BSP and Device Driver Development Guide

Edition 4.6.6, for 4.6.6

30 August 2003

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 2002-11-25.11

The RTEMS Project is hosted at `http://www.rtems.com`. Any inquiries concerning RTEMS, its related support components, its documentation, or any custom services for RTEMS should be directed to the contacts listed on that site. A current list of RTEMS Support Providers is at `http://www.rtems.com/support.html`.

Table of Contents

1 Introduction

Before reading this documentation, it is strongly advised to read the RTEMS Development Environment Guide to get acquainted with the RTEMS directory structure. This document describes how to do a RTEMS Board Support Package, i.e. how to port RTEMS on a new target board. Discussions are provided for the following topics:

- RTEMS Board Support Package Organization
- Makefiles and the Linker Command Script
- Board Initialization Sequence
- Device Drivers Including:
    - Console Driver
    - Clock Driver
    - Timer Driver
    - Real-Time Clock Driver
    - Non-Volatile Memory Driver
    - Networking Driver
    - Shared Memory Support Driver
    - Analog Driver
    - Discrete Driver

The original version of this manual was written by Geoffroy Montel <g_montel@yahoo.com>. When he started development of the gen68340 BSP, this manual did not exist. He wrote the initial version of this manual as the result of his experiences. At that time, this document was viewed internally as the most important "missing manual" in the RTEMS documentation set.

2 Target Dependent Files

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.

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 `c/src/lib/libbsp`. The BSP source directory is further subdivided based on the CPU family and BSP.

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

2.3 Peripheral Dependent

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

The source code for the reusable peripheral driver library may be found in the directory `c/src/lib/libchip`. 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. If you need help in making these modifications to RTEMS, please consider using one of the RTEMS Service Providers. The current list of these is at `http://www.rtems.com/support.html`.

2.5 CPU Dependent Executive Files

The CPU dependent files in the RTEMS executive source code are found in the following directory:

```
cpukit/score/cpu/CPU
```

where *CPU* is replaced with the CPU family name.

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 CPU Dependent Support Files

The CPU dependent support files contain routines which aid in the development of applications using that CPU family. For example, the support routines may contain standard trap handlers for alignment or floating point exceptions or device drivers for peripheral controllers found on the CPU itself. This class of code may be found in the following directory:

```
c/src/lib/libcpu/CPU
```

CPU model dependent support code is found in the following directory:

```
c/src/lib/libcpu/CPU/CPU_MODEL
```

2.7 Board Support Package Structure

The BSPs are all under the c/src/lib/libbsp directory. Below this directory, there is a subdirectory for each CPU family. Each BSP is found under the subdirectory for the appropriate processor family (m68k, powerpc, etc.). In addition, there is source code available which may be shared across all BSPs regardless of the CPU family or just across BSPs within a single CPU family. This results in a BSP using the following directories:

```
c/src/lib/libbsp/shared
c/src/lib/libbsp/CPU/shared
c/src/lib/libbsp/CPU/BSP
```

Under each BSP specific directory, there is a collection of subdirectories. For commonly provided functionality, the BSPs follow a convention on subdirectory naming. The following list describes the commonly found subdirectories under each BSP.

- **console**: is technically the serial driver for the BSP rather than just a console driver, it deals with the board UARTs (i.e. serial devices).
- **clock**: support for the clock tick – a regular time basis to the kernel.
- **timer**: support of timer devices.
- **rtc**: support for the hardware real-time clock.
- **nvmem**: support for non-volatile memory such as EEPROM or Flash.
- **network**: the Ethernet driver.
- **shmsupp**: support of shared memory driver MPCI layer in a multiprocessor system,
- **include**: include files for this BSP.
- **wrapup**: bundles all the components necessary to construct the BSP library.

The build order of the BSP is determined by the Makefile structure. This structure is discussed in more detail in the Chapter 3 [Makefiles], page 7 chapter.

NOTE: This manual refers to the gen68340 BSP for numerous concrete examples. You should have a copy of the gen68340 BSP available while reading this piece of documentation. This BSP is located in the following directory:

```
c/src/lib/libbsp/m68k/gen68340
```

Later in this document, the $BSP340_ROOT label will be used to refer to this directory.

3 Makefiles

This chapter discusses the Makefiles associated with a BSP. It does not describe the process of configuring, building, and installing RTEMS. This chapter will not provide detailed information about this process. Nonetheless, it is important to remember that the general process consists of three parts:

- configure
- build
- install

During the configure phase, a number of files are generated. These generated files are tailored for the specific host/target combination by the configure script. This set of files includes the Makefiles used to actually compile and install RTEMS.

During the build phase, the source files are compiled into object files and libraries are built.

During the install phase, the libraries, header files, and other support files are copied to the BSP specific installation point. After installation is successfully completed, the files generated by the configure and build phases may be removed.

3.1 Makefiles Used During The BSP Building Process

RTEMS uses the **GNU automake** and **GNU autoconf** automatic configuration package. Consequently, there are a number of automatically generated files in each directory in the RTEMS source tree. The `bootstrap` script found in the top level directory of the RTEMS source tree is executed to produce the automatically generated files. That script must be run from a directory with a `configure.ac` file in it.

There is a file named `Makefile.am` in each directory of a BSP. This file is used by **automake** to produce the file named `Makefile.in` which is also found in each directory of a BSP. The configure process specializes the `Makefile.in` files at the time that RTEMS is configured for a specific development host and target. Makefiles are automatically generated from the `Makefile.in` files. It is necessary for the BSP developer to provide the `Makefile.am` files and generate the `Makefile.in` files. Most of the time, it is possible to copy the `Makefile.am` from another similar directory and edit it.

The `Makefile` files generated are processed when configuring and building RTEMS for a given BSP.

The BSP developer is responsible for generating `Makefile.am` files which properly build all the files associated with their BSP. There are generally three types of Makefiles in a BSP source tree:

- Directory Makefiles
- Source Directory Makefiles
- Wrapup Makefile

3.1.1 Directory Makefiles

The Directory class of Makefiles directs the build process through a set of subdirectories in a particular order. This order is usually chosen to insure that build dependencies are properly processed. Most BSPs only have one Directory class Makefile. The `Makefile.am` in the BSP root directory (`c/src/lib/libbsp/CPU/BSP`) specifies following Makefile fragment shows how a BSP would specify the directories to be built and their order:

```
SUB_DIRS=include start340 startup clock console timer \
    network wrapup
```

3.1.2 Source Directory Makefiles

There is a `Makefile.am` in most of the directories in a BSP. This class of Makefile lists the files to be built as part of the driver. When adding new files to an existing directory, Do not forget to add the new files to the list of files to be built in the `Makefile.am` and run `bootstrap`.

NOTE: The `Makefile.am` files are ONLY processed by `bootstrap` and the resulting `Makefile.in` files are only processed during the configure process of a RTEMS build. Therefore, when developing a BSP and adding a new file to a `Makefile.am`, the already generated `Makefile` will not automatically include the new references unless you configured RTEMS with the `--enable-maintainer-mode` option. Otherwise, the new file not being be taken into account!

If adding a new directory, be sure to add it to the list of automatically generated files in the BSP's `configure.ac` script.

3.1.3 Wrapup Makefile

This class of Makefiles produces a library. The BSP wrapup Makefile is responsible for producing the library `libbsp.a` which is later merged into the `librtemsall.a` library. This Makefile specifies which BSP components are to be placed into the library as well as which components from the CPU dependent support code library. For example, this component may choose to use a default exception handler from the CPU dependent support code or provide its own.

This Makefile makes use of the `foreach` construct in **GNU make** to pick up the required components:

```
BSP_PIECES=startup clock console timer
CPU_PIECES=
GENERIC_PIECES=

# bummer; have to use $foreach since % pattern subst
#              rules only replace 1x
OBJS=$(foreach piece, $(BSP_PIECES), ../$(piece)/$(ARCH)/$(piece).o) \
 $(foreach piece, $(CPU_PIECES), \
```

```
    ../../../../libcpu/$(RTEMS_CPU)/$(piece)/$(ARCH)/$(piece).o) \
  $(wildcard \
    ../../../../libcpu/$(RTEMS_CPU)/$(RTEMS_CPU_MODEL)/fpsp/$(ARCH)/fpsp.rel) \
  $(foreach piece, \
    $(GENERIC_PIECES), ../../../$(piece)/$(ARCH)/$(piece).o)
```

The variable `OBJS` is the list of "pieces" expanded to include path information to the appropriate object files. The `wildcard` function is used on pieces of `libbsp.a` which are optional and may not be present based upon the configuration options.

3.2 Makefiles Used Both During The BSP Design and its Use

When building a BSP or an application using that BSP, it is necessary to tailor the compilation arguments to account for compiler flags, use custom linker scripts, include the RTEMS libraries, etc.. The BSP must be built using this information. Later, once the BSP is installed with the toolset, this same information must be used when building the application. So a BSP must include a build configuration file. The configuration file is `make/custom/BSP.cfg`.

The configuration file is taken into account when building one's application using the RTEMS template Makefiles (`make/templates`). It is strongly advised to use these template Makefiles since they encapsulate a number of build rules along with the compile and link time options necessary to execute on the target board.

There is a template Makefile provided for each of class of RTEMS Makefiles. The purpose of each class of RTEMS Makefile is to:

- call recursively the makefiles in the directories beneath the current one,
- build a library, or
- build an executable.

The following is a shortened and heavily commented version of the make customization file for the gen68340 BSP. The original source for this file can be found in the `make/custom` directory.

```
# The RTEMS CPU Family and Model
RTEMS_CPU=m68k
RTEMS_CPU_MODEL=mcpu32

include $(RTEMS_ROOT)/make/custom/default.cfg

# The name of the BSP directory used for the actual source code.
# This allows for build variants of the same BSP source.
RTEMS_BSP_FAMILY=gen68340

# CPU flag to pass to GCC
```

```
CPU_CFLAGS = -mcpu32

# optimization flag to pass to GCC
CFLAGS_OPTIMIZE_V=-O4 -fomit-frame-pointer

# The name of the start file to be linked with.  This file is the first
# part of the BSP which executes.
START_BASE=start340

# This make-exe macro is used in template makefiles to build the
# final executable. Any other commands to follow, just as using
# objcopy to build a PROM image or converting the executable to binary.

define make-exe
$(CC) $(CFLAGS) $(CFLAGS_LD) -o $(basename $@).exe $(LINK_OBJS)
$(NM) -g -n $(basename $@).exe > $(basename $@).num
$(SIZE) $(basename $@).exe
endif
```

3.2.1 Creating a New BSP Make Customization File

The basic steps for creating a `make/custom` file for a new BSP are as follows:

- copy any `.cfg` file to `BSP.cfg`
- modify RTEMS_CPU, RTEMS_CPU_MODEL, RTEMS_BSP_FAMILY, RTEMS_BSP, CPU_CFLAGS, START_BASE, and make-exe rules.

It is generally easier to copy a `make/custom` file from a BSP similar to the one being developed.

4 Linker Script

4.1 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.

4.2 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.

4.3 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
+------------------+
```

4.4 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
    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.
       */

      *(.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 = .;
} >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
}
```

4.5 Initialized Data

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

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

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

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

```
+--------------+     +--------------------+
| .data    RAM |     | .data          RAM |
+--------------+     +--------------------+
| .bss     RAM |     | .bss           RAM |
+--------------+     +--------------------+         +----------------+
| .text    ROM |     | .text          ROM |         |     .text      |
+--------------+     +--------------------+         +----------------+
                     | copy of .data  ROM |         | copy of .data  |
                     +--------------------+         +----------------+

      Step 1                Step 2                        Step 3
```

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

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

5 Miscellaneous Support Files

5.1 GCC Compiler Specifications File

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

```
%rename cpp old_cpp
%rename lib old_lib
%rename endfile old_endfile
%rename startfile old_startfile
%rename link old_link

*cpp:
%(old_cpp) %{qrtems: -D__embedded__} -Asystem(embedded)

*lib:
%{!qrtems: %(old_lib)} %{qrtems: --start-group \
%{!qrtems_debug: -lrtemsall} %{qrtems_debug: -lrtemsall_g} \
-lc -lgcc --end-group  ecrtn%O%s \
%{!qnolinkcmds: -T linkcmds%s}}

*startfile:
%{!qrtems: %(old_startfile)} %{qrtems:  ecrti%O%s \
%{!qrtems_debug: startsim.o%s} \
%{qrtems_debug: startsim_g.o%s}}

*link:
%{!qrtems: %(old_link)} %{qrtems: -Qy -dp -Bstatic \
-T linkcmds%s -e _start -u __vectors}
```

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

The `*cpp` definition specifies that when `-qrtems` is specified, predefine the preprocessor symbol `__embedded__`.

The `*lib` section insures that the RTEMS library, BSP specific linker script, gcc support library, and the EABI specific `ecrtn` file are used.

The `*startfile` section specifies that the BSP specific file `startsim.o` will be used instead of `crt0.o`. In addition, the EABI specific file `ecrti.o` will be linked in with the executable.

The `*link` section specifies the arguments that will be passed to the linker.

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

5.2 README Files

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

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

5.3 times

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

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

5.4 Tools Subdirectory

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

- `powerpc/psim` includes scripts to ease use of the simulator
- `m68k/mvme162` includes a utility to download across the VMEbus into target memory if the host is a VMEbus board in the same chasis.
- `unix/posix` includes scripts to run the tests automatically on this BSP.

5.5 bsp.h Include File

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

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

There are a number of fields in this file that are used only by the RTEMS Test Suites. The following is a list of these:

- `MAX_LONG_TEST_DURATION` - the longest length of time a "long running" test should run.
- `MAX_SHORT_TEST_DURATION` - the longest length of time a "short running" test should run.
- `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.

5.6 Calling Overhead File

The file `include/coverhd.h` contains the overhead associated with invoking each directive. This overhead consists of the execution time required to package the parameters as well as to execute the "jump to subroutine" and "return from subroutine" sequence. The intent of this file is to help separate the calling overhead from the actual execution time of a directive. This file is only used by the tests in the RTEMS Timing Test Suite.

The numbers in this file are obtained by running the "Timer Overhead" `tmoverhd` test. The numbers in this file may be 0 and no overhead is subtracted from the directive execution times reported by the Timing Suite.

5.7 sbrk() Implementation

If the BSP wants to dynamically extend the heap used by the C Library memory allocation routines (i.e. `malloc` family), then this routine must be functional. The following is the prototype for this routine:

```
void * sbrk(size_t increment)
```

The `increment` amount is based upon the `sbrk_amount` parameter passed to the `RTEMS_Malloc_Initialize` during system initialization. See Section 7.3.5 [Initialization Code RTEMS Pretasking Callback], page 30 for more information.

There is a default implementation which returns an error to indicate that the heap can not be extended. This implementation can be found in `c/src/lib/libbsp/shared/sbrk.c`. Many of the BSPs use this shared implementation. In order to use this implementation, the file `Makefile.am` in the BSP's `startup` directory must be modified so that the `$VPATH` variable searches both the `startup` directory and the shared directory. The following illustates the `VPATH` setting in the PowerPC psim BSP's `startup/Makefile.am`:

```
VPATH = @srcdir@:@srcdir@/../../../shared
```

This instructs make to look in all of the directories in the `VPATH` for the source files. The directories will be examined in the order they are specified.

5.8 bsp_cleanup() - Cleanup the Hardware

The `bsp_cleanup()` is the last C code invoked. Most of the BSPs use the same shared version of `bsp_cleanup()` that does nothing. This implementation is located in the following file:

```
c/src/lib/libbsp/shared/bspclean.c
```

The `bsp_cleanup()` routine can be used to return to a ROM monitor, insure that interrupt sources are disabled, etc.. This routine is the last place to insure a clean shutdown of the hardware.

5.9 set_vector() - Install an Interrupt Vector

The `set_vector` 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: `set_vector` is provided by the majority of BSPs but not all. In particular, the i386 BSPs use a different scheme.

6 Ada95 Interrupt Support

6.1 Introduction

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

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

6.2 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:

1. Write an interrupt/trap handler that will raise the corresponding signal depending on the interrupt/trap number.
2. Install the interrupt handler for all interrupts/traps that will be handled by gnat (including spurious).
3. 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.

6.3 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.

6.4 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.

7 Initialization Code

7.1 Introduction

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

Most of the examples in this chapter will be based on the gen68340 BSP initialization code. Like most BSPs, the initialization for this BSP is divided into two subdirectories under the BSP source directory. The gen68340 BSP source code is in the following directory:

```
c/src/lib/libbsp/m68k/gen68340
```

The following source code files are in this subdirectory.

- `start340`: assembly language code which contains early initialization routines
- `startup`: C code with higher level routines (RTEMS initialization related)

NOTE: The directory `start340` is simply named `start` or start followed by a BSP designation.

In the `start340` directory are two source files. The file `startfor340only.s` is the simpler of these files as it only has initialization code for a MC68340 board. The file `start340.s` contains initialization for a 68349 based board as well.

7.2 Required Global Variables

Although not strictly part of initialization, there are a few global variables assumed to exist by many support components. These global variables are usually declared in the file `startup/bspstart.c` that provides most of the BSP specific initialization. The following is a list of these global variables:

- `BSP_Configuration` is the BSP's writable copy of the RTEMS Configuration Table.
- `Cpu_table` is the RTEMS CPU Dependent Information Table.
- `bsp_isr_level` is the interrupt level that is set at system startup. It will be restored when the executive returns control to the BSP.

7.3 Board Initialization

This section describes the steps an application goes through from the time the first BSP code is executed until the first application task executes. The routines invoked during this will be discussed and their location in the RTEMS source tree pointed out.

7.3.1 Start Code - Assembly Language Initialization

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

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

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

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

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

7.3.2 boot_card() - Boot the Card

The `boot_card()` is the first C code invoked. Most of the BSPs use the same shared version of `boot_card()` which is located in the following file:

```
c/src/lib/libbsp/shared/main.c
```

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

- initializes the shared fields of the CPU Configuration Table (variable name `Cpu_table`) to a default state,
- copies the application's RTEMS Configuration Table (variable name `Configuration`) to the BSP's Configuration Table (variable name `BSP_Configuration`) so it can be modified as necessary without copying the original table,
- invokes the BSP specific routine `bsp_start()`,
- invokes the RTEMS directive `rtems_initialize_executive_early()` to initialize the executive, C Library, and all device drivers but return without initiating multitasking or enabling interrupts,
- invokes the shared `main()` in the same file as `boot_card()` which does not return until the `rtems_shutdown_executive` directive is called, and
- invokes the BSP specific routine `bsp_cleanup()` to perform any necessary board specific shutdown actions.

It is important to note that the executive and much of the support environment must be initialized before invoking `main()`.

7.3.3 bsp_start() - BSP Specific Initialization

This is the first BSP specific C routine to execute during system initialization. This routine often performs required fundamental hardware initialization such as setting bus controller registers that do not have a direct impact on whether or not C code can execute. The source code for this routine is usually found in the following file:

```
c/src/lib/libbsp/CPU/BSP/startup/bspstart.c
```

This routine is also responsible for overriding the default settings in the CPU Configuration Table and setting port specific entries in this table. This may include increasing the maximum number of some types of RTEMS system objects to reflect the needs of the BSP and the base set of device drivers. This routine will typically also install routines for one or more of the following initialization hooks:

- BSP Pretasking Hook
- BSP Predriver Hook
- BSP Postdriver Hook

One of the most important functions performed by this routine is determining where the RTEMS Workspace is to be located in memory. All RTEMS objects and task stacks will be allocated from this Workspace. The RTEMS Workspace is distinct from the application heap used for `malloc()`. Many BSPs place the RTEMS Workspace area at the end of RAM although this is certainly not a requirement.

After completing execution, this routine returns to the `boot_card()` routine.

7.3.4 main() - C Main

This routine is the C main entry point. This is a special routine and the GNU Compiler Suite treats it as such. The GNU C Compiler recognizes `main()` and automatically inserts a call to the compiler run-time support routine `__main()` as the first code executed in `main()`.

The routine `__main()` initializes the compiler's basic run-time support library and, most importantly, invokes the C++ global constructors.

The precise placement of when `main()` is invoked in the RTEMS initialization sequence insures that C Library and non-blocking calls can be made in global C++ constructors.

The shared implementation of this routine is located in the following file:

```
c/src/lib/libbsp/shared/main.c
```

In addition to the implicit invocation of `__main`, this routine performs some explicit initialization. This routine sets the variable `rtems_progname` and initiates multitasking via a call to the RTEMS directive `rtems_initialize_executive_late`. It is important to note that the executive does not return to this routine until the RTEMS directive `rtems_shutdown_executive` is invoked.

The RTEMS initialization procedure is described in the **Initialization Manager** chapter of the **RTEMS Application C User's Guide**. Please refer to that manual for more information.

7.3.5 RTEMS Pretasking Callback

The `pretasking_hook` entry in the RTEMS CPU Configuration Table may be the address of a user provided routine that is invoked once RTEMS API initialization is complete but before interrupts and tasking are enabled. No tasks – not even the IDLE task – have been created when this hook is invoked. The pretasking hook is optional.

Although optional, most of the RTEMS BSPs provide a pretasking hook callback. This routine is usually called `bsp_pretasking_hook` and is found in the file:

```
c/src/lib/libbsp/CPU/BSP/startup/bspstart.c
```

The `bsp_pretasking_hook()` routine is the appropriate place to initialize any support components which depend on the RTEMS APIs. Most BSPs set the debug level for the system and initialize the RTEMS C Library support in their implementation of `bsp_pretasking_hook()`. This initialization includes the application heap used by the `malloc` family of routines as well as the reentrancy support for the C Library.

The routine `bsp_libc_init` routine invoked from the `bsp_pretasking_hook()` routine is passed the starting address, length, and growth amount passed to `sbrk`. This "sbrk amount" is only used if the heap runs out of memory. In this case, the RTEMS malloc implementation will invoked `sbrk` to obtain more memory. See Section 5.7 [Miscellaneous Support Files sbrk() Implementation], page 23 for more details.

7.3.6 RTEMS Predriver Callback

The `predriver_hook` entry in the RTEMS CPU Configuration Table may be the address of a user provided routine that is is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

Most BSPs do not use this callback.

7.3.7 Device Driver Initialization

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

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

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

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

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

7.3.8 RTEMS Postdriver Callback

The `postdriver_hook` entry in the RTEMS CPU Configuration Table may be the address of a user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. Interrupts and tasking are disabled. The postdriver hook is optional.

Although optional, most of the RTEMS BSPs provide a postdriver hook callback. This routine is usually called `bsp_postdriver_hook` and is found in the file:

```
c/src/lib/libbsp/CPU/BSP/startup/bsppost.c
```

The `bsp_postdriver_hook()` routine is the appropriate place to perform initialization that must be performed before the first task executes but requires that a device driver be initialized. The shared implementation of the postdriver hook opens the default standard in, out, and error files and associates them with `/dev/console`.

7.4 The Interrupt Vector Table

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

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

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

7.4.1 Interrupt Vector Table on the gen68340 BSP

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

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

7.5 Chip Select Initialization

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

```
                     +-------------------+
         ------------|                   |
         ------------|                   |------------
         ------------|      Address      |------------
         ------------|      Decoder      |------------
         ------------|                   |------------
         ------------|                   |
                     +-------------------+
           CPU Bus                           Chip Select
```

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

7.6 Integrated Processor Registers Initialization

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

7.7 Data Section Recopy

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

This code performs the following actions:

- copies the .data section from ROM to its location reserved in RAM (see Section 4.5 [Linker Script Initialized Data], page 18 for more details about this copy),
- clear `.bss` section (all the non-initialized data will take value 0).

7.8 RTEMS-Specific Initialization

7.9 The RTEMS configuration table

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

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

The header file `<confdefs.h>` defines a constant table named `Configuration`. It is common practice for the BSP to copy this table into a modifiable copy named `BSP_Configuration`. This copy of the table is modified to define the base address of the RTEMS Executive Workspace as well as to reflect any BSP and device driver requirements not automatically handled by the application.

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

8 Console Driver

8.1 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 device drivers. A console driver can be used to do raw data processing in addition to the "normal" standard input and output device functions required of a console.

The serial driver may be called as the consequence of a C Library call such as `printf` or `scanf` or directly via the `read` or `write` system calls. There are two main functioning modes:

- console: formatted input/output, with special characters (end of line, tabulations, etc.) recognition and processing,
- raw: permits raw data processing.

One may think that two serial drivers are needed to handle these two types of data, but Termios permits having only one driver.

8.2 Termios

Termios is a standard for terminal management, included in the POSIX 1003.1b standard. It is commonly provided on UNIX implementations. Having RTEMS support for Termios is beneficial:

- from the user's side because it provides standard primitive operations to access the terminal and change configuration settings. These operations are the same under Unix and Rtems.
- from the BSP developer's side because it frees the developer from dealing with buffer states and mutual exclusions on them. Early RTEMS console device drivers also did their own special character processing.

Termios support includes:

- raw and console handling,
- blocking or non-blocking characters receive, with or without Timeout.

At this time, RTEMS documentation does not include a thorough discussion of the Termios functionality. For more information on Termios, type `man termios` on a Unix box or point a web browser at `http://www.freebsd.org/cgi/man.cgi`.

8.3 Driver Functioning Modes

There are generally two main functioning modes for an UART (Universal Asynchronous Receiver-Transmitter, i.e. the serial chip):

- polled mode
- interrupt driven mode

In polled mode, the processor blocks on sending/receiving characters. This mode is not the most efficient way to utilize the UART. 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 port is to be used primarily as a debug console. In a simple polled driver, the software will continuously check the status of the UART when it is reading or writing to the UART. Termios improves on this by delaying the caller for 1 clock tick between successive checks of the UART on a read operation.

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 UART. The interrupt subroutine 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 UART (the exact number depends on the UART) 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 UART is idle, it is typically necessary to prime the transmitter before the first interrupt will occur.

8.4 Serial Driver Functioning Overview

The following Figure shows how a Termios driven serial driver works:

```
Figure not included in this draft
```

The following list describes the basic flow.

- the application programmer uses standard C library call (printf, scanf, read, write, etc.),
- C library (in fact that's Cygnus Newlib) calls RTEMS system call interface. This code can be found in the `c/src/lib/libc` directory.
- Glue code calls the serial driver entry routines.

8.4.1 Termios and Polled I/O

The following functions are provided by the driver and invoked by Termios for simple character input/output. The specific names of these routines are not important as Termios invokes them indirectly via function pointers.

8.4.1.1 pollWrite

The `pollWrite` routine is responsible for writing `len` characters from `buf` to the serial device specified by `minor`.

```
int pollWrite (int minor, const char *buf, int len)
{
  for (i=0; i<len; i++) {
     put buf[i] into the UART channel minor
     wait for the character to be transmitted
     on the serial line
  }
  return 0
}
```

8.4.1.2 pollRead

The `pollRead` routine is responsible for reading a single character from the serial device specified by `minor`. If no character is available, then the routine should return -1.

```
int pollRead(int minor)
{
   read status of UART
   if status indicates a character is available
     return character
   return -1
}
```

8.4.2 Termios and Interrupt Driven I/O

The UART generally generates interrupts when it is ready to accept or to emit a number of characters. In this mode, the interrupt subroutine is the core of the driver.

8.4.2.1 InterruptHandler

The `InterruptHandler` is responsible for processing asynchronous interrupts from the UART. There may be multiple interrupt handlers for a single UART. Some UARTs 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 `InterruptHandler` will have to check the status of the UART and determine what caused the interrupt. The following describes the operation of an `InterruptHandler` which has to do this:

```
rtems_isr InterruptHandler (rtems_vector_number v)
{
  check whether there was an error

  if some characters were received:
     Ask Termios to put them on his input buffer

  if some characters have been transmitted
        (i.e. the UART output buffer is empty)
     Tell TERMIOS that the characters have been
     transmitted. The TERMIOS routine will call
     the InterruptWrite function with the number
     of characters not transmitted yet if it is
     not zero.
}
```

8.4.2.2 InterruptWrite

The `InterruptWrite` is responsible for telling the UART that the `len` characters at `buf` are to be transmitted.

```
static int InterruptWrite(int minor, const char *buf, int len)
{
  tell the UART to transmit len characters from buf
  return 0
}
```

The driver has to put the n first buf characters in the UART channel minor buffer (n is the UART channel size, n=1 on the MC68640). Generally, an interrupt is raised after these n characters being transmitted. So UART interrupts may have to be enabled after putting the characters in the UART.

8.4.3 Initialization

The driver initialization is called once during the RTEMS initialization process.

The `console_initialize` function has to:

- initialize Termios support: call `rtems_termios_initialize()`. If Termios has already been initialized by another device driver, then this call will have no effect.
- Initialize the UART: This procedure should be described in the UART manual. This procedure **MUST** be followed precisely. This procedure varies but usually consists of:
    - reinitialize the UART channels
    - set the channels configuration to the Termios default: 9600 bauds, no parity, 1 stop bit, and 8 bits per character
- If interrupt driven, register the console interrupt routine to RTEMS:

```
rtems_interrupt_catch(
    InterruptHandler, CONSOLE_VECTOR, &old_handler);
```

- enable the UART channels.
- register the device name: in order to use the console (i.e. being able to do printf/scanf on stdin, stdout, and stderr), some device must be registered as "/dev/console":

```
rtems_io_register_name ("dev/console", major, i);
```

8.4.4 Opening a serial device

The `console_open` function is called whenever a serial device is opened. The device registered as `"/dev/console"` is opened automatically during RTEMS initialization. For instance, if UART channel 2 is registered as "/dev/tty1", the `console_open` entry point will be called as the result of an `fopen("/dev/tty1", mode)` in the application.

The `console_open` function has to inform Termios of the low-level functions for serial line support; the "callbacks".

The gen68340 BSP defines two sets of callback tables:

- one with functions for polled input/output
- another with functions for interrupt driven input/output

This code can be found in the file `$BSPROOT/console/console.c`.

8.4.4.1 Polled I/O

Termios must be told the addresses of the functions that are to be used for simple character input/output, i.e. pointers to the `pollWrite` and `pollRead` functions defined earlier in Section 8.4.1 [Console Driver Termios and Polled I/O], page 36.

8.4.4.2 Interrupt Driven I/O

Driver functioning is quite different in this mode. There is no device driver read function to be passed to Termios. Indeed a `console_read` call returns the contents of Termios input buffer. This buffer is filled in the driver interrupt subroutine (see Section 8.4.2 [Console Driver Termios and Interrupt Driven I/O], page 37).

The driver is responsible for providing a pointer to the `InterruptWrite` function.

8.4.5 Closing a Serial Device

The `console_close` is invoked when the serial device is to be closed. This entry point corresponds to the device driver close entry point.

This routine is responsible for notifying Termios that the serial device was closed. This is done with a call to `rtems_termios_close`.

8.4.6 Reading Characters From a Serial Device

The `console_read` is invoked when the serial device is to be read from. This entry point corresponds to the device driver read entry point.

This routine is responsible for returning the content of the Termios input buffer. This is done by invoking the `rtems_termios_read` routine.

8.4.7 Writing Characters to a Serial Device

The `console_write` is invoked when the serial device is to be written to. This entry point corresponds to the device driver write entry point.

This routine is responsible for adding the requested characters to the Termios output queue for this device. This is done by calling the routine `rtems_termios_write` to add the characters at the end of the Termios output buffer.

8.4.8 Changing Serial Line Parameters

The `console_control` is invoked when the line parameters for a particular serial device are to be changed. This entry point corresponds to the device driver io_control entry point.

The application write is able to control the serial line configuration with Termios calls (such as the `ioctl` command, see the Termios documentation for more details). If the driver is to support dynamic configuration, then is must have the `console_control` piece of code. Refer to the gen68340 BSP for an example of how it is done. Basically `ioctl` commands call `console_control` with the serial line configuration in a Termios defined data structure. The driver is responsible for reinitializing the UART with the correct settings.

9 Clock Driver

9.1 Introduction

The purpose of the clock driver is to provide 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.

The clock driver is located in the `clock` directory of the BSP.

9.2 Clock Driver Global Variables

This section describes the global variables expected to be provided by this driver.

9.2.1 Major and Minor Number

The major and minor numbers of the clock driver are made available via the following variables.

- rtems_device_major_number rtems_clock_major;
- rtems_device_minor_number rtems_clock_minor;

The clock device driver is responsible for declaring and initializing these variables. These variables are used by other RTEMS components – notably the Shared Memory Driver.

NOTE: In a future RTEMS version, these variables may be replaced with the clock device driver registering **/dev/clock**.

9.2.2 Ticks Counter

Most of the clock device drivers provide a global variable that is simply a count of the number of clock driver interrupt service routines that have occured. This information is valuable when debugging a system. This variable is declared as follows:

```
volatile rtems_unsigned32 Clock_driver_ticks;
```

9.3 Initialization

The initialization routine is responsible for programming the hardware that will periodically generate an interrupt. A programmable interval timer is commonly used as the source of the clock tick.

The device should be programmed such that an interrupt is generated every m microseconds, where m is equal to `BSP_Configuration.microseconds_per_tick`. Sometimes the periodic interval timer can use a prescaler so you have to look carefully at your user's manual to determine the correct value.

You must use the RTEMS primitive `rtems_interrupt_catch` to install your clock interrupt service routine:

```
rtems_interrupt_catch (Clock_ISR, CLOCK_VECTOR, &old_handler);
```

Since there is currently not a driver entry point invoked at system shutdown, many clock device drivers use the `atexit` routine to schedule their `Clock_exit` routine to execute when the system is shutdown.

By convention, many of the clock drivers do not install the clock tick if the `ticks_per_timeslice` field of the Configuration Table is 0.

9.4 System shutdown

Many drivers provide the routine `Clock_exit` that is scheduled to be run during system shutdown via the `atexit` routine. The `Clock_exit` routine will disable the clock tick source if it was enabled. This can be used to prevent clock ticks after the system is shutdown.

9.5 Clock Interrupt Subroutine

It only has to inform the kernel that a ticker has elapsed, so call :

```
rtems_isr Clock_isr( rtems_vector_number vector )
{
  invoke the rtems_clock_tick() directive to announce the tick
  if necessary for this hardware
    reload the programmable timer
}
```

9.6 IO Control

The clock driver must supply a handler for the IO control device driver entry point. This functionality is used by other components – notably the Shared Memory Driver to install a wrapper for the clock interrupt service routine. The following shows the functionality required:

```
rtems_device_driver Clock_control(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void *pargp
)
{
  error check the argument pointer parameter

  if the command is "ISR"
    invoke the clock interrupt service routine
  else if the command is "NEW"
    install the requested handler
}
```

10 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.

10.1 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.

10.1.1 Timer_initialize

Initialize the timer source.

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

10.1.2 Read_timer

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

```
int Read_timer(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.

10.1.3 An Empty Function

This routine is invoked by the RTEMS Timing Test Suite to measure the cost of invoking a subroutine.

```
rtems_status_code Empty_function (void)
{
  return RTEMS_SUCCESSFUL;
}
```

10.1.4 Set_find_average_overhead

This routine is invoked by the "Check Timer" (`tmck`) test in the RTEMS Timing Test Suite. It makes the `Read_timer` 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 Set_find_average_overhead(rtems_boolean find_flag)
{
    disable the subtract overhead feature
}
```

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

10.2 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.

11 Real-Time Clock Driver

11.1 Introduction

The Real-Time Clock (**RTC**) driver is responsible for providing an interface to an **RTC** device. [NOTE: In this chapter, the abbreviation **TOD** is used for **Time of Day**.] 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

The reference implementation for a real-time clock driver can be found in `c/src/lib/libbsp/shared/tod.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 `c/src/lib/lib/libchip/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 uses the shared libchip implementation of the RTC driver. Its `RTC_Table` configuration table can be found in `c/src/lib/libbsp/powerpc/dmv177/tod/config.c`.

11.2 Initialization

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

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

```
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.

11.3 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
}
```

11.4 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
}
```

11.5 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
}
```

11.6 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
}
```

11.7 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
}
```

12 ATA Driver

12.1 Terms

ATA device - physical device attached to an IDE controller

12.2 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`.

12.3 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
...	...	...	

12.4 ATA Driver Architecture

12.4.1 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 */
    rtems_unsigned32  cnt;    /* Number of sectors to be exchanged */
    rtems_unsigned32  cbuf;   /* number of current buffer from breq in use */
    rtems_unsigned32  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 {
    rtems_boolean 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;
```

12.4.2 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)

13 IDE Controller Driver

13.1 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 `$RTEMS_SRC_ROOT/c/src/libchip/ide`. This driver is based on the libchip concept and allows to work with any of the IDE Controller chips simply by appropriate configuration of BSP. Drivers for a particular IDE Controller chips locate in the following directories: drivers for well-known IDE Controller chips locate into `$RTEMS_SRC_ROOT/c/src/libchip/ide`, drivers for IDE Controller chips integrated with CPU locate into `$RTEMS_SRC_ROOT/c/src/lib/libcpu/myCPU` and drivers for custom IDE Controller chips (for example, implemented on FPGA) locate into `$RTEMS_SRC_ROOT/c/src/lib/libbsp/myBSP`. 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.

13.2 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
}
```

13.3 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
}
```

13.4 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
}
```

13.5 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,
                                    rtems_unsigned32          *cbuf,
                                    rtems_unsigned32          *pos)
{
  get IDE Controller chip configuration information from
  IDE_Controller_Table by minor number

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

13.6 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,
                                     rtems_unsigned32          *cbuf,
                                     rtems_unsigned32          *pos)
{
  get IDE Controller chip configuration information from
  IDE_Controller_Table by minor number

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

14 Non-Volatile Memory Driver

The Non-Volatile driver is responsible for providing an interface to various types of non-volatile memory. These types of memory include, but are not limited to, Flash, EEPROM, and battery backed RAM. The capabilities provided by this class of device driver are:

- Initialize the Non-Volatile Memory Driver
- Optional Disable Read and Write Handlers
- Open a Particular Memory Partition
- Close a Particular Memory Partition
- Read from a Particular Memory Partition
- Write to a Particular Memory Partition
- Erase the Non-Volatile Memory Area

There is currently only one non-volatile device driver included in the RTEMS source tree. The information provided in this chapter is based on drivers developed for applications using RTEMS. It is hoped that this driver model information can form the basis for a standard non-volatile memory driver model that can be supported in future RTEMS distribution.

14.1 Major and Minor Numbers

The **major** number of a device driver is its index in the RTEMS Device Address Table.

A **minor** number is associated with each device instance managed by a particular device driver. An RTEMS minor number is an `unsigned32` entity. Convention calls dividing the bits in the minor number down into categories that specify an area of non-volatile memory and a partition with that area. This results in categories like the following:

- **area** - indicates a block of non-volatile memory
- **partition** - indicates a particular address range with an area

From the above, it should be clear that a single device driver can support multiple types of non-volatile memory in a single system. The minor number is used to distinguish the types of memory and blocks of memory used for different purposes.

14.2 Non-Volatile Memory Driver Configuration

There is not a standard non-volatile driver configuration table but some fields are common across different drivers. The non-volatile memory driver configuration table is typically an array of structures with each structure containing the information for a particular area of non-volatile memory. The following is a list of the type of information normally required to configure each area of non-volatile memory.

memory_type is the type of memory device in this area. Choices are battery backed RAM, EEPROM, Flash, or an optional user-supplied type. If the user-supplied type is configured, then the user is responsible for providing a set of routines to program the memory.

memory is the base address of this memory area.

attributes is a pointer to a memory type specific attribute block. Some of the fields commonly contained in this memory type specific attribute structure area:

use_protection_algorithm is set to TRUE to indicate that the protection (i.e. locking) algorithm should be used for this area of non-volatile memory. A particular type of non-volatile memory may not have a protection algorithm.

access is an enumerated type to indicate the organization of the memory devices in this memory area. The following is a list of the access types supported by the current driver implementation:

- simple unsigned8
- simple unsigned16
- simple unsigned32
- simple unsigned64
- single unsigned8 at offset 0 in an unsigned16
- single unsigned8 at offset 1 in an unsigned16
- single unsigned8 at offset 0 in an unsigned32
- single unsigned8 at offset 1 in an unsigned32
- single unsigned8 at offset 2 in an unsigned32
- single unsigned8 at offset 3 in an unsigned32

depth is the depth of the progamming FIFO on this particular chip. Some chips, particularly EEPROMs, have the same programming algorithm but vary in the depth of the amount of data that can be programmed in a single block.

number_of_partitions is the number of logical partitions within this area.

Partitions is the address of the table that contains an entry to describe each partition in this area. Fields within each element of this table are defined as follows:

offset is the offset of this partition from the base address of this area.

length is the length of this partition.

By dividing an area of memory into multiple partitions, it is possible to easily divide the non-volatile memory for different purposes.

14.3 Initialize the Non-Volatile Memory Driver

At system initialization, the non-volatile memory driver's initialization entry point will be invoked. As part of initialization, the driver will perform whatever initializatin is required on each non-volatile memory area.

The discrete I/O driver may register device names for memory partitions of particular interest to the system. Normally this will be restricted to the device `"/dev/nv_memory"` to indicate the entire device driver.

14.4 Disable Read and Write Handlers

Depending on the target's non-volatile memory configuration, it may be possible to write to a status register and make the memory area completely inaccessible. This is target dependent and beyond the standard capabilities of any memory type. The user has the optional capability to provide handlers to disable and enable access to a partiticular memory area.

14.5 Open a Particular Memory Partition

This is the driver open call. Usually this call does nothing other than validate the minor number.

With some drivers, it may be necessary to allocate memory when a particular device is opened. If that is the case, then this is often the place to do this operation.

14.6 Close a Particular Memory Partition

This is the driver close call. Usually this call does nothing.

With some drivers, it may be necessary to allocate memory when a particular device is opened. If that is the case, then this is the place where that memory should be deallocated.

14.7 Read from a Particular Memory Partition

This corresponds to the driver read call. After validating the minor number and arguments, this call enables reads from the specified memory area by invoking the user supplied "enable reads handler" and then reads the indicated memory area. When invoked the `argument_block` is actually a pointer to the following structure type:

```
typedef struct {
  rtems_unsigned32   offset;
  void              *buffer;
  rtems_unsigned32   length;
  rtems_unsigned32   status;
}   Non_volatile_memory_Driver_arguments;
```

The driver reads `length` bytes starting at `offset` into the partition and places them at `buffer`. The result is returned in `status`.

After the read operation is complete, the user supplied "disable reads handler" is invoked to protect the memory area again.

14.8 Write to a Particular Memory Partition

This corresponds to the driver write call. After validating the minor number and arguments, this call enables writes to the specified memory area by invoking the "enable writes handler", then unprotecting the memory area, and finally actually writing to the indicated memory area. When invoked the `argument_block` is actually a pointer to the following structure type:

```
typedef struct {
  rtems_unsigned32   offset;
  void              *buffer;
  rtems_unsigned32   length;
  rtems_unsigned32   status;
}   Non_volatile_memory_Driver_arguments;
```

The driver writes `length` bytes from `buffer` and writes them to the non-volatile memory starting at `offset` into the partition. The result is returned in `status`.

After the write operation is complete, the "disable writes handler" is invoked to protect the memory area again.

14.9 Erase the Non-Volatile Memory Area

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, the specified area of non-volatile memory is erased.

15 Networking Driver

15.1 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 `c/src/lib/libbsp/m68k/gen68360/network` directory in the RTEMS source code distribution. Having a copy of this driver at hand when reading the following notes will help significantly.

15.2 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.

15.3 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`.

15.4 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.

15.5 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.

15.6 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.

15.7 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.

15.8 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.

15.9 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.

15.10 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.

15.11 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.

15.12 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`.

16 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**.

16.1 Shared Memory Configuration Table

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

```
typedef volatile rtems_unsigned32 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)( rtems_unsigned32 );
  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 is either INTR_MODE or POLLED_MODE to indicate how the node will be informed of incoming messages.

cause_intr

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.

16.2 Primitives

16.2.1 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
}
```

16.2.2 Get Configuration

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

```
void Shm_Get_configuration(
  rtems_unsigned32   localnode,
  shm_config_table **shmcfg
)
{
   fill in the Shared Memory Configuration Table
}
```

16.2.3 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.

16.2.3.1 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;
}
```

16.2.3.2 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
    }
}
```

16.2.3.3 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
)
{
  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
}
```

16.3 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.

17 Analog Driver

The Analog driver is responsible for providing an interface to Digital to Analog Converters (DACs) and Analog to Digital Converters (ADCs). The capabilities provided by this class of device driver are:

- Initialize an Analog Board
- Open a Particular Analog
- Close a Particular Analog
- Read from a Particular Analog
- Write to a Particular Analog
- Reset DACs
- Reinitialize DACS

Most analog devices are found on I/O cards that support multiple DACs or ADCs on a single card.

There are currently no analog device drivers included in the RTEMS source tree. The information provided in this chapter is based on drivers developed for applications using RTEMS. It is hoped that this driver model information can form the basis for a standard analog driver model that can be supported in future RTEMS distribution.

17.1 Major and Minor Numbers

The **major** number of a device driver is its index in the RTEMS Device Address Table.

A **minor** number is associated with each device instance managed by a particular device driver. An RTEMS minor number is an `unsigned32` entity. Convention calls dividing the bits in the minor number down into categories like the following:

- **board** - indicates the board a particular device is located on
- **port** - indicates the particular device on a board.

From the above, it should be clear that a single device driver can support multiple copies of the same board in a single system. The minor number is used to distinguish the devices.

17.2 Analog Driver Configuration

There is not a standard analog driver configuration table but some fields are common across different drivers. The analog driver configuration table is typically an array of structures with each structure containing the information for a particular board. The following is a list of the type of information normally required to configure an analog board:

board_offset
is the base address of a board.

DAC_initial_values
is an array of the voltages that should be written to each DAC during initialization. This allows the driver to start the board in a known state.

17.3 Initialize an Analog Board

At system initialization, the analog driver's initialization entry point will be invoked. As part of initialization, the driver will perform whatever board initialization is required and then set all outputs to their configured initial state.

The analog driver may register a device name for each DAC and ADC in the system.

17.4 Open a Particular Analog

This is the driver open call. Usually this call does nothing other than validate the minor number.

With some drivers, it may be necessary to allocate memory when a particular device is opened. If that is the case, then this is often the place to do this operation.

17.5 Close a Particular Analog

This is the driver close call. Usually this call does nothing.

With some drivers, it may be necessary to allocate memory when a particular device is opened. If that is the case, then this is the place where that memory should be deallocated.

17.6 Read from a Particular Analog

This corresponds to the driver read call. After validating the minor number and arguments, this call reads the indicated device. Most analog devices store the last value written to a DAC. Since DACs are output only devices, saving the last written value gives the appearance that DACs can be read from also. If the device is an ADC, then it is sampled.

NOTE: Many boards have multiple analog inputs but only one ADC. On these boards, it will be necessary to provide some type of mutual exclusion during reads. On these boards, there is a MUX which must be switched before sampling the ADC. After the MUX is switched, the driver must delay some short period of time (usually microseconds) before the signal is stable and can be sampled. To make matters worse, some ADCs cannot respond to wide voltage swings in a single sample. On these ADCs, one must do two samples when the voltage swing is too large. On a practical basis, this means that the driver usually ends up double sampling the ADC on these systems.

The value returned is a single precision floating point number representing the voltage read. This value is stored in the `argument_block` passed in to the call. By returning the voltage, the caller is freed from having to know the number of bits in the analog and board dependent conversion algorithm.

17.7 Write to a Particular Analog

This corresponds to the driver write call. After validating the minor number and arguments, this call writes the indicated device. If the specified device is an ADC, then an error is usually returned.

The value written is a single precision floating point number representing the voltage to be written to the specified DAC. This value is stored in the `argument_block` passed in to the call. By passing the voltage to the device driver, the caller is freed from having to know the number of bits in the analog and board dependent conversion algorithm.

17.8 Reset DACs

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, all of the DACs are written to 0.0 volts.

17.9 Reinitialize DACS

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, all of the DACs are written with the initial value configured for this device.

17.10 Get Last Written Values

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, the following information is returned to the caller:

- last value written to the specified DAC
- timestamp of when the last write was performed

18 Discrete Driver

The Discrete driver is responsible for providing an interface to Discrete Input/Outputs. The capabilities provided by this class of device driver are:

- Initialize a Discrete I/O Board
- Open a Particular Discrete Bitfield
- Close a Particular Discrete Bitfield
- Read from a Particular Discrete Bitfield
- Write to a Particular Discrete Bitfield
- Reset DACs
- Reinitialize DACS

Most discrete I/O devices are found on I/O cards that support many bits of discrete I/O on a single card. This driver model is centered on the notion of reading bitfields from the card.

There are currently no discrete I/O device drivers included in the RTEMS source tree. The information provided in this chapter is based on drivers developed for applications using RTEMS. It is hoped that this driver model information can form the discrete I/O driver model that can be supported in future RTEMS distribution.

18.1 Major and Minor Numbers

The **major** number of a device driver is its index in the RTEMS Device Address Table.

A **minor** number is associated with each device instance managed by a particular device driver. An RTEMS minor number is an `unsigned32` entity. Convention calls for dividing the bits in the minor number down into categories that specify a particular bitfield. This results in categories like the following:

- **board** - indicates the board a particular bitfield is located on
- **word** - indicates the particular word of discrete bits the bitfield is located within
- **start** - indicates the starting bit of the bitfield
- **width** - indicates the width of the bitfield

From the above, it should be clear that a single device driver can support multiple copies of the same board in a single system. The minor number is used to distinguish the devices.

By providing a way to easily access a particular bitfield from the device driver, the application is insulated with knowing how to mask fields in and out of a discrete I/O.

18.2 Discrete I/O Driver Configuration

There is not a standard discrete I/O driver configuration table but some fields are common across different drivers. The discrete I/O driver configuration table is typically an array of structures with each structure containing the information for a particular board. The following is a list of the type of information normally required to configure an discrete I/O board:

board_offset is the base address of a board.

relay_initial_values is an array of the values that should be written to each output word on the board during initialization. This allows the driver to start with the board's output in a known state.

18.3 Initialize a Discrete I/O Board

At system initialization, the discrete I/O driver's initialization entry point will be invoked. As part of initialization, the driver will perform whatever board initializatin is required and then set all outputs to their configured initial state.

The discrete I/O driver may register a device name for bitfields of particular interest to the system. Normally this will be restricted to the names of each word and, if the driver supports it, an `"all words"`.

18.4 Open a Particular Discrete Bitfield

This is the driver open call. Usually this call does nothing other than validate the minor number.

With some drivers, it may be necessary to allocate memory when a particular device is opened. If that is the case, then this is often the place to do this operation.

18.5 Close a Particular Discrete Bitfield

This is the driver close call. Usually this call does nothing.

With some drivers, it may be necessary to allocate memory when a particular device is opened. If that is the case, then this is the place where that memory should be deallocated.

18.6 Read from a Particular Discrete Bitfield

This corresponds to the driver read call. After validating the minor number and arguments, this call reads the indicated bitfield. A discrete I/O devices may have to store the last value written to a discrete output. If the bitfield is output only, saving the last written value gives the appearance that it can be read from also. If the bitfield is input, then it is sampled.

NOTE: Many discrete inputs have a tendency to bounce. The application may have to take account for bounces.

The value returned is an `unsigned32` number representing the bitfield read. This value is stored in the `argument_block` passed in to the call.

NOTE: Some discrete I/O drivers have a special minor number used to access all discrete I/O bits on the board. If this special minor is used, then the area pointed to by `argument_block` must be the correct size.

18.7 Write to a Particular Discrete Bitfield

This corresponds to the driver write call. After validating the minor number and arguments, this call writes the indicated device. If the specified device is an ADC, then an error is usually returned.

The value written is an `unsigned32` number representing the value to be written to the specified bitfield. This value is stored in the `argument_block` passed in to the call.

NOTE: Some discrete I/O drivers have a special minor number used to access all discrete I/O bits on the board. If this special minor is used, then the area pointed to by `argument_block` must be the correct size.

18.8 Disable Discrete Outputs

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, the discrete outputs are disabled.

NOTE: It may not be possible to disable/enable discrete output on all discrete I/O boards.

18.9 Enable Discrete Outputs

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, the discrete outputs are enabled.

NOTE: It may not be possible to disable/enable discrete output on all discrete I/O boards.

18.10 Reinitialize Outputs

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, the discrete outputs are rewritten with the configured initial output values.

18.11 Get Last Written Values

This is one of the IOCTL functions supported by the I/O control device driver entry point. When this IOCTL function is invoked, the following information is returned to the caller:

- last value written to the specified output word
- timestamp of when the last write was performed

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.