framework by calling `write()` function.

CALLING SEQUENCE:

```
ssize_t write( int fd, struct can_frame *frame, size_t count );
```

DESCRIPTION:

Passes CAN frame represented by `can_frame` structure to the network. Return values comply with POSIX standard. Write size `count` can be calculated with `can_framesize()` function. It is possible to write just one frame with a single call. Passing incorrect frame length (less than the header size or larger than maximum CAN frame size) results in write error.

User can check whether the messages were transferred from RTEMS framework to the physical network by calling ioctl `RTEMS_CAN_WAIT_TX_DONE`.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_WAIT_TX_DONE, &timeout );
```

DESCRIPTION:

Waits with timeout until all frames are transferred to the network. The timeout is defined as a pointer to `timespec` structure. The timeout is specified as a relative timeout. Returns 0 on success and `ETIME` on timeout.

This call applies to TX FIFO queues at once for a given file descriptor.

Polling in nonblocking mode can be done with `RTEMS_CAN_POLL_TX_READY` ioctl call.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_POLL_TX_READY, &timeout );
```

DESCRIPTION:

Implements polling function on outgoing edges. Timeout is defined with `timespec` structure. The timeout is specified as a relative timeout. It waits until there is an available frame in any of the input FIFOs or until timeout.

10.2.4 Frame Reception

Frame is received from the CAN framework by calling `read()` function.

CALLING SEQUENCE:

```
ssize_t read( int fd, struct can_frame *frame, size_t count );
```

DESCRIPTION:

Reads CAN frame represented by `can_frame` from the network. Return values comply with POSIX standard. The call returns error if read size specified by `count` is less than the length of the frame header. It is possible to read only a single frame with one read call.

Polling in nonblocking mode can be done with `RTEMS_CAN_POLL_RX_AVAIL` ioctl call.

CALLING SEQUENCE:

```
ssize_t ioctl( fd, RTEMS_CAN_POLL_RX_AVAIL, &timeout );
```

DESCRIPTION:

Implements polling function on incoming edges. Timeout is defined with `timespec` structure. It waits until there is an available frame in any of the input FIFOs or until timeout.

10.2.5 Error Reporting

There are two flags for error reporting: CAN_FRAME_TXERR and CAN_FRAME_ERR. First flag is used to report frame transmission error. In this case, the controller should send the frame that caused the error back to its opened instance with added CAN_FRAME_TXERR flag. The message should not be changed in any other way.

It is possible to receive various CAN bus related error through error messages sent from the controller to the application. Flag CAN_FRAME_ERR is used for that. If this flag is set, received frame has a special format and shall be looked up as an error frame.

For generated error frame, identifier field is used to store the information about error type. Following types are supported.

- CAN_ERR_ID_TXTIMEOUT,
- CAN_ERR_ID_LOSTARB,
- CAN_ERR_ID_CRTL,
- CAN_ERR_ID_PROT,
- CAN_ERR_ID_TRX,
- CAN_ERR_ID_ACK.
- CAN_ERR_ID_BUSOFF.
- CAN_ERR_ID_BUSERROR,
- CAN_ERR_ID_RESTARTED,
- CAN_ERR_ID_CNT, and
- CAN_ERR_ID_INTERNAL.

Additionally, 31st bit of CAN identifier is set to logical one. This is another check that indicates it is not a regular frame but error one. Having error types located in CAN frame identifier brings the possibility to create new RX queues with identifier mask set in such way that only some of these errors are propagated to the application.

The additional information providing deeper description of raised error are also available in data fields for some error types. Only a standard frame with 8 bytes long data field is used.

The first byte (8 bits) of the data field keeps the detailed information regarding lost arbitration error (CAN_ERR_ID_LOSTARB). This basically just informs in that bit the arbitration was lost. Another field stores controller related problems (CAN_ERR_ID_CRTL). This includes RX or TX overflows and the controller changing its error state (error active, warning, passive).

Protocol related violations (CAN_ERR_ID_PROT) are stored in the third and the fourth data field. The first informs what kind of violation is present. This may be incorrect bit stuffing, controller incapability to generate dominant or recessive bit or bus overload for example. The latter field provides a location of this violation.

Transceiver status (CAN_ERR_ID_TRX) is located in the fifth data field. This is used to report hardware layer issues as missing wire or wire being short-circuited to ground or supply voltage. The sixth data field is reserved and not used. The infrastructure also reports number of the current values of TX and RX error counter (CAN_ERR_ID_CNT). These data are passed through seventh and eight data fields for transmission and reception, respectively.

10.3 Driver Interface

```
#include <dev/can/can.h>
#include <dev/can/can-devcommon.h>
```

The includes listed above are required to use the functions described in this section.

The infrastructure provides several functions ensuring an interface between FIFO queue side of the stack and the controller's specific implementation. Controller's driver should use these functions to access the FIFOs.

The driver is forbidden to access the CAN framework from an interrupt handler. Instead, it should utilize a worker thread waiting on a semaphore and triggered when there are interrupts to be processed (received frame, send done, error and so on) or where there are frames to be transmitted (this information is triggered from the CAN framework, see the next section for the detailed description).

For examples of CAN controller's driver see:

- CAN virtual driver
- CTU CAN FD driver

10.3.1 Chip Initialization

The chip initialization function should allocate a `rtems_can_chip` structure. This structure holds the controller's ends of FIFO queues as well as the chip's private structure. This structure is chip specific, and it is used to store the chip specific data. The controller has to allocate and initialize its side of the FIFO queues. The structure to be allocated is `rtems_can_queue_ends_dev` located on a `qends_dev` field of `rtems_can_chip` structure and initialization is done with a `rtems_can_queue_ends_init_chip()` function call.

CALLING SEQUENCE:

```
int rtems_can_queue_ends_init_chip (
  struct rtems_can_chip *chip,
  const char *name
);
```

DESCRIPTION:

Initializes controller's side of ends defined in `chip` structure and connects them to the FIFO queues. It also creates a worker binary semaphore `worker_sem` named `name` and used by the framework to inform the controller's side about new frame to be transmitted. The controller may also use this semaphore to trigger its worker thread in case of an interrupt (received frame for example).

Driver should also register several functions used by ioctl calls (start chip, stop chip, set bit rate for example). These functions are assigned through `rtems_can_chip_ops` structure.

10.3.2 Frame Transmission

The controller retrieves the slots (frames) from FIFO queues (edges) and sends them to the network. The naming of the functions utilizes `_outslot` suffix, because the controller's side takes the frames from the FIFO's outputs.

The framework informs the driver about the new message to be transmitted by posting a `worker_sem` semaphore. The driver then may call the function `rtems_can_queue_test_outslot()` to obtain the oldest (least recently added to the FIFO from an application) slot from the highest priority queue. Note that this uses the priority of the queues as described in the previous sections, not the priority of a CAN frame (i.e. the frame's identifier value). However, these queues may have the filter set up in such a way to accept only a certain identifier range.

CALLING SEQUENCE:

```
int rtems_can_queue_test_outslot(
  struct rtems_can_queue_ends **qends,
  struct rtems_can_queue_edge **qedgep,
  struct rtems_can_queue_slot **slotp
);
```

DESCRIPTION:

Tests and retrieves the oldest ready slot from the highest priority active queue (priority class).

The slot can subsequently be put into the controller's hardware buffer and sent to the network. Function `rtems_can_queue_test_outslot()` does not free the slot's space in the FIFO queue. The controller should inform the framework to free the space by calling the `rtems_can_queue_free_outslot()` function once the frame is successfully transmitted or the transmission results in an error.

CALLING SEQUENCE:

```
int rtems_can_queue_free_outslot(
  struct rtems_can_queue_ends *qends,
  struct rtems_can_queue_edge *qedge,
  struct rtems_can_queue_slot *slot
);
```

DESCRIPTION:

Releases processed slot previously acquired by a function `rtems_can_queue_test_outslot()` call.

The framework also provides a unique feature to push the frames back to the correct FIFO and schedule the slot later processing. This is useful in case the frame put into a hardware buffer is aborted. The abort might be used when some later scheduled low-priority frame occupies the hardware TX buffer, which is urgently demanded for a higher priority pending message from other FIFO for example.

CALLING SEQUENCE:

```
int rtems_can_queue_push_back_outslot(
  struct rtems_can_queue_ends *qends,
  struct rtems_can_queue_edge *qedge,
  struct rtems_can_queue_slot *slot
);
```

DESCRIPTION:

Reschedules slot previously acquired with a `rtems_can_queue_test_outslot()` function call for a second time processing.

Previously described function `rtems_can_queue_test_outslot()` already takes the slot from the FIFO when called. This is not convenient in case there is no free space in the controller's hardware TX buffers. We would rather just check whether there is some pending message from higher priority class compared to the priority classes presented in buffers. This can be done with `rtems_can_queue_pending_outslot_prio()`.

CALLING SEQUENCE:

```
int rtems_can_queue_pending_outslot_prio(
  struct rtems_can_queue_ends *qends,
  int prio_min
);
```

DESCRIPTION:

Tests whether there is ready slot for given ends and minimum priority to be considered. Negative value informs this is not a case, positive value informs about the available slot priority class.

10.3.3 Frame Reception

Upon successful frame reception, the controller has to pass the frame to FIFO edges. This is done with `rtems_can_queue_filter_frame_to_edges()`

CALLING SEQUENCE:

```
int rtems_can_queue_filter_frame_to_edges(
  struct rtems_can_queue_ends *qends,
  struct rtems_can_queue_edge *src_edge,
  struct can_frame *frame,
  unsigned int flags2add
);
```

DESCRIPTION:

Sends a message (frame) defined with `frame` argument to all outgoing edges connected to the given ends (`qends`) with additional flags defined by `flags2add` argument. Argument `src_edge` defines an optional source edge for echo detection. This is used to correctly filter echo frames.

The controller also should use this function to send an echo frame with additional flag `CAN_FRAME_LOCAL` upon a successful frame transmission or `CAN_FRAME_TXERR` frame when transmission error occurs.

10.3.4 Worker Thread Example

This is a largely simplified example of a possible worker thread for CAN device driver. The thread is used both for interrupt processing (semaphore trigger by the interrupt handler) and frame sender. The interrupt handler should disable interrupts before posting the semaphore as all interrupts are handled in the worker. The worker should enable them before waiting on the semaphore.

```
static rtems_task worker( rtems_task_argument arg )
{
  struct rtems_can_chip *chip = (struct rtems_can_chip *)arg;
  struct rtems_can_queue_ends *qends = &chip->qends_dev->base;

  while ( 1 ) {
    /* This should be another while loop that handles all interrupts */
    process_interrupts( chip );

    if ( buffer_has_free_space() ) {
      ret = rtems_can_queue_test_outslot( qends, &qedge, &slot );
      if ( ret >= 0 ) {
        /* Send frame located in slot->frame */
      }
    } else {
      /* Check for the possible higher priority frames with
       * rtems_can_queue_pending_outslot_prio() call.
       * This has sense only if the controller supports frame
       * abort from HW buffers.
       */
    }

    /* Enable interrupts and wait on semaphore */
    interrupts_enable( chip );
    rtems_binary_semaphore_wait( &chip->qends_dev->worker_sem );
  }
}
```

10.4 Registering CAN Bus

```
#include <dev/can/can-bus.h>
#include <dev/can/controller-dependent.h>
```

The headers listed above have to be used ti provide the described functions. Header <dev/can/controller-dependent.h> represents the controller's specific header. This header usually declares the controller's initialization function.

Once initialized (rtems_can_chip structure allocated and obtained from a controller specific function), the device can be registered into /dev namespace by rtems_can_bus_register() function.

CALLING SEQUENCE:

```
int rtems_can_bus_register(
  struct rtems_can_bus *bus,
  const char *bus_path
);
```

DESCRIPTION:

Registers CAN devices in structure rtems_can_bus to /dev namespace with path bus_path. The path may follow the standard /dev/canX naming, or it can be a different name selected by the user/BSP. Structure rtems_can_bus represents one CAN device and is defined as:

```
struct rtems_can_bus {
    struct rtems_can_chip *chip;
};
```

The device can be opened by open() function once registered into /dev namespace. It is possible to open one device multiple times.

10.4.1 Example

The entire process of initialization and registration is demonstrated on virtual CAN controller in the code below. Note that the user has to specify the path to which the controller is registered, and this path has to be unique. Chip specific function xxx_initialize() may also have different input parameters for different chips or can even have a different name according to the chip's specific implementation.

```
#include <dev/can/can-bus.h>
#include <dev/can/can-virtual.h>

/* Allocate can_bus structure */
struct rtems_can_bus bus = malloc( sizeof( struct rtems_can_bus ) );

/* Initialize virtual CAN controller */
bus->chip = rtems_virtual_initialize();

/* Register controller as dev/can0, returns 0 on success */
int ret = rtems_can_bus_register( bus, "dev/can0" );
```

CHAPTER

ELEVEN

I2C DRIVER

The Inter-Integrated Circuit (I2C, I^2C, 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

CHAPTER

TWELVE

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

CHAPTER

THIRTEEN

REAL-TIME CLOCK DRIVER

13.1 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 ports of the DMV177's `RTC_Table` configuration table is below:

```
#include <bsp.h>
#include <libchip/rtc.h>
#include <libchip/icm7170.h>

bool dmv177_icm7170_probe(int minor);

rtc_tbl RTC_Table[] = {
  { "/dev/rtc0",                /* sDeviceName */
    RTC_ICM7170,                /* deviceType */
    &icm7170_fns,               /* pDeviceFns */
    dmv177_icm7170_probe,       /* deviceProbe */
    (void *) ICM7170_AT_1_MHZ,  /* pDeviceParams */
    DMV170_RTC_ADDRESS,         /* ulCtrlPort1 */
    0,                          /* ulDataPort */
    icm7170_get_register_8,     /* getRegister */
    icm7170_set_register_8,     /* setRegister */
  }
};
unsigned long RTC_Count = (sizeof(RTC_Table)/sizeof(rtc_tbl));
rtems_device_minor_number RTC_Minor;

```

(continues on next page)

(continued from previous page)

```
22 bool dmv177_icm7170_probe(int minor)
23 {
24   volatile uint16_t *card_resource_reg;
25   card_resource_reg = (volatile uint16_t *) DMV170_CARD_RESORCE_REG;
26   if ( (*card_resource_reg & DMV170_RTC_INST_MASK) == DMV170_RTC_INSTALLED )
27     return TRUE;
28   return FALSE;
29 }
```

13.2 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:

```
rtems_device_driver rtc_initialize(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor_arg,
  void                      *arg
)
{
  for each RTC configured in RTC_Table
    if the deviceProbe for this RTC indicates it is present
      set RTC_Minor to this device
      set RTC_Present to TRUE
      break out of this loop

    if RTC_Present is not TRUE
      return RTEMS_INVALID_NUMBER to indicate that no RTC is present

    register this minor number as the "/dev/rtc"

    perform the deviceInitialize routine for the preferred RTC chip

    for RTCs past this one in the RTC_Table
      if the deviceProbe for this RTC indicates it is present
        perform the deviceInitialize routine for this RTC chip
        register the configured name for this RTC
}
```

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.

13.3 setRealTimeToRTEMS

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

```
void setRealTimeToRTEMS(void)
{
  if no RTCs are present
    return

  invoke the deviceGetTime routine for the preferred RTC
  set the RTEMS TOD using rtems_clock_set
}
```

13.4 setRealTimeFromRTEMS

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

```
void setRealTimeFromRTEMS(void)
{
  if no RTCs are present
    return

  obtain the RTEMS TOD using rtems_clock_get
  invoke the deviceSetTime routine for the preferred RTC
}
```

13.5 getRealTime

The `getRealTime` returns the preferred RTC TOD to the caller.

```
void getRealTime( rtems_time_of_day *tod )
{
  if no RTCs are present
  return

  invoke the deviceGetTime routine for the preferred RTC
}
```

13.6 setRealTime

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

```
void setRealTime( rtems_time_of_day *tod )
{
  if no RTCs are present
    return

  invoke the deviceSetTime routine for the preferred RTC
}
```

13.7 checkRealTime

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

```
int checkRealTime( void )
{
  if no RTCs are present
    return -1

  obtain the RTEMS TOD using rtems_clock_get
  get the TOD from the preferred RTC using the deviceGetTime routine
  convert the RTEMS TOD to seconds
  convert the RTC TOD to seconds

  return the RTEMS TOD in seconds - RTC TOD in seconds
}
```

CHAPTER

FOURTEEN

NETWORKING DRIVER

14.1 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.

14.2 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.

14.3 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`.

14.4 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:

```
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.

```
-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.

14.5 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.

```
int rtems_bsdnet_parse_driver_name (
  const struct rtems_bsdnet_ifconfig  *config,
  char                                **namep
);
```

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.

14.6 Write the Driver Start Function.

This function is called each time the network stack wants to start the transmitter. This occures 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.

14.7 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:

```
rtems_id rtems_bsdnet_newproc(
  char *name,
  int   stacksize,
  void  (*entry)(void *),
  void *arg
);
```

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.

14.8 Write the Driver Transmit Task

This task is reponsible 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.

14.9 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.

14.10 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.

14.11 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.

14.12 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`.

CHAPTER

FIFTEEN

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()`

15.1 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()`, `ioctls()` to get the frame buffer info, `read()` and/or `write()`, and `close()`.

15.2 Driver Function Overview

15.2.1 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:

```
rtems_device_driver frame_buffer_initialize(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *arg)
{
  rtems_status_code status;

  printk( "frame buffer driver initializing..\n" );

  /*
   * Register the device
   */
  status = rtems_io_register_name("/dev/fb0", major, 0);
  if (status != RTEMS_SUCCESSFUL)
  {
    printk("Error registering frame buffer device!\n");
    rtems_fatal_error_occurred( status );
  }

  /*
   * graphics hardware initialization goes here for non-console
   * devices
   */

  return RTEMS_SUCCESSFUL;
}
```

15.2.2 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:

```
rtems_device_driver frame_buffer_close(
  rtems_device_major_number  major,
```

(continues on next page)

(continued from previous page)

```
  rtems_device_minor_number  minor,
  void                      *arg
)
{
  if (pthread_mutex_unlock(&mutex) == 0) {
    /* restore previous state.  for VGA this means return to text mode.
     * leave out if graphics hardware has been initialized in
     * frame_buffer_initialize()
     */
    ega_hwterm();
    printk( "FBVGA close called.\n" );
    return RTEMS_SUCCESSFUL;
  }
  return RTEMS_UNSATISFIED;
}
```

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

15.2.3 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:

```
rtems_device_driver frame_buffer_close(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *arg)
{
  pthread_mutex_unlock(&mutex);

  /* TODO check mutex return value, RTEMS_UNSATISFIED if it failed.  we
   * don't want to unconditionally call ega_hwterm()... */
  /* restore previous state.  for VGA this means return to text mode.
   * leave out if graphics hardware has been initialized in
   * frame_buffer_initialize() */
  ega_hwterm();
  printk( "frame buffer close called.\n" );
  return RTEMS_SUCCESSFUL;
}
```

15.2.4 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:

```
rtems_device_driver frame_buffer_read(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                       *arg
)
{
  rtems_libio_rw_args_t *rw_args = (rtems_libio_rw_args_t *)arg;
  rw_args->bytes_moved = ((rw_args->offset + rw_args->count) > fb_fix.smem_len ) ?
                          (fb_fix.smem_len - rw_args->offset) : rw_args->count;
  memcpy(rw_args->buffer,
         (const void *) (fb_fix.smem_start + rw_args->offset),
         rw_args->bytes_moved);
  return RTEMS_SUCCESSFUL;
}
```

15.2.5 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:

```
rtems_device_driver frame_buffer_write(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                       *arg
)
{
  rtems_libio_rw_args_t *rw_args = (rtems_libio_rw_args_t *)arg;
  rw_args->bytes_moved = ((rw_args->offset + rw_args->count) > fb_fix.smem_len ) ?
                          (fb_fix.smem_len - rw_args->offset) : rw_args->count;
  memcpy((void *) (fb_fix.smem_start + rw_args->offset),
         rw_args->buffer,
         rw_args->bytes_moved);
  return RTEMS_SUCCESSFUL;
}
```

15.2.6 Frame Buffer IO Control

The frame buffer driver allows several ioctls, 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:

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

```
rtems_device_driver frame_buffer_control(
  rtems_device_major_number  major,
```

(continues on next page)

(continued from previous page)

```
  rtems_device_minor_number  minor,
  void                       *arg
)
{
  rtems_libio_ioctl_args_t *args = arg;

  printk( "FBVGA ioctl called, cmd=%x\n", args->command  );

  switch( args->command ) {
    case FBIOGET_FSCREENINFO:
      args->ioctl_return =  get_fix_screen_info( ( struct fb_fix_screeninfo * ) args->buffer );
      break;
    case FBIOGET_VSCREENINFO:
      args->ioctl_return =  get_var_screen_info( ( struct fb_var_screeninfo * ) args->buffer );
      break;
    case FBIOPUT_VSCREENINFO:
      /* not implemented yet*/
      args->ioctl_return = -1;
      return RTEMS_UNSATISFIED;
    case FBIOGETCMAP:
      args->ioctl_return =  get_palette( ( struct fb_cmap * ) args->buffer );
      break;
    case FBIOPUTCMAP:
      args->ioctl_return =  set_palette( ( struct fb_cmap * ) args->buffer );
      break;
    default:
      args->ioctl_return = 0;
      break;
  }

  return RTEMS_SUCCESSFUL;
}
```

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

CHAPTER

SIXTEEN

ADA95 INTERRUPT SUPPORT

16.1 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 <mailto:jgais@ws.estec.esa.nl> who also wrote this chapter.

16.2 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.

16.3 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.

16.4 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.

CHAPTER

SEVENTEEN

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 mulitprocessing 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*.

17.1 Shared Memory Configuration Table

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

```
typedef volatile uint32_t vol_u32;

typedef struct {
  vol_u32 *address;        /* write here for interrupt    */
  vol_u32  value;          /* this value causes interrupt */
  vol_u32  length;         /* for this length (0,1,2,4)   */
} Shm_Interrupt_information;

struct shm_config_info {
  vol_u32           *base;                    /* base address of SHM         */
  vol_u32            length;                  /* length (in bytes) of SHM    */
  vol_u32            format;                  /* SHM is big or little endian */
  vol_u32          (*convert)();              /* neutral conversion routine  */
  vol_u32            poll_intr;               /* POLLED or INTR driven mode  */
  void             (*cause_intr)( uint32_t );
  Shm_Interrupt_information Intr;             /* cause intr information      */
};

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.

17.2 Primitives

17.2.1 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.

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

17.2.2 Get Configuration

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

```
void Shm_Get_configuration(
  uint32_t           localnode,
  shm_config_table **shmcfg
)
{
  fill in the Shared Memory Configuration Table
}
```

17.2.3 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:

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

17.2.3.1 Initializing a Shared Lock

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

```
void Shm_Initialize_lock(
  Shm_Locked_queue_Control *lq_cb
)
{
  lq_cb->lock = LQ_UNLOCKED;
}
```

17.2.3.2 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 unavailble, 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:

```
void Shm_Lock(
  Shm_Locked_queue_Control *lq_cb
)
{
  disable processor interrupts
    set Shm_isrstat to previous interrupt disable level

  while ( TRUE ) {
    atomically attempt to acquire the lock
    if the lock was acquired
      return
    delay some small period of time
  }
}
```

17.2.3.3 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:

```
void Shm_Unlock(
  Shm_Locked_queue_Control *lq_cb
)
{
```

(continues on next page)

(continued from previous page)

```
  set the lock to the unlocked value
  reenable processor interrupts to their level prior
    to the lock being acquired.  This value was saved
    in the global variable Shm_isrstat
}
```

17.3 Installing the MPCI 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:

```
void Shm_setvec( void )
{
  install the interprocessor communications ISR
}
```

On a polled node, this routine would be empty.

CHAPTER

EIGHTEEN

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.

18.1 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.

18.1.1 benchmark_timer_initialize

Initialize the timer source.

```
void benchmark_timer_initialize(void)
{
  initialize the benchmark timer
}
```

18.1.2 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`.

```
benchmark_timer_t benchmark_timer_read(void)
{
  stop time = read the hardware timer
  if the subtract overhead feature is enabled
    subtract overhead from stop time
  return the stop time
}
```

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 harware timer value change is sufficiently small. This is intended to indicate that the execution time is below the resolution of the timer.

18.1.3 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.

```
void benchmark_timer_disable_subtracting_average_overhead(bool find_flag)
{
  disable the subtract overhead feature
}
```

The `benchmark_timer_find_average_overhead` variable is used to indicate the state of the "subtract overhead feature".

18.2 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.

CHAPTER

NINETEEN

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.

19.1 Terms

ATA device - physical device attached to an IDE controller

19.2 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`.

19.3 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:

```
rtems_device_driver ata_initialize(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *arg
)
{
  initialize internal ATA driver data structure

  for each IDE controller successfully initialized by the IDE Controller driver
    if the controller is interrupt driven
      set up interrupt handler

    obtain information about ATA devices attached to the controller
    with help of EXECUTE DEVICE DIAGNOSTIC command

    for each ATA device detected on the controller
      obtain device parameters with help of DEVICE IDENTIFY command

      register new ATA device as new block device in the system
}
```

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
...	...	...	...

19.4 ATA Driver Architecture

19.4.1 ATA Driver Main Internal Data Structures

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

```
/* ATA request */
typedef struct ata_req_s {
  Chain_Node        link;   /* link in requests chain */
  char              type;   /* request type */
  ata_registers_t   regs;   /* ATA command */
  uint32_t          cnt;    /* Number of sectors to be exchanged */
  uint32_t          cbuf;   /* number of current buffer from breq in use */
  uint32_t          pos;    /* current position in 'cbuf' */
  blkdev_request   *breq;   /* blkdev_request which corresponds to the ata␣
↪request */
  rtems_id          sema;   /* semaphore which is used if synchronous
                             * processing of the ata request is required */
  rtems_status_code status; /* status of ata request processing */
  int               error;  /* error code */
} 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:

```
/*
 * This structure describes controller state, devices configuration on the
 * controller and chain of ATA requests to the controller.
*/
typedef struct ata_ide_ctrl_s {
  bool          present;   /* controller state */
  ata_dev_t     device[2]; /* ata devices description */
  Chain_Control reqs;      /* requests chain */
} 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):

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

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

ATA driver defines the following internal events:

```
/* ATA driver events */
typedef enum ata_msg_type_s {
  ATA_MSG_GEN_EVT = 1,     /* general event */
  ATA_MSG_SUCCESS_EVT,     /* success event */
  ATA_MSG_ERROR_EVT,       /* error event */
  ATA_MSG_PROCESS_NEXT_EVT /* process next ata request event */
} ata_msg_type_t;
```

19.4.2 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)

CHAPTER

TWENTY

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.

20.1 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.

20.2 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:

```
rtems_device_driver ide_controller_initialize(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor_arg,
  void                      *arg
)
{
  for each IDE Controller chip configured in IDE_Controller_Table
    if (BSP dependent probe(if exists) AND device probe for this IDE chip
        indicates it is present)
      perform initialization of the particular chip
      register device with configured name for this chip
}
```

20.3 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.

```
void ide_controller_read_register(
  rtems_device_minor_number  minor,
  unsigned32                 reg,
  unsigned32                *value
)
{
  get IDE Controller chip configuration information from
  IDE_Controller_Table by minor number

  invoke read register routine for the chip
}
```

20.4 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.

```
void ide_controller_write_register(
  rtems_device_minor_number minor,
  unsigned32                reg,
  unsigned32                value
)
{
  get IDE Controller chip configuration information from
  IDE_Controller_Table by minor number

  invoke write register routine for the chip
}
```

20.5 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.

```
void ide_controller_read_data_block(
  rtems_device_minor_number  minor,
  unsigned16                 block_size,
  blkdev_sg_buffer          *bufs,
  uint32_t                  *cbuf,
  uint32_t                  *pos
)
{
  get IDE Controller chip configuration information from
  IDE_Controller_Table by minor number

  invoke read data block routine for the chip
}
```

20.6 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.

```
void ide_controller_write_data_block(
  rtems_device_minor_number  minor,
  unsigned16                 block_size,
  blkdev_sg_buffer          *bufs,
  uint32_t                  *cbuf,
  uint32_t                  *pos
)
{
  get IDE Controller chip configuration information from
  IDE_Controller_Table by minor number

  invoke write data block routine for the chip
}
```

CHAPTER

TWENTYONE

COMMAND AND VARIABLE INDEX

There are currently no Command and Variable Index entries.

CHAPTER

TWENTYTWO

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.

22.1 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.

22.2 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.

22.3 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 bsps/ 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.

22.4 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. .

22.5 The Structure of the bsps/ directory

All BSPs are found within the bsps/ 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.

```
$ cd bsps
$ ls
aarch64  i386     m68k        mips   nios2   or1k     riscv   sparc
arm      include  microblaze  moxie  no_cpu  powerpc  shared  x86_64
```

If we cd into a specific architecture, we see that a similar structure is employed. bsps/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.

```
$ cd arm
$ ls
altera-cyclone-v  fvp       lpc176x          shared    xilinx-versal-rpu
atsam             gumstix   lpc24xx          smdk2410  xilinx-zynq
beagle            imx       lpc32xx          stm32f4   xilinx-zynqmp
csb336            imxrt     raspberrypi      stm32h7   xilinx-zynqmp-rpu
csb337            include   realview-pbx-a9  tms570
edb7312           lm3s69xx  rtl22xx          xen
```

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.

```
$ cd raspberrypi
$ ls
config  console  gpio  i2c  include  irq  README.md  spi  start
```

Another way to get an idea of the structure of bsps/ 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.

```
$ pwd
~/rtems/bsps/arm/raspberrypi
$ tree -f
.
├── ./config
│   ├── ./config/raspberrypi2.cfg
│   ├── ./config/raspberrypi.cfg
│   └── ./config/raspberrypi.inc
├── ./console
│   ├── ./console/console-config.c
│   ├── ./console/fb.c
│   ├── ./console/fbcons.c
│   ├── ./console/font_data.h
│   └── ./console/outch.c
├── ./gpio
```

(continues on next page)

(continued from previous page)

```
16 │   ├── ./gpio/gpio-interfaces-pi1-rev2.c
17 │   └── ./gpio/rpi-gpio.c
18 ├── ./i2c
19 │   └── ./i2c/i2c.c
20 ├── ./include
21 │   ├── ./include/bsp
22 │   │   ├── ./include/bsp/fbcons.h
23 │   │   ├── ./include/bsp/i2c.h
24 │   │   ├── ./include/bsp/irq.h
25 │   │   ├── ./include/bsp/mailbox.h
26 │   │   ├── ./include/bsp/mmu.h
27 │   │   ├── ./include/bsp/raspberrypi.h
28 │   │   ├── ./include/bsp/rpi-fb.h
29 │   │   ├── ./include/bsp/rpi-gpio.h
30 │   │   ├── ./include/bsp/spi.h
31 │   │   ├── ./include/bsp/usart.h
32 │   │   └── ./include/bsp/vc.h
33 │   ├── ./include/bsp.h
34 │   └── ./include/tm27.h
35 ├── ./irq
36 │   └── ./irq/irq.c
37 ├── ./README.md
38 ├── ./spi
39 │   └── ./spi/spi.c
40 └── ./start
41     ├── ./start/bspreset.c
42     ├── ./start/bspsmp.c
43     ├── ./start/bspsmp_init.c
44     ├── ./start/bspstart.c
45     ├── ./start/bspstarthooks.c
46     ├── ./start/cmdline.c
47     ├── ./start/mailbox.c
48     ├── ./start/timer.c
49     ├── ./start/vc.c
50     └── ./start/vc_defines.h
51
52 9 directories, 37 files
```

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 bsps/ 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 bsps/ 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 bsps/ directory, we get an idea of how we should structure the Doxygen groups we create. More on this in the next section.

22.6 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 bsps/ 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 >][doxygen recommendations <wiki:developer/coding/doxygen >]. , which also contains a fair amount of information that will not be covered here.

22.7 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.

22.8 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.

```
bsps/arm/raspberrypi/include/bsp.h:

/**
 * @defgroup arm_raspberrypi Raspberry Pi Support
 *
 * @ingroup bsp_arm
 *
 * @brief Raspberry Pi support package
 *
 */
```

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.

22.9 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:

```
@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 bsps/ directory itself. We do this by creating a hierarchy of groups for each CPU architecture and each BSP.

22.10 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.

22.11 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>
```

22.12 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:

```
Header type 1: used to add files to groups, always found at the␣
↪beginning of a file
/**
 * @file
 *
 * @ingroup raspberrypi
 *
 * @brief Global BSP definitions.
 */

/*
 *  Copyright (c) YYYY NAME
 *
 *   <LICENSE TERMS>
 */
```

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:

```
Header type 2: Used to nest groups, found anywhere within a file
/**
 * @defgroup arm_raspberrypi Raspberry Pi Support
 *
 * @ingroup bsp_arm
 *
 * @brief Raspberry Pi Support Package
 */
```

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_raspberry pi 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.

22.13 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 bsps/ you would do:

```
export r=~/rtems
./do_doxygen bsps
```

22.14 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.

22.15 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:

```
  <----- This is your group name -------> <--usual description -->
@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.= Structure Conventions =

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.

22.16 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.

22.17 @defgroups 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:

```
/**
 *  @defgroup bsp_arm ARM
 *
 *  @ingroup bsp_kit
 *
 *  @brief ARM Board Support Packages
 */

/**
 *  @defgroup arm_shared ARM Shared Modules
 *
 *  @ingroup bsp_arm
 *
 *  @brief ARM Shared Modules
 */
```

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 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.

22.18 @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:

```
/**
  *  @defgroup *architecture*_*BSP* *name*
  *
  *  @ingroup bsp_*architecture*
  *
  *  @brief *BSP* Support Package
  */
```

22.19 @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:

22.20 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 subgroups like **raspberry_interrupt** to group files responsible for handling interrupts, **raspberrypi_usart** to group files responsible for implementing USART support, and many other subgroups.

22.21 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.

22.22 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.

22.23 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.

22.24 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.

22.25 @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.

22.26 @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 bsps/ 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 bsps/ 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.

22.27 @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 bsps/shared 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

C