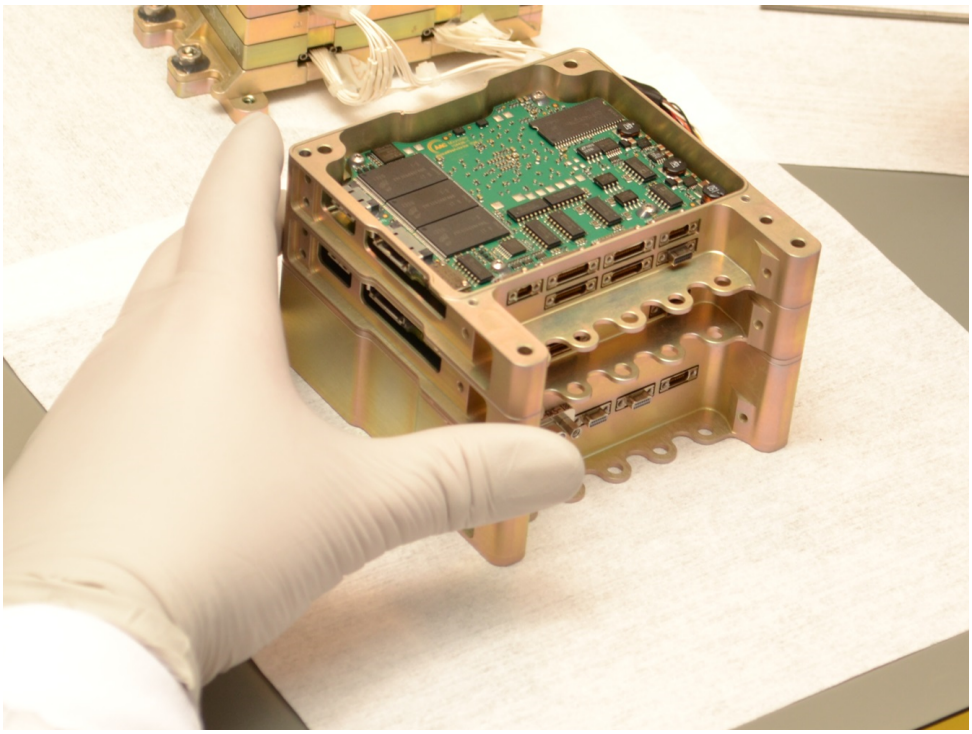
# Sirius TCM User Manual

**v1.22.0**



# TABLE OF CONTENTS

# 1. Introduction

The AAC Clyde space Sirius line of products, which will be referred to as "the Sirius products" in this document, consist of:

- Sirius OBC
- Sirius TCM
- Sirius TCM - with TCM Core Application

This manual describes the functionality and usage of the Sirius Leon3 TCM.

The Sirius OBC or Sirius TCM differ in certain areas such as the SoC, interfaces etc. see the electrical and mechanical ICD documents, [RD1] and [RD2], for details on the interfaces.

## 1.1. Applicable releases

This version of the manual is applicable to the following software releases:

Sirius Leon3 TCM: v1.22.0

## 1.2. Intended users

This manual is written for software engineers writing their own application software for the AAC Clyde Space Sirius Leon3 TCM. The electrical and mechanical interface is described in more detail in the electrical and mechanical ICD document [RD2] .

## 1.3. Getting support

If you encounter any problem using the Sirius products or another AAC Clyde Space product, please use the following address to get help:

Email: support@aac-clydespace.com

## 1.4. Reference documents

[RD1] “Sirius OBC electrical and mechanical ICD,” 205-088. AAC Clyde Space.

[RD2] “Sirius TCM electrical and mechanical ICD,” 205-089. AAC Clyde Space.

[RD3] “GRLIB IP Core User’s Manual,” GRIP, May 2019, Version 2019.2.

[RD4] “Sirius TCM Application User Manual,” 206-308. AAC Clyde Space.

[RD5] “Electrostatics - Part 5-1: Protection of electronic devices from electrostatic phenomena - General requirements,” SS-EN 61340-5-1.

[RD6] “ GRMON3 User’s Manual,” GRMON3-UM, June 2019, Version 3.1.0.

[RD7] “RTEMS C User Manual,” Edition 4.11.

[RD8] “Sirius SoC Configuration Document,” 206-222. AAC Clyde Space.

[RD9] “Asynchronous Receiver/Transmitter with FIFOs,” SNLS378B.

[RD10] “SpaceWire - Links, nodes, routers and networks,” ECSS-E-ST-50-12C.

[RD11] “RTEMS BSP and Device Driver Development Guide,” Edition 4.11.

[RD12] “TM Space Data Link Protocol,” CCSDS 132.0-B-2.

[RD13] “TC Space Data Link Protocol,” CCSDS 232.0-B-2.

[RD14] “RTEMS POSIX User Manual,” Edition 4.11.

[RD15] “Space engineering - Telemetry and telecommand packet utilization,” ECSS-E-ST-70-41C.

[RD16] “The SPARC Architecture Manual,” sparcv8, SAV080SI9308, Version 8.

[RD17] “TM Synchronization and Channel Coding,” CCSDS 131.0-B-4.

# 2. System overview

## 2.1. Description

The Sirius OBC and Sirius TCM products are depicted in Figure 3.1 and Figure 3.2.

In addition to the external interfaces, the Sirius products also include both a debugger interface for downloading and debugging software applications and a JTAG interface for programming the FPGA during manufacturing.

The FPGA firmware implements a SoC built around a LEON3FT processor [RD3] running at a system frequency of 50 MHz and with the following key peripherals:

- Error manager - error handling, tracking and log of e.g. memory error detection.
- SDRAM controller - 64 MB data + 64 MB EDAC running @100MHz.
- Spacecraft Elapsed Timer (SCET) - including a PPS (Pulse Per Second) time synchronization interface for accurate time measurement with a resolution of 15 µs.
- SpaceWire - including a three-port SpaceWire router, for communication with external peripheral units.
- UARTs - RS422 and RS485 line drivers on the board with line driver mode set by software.
- GPIOs
- Watchdog - a fail-safe mechanism to prevent a system lockup
- System flash - 2 GB of EDAC-protected flash for storing boot images in multiple copies.
- Pulse command inputs - for reset to a specific software image
- NVRAM - for storage of metadata and other data that requires a large number of writes that shall survive loss of power.

For the Sirius TCM the following additional peripherals are included in the SoC:

- CCSDS - communications IP with RS422/LVDS interfaces for radio communication and an UMBI interface for communication with EGSE.
- Mass memory - 32GB of EDAC-protected NAND flash based, for storage of mission critical data.

For the Sirius OBC:

- An analog interface is included for external analog measurements.

The input power supply provided to the Sirius products shall be between +4.5 and +16 VDC. Power consumption is highly dependent on activities and peripheral loads and ranges from 1.2 W to 2 W.

## 2.2. OBC/TCM peripherals

Figure 2.1 shows an overview of the System-on-Chip (SoC) together with the peripheral circuitry of the Sirius OBC and Sirius TCM products. The color coding in the figure shows what parts are included for which products. The CPU is a LEON3FT.

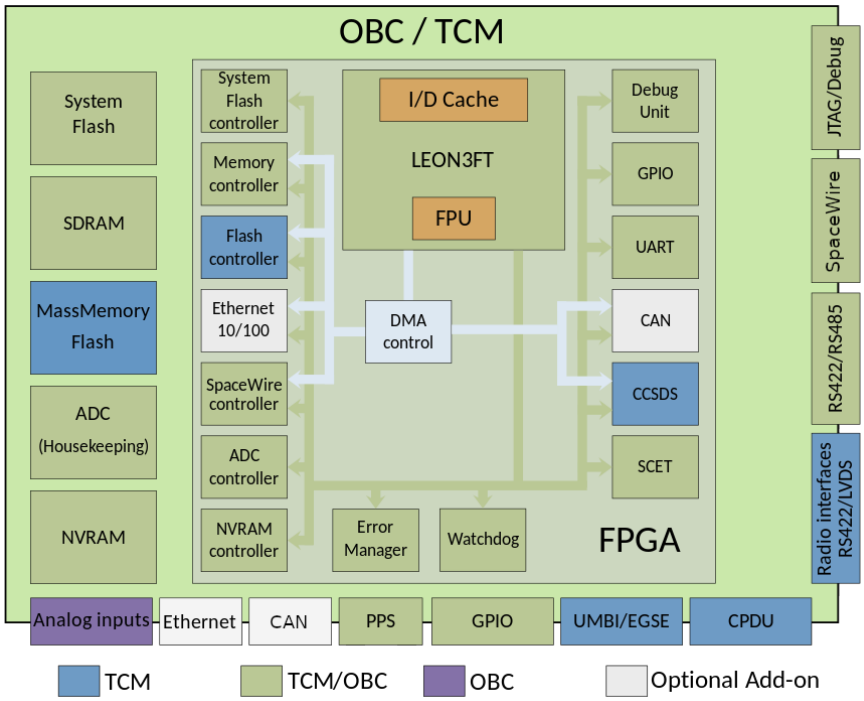



Figure 2.1 - The Sirius OBC / Sirius TCM SoC Overview

## 2.3. Fault tolerant design

The Sirius OBC and Sirius TCM are both fault tolerant by design to withstand the environmental loads that the modules are subjected to when used in space applications. The following error mitigation techniques are used.

- Continuous EDAC scrubbing of SDRAM data with at least 1 bit error correction and 2 bit error detection for each 16-bit word. Non-correctable errors cause a processor interrupt to allow the software to handle the error differently depending on in which section of the memory it appeared, unless the error appear in the execution path (see below).
- EDAC checking of instructions before execution and on data used in the

instruction (at least 1 bit error correction and 2 bit error detection as described in the previous point). Non-correctable errors cause automatic reboot.

- Parity checking of Instruction and Data caches when they are enabled. Errors cause a processor interrupt with a cache reload as the default error handling.
- Parity checking of peripheral FIFOs. Errors cause processor interrupt.
- EDAC checking on system flash with double bit error correction and extended bit error detection in combination with interleaving that corrects bursts with up to 16 bits in error.
- Triple Modular Redundancy (TMR) on all FPGA flip-flops
- All software stored in boot flash is, in addition to the EDAC protection of the flash data, encoded with a header for checksum and length. Each boot image is stored in three copies to allow for an automatic fallback option if the ECC and/or length check fails on one copy.
- Watchdog, tripping leads to automatic reboot of the device.
- Advanced Error Manager keeping the detected failures during reset/reboot for later analysis.

## 2.4. Usage and concept

This section describes the concept and normal intended use for the Sirius OBC and Sirius TCM in the default product configuration.

### 2.4.1. Combined setup

The OBC and TCM are intended to be used together to form the data processing and data handling portion of an on-board satellite system.

The OBC and TCM connect via spacewire, which provides the main interface for both commanding and data transfers.

Figure 2.2 shows an overview of an example setup with the OBC, TCM, a radio, and a pair of payloads in a suggested normal setup.

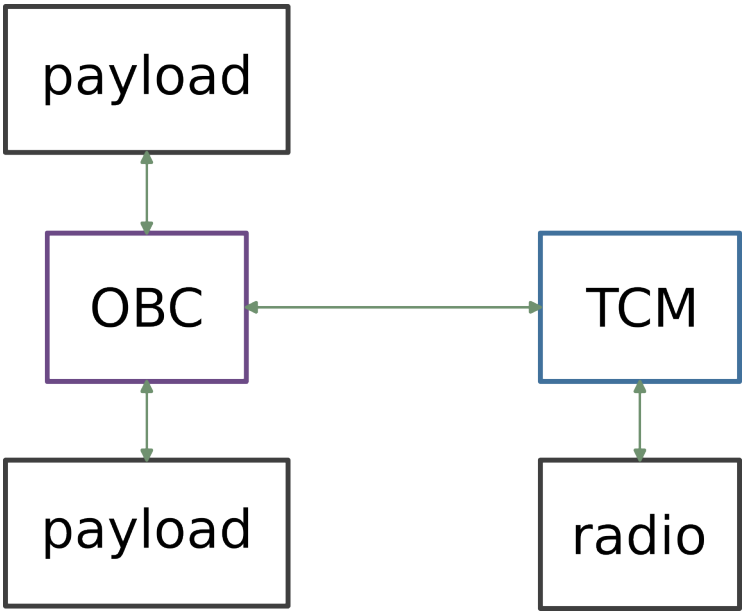



Figure 2.2 - Conceptual design of an on-board data handling system

### 2.4.2. OBC concept

The OBC provides a platform for hosting mission-specific flight software developed by the user, it is intended to handle the overall command and control handling of the on-board satellite system.

The OBC is also intended to handle the main data processing, and several interfaces for connecting to payloads and other on-board modules are provided.

The OBC Board Support Package (BSP) contains the RTEMS operation system along with drivers (see Chapter 5). for use when developing its software.

### 2.4.3. TCM concept

#### 2.4.3.1. Use with pre-programmed flight software

The TCM contains pre-programmed flight software [RD4]. This software is conceptually passive and relies on external command and control, intended to be provided by the OBC.

The TCM is intended to be connected to a radio and provide a TM/TC communications interface for use by the OBC. The TCM also provides a data storage interface which can be used by the OBC for both custom data and pre-prepared telemetry for later downlinking.

The TCM is configured by the user to fit the specific mission parameters [RD4].

#### 2.4.3.2. Use without pre-programmed flight software

The TCM may be used without the pre-programmed flight software and a TCM BSP is provided to allow the user to develop mission-specific software on the TCM, in a similar procedure as is normal for the OBC.

Using the TCM without the pre-programmed flight software is normally not the main intended use.

## 2.5. Manual chapters overview

Information on how to connect to the Leon3 processor to load/debug software can be found in Section 3.5. An introduction to how to build software for the boards is in Chapter 4.

Different aspects of how to use the System Flash and the board bootloader can be found in Chapters 3.6, 7, 5.12, 8 and 9.

Non-volatile RAM structure and usage is detailed Chapters 7 and 5.11.

How to use the different peripheral units in the System-on-Chip in an RTEMS application can be found in the subsections of Chapter 5.

# 3. Setup and operation

## 3.1. User prerequisites

The following hardware and software are needed for the setup and operation of the Sirius products.

**PC computer**

- 1 GB free space for installation (minimum)
- Debian 10 or Debian 11 64-bit with super user rights
- USB 2.0

**JTAG debugger**

- AAC JTAG debugger hardware including harness (104452)

**Recommended applications and software packages**

- Installed serial communication terminal, e.g. *gtkterm* or *minicom*
- GPG for encryption/decryption of files containing sensitive data
- Host build system, e.g. the debian package build-essential
- AAC toolchain for LEON3 with RTEMS 4.11
- BCC2 bare metal toolchain from Frontgrade Gaisler

**For FPGA update capabilities**

- Microsemi FlashPro Express v11.9 (www.microsemi.com/products/fpga-soc/design-resources/programming/flashpro#software)
- FlashPro5 programmer

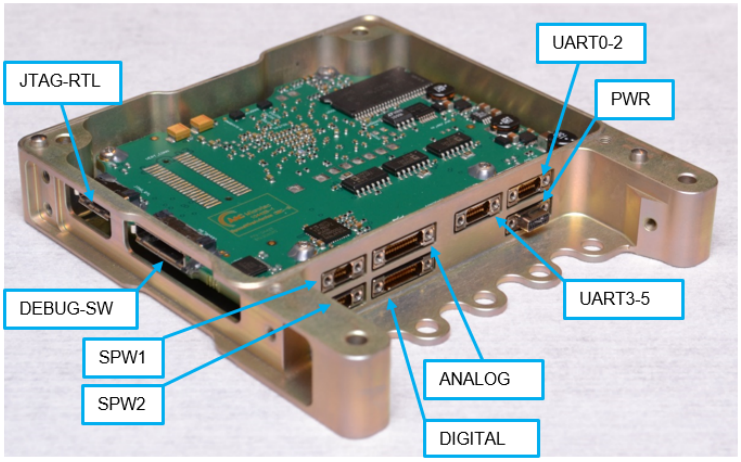



Figure 3.1 - Sirius OBC with connector naming

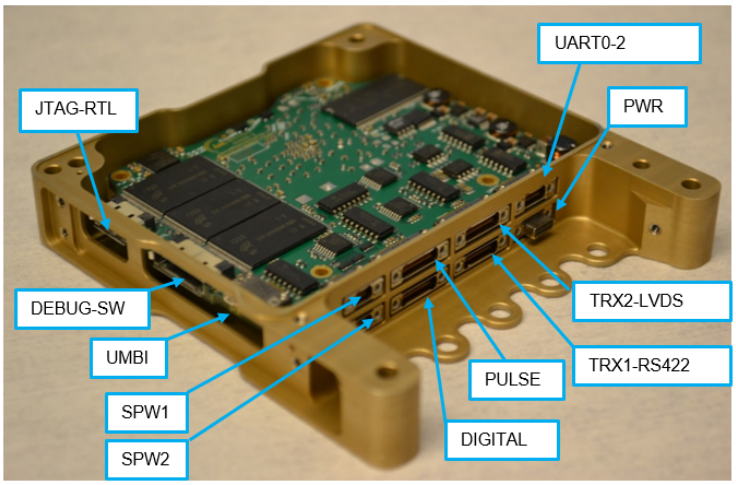



Figure 3.2 - Sirius TCM with connector naming

## 3.2. Connecting cables to the Sirius products

- All products and ingoing material shall be handled with care to prevent damage of any kind.
- ESD protection and other protective measures shall be considered. Handling should be performed according to applicable ESD requirement standards such as [RD5] or equivalent.
- Ensure that all mating connectors have the same zero reference (ground) before connecting.
- Connect the nano-D connector to the PWR connector with 4.5 - 16 V DC. The units will nominally draw about 260-300 mA @5V DC.
- The AAC debugger is mainly used for development of custom software for the Sirius OBC or Sirius TCM and has both a debug UART for monitoring and a JTAG

interface for debug capabilities. It is also used for programming an image to the system flash memory. For further information refer to Section 3.6. When it is to be used, connect the 104452 AAC Debugger to the DEBUG-SW connector. Connect the adapter USB-connector to the host PC.

- For FPGA updating only: Connect a FlashPro5 programmer to the JTAG-RTL connector using the 104470 FPGA programming cable assembly. For further information how to update the SoC refer to Chapter 12.
- For connecting the SpaceWire interface, connect the nano-D connector to connector SPW1 or SPW2.

For more detailed information about the connectors, see [RD2] .

## 3.3. Installation of toolchain

This chapter describes instructions for installing the AAC toolchains.

### 3.3.1. Supported Operating Systems

- Debian 10 64-bit
- Debian 11 64-bit

When installing Debian, we recommend using the “netinst” (network install) method. Images for installing are available via www.debian.org/releases/bullseye/debian-installer/

To install the toolchain below, a Debian package server mirror must be added, either in the installation procedure (also required during network install) or after installation.

On Debian 11 some packages required to build the BSP have been noted to not be installed by default. These need to be installed in order to configure and build:

```
sudo apt-get update
sudo apt-get install m4 autoconf
```

### 3.3.2. Installation Steps

#### 3.3.2.1. Store the key for the AAC package archive

In order to obtain the key to verify the packages, run the following commands

```
wget -O key.asc http://repo.aacmicrotec.com/archive/key.asc
gpg --dearmor --yes --output key.gpg key.asc
sudo mkdir -p /etc/apt/keyrings
```

```
sudo cp key.gpg /etc/apt/keyrings/aac-repo.gpg
```

#### 3.3.2.1.1. Old AAC package archive key in global trusted apt keyring

Previous toolchain instructions described installing the AAC package archive key in the global trusted apt keyring, this is no longer recommended practise in Debian. If the AAC package key has previously been added to the global trusted apt keyring, it can be removed via

```
sudo apt-key del "39D5 F87E 457C 8EA5 0DEE B148 FA81 C4F9 0257 7CF0"
```

where the argument string is the fingerprint of the AAC package archive key.

**NOTE**

If the key is deleted from the global trusted apt keyring it must instead be available in an individual keyring and the package archive source files must be rewritten to use it via the `signed-by` option, as described in Section 3.3.2.1 and Section 3.3.2.2.

### 3.3.2.2. Add the package archive as a source

In order to add the AAC package archive as a source; create a new repository source file and open it, for example via

```
sudoedit /etc/apt/sources.list.d/aac-repo.list
```

add the following lines to the file

```
1 deb [signed-by=/etc/apt/keyrings/aac-repo.gpg]
  http://repo.aacmicrotec.com/archive/ aac/
2 deb-src [signed-by=/etc/apt/keyrings/aac-repo.gpg]
  http://repo.aacmicrotec.com/archive/ aac/
```

then save and close the file.

### 3.3.2.3. Install the packages

In order to install the packages, run the following commands

```
sudo apt update
```

```
sudo apt install aac-sparc-toolchain
```

#### 3.3.2.4. Setup PATH

The toolchain PATH setup file can be sourced manually to use the toolchain in the current instance of the shell via

```
. /opt/aac-sparc/aac-path.sh
```

In order to make the toolchain available automatically in all instances of the bash shell, which is recommended for convenience; open the bash run commands file for the current user in an editor, for example via

```
editor ~/.bashrc
```

add the following lines to the end of the file

```
# AAC LEON3 toolchain PATH setup
if [ -f /opt/aac-sparc/aac-path.sh ]; then
. /opt/aac-sparc/aac-path.sh >/dev/null
fi
```

then save and close the file.

New instances of the bash shell will now automatically have access to the toolchain.

### 3.3.3. AAC toolchain is RTEMS-only

| | |
|---|---|
| **NOTE** | The AAC toolchain for LEON3 only supports RTEMS application development, for bare metal software the BCC2 toolchain from Cobham Gaisler is recommended (available at www.gaisler.com/index.php/downloads/compilers). |

## 3.4. Installing the Board Support Package (BSP)

Board support packages can be found at repo.aacmicrotec.com/bsp. Download the file `aac-<cpu>-<board>-bsp-<version>.tar.bz2`, where `<cpu>` is the processor type (currently only leon3); `<board>` is obc-s or tcm-s; and `<version>` is the wanted version number of that BSP; and extract it to a directory of your choice.

The extracted directory `aac-<cpu>-<board>-bsp` now contains the drivers for both bare-metal applications and RTEMS. See the included README and Section 4.1 for build instructions.

## 3.5. Deploying a Sirius application

### 3.5.1. Establish a debugger connection to the Sirius products

The Sirius products are shipped with debuggers that connect to a PC via USB and have two interfaces towards the board:

- One JTAG interface to the SoC debug unit.
- One debug UART to exchange information with the running software.

### 3.5.2. JTAG connection

To communicate with the debug unit in LEON3 based SoC's the program GRMON from Frontgrade Gaisler is used. This is not included in the AAC toolchain package as it requires a special license and thus needs to be installed separately.

GRMON3 Pro version 3.0.10 or higher is required. This can be downloaded from Gaisler at www.gaisler.com/index.php/downloads/debug-tools. For further instructions please refer to the GRMON3 manual, which is available at www.gaisler.com/doc/grmon3.pdf.

GRMON3 can be used as a standalone debug monitor to load and run applications, set breakpoints and read/write system registers and memory, and it is scriptable using TCL. It can also run as a server for the GNU Debugger if that interface is preferred.

### 3.5.3. Setup a serial terminal to the device debug UART

The device debug UART may be used as a debug interface for printf output etc.

A serial communication terminal such as minicom or gtkterm is necessary to communicate with the Sirius product, using these settings:

Baud rate: 115200
Data bits: 8
Stop bits: 1
Parity: None
Hardware flow control: Off

On a clean system with no other USB-to-serial devices connected, the serial port will appear as `/dev/ttyUSB1`. However, the numbering may change when other USB devices are connected, and the user must make sure to use the correct device number

to communicate to the board's debug UART.

On Debian, a more foolproof way of identifying the terminal to use is the by-id mechanism using the serial number of the debugger obtained in Section 3.5.4. When the AAC debugger is connected the system automatically creates named symbolic links to the device files under `/dev/serial/by-id`. The interface to use is `usb-AAC_Microtec_JTAG_Debugger_FTZ7QCMF-if01-port0`, where `FTZ7QCMF` is the serial number in this case. The debug UART is on `if01`, while `if00` is used for the JTAG interface (any serial device created for `if00` should disappear when a debug monitor is started).

### 3.5.4. Using multiple debuggers on the same PC

In order to use multiple debuggers connected to the same PC, each instance of run_aac_debugger.sh must be configured to connect to the specific debugger serial number and to use unique ports.

To determine the serial number for a specific device, run the following command before connecting the debugger:

```
sudo tail -f /var/log/kern.log
```

This initially prints the last 10 lines of the kernel log file, which can be ignored. When plugging in the debugger USB cable into the PC, this should produce new output similar to:

```
[363061.959120] usb 1-1.3.3.3: new full-speed USB device number 15 using
ehci_hcd
[363062.058152] usb 1-1.3.3.3: New USB device found, idVendor=0403,
idProduct=6010
[363062.058176] usb 1-1.3.3.3: New USB device strings: Mfr=1, Product=2,
SerialNumber=3
[363062.058194] usb 1-1.3.3.3: Product: JTAG Debugger
[363062.058207] usb 1-1.3.3.3: Manufacturer: AAC Microtec
[363062.058220] usb 1-1.3.3.3: SerialNumber: FTZ7QCMF
```

where `FTZ7QCMF` is the serial number for the debugger.

For GRMON3 the port to use for the GDB server needs to be unique. The default is `50001`.

For example, two debuggers with serial numbers `FTZ7QCMF` and `FTZ7IB10` can be setup via

```
run_aac_debugger.sh -s FTZ7QCMF -g 50001
run_aac_debugger.sh -s FTZ7IB10 -g 50002
```

Two instances of GDB can then be opened and connected to the different debuggers through the chosen ports.

### 3.5.5. Alternative USB library for GRMON

Some versions of GRMON have had issues communicating with the USB connected debugger hardware, particularly when dumping memory. This shows as error messages at the GRMON3 prompt noting “usb bulk write failed”, “usb bulk read failed” or similar. These come from the open source libftdi and libusb libraries included with GRMON. In case of such issues a workaround is to use the proprietary D2XX library from FTDI instead.

To install the library, download the D2XX driver package for linux from FTDI: ftdichip.com/drivers/d2xx-drivers/

The package contains a lot of examples and things needed to build applications that communicate with FTDI USB devices, but the only thing needed here is the file `libftd2xx.so.<version>`. This can be extracted and copied to a suitable directory on the computer running GRMON, for example `/usr/local/lib`. Then a symbolic link should be created in the same directory so that there appears to be a file without the version:

```
sudo ln -s libftd2xx.so.1.2.27 libftd2xx.so
```

GRMON can then be started with this library instead of the included open source libftdi:

```
LD_LIBRARY_PATH=/usr/local/lib /opt/grmon-pro-3.3.2/linux/bin64/grmon -v -abaud
115200 -ftdi d2xx -ftdigpio 0x08100000 -gdb 50001 -stack 0x04000000
```

To handle multiple debugger units connected to the same computer when using the D2XX library, the user can select the unit to use by serial number by adding the command line switch `-jtagserial FTZ7QCMF`, or alternatively listing the available debuggers using

```
LD_LIBRARY_PATH=/usr/local/lib /opt/grmon-pro-3.3.2/linux/bin64/grmon -ftdi d2xx
-jtaglist
```

and selecting the wanted unit using `-jtagcable <num>`.

### 3.5.6. Loading an application on LEON3

An application can either be loaded only to the board SDRAM, which is easier and typically used during the development stages, or to the system flash (see Section 3.6). In this manual it is done using GDB, but it could also be done using only GRMON (see sections 3.4.2 and 3.4.3 in the GRMON3 User’s Manual [RD6]). From GDB the user can also pass commands to GRMON by prefixing them with the GDB command `monitor`.

1. Start GDB with the following command from a shell to debug RTEMS executables:

   ```
   sparc-aac-rtems4.11-gdb
   ```

2. When GDB has opened successfully, connect to the hardware through the GRMON server using the GDB command target.

   ```
   target extended-remote localhost:50001
   ```

3. Specify the executable file for GDB to work with. Make sure the file is in ELF format.

   ```
   file <path/to/executable>
   ```

4. Transfer into the target RAM

   ```
   load
   ```

5. Start the application.

   ```
   run
   ```

### 3.5.7. Debugging software

Halting and reloading software via GRMON or GDB may leave peripheral units in an unknown state, and thus give unexpected behavior, especially if there is communication running on SpaceWire and UARTs. When working with software through the debugger it is good to start from a system reset, preferably with a very simple software in flash.

The Watchdog timer is enabled by default and can only be disabled when the debugger is connected. To avoid unexpected resets while debugging it is good to have a prepared command in GRMON or GDB to disable the Watchdog as soon as possible after software is halted. See Section 5.2 for more details about the Watchdog.

In GRMON: `wmem 0xCB000000 0x0`

In GDB: `set *(unsigned int) 0xCB000000 = 0`

A manual reset can be triggered through the Error Manager (see Section 5.2).

In GRMON: `wmem 0xC0000000 0xFFFFFFFF`

In GDB: `set *(unsigned int) 0xC0000000 = 0xFFFFFFFF`

If GRMON gives the error “CPU not in debug mode” when executing a command, that usually means that the board has reset, and the Debug Support Unit in the SoC is not in control of the CPU. To take back control the attach command is used.

In GRMON: `attach`

In GDB: `monitor attach`

This should be immediately followed by disabling the Watchdog to avoid losing the connection again.

## 3.6. Programming an application (boot image) to system flash

To have an application start automatically when the board is powered the application image must be programmed to the system flash. This is done by taking the boot image binary and building it into the NAND flash programming application. The NAND flash programming application is then uploaded to the target and started using GDB, as described in the previous section. The maximum recommended size for the boot image is 16 MB. The `nandflash_program` application can be found in the BSP.

The below instructions assume that the toolchain is in the PATH, see Section 3.3 for how to accomplish this.

1. Compile the boot image binary according to the rules for that program.
2. Ensure that this image is in a binary-only format and not ELF. This can be accomplished with the help of the GCC objcopy tool included in the toolchain:

   ```
   sparc-aac-rtems4.11-objcopy -O binary boot_image.elf boot_image.bin
   ```

3. See Section 3.4 for installing the BSP and enter

   ```
   cd path/to/bsp/aac-<cpu>-<board>-bsp/src/nandflash_program/src
   ```

4. Now, compile the nandflash-program application, bundling it together with the boot image binary.

   ```
   make nandflash-program.elf PROGRAMMINGFILE=/path/to/boot_image.bin
   ```

5. Load the nandflash-program.elf onto the target RAM with the help of GDB and execute it, see Section 3.5.6. The programmer application will output progress information on the debug UART.

## 3.7. Re-initialising the NVRAM

In some situations, it may be desirable to clear and re-initialise the NVRAM from scratch, for example if a test application has written data to the NVRAM which does not match the expected format for the system flash bad block table.

Clearing the NVRAM will cause loss of the following data, which should be read out, backed up, and written back after re-initialising if critical:

- Bad block markings for discovered bad blocks in the system flash (Both OBC and TCM), may degrade reliability if cleared.
- Bad block markings for discovered bad blocks in the mass memory (TCM with the TCM core application software), may degrade reliability if cleared.
- Ongoing operation markers for the mass memory handler (TCM with TCM core application), may cause partial loss of stored partition data if cleared.
- Internal write pointers for the mass memory handler (TCM with TCM core application), may case loss of start and end location in a completely full partition if cleared.

The following steps are required in order to clear and re-initialise the NVRAM:

1. Compile and run the nvram_clear application using the debugger. This application is located in the src/example/ directory in the OBC or TCM BSP; the steps for compiling it are described in Section 4.1. This will clear the NVRAM.
2. Program a boot image to the system flash as described in Section 3.6. This will initialize the system flash bad block table in the NVRAM.

# 4. Software development

The RTEMS OS is the recommended way to develop and deploy applications to the Sirius products.

The toolchain (see Section 3.3) provides RTEMS development tools with the `<arch>-aac-rtems4.11-` prefix, and the BSP provides drivers with the `_rtems` postfix for use with RTEMS. The BSP also provides RTEMS application code examples in the `src/example/` directory.

The RTEMS drivers are documented in Chapter 5 in this manual.

Bare-metal toolchain and bare-metal drivers in the BSP are also available, but these are currently not supported for general application development, and documentation for these drivers is not included in this manual.

## 4.1. RTEMS step-by-step compilation

### 4.1.1. Compiling the BSP and compiling an example

The BSP is supplied with an example of how to write an application for RTEMS and engage all the available drivers.

Please note that the toolchain described in Section 3.3 needs to be installed and the BSP unpacked as described in Section 3.4.

The following instructions detail how to build the RTEMS environment and a test application

1. Enter the BSP src directory
   `cd path/to/bsp/aac-<cpu>-<board>-bsp/src/`

2. Run make to build the RTEMS target
   make

3. Once the build is complete, the build target directory is librtems
4. Set the `RTEMS_MAKEFILE_PATH` environment variable to point to the librtems directory containing Makefile.inc:
   `export RTEMS_MAKEFILE_PATH=path/to/librtems/sparc-aac-rtems4.11/leon3/`

5. Enter the example directory and build the test application by issuing
   ```
   cd example
   make
   ```

Load the resulting application using the debugger according to the instructions in Section 3.5.

#### 4.1.2. Compiling the BSP with debug output removed

During development, debug output from the RTEMS drivers can be very useful for detecting errors. During flight, debug output is unlikely to be useful (it is expected that the debug UART will be disconnected) and may decrease performance in case of large amounts of warnings/errors.

The RTEMS BSP can be compiled without debug output by replacing the make command in step 2. above with instead:

```
make clean
make BSP_AAC_DISABLE_DEBUG_OUTPUT=y
```

(The make clean command is only required if the BSP has previously been compiled with a different configuration.)

## 4.2. RTEMS floating-point considerations

For LEON3, RTEMS saves the FPU (Floating Point Unit) register file and FSR (Floating Point Status Register) register across context switches and disables the FPU temporarily during interrupts to avoid that a faulty ISR (Interrupt Service Routine) thrashes the FPU state. If an ISR needs to use FPU it is responsible to save and restore the FPU context itself using the RTEMS API. Due to the SPARC ABI the OS only needs to save the FPU context on interrupts since the ABI states that FPU context is clobbered on function calls.

When creating RTEMS classic tasks the `RTEMS_FLOATING_POINT` option must be set if the task will execute FP instructions. Otherwise the CPU will generate a fp_disabled trap (trap type tt=`0x04`) on the first FP instruction executed by the task.

The `RTEMS Init()` task is by default configured without the `RTEMS_FLOATING_POINT` option. To enable `RTEMS_FLOATING_POINT` in the `Init()` task, the following configuration statement can be used:

```
#define CONFIGURE_INIT_TASK_ATTRIBUTES RTEMS_FLOATING_POINT
```

Note that the RTEMS BSPs for the Sirius products are built using the floating-point instructions. This means RTEMS libraries may contains floating point instructions which require the calling task to have a floating-point context (`RTEMS_FLOATING_POINT`) to avoid an exception.

For more information about floating-point usage in RTEMS, please refer to section 7.2.7 in [RD7]. For details about the floating-point unit in the LEON3 systems see [RD 3].

## 4.3. Software disclaimer of warranty

This source code is provided "as is" and without warranties as to performance or merchantability. The author and/or distributors of this source code may have made statements about this source code. Any such statements do not constitute warranties and shall not be relied on by the user in deciding whether to use this source code.

# 5. RTEMS

## 5.1. Introduction

This section presents the RTEMS drivers. Figure 5.1 shows a block diagram representing the driver functionality access via the RTEMS API.

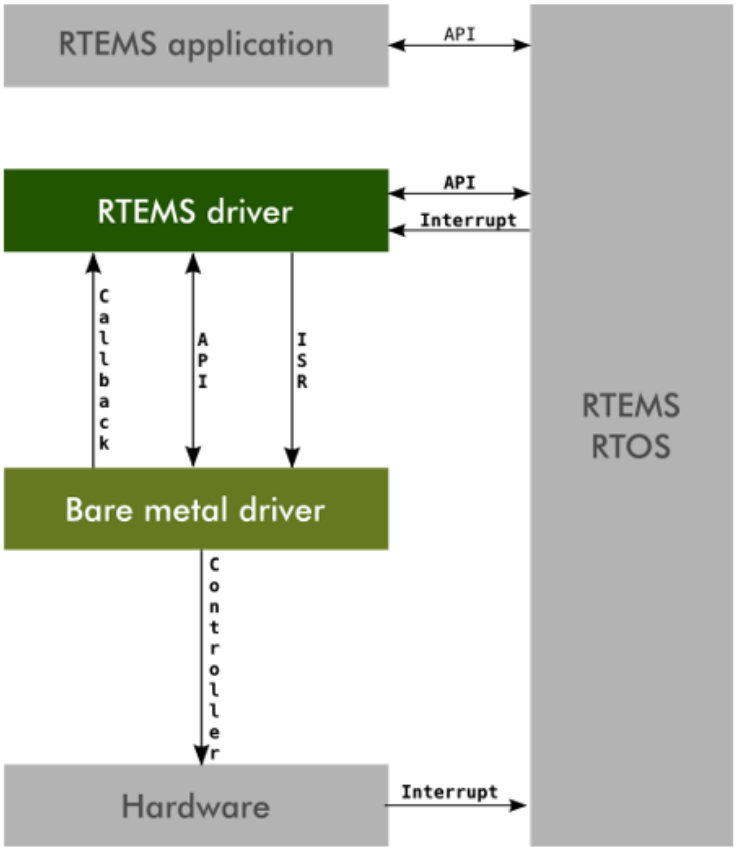



Figure 5.1 - Functionality acces via RTEMS API

## 5.2. Watchdog

### 5.2.1. Description

This section describes the driver as one utility for accessing the watchdog device. The watchdog is enabled from boot and cannot be disabled unless the debugger is connected. If the watchdog device file is not written to within a set time, it will trigger a reset of the board.

### 5.2.2. API

This API represents the driver interface from a user application's perspective for the RTEMS driver.

The driver functionality is accessed through RTEMS POSIX API for ease of use. In case of failure on a function call, the *errno* value is set for determining the cause.

| NOTE | The watchdog is enabled by default and can only be disabled if the debugger is connected. |
|---|---|

#### 5.2.2.1. Function: `int open(…)`

Opens access to the driver.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `filename` | `char *` | in | The absolute path to the file that is to be opened. Watchdog device is defined as `RTEMS_WATCHDOG_DEVICE_NAME` (/dev/watchdog) |
| `oflags` | `int` | in | A bitwise 'or' separated list of values that determine the method in which the file is to be opened (whether it should be read only, read/write). |

| Return value | Description |
|---|---|
| `>0` | A file descriptor for the device on success |
| `-1` | see *errno* values |
| **errno values** | |
| `ENFILE` | File descriptor limit reached |
| `ENOENT` | Invalid path |

#### 5.2.2.2. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |
| `-1` | see *errno* values |
| **errno values** | |
| `EBADF` | Invalid file descriptor |

#### 5.2.2.3. Function: `ssize_t write(…)`

Any data is accepted as a watchdog kick.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `buf` | `void *` | in | Character buffer to read data from |
| `nbytes` | `size_t` | in | Number of bytes to write |

| Return value | Description |
|---|---|
| `*` | n Number of bytes that were written. |
| `-1` | see *errno* values |
| **errno values** | |
| `EBADF` | Invalid file descriptor |
| `EINVAL` | Invalid `buf` argument (`NULL`) |

#### 5.2.2.4. Function: `int ioctl(…)`

Ioctl allows for disabling/enabling of the watchdog and setting of the timeout.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `cmd` | `int` | in | Command to send |
| `val` | `int` | in | Data to write |

| Command table | Val interpretation |
|---|---|
| `WATCHDOG_ENABLE_IOCTL` | `1` = Enables the watchdog (default)<br>`0` = Disables the watchdog<br><br>**NOTE:** It's only possible to disable the watchdog when the debugger is connected. |
| `WATCHDOG_SET_TIMEOUT_IOCTL` | `0` - `255` = Number of seconds until the watchdog barks |

| Return value | Description |
|---|---|
| `0` | Command executed successfully |
| `-1` | see *errno* values |
| **errno values** | |
| `EBADF` | Invalid file descriptor |
| `EINVAL` | Invalid data in `val` argument or invalid `cmd` argument |

### 5.2.3. Usage description

The following `#define` needs to be set by the user application to be able to use the watchdog:

- `CONFIGURE_APPLICATION_NEEDS_WDT_DRIVER`

The RTEMS driver must be opened before it can access the watchdog device. Once opened, all provided operations can be used as described in the RTEMS API defined in Section 5.2.2 and, if desired, the access can be closed when not needed.

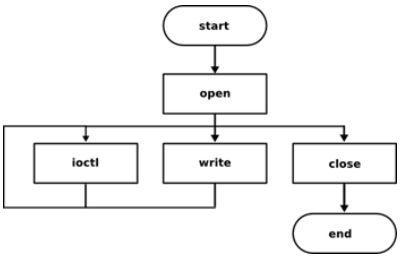



Figure 5.2 - RTEMS driver usage description

| NOTE | All calls to RTEMS driver are blocking calls. |
|---|---|

#### 5.2.3.1. RTEMS application example

In order to use the watchdog driver on the RTEMS environment, the following code structure is suggested to be used:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <bsp/wdt_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_WDT_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MAXIMUM_DRIVERS
#define CONFIGURE_MAXIMUM_TASKS                 2 /* Idle & Init */
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 1
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT
#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

rtems_task Init(rtems_task_argument ignored)
{
  int fd = open(RTEMS_WATCHDOG_DEVICE_NAME, O_WRONLY);
  ioctl(fd, WATCHDOG_ENABLE_IOCTL, WATCHDOG_DISABLE);
  ioctl(fd, WATCHDOG_SET_TIMEOUT_IOCTL, 10);
  ioctl(fd, WATCHDOG_ENABLE_IOCTL, WATCHDOG_ENABLE);
  while (1) {
    sleep(9);
    const unsigned char payload = WATCHDOG_KICK;
    write(fd, &payload, sizeof(payload));
  }
}
```

- Inclusion of `<fcntl.h> and <unistd.h>` are required for using the POSIX functions open, close, lseek, read and write.
- Inclusion of `<errno.h>` is required for retrieving error values on failures.
- Inclusion of `<bsp/wdt_rtems.h>` is required for accessing watchdog device name `RTEMS_WATCHDOG_DEVICE_NAME`.

If the application is run directly via GDB (not via the bootrom), `CONFIGURE_APPLICATION_NEEDS_ERROR_MANAGER_DRIVER` must be defined in order to initialise the error manager and enable board reset on watchdog timeout.

## 5.3. Error Manager

### 5.3.1. Description

The error manager driver is a software abstraction layer meant to simplify the usage of the error manager for the application writer.

This section describes the driver as one utility for accessing the error manager device.

### 5.3.2. API

This API represents the driver interface from a user application's perspective for the RTEMS driver.

The driver functionality is accessed through the RTEMS POSIX API for ease of usage. In case of failure on a function call, the `errno` value is set for determining the cause.

The error manager driver does not support writing nor reading to the device file. Instead, register accesses are performed using ioctls.

The driver exposes a message queue for receiving interrupt-driven events such as some non-fatal uncorrectable errors found in the CPU working memory.

#### 5.3.2.1. Struct: `errman_latest_reset_info_t`

| Type | Name | Purpose |
|---|---|---|
| `uint32_t` | `scet_seconds` | The SCET seconds at time of latest reset. Zero following a hard reset or power-up. |
| `uint16_t` | `scet_subseconds` | The SCET subseconds at time of latest reset. Zero following a hard reset or power-up. |
| `uint8_t` | `cause` | Latest cause of reset, encoded as:<br><br>`0x0` - Power-Up<br>`0x1` - Watchdog<br>`0x2` - Manual (SW initiated)<br>`0x3` - CPDU (safe image)<br>`0x4` - CPDU (default image)<br>`0x5` - Uncorrectable error found by CPU |
| `uint8_t` | RESERVED | |

#### 5.3.2.2. Function: `int open(…)`

Opens access to the device. The device driver allows multiple readers but only one

writer at a time.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `filename` | `char*` | in | The absolute path to the file that is to be opened. Error manager device is defined as RTEMS_ERRMAN_DEVICE_NAME. |
| `oflags` | `int` | in | Specifies one of the access modes in the following table. |

| Access mode | Description |
|---|---|
| `O_RDONLY` | Open for reading only |
| `O_WRONLY` | Open writing only |
| `O_RDWR` | Open for reading and writing |

| Return value | Description |
|---|---|
| `fd` | A file descriptor for the device on success |
| `-1` | see *errno* values |
| **errno values** | |
| `EALREADY` | Device already opened |

#### 5.3.2.3. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |

#### 5.3.2.4. Function: `int ioctl(…)`

Ioctl allows for disabling/enabling functionality of the error manager, setting of the timeout and reading out counter values.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |

| Argument | Type | Direction | Description |
|---|---|---|---|
| `cmd` | `uint32_t` | in | Command to send |
| `val` | `uint32_t` / `uint32_t*` | in / out | Value to write or a pointer to a buffer where data will be written |

| Command table | Val type | Description |
|---|---|---|
| `ERRMAN_GET_SR_IOCTL` | `uint32_t*` | Get the status register. This register is defined in Table 5.1 |
| `ERRMAN_GET_CF_IOCTL` | `uint32_t*` | Gets the Carry flag register. This register is defined in Table 5.2 |
| `ERRMAN_GET_SELFW_IOCTL` | `uint32_t*` | Points to which boot firmware that will be loaded and executed upon system reboot.<br><br>`0x0`: Programmable FW from Power on<br>`0x1`: Programmable FW, Backup copy<br>`0x2`: Programmable FW, Backup copy<br>`0x3`: Safe FW<br>`0x4`: Safe FW, Backup copy<br>`0x5`: Safe FW, Backup copy |
| `ERRMAN_GET_RUNFW_IOCTL` | `uint32_t*` | Gets the currently running firmware<br><br>`0x0`: Programmable FW from Power on<br>`0x1`: Programmable FW, Backup copy<br>`0x2`: Programmable FW, Backup copy<br>`0x3`: Safe FW<br>`0x4`: Safe FW, Backup copy<br>`0x5`: Safe FW, Backup copy |
| `ERRMAN_GET_SCRUBBER_IOCTL` | `uint32_t*` | Gets the state of the memory scrubber.<br><br>`0` = Scrubber is disabled<br>1 = Scrubber is enabled. |
| ~~`ERRMAN_GET_RESET_ENABLE_IOCTL`~~ | ~~`uint32_t*`~~ | ~~**Gets the reset enable state.**~~<br><br>~~**`0` = Soft reset is disabled.**~~<br>~~**1 = Soft reset is enabled**~~<br><br>The command is deprecated and might be removed in future releases. |

| Command table | Val type | Description |
|---|---|---|
| ERRMAN_GET_WDT_ERRCNT_IOCTL | uint32_t* | Gets the count of watchdog resets. This register can store a value up to 15 and then wraps.<br><br>After a wrap, the corresponding carry-flag bit is set in the carry flag register, see Table 5.2. |
| ERRMAN_GET_EDAC_SINGLE_ERRCNT_IOCTL | uint32_t* | Gets the error counts of the correctable errors in the CPU working memory. See Table 5.3 for interpretation of the register.<br><br>After a wrap to any of the error counters, the corresponding carry-flag bit is set in the carry-flag register, see Table 5.2 |
| ERRMAN_GET_EDAC_MULTI_ERRCNT_IOCTL | uint32_t* | Gets the error counts of the uncorrectable errors in the CPU working memory. See Table 5.4 for interpretation of the register.<br><br>After a wrap to any of the error counters, the corresponding carry-flag bit is set in the carry-flag register, see Table 5.2 |
| ~~ERRMAN_GET_CPU_PARITY_ERRCNT_IOCTL~~ | ~~uint32_t*~~ | **~~Gets the CPU Parity error count register.~~**<br><br>Always reads 0. The command is deprecated and might be removed in future releases. |
| ERRMAN_GET_SYS_SINGLE_ERRCNT_IOCTL | uint32_t* | Gets the error count of the correctable errors in the System Flash. This register is 4 bit wide and will wrap upon overflow. |
| ERRMAN_GET_SYS_MULTI_ERRCNT_IOCTL | uint32_t* | Gets the error count of the uncorrectable errors in the System Flash. This register is 4 bit wide and will wrap upon overflow. |
| ERRMAN_GET_MMU_SINGLE_ERRCNT_IOCTL | uint32_t* | Gets the error count of the correctable errors in the Mass Memory. This register is 4 bit wide and will wrap upon overflow. |

| Command table | Val type | Description |
|---|---|---|
| ERRMAN_GET_MMU_MULTI_ERRCNT_IOCTL | uint32_t* | Gets the error count of the uncorrectable errors in the Mass Memory. This register is 4 bit wide and will wrap upon overflow. |
| ERRMAN_GET_NVRAM_SINGLE_ERRCNT_IOCTL | uint32_t* | Gets the error count of the correctable single-bit errors in the NVRAM. This register is 4 bit wide and will wrap upon overflow |
| ERRMAN_GET_NVRAM_DOUBLE_ERRCNT_IOCTL | uint32_t* | Gets the error count of the correctable double-bit errors in the NVRAM. This register is 4 bit wide and will wrap upon overflow |
| ERRMAN_GET_NVRAM_MULTI_ERRCNT_IOCTL | uint32_t* | Gets the error count of the uncorrectable errors in the NVRAM. This register is 4 bit wide and will wrap upon overflow |
| ERRMAN_GET_LAST_RESET_CAUSE_IOCTL | errman_latest_reset_info_t* | Gets the last reset cause and its corresponding timestamp. |
| ERRMAN_GET_LATEST_BOOT_STATUS_IOCTL | uint32_t* | Gets the latest boot status.<br><br>See Table 5.5 for details. |
| ERRMAN_SET_SR_IOCTL | uint32_t | Sets the status register. See Table 5.1 for register definition. |
| ERRMAN_SET_CF_IOCTL | uint32_t | Sets the carry flag register, see Table 5.2 for register definition. |
| ERRMAN_SET_SELFW_IOCTL | uint32_t | Sets the next boot firmware.<br><br>0x0: Programmable FW from Power on<br>0x1: Programmable FW, Backup copy<br>0x2: Programmable FW, Backup copy<br>0x3: Safe FW<br>0x4: Safe FW, Backup copy<br>0x5: Safe FW, Backup copy |
| ERRMAN_RESET_SYSTEM_IOCTL | uint32_t | Performs a software reset (value ignored.) |

| Command table | Val type | Description |
|---|---|---|
| ERRMAN_SET_SCRUBBER_IOCTL | uint32_t | Sets the state of the memory scrubber.<br><br>1 = On,<br>0 = Off.<br><br>The scrubber is a vital part of keeping the SDRAM free from errors. |
| ~~**ERRMAN_SET_RESET_ENABLE_IOCTL**~~ | ~~**uint32_t***~~ | ~~**Sets the reset enable state.**~~<br><br>~~**0 = Soft reset is disabled.**~~<br>~~**1 = Soft reset is enabled**~~<br><br>The command is deprecated and might be removed in future releases. |
| ERRMAN_SET_WDT_ERRCNT_IOCTL | uint32_t | Sets the error count of watchdog resets. The counter width is 4 bits i. e. 15 is the maximum value that can be written. |
| ERRMAN_SET_EDAC_SINGLE_ERRCNT_IOCTL | uint32_t | Sets the error count of the correctable errors in the CPU working memory. See Table 5.3 for register definition. |
| ERRMAN_SET_EDAC_MULTI_ERRCNT_IOCTL | uint32_t | Sets the error count of the uncorrectable errors in the CPU working memory. See Table 5.4 for register definitions. |
| ~~**ERRMAN_SET_CPU_PARITY_ERRCNT_IOCTL**~~ | ~~**uint32_t***~~ | ~~**Sets the CPU Parity error count register.**~~<br><br>The command is deprecated and might be removed in future releases. |
| ERRMAN_SET_SYS_SINGLE_ERRCNT_IOCTL | uint32_t | Sets the error count of the correctable errors in the System Flash. This register is 4 bit wide. |
| ERRMAN_SET_SYS_MULTI_ERRCNT_IOCTL | uint32_t | Sets the error count of the uncorrectable errors in the System Flash. This register is 4 bit wide. |
| ERRMAN_SET_MMU_SINGLE_ERRCNT_IOCTL | uint32_t | Sets the error count of the correctable errors in the Mass Memory. This register is 4 bit wide. |

| Command table | Val type | Description |
|---|---|---|
| ERRMAN_SET_MMU_MULTI_ERRCNT_IOCTL | uint32_t | Sets the error count of the uncorrectable errors in the Mass Memory. This register is 4 bit wide. |
| ERRMAN_SET_NVRAM_SINGLE_ERRCNT_IOCTL | uint32_t | Sets the error count of the correctable single-bit errors in the NVRAM. This register is 4 bit wide |
| ERRMAN_SET_NVRAM_DOUBLE_ERRCNT_IOCTL | uint32_t | Sets the error count of the correctable double-errors in the NVRAM. This register is 4 bit wide |
| ERRMAN_SET_NVRAM_MULTI_ERRCNT_IOCTL | uint32_t | Sets the error count of the uncorrectable errors in the NVRAM. This register is 4 bit wide |

| Return value | Description |
|---|---|
| 0 | Command executed successfully |
| -1 | See *errno* values |
| **errno values** | |
| EBADF | File descriptor not opened for writing. |
| EINVAL | Invalid IOCTL command. |

#### 5.3.2.4.1. Status register

Table 5.1 - Status register

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 31:23 | RESERVED | | |
| 22:20 | ERRMAN_RESET_CAUSE | R | Cause of reset encoded as:<br><br>0x0 – Power-Up<br>0x1 – Watchdog<br>0x2 – Manual (SW initiated)<br>0x3 – CPDU (safe image)<br>0x4 – CPDU (default image)<br>0x5 - Uncorrectable error found by CPU |
| 19 | RESERVED | | |

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 18 | ERRMAN_MNVERRFLG | R/W | An uncorrectable error has been detected in the NVRAM.<br><br>Clear flag by writing a 1. |
| 17 | ERRMAN_DNVERRFLG | R/W | A correctable double-bit error has been detected in the NVRAM.<br><br>Clear flag by writing a 1. |
| 16 | ERRMAN_SNVERRFLG | R/W | A correctable single-bit error has been detected in the NVRAM.<br><br>Clear flag by writing a 1. |
| 15 | ERRMAN_MMMERRFLG | R/W | An uncorrectable error in the Mass Memory has been detected.<br><br>Clear flag by writing a 1. |
| 14 | ERRMAN_SMMERRFLG | R/W | A correctable error in the Mass Memory has been detected.<br><br>Clear flag by writing a 1. |
| 13 | ERRMAN_MSYSERRFLG | R/W | An uncorrectable error in the System Flash has been detected.<br><br>Clear flag by writing a 1. |
| 12 | ERRMAN_SSYSERRFLG | R/W | A correctable error in the System Flash has been detected.<br><br>Clear flag by writing a 1. |
| 11 | ERRMAN_PULSEFLG | R/W | A pulse command has been received and caused a reset.<br><br>Clear flag by writing a 1 |
| 10 | RESERVED | | |
| 9 | ERRMAN_MEMCLR | R | This signal is set from the scrubber unit function in the memory controller. This bit is set when memory has been cleared after reset. |
| 8 | RESERVED | | |

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 7 | RESERVED | | |
| 6 | ERRMAN_MEOTHFLG | R/W | An uncorrectable error has been detected in the CPU working memory, by the scrubber or as part of a DMA access.<br><br>Clear flag by writing a 1 |
| 5 | ERRMAN_SEOTHFLG | R/W | A correctable error has been detected in the CPU working memory, by the scrubber or as part of a DMA access. The error was corrected before returning the data, but it has not been corrected in memory.<br><br>Clear flag by writing a 1. |
| 4 | ERRMAN_MECRIFLG | R/W | An uncorrectable error has been detected by the CPU when reading instructions or data from the CPU working memory.<br><br>Clear flag by writing a 1. |
| 3 | ERRMAN_SECRIFLG | R/W | A correctable error has been detected by the CPU when reading instructions or data from the CPU working memory. The error was corrected before returning the data, but it has not been corrected in memory.<br><br>Clear flag by writing a 1 |
| 2 | ERRMAN_WDTFLG | R/W | A watchdog timer reset has been detected.<br><br>Clear flag by writing a 1 |
| 1 | ERRMAN_RFLG | R/W | A manual reset has been detected.<br><br>Clear flag by writing a 1 |

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 0 | ERRMAN_IFLAG | R/W | Error Manager Interrupt Flag:<br><br>0 = No interrupt pending<br>1 = Interrupt pending<br><br>Clear flag by writing a 1 |

#### 5.3.2.4.2. Carry flag register

Table 5.2 - Carry flag register

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 31:19 | RESERVED | | |
| 18 | ERRMAN_MNVERRCFLG | R/W | Carry flag set when an overflow has occurred to the error counter for uncorrectable errors in the NVRAM<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 17 | ERRMAN_DNVERRCFLG | R/W | Carry flag set when an overflow has occurred to the error counter for correctable double-bit errors in the NVRAM<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 16 | ERRMAN_SNVERRCFLG | R/W | Carry flag set when an overflow has occurred to the error counter for correctable single-bit errors in the NVRAM<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 15 | ERRMAN_MMMERRCFLG | R/W | Carry flag set when an overflow has occurred to the error counter for uncorrectable errors in the Mass Memory<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 14 | ERRMAN_SMMERRCFLG | R/W | Carry flag set when an overflow has occurred to the error counter for correctable errors in the Mass Memory<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 13 | ERRMAN_MSYSERRCFLG | R/W | Carry flag set when an overflow has occurred to the error counter for uncorrectable errors in the System Flash<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 12 | ERRMAN_SSYSERRCFLG | R/W | Carry flag set when an overflow has occurred to the error counter for correctable errors in the System Flash<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 11:7 | RESERVED | | |

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 6 | ERRMAN_MEOFLG | R/W | Carry flag set when an overflow has occurred to the error counter for uncorrectable errors found during scrubbing, or through a DMA access to, the CPU working memory<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing 1) |
| 5 | ERRMAN_SEOFLG | R/W | Carry flag set when an overflow has occurred to the error counter for correctable errors found during scrubbing, or through a DMA access to, the CPU working memory<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 4 | ERRMAN_MECFLG | R/W | Carry flag set when an overflow has occurred to the error counter for uncorrectable errors found during CPU access to working memory<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing 1) |
| 3 | ERRMAN_SECFLG | R/W | Carry flag set when an overflow has occurred to the error counter for correctable errors found during CPU access to working memory<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 2 | ERRMAN_WDTCFLG | R/W | Carry flag set when an overflow has occurred to the counter for watchdog resets<br><br>0 – No CF set<br>1 – Counter overflow (Cleared by writing a 1) |
| 1:0 | RESERVED | - | |

#### 5.3.2.4.3. Register for correctable errors in CPU working memory

Table 5.3 - Register for correctable errors in CPU working memory

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 31:20 | RESERVED | - | |
| 19:16 | ERRMAN_SENOCNT_SDRAM | R/W | Error counter for correctable errors found by the scrubber, or through a DMA access, in CPU working memory |
| 15:4 | RESERVED | - | |
| 3:0 | ERRMAN_SECRICNT_SDRAM | R/W | Error counter for correctable errors found during CPU access to working memory |

#### 5.3.2.4.4. Register for uncorrectable errors in CPU working memory

Table 5.4 - Register for uncorrectable errors in CPU working memory

| Bit pos. | Name | Direction | Description |
|---|---|---|---|
| 31:20 | RESERVED | - | |
| 19:16 | ERRMAN_MENOCNT | R/W | Error counter for uncorrectable errors found by the scrubber, or through a DMA access, in CPU working memory |
| 15:4 | RESERVED | - | |
| 3:0 | ERRMAN_MECRICNT | R/W | Error counter for uncorrectable errors found during CPU access to working memory |

#### 5.3.2.4.5. Latest boot status register

Indicates the status of the latest failed boot (if any, otherwise latest successful boot). Will be cleared upon read. The format is defined by the bootrom but is reproduced here for convenience.

Table 5.5 - Latest boot status register

| Bit pos. | Description |
|---|---|
| 31:28 | The first SW image in the current boot sequence which failed to boot. If none failed to boot, the current successfully booted SW image.<br><br>0x0 – Updated image copy #3<br>0x1 – Updated image copy #2<br>0x2 – Updated image copy #1<br>0x3 – Safe image copy #3<br>0x4 – Safe image copy #2<br>0x5 – Safe image copy #1 |
| 27:8 | Reserved |
| 7:0 | Latest boot step successfully passed for the given SW image. If an SW image failed to boot, the subsequent step is the step which failed.<br><br>0x01 – Init<br>0x02 – Init timer<br>0x03 – Init UART<br>0x04 – Read SoC info<br>0x05 – Wait for scrubber<br>0x06 – Read bad-block table<br>0x07 – Set image<br>0x08 – Check bad-block table<br>0x09 – Get SCET before load<br>0x0A – Init sysflash<br>0x0B – Load image<br>0x0C – Compute load time<br>0x0D – Verify checksum<br>0x0E – Handover to boot image |

For example:

- 0x0000000E indicates a successful boot of updated image copy #3.
- 0x30000005 indicates a failed boot of safe image copy #3, where an error occurred during the read of the bad block table.

### 5.3.3. Usage description

The following `#define` needs to be set by the user application to be able to use the error manager:

- `CONFIGURE_APPLICATION_NEEDS_ERROR_MANAGER_DRIVER`

By defining this as part of RTEMS configuration, the driver will automatically be initialised at boot up.

The RTEMS driver must be opened before it can access the error manager device. Once opened, all provided operations can be used as described in the RTEMS API defined in Section 5.3.2. And, if desired, the access can be closed when not needed.

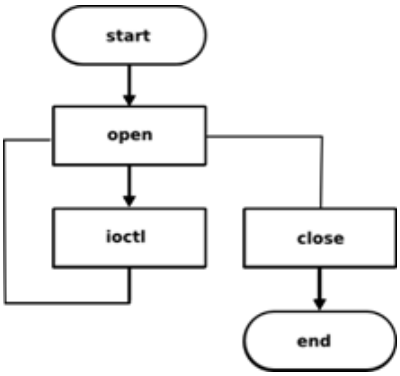



Figure 5.3 - RTEMS driver usage description

#### 5.3.3.1. Interrupt message queue

The error manager RTEMS driver exposes a message queue service which can be subscribed to. The name of the queue is "'E', 'M', 'G', 'R'".

A subscriber must inspect the message according to the following table to determine whether to take action or not. Multiple subscribers are allowed, and all subscribers will be notified upon a message.

| Message | Description |
|---|---|
| `ERRMAN_IRQ_EDAC_MULTIPLE_ERR_OTHER` | An uncorrectable error has been detected in the CPU working memory, by the scrubber or as part of a DMA access. |

#### 5.3.3.2. RTEMS application example

In order to use the error manager driver on RTEMS environment, the following code structure is suggested to be used:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <bsp/error_manager_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_ERROR_MANAGER_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER

#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 30
#define CONFIGURE_MAXIMUM_DRIVERS                10
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS          20
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES 20

#define CONFIGURE_INIT
#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

rtems_task Init(rtems_task_argument ignored)
{
  int fd;
  uint32_t status_register;

  fd = open(RTEMS_ERRMAN_DEVICE_NAME, O_RDONLY);

  /* Get the status register */
  ioctl(fd, ERRMAN_GET_SR_IOCTL, &status_register);
  /* Previous watch dog timer reset detected? */
  if (status_register & ERRMAN_WDTFLAG) {
    printf("Watchdog barked.\n");
  } else {
    printf("Watchdog did not bark.\n");
  }
}
```

- Inclusion of `<fcntl.h>` and `<unistd.h>` are required for using the POSIX functions: open, close, ioctl.
- Inclusion of `<errno.h>` is required for retrieving error values on failures.
- Inclusion of `<bsp/error_manager.h>` is required for accessing error manager device name `RTEMS_ERROR_MANAGER_DEVICE_NAME`.

#### 5.3.4. Limitations

Many of the error mechanisms are currently unverifiable outside of radiation testing due to the lack of mechanisms of injecting errors in this release.

## 5.4. SCET

### 5.4.1. Description

The scet driver is a software abstraction layer meant to simplify the usage of the scet for the application writer. This section describes the driver as one utility for accessing the scet device.

#### 5.4.1.1. Overview

The main purpose of the SCET IP and driver is to track the time since power on and to act as a source of timestamps. It is designed to be included in several different units in a system and for time synchronization between these units. Each SCET have the ability to receive and/or transmit PPS signals. The SCET has also been enhanced with general-purpose triggers and PPS signaling. Figure 5.4 shows an overview of SCET. It is shortly described below the figure.

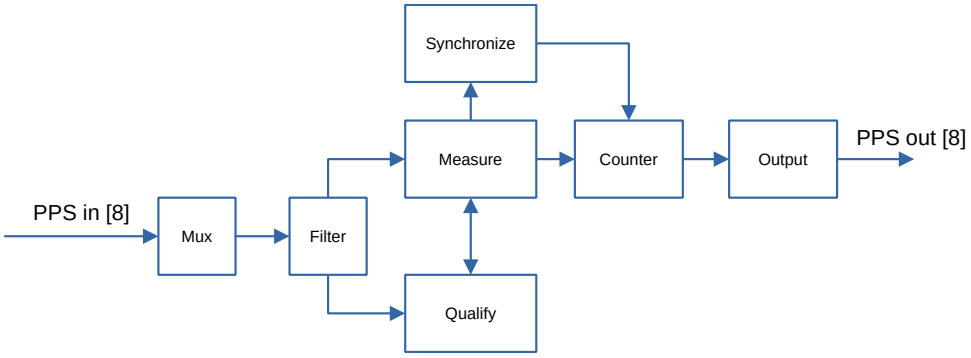



Figure 5.4 - A simplified block diagram of the SCET.

The *mux* allows the selection of one of several PPS inputs to use as synchronization reference (Section 5.2.2.4). The *filter* is configured to only allow PPS input pulses of 1 µs or longer (Section 5.4.1.6). The *measure* component measures the interval of the incoming PPS in order to be able to generate an output of the same interval, and to qualify the PPS to a user-defined window. *Qualify* performs this qualification by evaluating the incoming PPS pulse in relation to this window. *Synchronize* makes sure to, once, synchronize the internal timing to the incoming PPS and to align the PPS output with the PPS input. *Counter* keeps track of the internal timing based on the reference interval measured, and *output* outputs a PPS of length 1 ms (Section 5.4.1.7) on the enabled PPS outputs (Section 5.2.2.4).

The SCET counts in seconds and subseconds, with a subsecond being one $2^{-16}$th of a second, roughly equivalent to 15.3 µs.

#### 5.4.1.2. Timing modes

The SCET can either be *free running* or synchronize to one of several input PPS signals. In the free-running (or internal) mode, the SCET does not synchronize to any external signal. It simply keeps track of the *relative* time since power on without correlation with anything else.

In external mode, the SCET attempts to synchronize the internal time to one of several external PPS signals. When it does so, it will also monitor this PPS signal to make sure it arrives as expected within a user-set window. If synchronization to this input PPS is lost (the PPS arrives *ahead* of the window, or the window *expires*), it will fall back to internal mode and output PPS signals (if configured) with the latest measured PPS interval. This requires software interaction to resynchronize to the incoming PPS pulse, possibly by qualifying a number of PPS pulses and then switching back to external mode. The PPS monitoring will issue messages on the SCET message queue in RTEMS (Section 5.4.2.6) to notify the application if the PPS is qualified or not, but also if the window expired.

The synchronization of the internal *relative* time (start/end of a second) is based on an incoming PPS pulse that nominally arrives once per second. Once synchronized, the SCET supports adjusting the current seconds so as to also get an understanding of the *absolute* time (Section 5.2.2.3). To increase the resolution of this time, the concept of subseconds is used. The reference of how long a subsecond is depends on how long the actual second is, as determined by this external PPS signal, which is subjected to jitter. There are two different methods to handle this.

1. Truncate the count of a second (wrap subseconds earlier than $2^{16}$ - 1) or extend the count of a second (wrap subseconds later than $2^{16}$ - 1, which reduces resolution).
2. Adjust the length of a subsecond such that there are always $2^{16}$ subseconds in a second.

The second example is the method implemented, which requires continuous measurement of the incoming PPS interval as we at the start of the internal second need to know the length of it. This also has the effect of delaying reaction of PPS out to changes in PPS in, which causes a constant offset (with a constant incoming interval) in the alignment. A varying incoming interval (e.g. due to jitter) will however cause a kind of oscillation in the alignment offset. This offset is believed to be limited to ± 4J, where J is the cycle-to-cycle jitter. An example is illustrated in Figure 5.5.

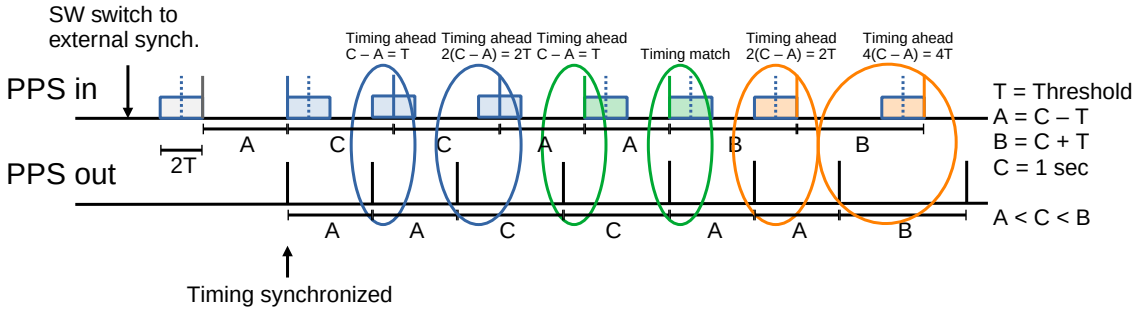



Figure 5.5 - Illustration of how the input PPS jitter affects the output PPS alignment.

#### 5.4.1.3. Threshold

In order to synchronize to an external PPS signal the PPS threshold needs to be configured. This sets the tolerable limits for the variation of the PPS input. An incoming PPS will either be located in this window (*qualified*) or outside this window (*unqualified*). In addition, the window can also expire causing an *expired* notification (Section 5.4.2.6). Note that these messages are generated independently of the actual mode configured, and that the PPS threshold is also used before attempting to synchronize to the external PPS. Figure 5.6 illustrates how to interpret the PPS threshold.

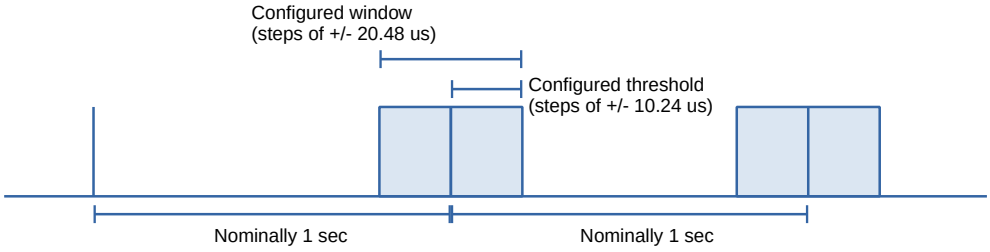



Figure 5.6 - Illustration of how to interpret the PPS threshold.

The qualification window resulting from different threshold values is show in Table 5.6.

Table 5.6 - Threshold values and resulting qualification window

| Value | Window size |
|---|---|
| `0` (default) | ±10.24µs |
| 1 | ±20.48µs |
| *N* | ±(1 + *N*)10.24µs |
| 65535 | ~ ±0.67s |

#### 5.4.1.4. Synchronization

In order to ensure that the PPS input is stable it should be verified that a sufficient number of consecutive qualified interrupts (without unqualified or expired in between) have been observed/received. Consider the example given in Figure 5.7, which is further explained below it.

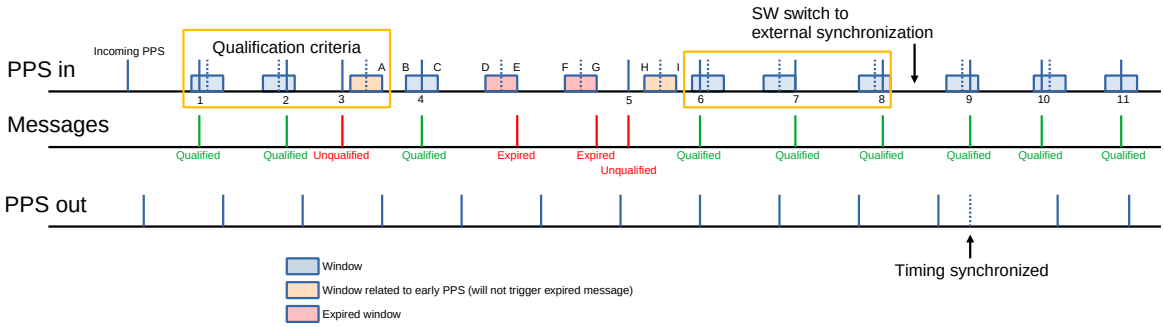



Figure 5.7 - Illustrates an example synchronization sequence.

A qualification criterium of 3 PPS pulses is used. The threshold has been configured to an acceptable value. Ignoring the first pulse, which is just illustrated to create a reference point for the rest, there is the following sequence:

1. PPS signal 1 is input as expected, within the configured window. It is thus qualified.
2. PPS signal 2 is also qualified.
3. PPS signal 3 is input ahead of the window and thus gives rise to an unqualified message. The reference to where the next PPS is expected to be located is still in relation to this early PPS, which is why point A is not causing the window to expire (the window is now located between point B and C). There are now zero consecutive qualified PPS signals.
4. PPS signal 4 is again qualified as it arrives in the window.
5. There is then no PPS input between point D and E, nor between F and G. Each of these causes the generation of expired messages. There are now, again, zero consecutive qualified PPS signals.
6. Suddenly the PPS returns at an arbitrary point, so PPS signal 5 falls outside of the orange window between point H and I and is thus unqualified.
7. PPS signals 6, 7 and 8 are then all input inside the window and there are thus 3 consecutive qualified PPS signals, which is the qualification criterium in this example.
8. The application switches to external mode.
9. PPS signal 9 is also qualified as it arrives in the window. This causes the internal time to synchronize. Note however that a PPS pulse is not generated at the time of

synchronization, but the following PPS signals will be synchronized.

10. PPS signal 10 is also qualified, and on the arrival of it the timing will be synchronized and the PPS output aligned to the PPS input.
11. PPS signals 11 and onwards continue to arrive within the window and are thus qualified. The time is synchronized.

#### 5.4.1.5. Losing synchronization

The synchronization can be lost in three different ways, although there are variations in exactly how they occur. Figure 5.8 illustrates an incoming PPS with an overall fixed interval, subjected to jitter.

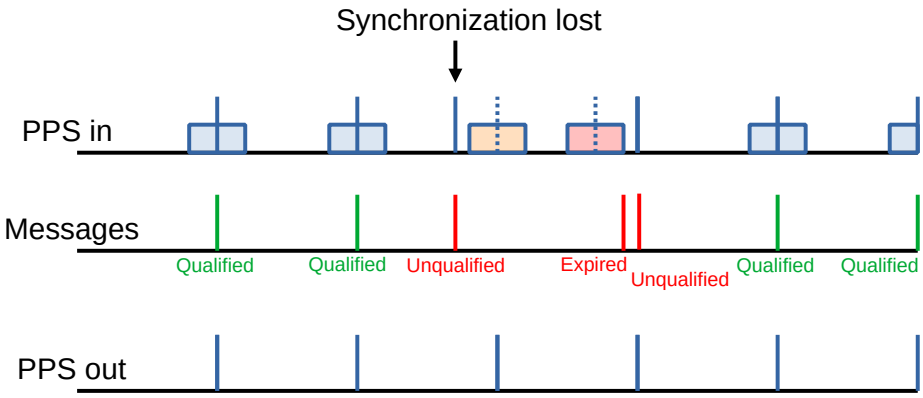



Figure 5.8 - Illustration of synchronization lost due to a PPS pulse determined to early.

The loss of synchronization is shown, caused by an incoming PPS pulse that is determined to be to *early*. It can be seen that qualified messages are generated when synchronized and then an unqualified message when the PPS arrives early (ahead of the window indicated in orange). This causes a loss of synchronization and revert to internal timing. As the next window is relative to the previously received PPS signal (qualified or not), the window for the next pulse is the red window (which is at the same distance from the PPS input as the distance between the blue windows, before losing synchronization). Since the synchronization was lost due to jitter, this example illustrates the next PPS to arrive at the nominal distance, without jitter. The window thus expires before the PPS arrives, which causes an expired message. The late PPS (actually determined to early, since the previous window expired) then causes an unqualified message. Again, the window is aligned based on the previous PPS input and thus the next PPS falls inside the window (again, assuming nominal interval). Within the illustrated time frame it continues to enter qualified. Synchronization is again found, but the application SW needs to actively change to external mode in order to actually synchronize the timing and the PPS output.

Figure 5.9 illustrates the sudden disconnect of the external PPS source.

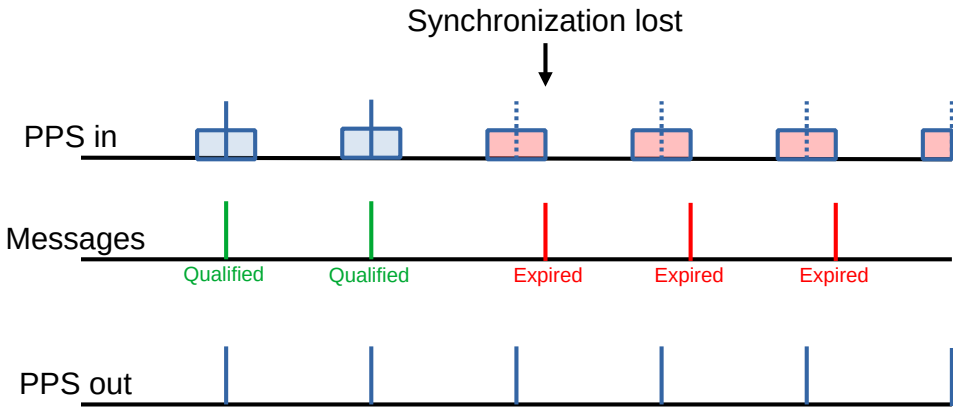



Figure 5.9 - Illustration of synchronization lost due to the PPS window expiring and PPS never arriving.

It can be seen that qualified messages are generated when synchronized and then when the window expires we lose synchronization and revert to internal mode. As the PPS does not arrive again within the illustrated time frame, there are continuous expired-messages generated.

Finally, Figure 5.10 illustrates a case where the incoming PPS interval suddenly changes and in this case causes the PPS to be determined late.

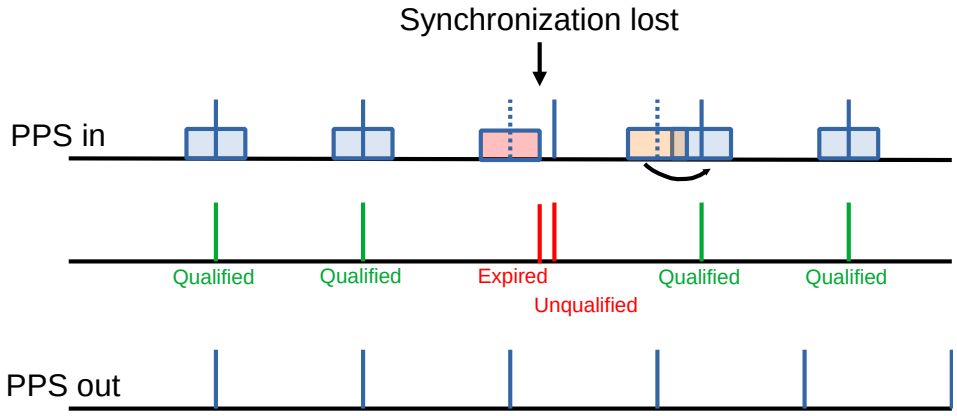



Figure 5.10 - Illustration of synchronization lost due to the PPS determined to late.

It can be seen that qualified messages are generated when synchronized and then when the window expires we lose synchronization and revert to internal mode. This expiration generates an expired message and causes the orange window to be created in reference to the middle point of the expired red window. The incoming PPS is thus seen *early*, because it is referenced to the new, orange, window and causes an unqualified message to be generated. This in turn *moves* the orange window to be in relation to the previously incoming PPS. The next PPS falls within this window and it thus qualified again. Within the illustrated time frame it continues to enter qualified. We have found synchronization again, but SW needs to actively change to external

mode in order to actually synchronize the timing and the PPS output.

#### 5.4.1.6. Input filter

The incoming PPS has a requirement of 1 µs, which can be illustrated as Figure 5.11. Note that the 1 µs delay of *PPS* in the figure is irrelevant. It is simply a qualification strobe indicating an incoming PPS exceeding the length requirements. When synchronized, this filter delay will be adjusted for.

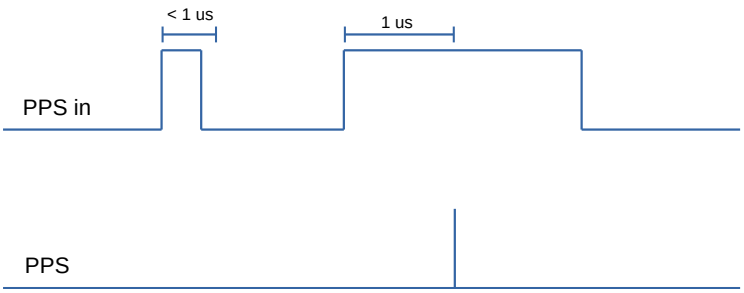



Figure 5.11 - Illustration of PPS input requirements.

#### 5.4.1.7. PPS output

The generated PPS signals have a pulse length of 1 ms, which is illustrated in Figure 5.12.

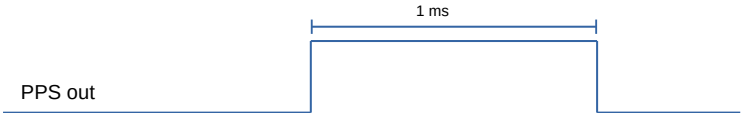



Figure 5.12 - Illustration of PPS output pulse.

#### 5.4.1.8. General-purpose triggers

To be able to provide more accurate time stamping on external events, the SCET has a number of general-purpose triggers. When a trigger fires, the SCET will sample a subset (24 bits) of the current time allowing SW to match the external event to the SCET time regardless of current software state. The exact functionality connected to each general-purpose trigger and the number available is dependent on the system mapping of the SCET, e.g. in a System-On-Chip (SoC), see detailed description in [RD8].

### 5.4.2. API

This API represents the driver interface of the module from an RTEMS user application's perspective.

The driver functionality is accessed through the RTEMS POSIX API for ease of use. In case of a failure on a function call, the *errno* value is set for determining the cause.

SCET counter accesses can be done by reading or writing to the device file, modifying the second and subsecond counter values. The SCET RTEMS driver also supports a number of different IOCTLs for other operations which isn't specifically affecting the SCET counter registers.

For event signaling, the SCET driver has a number of message queues, allowing the application to act upon different events.

#### 5.4.2.1. Function: `int open(…)`

Opens access to the driver. The device driver allows multiple readers but only one writer at a time.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `filename` | `char*` | in | The absolute path to the file that is to be opened. SCET device is defined as `RTEMS_SCET_DEVICE_NAME`. |
| `oflags` | `int` | in | Specifies one of the access modes in the following table. |

| Flags | Description |
|---|---|
| `O_RDONLY` | Open for reading only. |
| `O_WRONLY` | Open for writing only. |
| `O_RDWR` | Open for reading and writing. |

| Return value | Description |
|---|---|
| `>0` | A file descriptor for the device on success |
| `-1` | see *errno* values |
| **errno values** | |
| `EALREADY` | Device already opened for writing |
| `EIO` | Internal RTEMS error |

#### 5.4.2.2. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |

#### 5.4.2.3. Function: `ssize_t read(…)`

Reads the current SCET value, consisting of second and subsecond counters. Both counter values are guaranteed to be sampled at the same moment.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `buf` | `void*` | in | Pointer to a 6-byte buffer where the timestamp will be stored. The first four bytes are the seconds and the last two bytes are the subseconds. |
| `count` | `size_t` | in | Number of bytes to read, must be set to 6. |

| Return value | Description |
|---|---|
| `>=0` | Number of bytes that were read. |
| `-1` | See *errno* values |
| **errno values** | |
| `EBADF` | File descriptor not opened for reading |
| `EINVAL` | Number of bytes to read, count, is not 6 |

#### 5.4.2.4. Function: `ssize_t write(…)`

Offsets the SCET by an offset specified by buf.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `buf` | `const void*` | in | Pointer to a 6-byte buffer where the offsets are stored. The first four bytes are the offset for the seconds and the last two bytes are the offset for the subseconds.<br><br>Values should be in two's complement. |
| `count` | `size_t` | in | Number of bytes to write, must be set to 6. |

| Return value | Description |
|---|---|
| `>=0` | Number of bytes that were written. |
| `-1` | See *errno* values |
| **errno values** | |
| `EBADF` | File descriptor not opened for writing |
| `EINVAL` | Number of bytes to write, count, is not 6 |

#### 5.4.2.5. Function: `int ioctl(…)`

Ioctl allows for any other SCET-related operation which isn't specifically aimed at reading and/or writing the SCET time value.

**NOTE** The number of available PPS inputs and outputs depend on the hardware configuration.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `cmd` | `int` | in | Command to send |
| `val` | `uint32_t/ uint32_t*` | in/out | Data according to the specific command. |

| Command table | Description |
|---|---|
| `SCET_SET_PPS_SOURCE_IOCTL` | Input value sets the PPS source.<br><br>`0` = External PPS source<br>1 = Internal PPS source (default) |
| `SCET_GET_PPS_SOURCE_IOCTL` | Returns the current PPS source<br><br>`0` = External PPS source<br>1 = Internal PPS source (default) |

| Command table | Description |
|---|---|
| SCET_SET_PPS_OUTPUT_PORT_IOCTL | Input bit field configures which PPS output drivers to enable.<br><br>Bit 0 is the output driver of PPS0.<br>Bit N is the output driver of PPSN.<br>Bit 7 is the output driver of PPS7.<br><br>Bit value 0 = The output driver is disabled<br>Bit value 1 = The output driver is enabled<br>(0 is default value of the bit field, all drivers disabled) |
| SCET_GET_PPS_OUTPUT_PORT_IOCTL | Returns the currently enabled PPS output drivers as a bit field.<br><br>Bit 0 is the output driver of PPS0.<br>Bit N is the output driver of PPSN.<br>Bit 7 is the output driver of PPS7.<br><br>Bit value 0 = The output driver is disabled<br>Bit value 1 = The output driver is enabled |
| SCET_SET_PPS_INPUT_PORT_IOCTL | Argument value sets the PPS input port.<br><br>Value 0 is PPS0<br>Value 1 is PPS1<br>Value N is PPSN<br>Value 7 is PPS7<br><br>(1 is default value, PPS1 is default input) |
| SCET_GET_PPS_INPUT_PORT_IOCTL | Returns the currently used PPS input port.<br><br>Value 0 is PPS0<br>Value 1 is PPS1<br>Value N is PPSN<br>Value 7 is PPS7 |
| SCET_SET_PPS_THRESHOLD_IOCTL | Input value configures the PPS threshold window. See Table 5.6 for details. |
| SCET_GET_PPS_THRESHOLD_IOCTL | Returns the currently configured PPS threshold window. |

| Command table | Description |
|---|---|
| SCET_GET_PPS_ARRIVE_COUNTER_IOCTL | Returns 24 bits of the SCET time sampled when PPS arrived.<br><br>Bit 23:16 contains lower 8 bits of second<br>Bit 15:0 contains subseconds |
| SCET_SET_GP_TRIGGER_LEVEL_IOCTL | Input bit field configures the trigger level of each trigger:<br><br>Bit 0 is trigger 0,<br>Bit N is trigger N,<br>Bit 7 is trigger 7.<br><br>Bit value 0 = trigger activates on 0 to 1 transition (rising edge)<br>Bit value 1 = trigger activates on 1 to 0 transition (falling edge).<br><br>(0 is default). |
| SCET_GET_GP_TRIGGER_LEVEL_IOCTL | Returns the currently configured level of the all GP triggers as a bit field:<br><br>Bit 0 is trigger 0,<br>Bit N is trigger N,<br>Bit 7 is trigger 7.<br><br>Bit value 0 = trigger activates on 0 to 1 transition (rising edge)<br>Bit value 1 = trigger activates on 1 to 0 transition (falling edge).<br><br>(0 is default). |
| SCET_SET_GP_TRIGGER_ENABLE_IOCTL | Input bit field selects which GP trigger(s) to enable:<br><br>Bit 0 is trigger 0,<br>Bit N is trigger N,<br>Bit 7 is trigger 7.<br><br>All triggers are disabled by default (0) |

| Command table | Description |
|---|---|
| SCET_GET_GP_TRIGGER_ENABLE_IOCTL | Returns which GP triggers that are enabled.<br><br>Bit 0 is trigger 0,<br>Bit N is trigger N,<br>Bit 7 is trigger 7. |
| SCET_GET_GP_TRIGGER_COUNTER_IOCTL | Input value selects which GP trigger SCET counter sample to read [0,7].<br><br>Returns 24 bits of the SCET counter sampled when the GP trigger became active.<br><br>Bit 23:16 contains lower 8 bits of second<br>Bit 15:0 contains subseconds |

| Return value | Description |
|---|---|
| >=0 | Data returned from get commands, or 0 for success in other cases |
| -1 | See *errno* values |
| **errno values** | |
| EBADF | File descriptor not opened for writing |
| EINVAL | Invalid value for command, or invalid command. |
| ENOTTY | Inappropriate I/O control operation, the command SCET_GET_PPS_O_EN_IOCTL was issued though a PPS input/output configuration, different than what can be reported by this command, is used. |

#### 5.4.2.5.1. Alternative PPS input/output control

The ioctl-commands SCET_SET_PPS_O_EN_IOCTL and SCET_GET_PPS_O_EN_IOCTL are deprecated but still functional and kept for backwards compatibility. Issuing the command SCET_SET_PPS_O_EN_IOCTL with the argument 1 is equivalent to issuing the commands SCET_SET_PPS_OUTPUT_PORT_IOCTL and SCET_SET_PPS_INPUT_PORT_IOCTL with the arguments 2 and 0 respectively.

If any PPS input/output configuration other than those described in the table below are in use, trying to read out the current PPS configuration with SCET_GET_PPS_O_EN_IOCTL will fail and return ENOTTY.

| Command table | Description |
|---|---|
| SCET_SET_PPS_O_EN_IOCTL | Input value configures if pps0 or pps1 is input and if pps1 is input or output.<br><br>0 = pps1 is input, no output ports are activated, (default)<br>1 = pps0 is input, pps1 is output |
| SCET_GET_PPS_O_EN_IOCTL | Returns whether the pps0 or pps1 signal is input and if pps1 is input or output.<br><br>0 = pps1 is input, no output ports are activated, (default)<br>1 = pps0 is input, pps1 is output |

#### 5.4.2.6. Event callback via message queue

The SCET driver exposes message queues for event messaging from the driver to the application. The queues do not use broadcast and thus only delivers each message to a single task.

The queues are limited to 10 pending messages. If the message queues are not continuously emptied they will fill up and new messages will be dropped while they are full. It is recommended to flush the queues to remove potentially outdated messages before starting to use them.

The SCET PPS message queue uses the rtems name `'S', 'P', 'P', 'S'` and provides PPS related messages as described in Table 5.7.

Table 5.7 - Driver message queue message types

| Event name | Description |
|---|---|
| SCET_PPS_EXPIRED | Threshold qualification window expired without any PPS signal detected |
| SCET_PPS_UNQUALIFIED | PPS signal detected before the threshold qualification window |
| SCET_PPS_QUALIFIED | PPS signal detected inside the threshold qualification window |

The SCET General purpose Task N, ‘S’, ‘G’, ‘T’, ‘n’, handles messages sent from the general purpose trigger n, with the number n ranging from 0 to up to the maximum defined for the particular SoC configuration, Table 5.8

Table 5.8 - General purpose trigger n message queue

| Event name | Description |
|---|---|
| SCET_INTERRUPT_STATUS_TRIGGERn | Trigger n was triggered |

#### 5.4.3. Usage description

The following `#define` needs to be set by the user application to be able to use the scet driver:

- `CONFIGURE_APPLICATION_NEEDS_SCET_DRIVER`

By defining this as part of RTEMS configuration, the driver will automatically be initialized at boot up.

##### 5.4.3.1. PPS synchronization procedure

In order to synchronise the SCET timer and the PPS output (if applicable) to an incoming PPS, the following procedure is suggested:

1. Set the PPS input port via the `SCET_SET_PPS_INPUT_PORT_IOCTL` command.
2. Set the PPS threshold via the `SCET_SET_PPS_THRESHOLD_IOCTL` command.
3. Flush the PPS event message queue via `rtems_message_queue_flush()`.
4. Receive from the PPS event message queue via `rtems_message_queue_receive()` until a sufficient number of consecutive qualified PPS event messages are observed, without any expired or unqualified PPS event messages between.
5. Switch to external PPS sync via the `SCET_SET_PPS_SOURCE_IOCTL` command.
6. Receive indefinitely from the PPS event message queue and ensure that only qualified PPS event messages are observed.
    a. If an expired or unqualified PPS event message is observed, restart from step 4.

Whenever an expired or unqualified event occurs, the system will automatically transition to internal synchronization; switching to external synchronization is always done explicitly by the application.

##### 5.4.3.2. RTEMS application example

In order to use the SCET driver in the RTEMS environment, the following code structure is suggested for use:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <assert.h>
#include <bsp/scet_rtems.h>

```

```
#define CONFIGURE_APPLICATION_NEEDS_SCET_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER

#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 30
#define CONFIGURE_MAXIMUM_DRIVERS                10
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS          20
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES 20

#define CONFIGURE_INIT

#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

static const int32_t secs_to_adjust = -10;
static const int16_t subsecs_to_adjust = 1000;

/* Adjust SCET time 10 seconds backwards and 1000
 * subseconds forwards */
rtems_task Init(rtems_task_argument ignored)
{
  int result;
  int scet_fd;
  uint32_t old_seconds;
  uint16_t old_subseconds;
  uint32_t new_seconds;
  uint16_t new_subseconds;
  uint8_t read_buffer[6];
  uint8_t write_buffer[6];

  scet_fd = open(RTEMS_SCET_DEVICE_NAME, O_RDWR);
  assert(scet_fd >= 0);

  result = read(scet_fd, read_buffer, 6);
  assert(result == 6);

  memcpy(&old_seconds, read_buffer, sizeof(uint32_t));
  memcpy(&old_subseconds, read_buffer + sizeof(uint32_t),
sizeof(uint16_t));

  printf("\nOld SCET time is %lu.%u\n", old_seconds, old_subseconds);
  printf("Adjusting seconds with %ld, subseconds
             with %
             d\n ",
             secs_to_adjust,
         subsecs_to_adjust);

  memcpy(write_buffer, &secs_to_adjust, sizeof(uint32_t));
  memcpy(write_buffer + sizeof(uint32_t), &subsecs_to_adjust,
sizeof(uint16_t));

```

```
57   result = write(scet_fd, write_buffer, 6);
58   assert(result == 6);
59
60   result = read(scet_fd, read_buffer, 6);
61   assert(result == 6);
62
63   memcpy(&new_seconds, read_buffer, sizeof(uint32_t));
64   memcpy(&new_subseconds, read_buffer + sizeof(uint32_t),
   sizeof(uint16_t));
65
66   printf("New SCET time is %lu.%u\n", new_seconds, new_subseconds);
67
68   result = close(scet_fd);
69   assert(result == 0);
70
71   rtems_task_delete(RTEMS_SELF);
72 }
```

- Inclusion of `<fcntl.h>` and `<unistd.h>` are required for using the POSIX functions: open, close, ioctl.
- Inclusion of `<errno.h>` is required for retrieving error values on failures.
- Inclusion of `<bsp/scet_rtems.h>` is required for accessing SCET device name `RTEMS_SCET_DEVICE_NAME` as well as other defines.

## 5.5. UART

### 5.5.1. Description

This section describes the driver as one utility for accessing the uart device.

#### 5.5.1.1. Driver

This driver is using the de facto standard interface for a 16550D UART given in [RD9] and as such has an 8-bit interface, but has been expanded to provide a faster and more delay-tolerant implementation.

#### 5.5.1.2. RX/TX buffer depth

The RX and TX FIFOs have been expanded to 128 characters compared to the original specification of 16 characters. To be backwards compatible as well as being able to utilize the larger depth of the FIFOs, a new parameter has been brought in called buffer depth. The set buffer depth decides how much of the FIFOs real depth it should base its calculations on. Buffer depth affects only the RX FIFO handling in the bare-metal case, but both RX and TX FIFOs handling in the RTEMS driver.

#### 5.5.1.3. Trigger levels

To be able to utilize the larger FIFOs, the meaning of the trigger levels have been changed. In the specification in [RD9], it defines the trigger levels as 1 character, 4 characters, 8 characters and 14 characters. This has now been changed to instead mean 1 character, 1/4 of the FIFO is full, 1/2 of the FIFO is full and the FIFO is 2 characters from the given buffer depth top. This results in the IP being fully backwards compatible, since a buffer depth of 16 characters would yield the same trigger levels as those given in [RD9]. Since the trigger is depending on the buffer depth, the trigger level have to be set after the buffer depth is changed.

#### 5.5.1.4. Modes

The UARTs can be set to operate in different modes using the ioctl call `UART_IOCTL_MODE_SELECT`, as given in Section 5.5.2.5.

When in RS-485 mode the UART IP will automatically disable the line driver (put it in a high impedance state) and only enable it while transmitting. When the UART does not have anything to transmit it will wait for 800 ns to allow the last bits to propagate through the circuit, then it will disable the driver. According to the data sheet the driver disable time is 100 ns, so within 1000 ns of the last bit being transmitted the driver should be in a high impedance state and the UART should be ready to receive.

RS-422 mode is the default mode. In this mode the transmitter and receiver are both

enabled.

In LOOPBACK-mode TX and RX are connected internally in the UART IP.

#### 5.5.2. API

This API represents the driver interface of the module from an RTEMS user application's perspective.

The driver functionality is accessed through the RTEMS POSIX API for ease of usage. In case of a failure on a function call, the *errno* value is set for determining the cause.

The driver allows one reader per UART and one writer per UART.

##### 5.5.2.1. Function: `int open(…)`

Opens access to the requested UART. Only blocking mode is supported. Upon each successful open call the device interface is reset to 115200 bps and its default mode according to the table below. See [RD8] for the current SoC configuration, including device name and characteristics for each UART device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `pathname` | `const char*` | in | The absolute path to the file that is to be opened. See [RD8] for uart naming. |
| `flags` | `int` | in | Specifies one of the access modes in the following table. |

| Flags | Description |
|---|---|
| `O_RDONLY` | Open for reading only. |
| `O_WRONLY` | Open for writing only. |
| `O_RDWR` | Open for reading and writing. |

| Return value | Description |
|---|---|
| `fd` | A file descriptor for the device on success |
| `-1` | See *errno* values |
| **errno values** | |
| `ENOENT` | Invalid filename |
| `EALREADY` | Device is already open |
| `EIO` | Failed to obtain internal resource |

#### 5.5.2.2. Function: `int close(…)`

Closes access to the device and disables the line drivers.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |

#### 5.5.2.3. Function: `ssize_t read(…)`

Read data from the UART. The call blocks until data is received from the UART RX FIFO unless UART read timeout is enabled. UART read timeout can be enabled with the ioctl `UART_IOCTL_READ_TIMEOUT_ENABLE`. The duration of the timeout can be set with the ioctl `UART_IOCTL_READ_TIMEOUT_DURATION_SET`. When UART timeout is enabled and read() has not received any data before the timer fires, read() will return minus one and set the status code `ETIME`.

**NOTE** The read call may return less data than requested.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `buf` | `void*` | in | Pointer to character buffer to write data to |
| `count` | `size_t` | in | Number of characters to read |

| Return value | Description |
|---|---|
| `>0` | Number of characters that were read. |
| `0` | A parity/framing/overrun error occurred. The RX data path has been flushed. Data was lost. |
| `-1` | see *errno* values |
| **errno values** | |
| `EIO` | Failed to get internal resource |
| `ETIME` | The read operation timed out and no packet was received. |

#### 5.5.2.4. Function: `ssize_t write(…)`

Write data to the UART. The write call is blocking until all data has been transmitted unless UART write timeout is enabled.

UART write timeout can be enabled with the ioctl command `UART_IOCTL_WRITE_TIMEOUT_ENABLE`. The duration of the timeout can be set with the ioctl `UART_IOCTL_WRITE_TIMEOUT_DURATION_SET`. When write timeout is enabled, if UART write() does not get an interrupt saying that the transmission was successful before the timer fires, write() will return minus one and set the status code `ETIME`.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `buf` | `const void*` | in | Pointer to character buffer to read data from |
| `count` | `size_t` | in | Number of characters to write |

| Return value | Description |
|---|---|
| `>=0` | Number of characters that were written. |
| `-1` | see *errno* values |
| **errno values** | |
| `EIO` | Failed to get internal resource |
| `ETIME` | The write operation timed out. |

#### 5.5.2.5. Function: `int ioctl(…)`

> **NOTE** Since the granularity of the system is 10ms, values not divisible by 10 ms will be truncated to the nearest multiple if 10ms. Setting a timeout less than 10 ms will result in a timeout of 0 ms.

The timeout configuration applies to all open file descriptors. If more than one UART device is opened, it is not possible to control the timeout configuration for a specific file descriptor.

Ioctl allows for toggling the RS422/RS485/Loopback mode and setting the baud rate. RS422/RS485 mode selection is not applicable for UART6 and UART7.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `cmd` | `int` | in | Command to send |

| Argument | Type | Direction | Description |
|---|---|---|---|
| `val` | `int` | in | Value to write or a pointer to a buffer where data will be written. |

| Command table | Type | Direction | Description |
|---|---|---|---|
| `UART_IOCTL_SET_BITRATE` | `uint32_t` | in | Set the bitrate of the line interface. Possible values:<br>`UART_B375000`<br>`UART_B347200`<br>`UART_B223200`<br>`UART_B153600`<br>`UART_B115200` (default)<br>`UART_B76800`<br>`UART_B57600`<br>`UART_B38400`<br>`UART_B19200`<br>`UART_B9600`<br>`UART_B4800`<br>`UART_B2400`<br>`UART_B1200` |
| `UART_IOCTL_MODE_SELECT` | `uint32_t` | in | Set the mode of the interface. Possible values:<br>`UART_RTEMS_MODE_RS422` (default)<br>`UART_RTEMS_MODE_RS485`<br>`UART_RTEMS_MODE_LOOPBACK` (TX connected to RX internally) |
| `UART_IOCTL_RX_FLUSH` | `uint32_t` | in | Flushes the RX software FIFO. Cannot be done on a write-only file descriptor. Indata is not used by driver. |
| `UART_IOCTL_SET_PARITY` | `uint32_t` | in | Set parity. Possible values:<br>`UART_PARITY_NONE` (default)<br>`UART_PARITY_ODD`<br>`UART_PARITY_EVEN` |
| `UART_IOCTL_SET_BUFFER_DEPTH` | `uint32_t` | in | Set the FPGA FIFO buffer depth. Possible values:<br>`UART_BUFFER_DEPTH_16` (default)<br>`UART_BUFFER_DEPTH_32`<br>`UART_BUFFER_DEPTH_64`<br>`UART_BUFFER_DEPTH_128` |

| Command table | Type | Direction | Description |
|---|---|---|---|
| UART_IOCTL_GET_BUFFER_DEPTH | uint32_t* | out | Get the current buffer depth. |
| UART_IOCTL_SET_TRIGGER_LEVEL | uint32_t | in | Set the FPGA RX FIFO trigger level. Have to be set after the buffer depth. Cannot be done on a write-only file descriptor. Possible values: UART_TRIGGER_LEVEL_1 = 1 character UART_TRIGGER_LEVEL_4 = 1/4 full UART_TRIGGER_LEVEL_8 = 1/2 full UART_TRIGGER_LEVEL_14 = buffer_depth - 2 (default) |
| UART_IOCTL_GET_TRIGGER_LEVEL | uint32_t* | out | Get the current trigger level. |
| UART_IOCTL_READ_TIMEOUT_ENABLE | uint32_t | in | 1 = Enables UART read timeout 0 = Disables UART read timeout (default) |
| UART_IOCTL_READ_TIMEOUT_DURATION_SET | uint32_t | in | Sets the duration of the timeout in milliseconds. Default is 1000 ms. |
| UART_IOCTL_WRITE_TIMEOUT_ENABLE | uint32_t | in | 1 = Enables UART write timeout 0 = Disables UART write timeout (default) |
| UART_IOCTL_WRITE_TIMEOUT_DURATION_SET | uint32_t | in | Sets the duration of the timeout in milliseconds. Default is 1000 ms. |

| Return value | Description |
|---|---|
| 0 | Command executed successfully |
| -1 | see *errno* values |
| **errno values** | |
| EBADF | Bad file descriptor for intended operation |
| EINVAL | Invalid value supplied to IOCTL |

#### 5.5.3. Usage description

The following `#define` needs to be set by the user application to be able to use the UARTs:

- `CONFIGURE_APPLICATION_NEEDS_UART_DRIVER`

By defining this as part of RTEMS configuration, the driver will automatically be initialized at boot up.

##### 5.5.3.1. RTEMS application example

In order to use the uart driver in the RTEMS environment, the following code structure is suggested to be used:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <bsp/uart_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_UART_DRIVER
#define CONFIGURE_SEMAPHORES 40

#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

#define CONFIGURE_INIT
rtems_task Init(rtems_task_argument argument);

rtems_task Init(rtems_task_argument ignored)
{
  /* ... */
}
```

- Inclusion of `<fcntl.h>` and `<unistd.h>` are required for using the POSIX functions: open, close, ioctl.
- Inclusion of `<errno.h>` is required for retrieving error values on failures.
- Inclusion of `<bsp/uart_rtems.h>` is required for accessing the uarts.

##### 5.5.3.2. Parity, framing and overrun error notification

Upon receiving a parity, framing or an overrun error the read call returns 0 and the internal RX queue is flushed.

### 5.5.4. Limitations

8 data bits only.
1 stop bit only.
No hardware flow control support.

## 5.6. Mass memory

### 5.6.1. Description

This section describes the mass memory driver's design and usability.

#### 5.6.1.1. General

The driver supports a number of independent operations on the mass memory. Logically the mass memory is divided into blocks and pages. There are MASSMEM_BLOCKS blocks starting from block number 0 and MASSMEM_PAGES_PER_BLOCK pages within each block starting from page 0.

#### 5.6.1.2. Data Structures

##### 5.6.1.2.1. Struct: `massmem_cid_t`

This struct is used as the target for reading the mass memory chip IDs. The byte array constants *massmem_cid_CHIP_TYPE_A* and *massmem_cid_CHIP_TYPE_B* are provided as chip id references for the possible chip types.

| Type | Name | Purpose |
|---|---|---|
| Array of 5 uint8_t | edac | Byte array for EDAC chip ID |
| Array of 5 uint8_t | chip0 | Byte array for chip 0 ID |
| Array of 5 uint8_t | chip1 | Byte array for chip 1 ID |
| Array of 5 uint8_t | chip2 | Byte array for chip 2 ID |
| Array of 5 uint8_t | chip3 | Byte array for chip 3 ID |

##### 5.6.1.2.2. Struct: `massmem_error_injection_t`

This struct is used as a specification when manually injecting errors when writing to the the mass memory.

| Type | Name | Purpose |
|---|---|---|
| uint8_t | edac_error_injection | Bits to be XOR:ed with generated EDAC byte |
| uint32_t | data_error_injection | Bits to be XOR:ed with supplied data |

#### 5.6.1.2.3. Struct: `massmem_ioctl_spare_area_args_t`

This struct is used by the RTEMS API as the target when reading or writing the spare area.

| Type | Name | Purpose |
|---|---|---|
| uint32_t | page_num | What page to read/write |
| uint32_t | offset | Byte offset into spare area to read or write. Must be 32 word (of 4 bytes) aligned. |
| uint8_t * | data_buf | Pointer to buffer in which the data is to be stored, or to the data that is to be written. |
| uint8_t * | edac_buf | Deprecated; this parameter will not be accessed. |
| uint32_t | size | Size to read/write in bytes. Must be 32 word (of 4 bytes) aligned. |

#### 5.6.1.2.4. Struct: `massmem_ioctl_error_injection_args_t`

This structure is used by the RTEMS API in order to perform a special write call to inject errors into the mass memory.

| Type | Name | Purpose |
|---|---|---|
| uint32_t | page_num | What page to write |
| uint8_t * | data_buf | Pointer to data to write |
| uint32_t | size | Size of data to write in bytes |
| massmem_error_injection_t * | error_injection | Pointer to error injection struct. See Section 5.6.1.2.2 for definition |

### 5.6.2. API

This API represents the driver interface from a user application's perspective for the RTEMS driver.

The driver functionality is accessed through RTEMS POSIX API for ease of usage. In

case of failure on a function call, *errno* value is set for determining the cause.

#### 5.6.2.1. Function: `int open(…)`

Opens access to the driver.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `filename` | `char *` | in | The absolute path to the file that is to be opened. Mass memory device is defined as MASSMEM_DEVICE_NAME. |
| `oflags` | `int` | in | Device must be opened by exactly one of the symbols defined in Table 5.9. |

| Return value | Description |
|---|---|
| `>0` | A file descriptor for the device. |
| `-1` | see *errno* values |
| | **errno values** |
| `EBADF` | The file descriptor *fd* is not an open file descriptor |
| `ENOENT` | Invalid filename |
| `EEXIST` | Device already opened. |

Table 5.9 - Flags for function `open()`

| Symbol | Description |
|---|---|
| `O_RDONLY` | Open for reading only |
| `O_WRONLY` | Open writing only |
| `O_RDWR` | Open for reading and writing |

#### 5.6.2.2. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |
| `-1` | see *errno* values |

| Return value | Description |
|---|---|
| | **errno values** |
| EBADF | The file descriptor *fd* is not an open file descriptor |

#### 5.6.2.3. Function: `off_t lseek(…)`

Sets page offset for read/ write operations.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| offset | off_t | in | Page number. |
| whence | int | in | Must be set to SEEK_SET. |

| Return value | Description |
|---|---|
| offset | Page number |
| -1 | see *errno* values |
| | **errno values** |
| EBADF | The file descriptor *fd* is not an open file descriptor |
| ESPIPE | *fd* is associated with a pipe, socket or FIFO. |
| EINVAL | *whence* is not a proper value. |
| EOVERFLOW | The resulting file offset would overflow **off_t**. |

#### 5.6.2.4. Function: `ssize_t read(…)`

Reads requested size of bytes from the device starting from the offset set in lseek.

| | |
|---|---|
| **NOTE** | For iterative read operations, lseek must be called to set page offset ***before*** each read operation. |

| | |
|---|---|
| **NOTE** | The character buffer location handed to read must be 32-bit aligned. |

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| buf | void * | in | Character buffer where to store the data |
| nbytes | size_t | in | Number of bytes to read into *buf.* |

| Return value | Description |
|---|---|
| `>0` | Number of bytes that were read. |
| `-1` | see *errno* values |
| | **errno values** |
| `EBADF` | The file descriptor *fd* is not an open file descriptor |
| `EINVAL` | Page offset set in lseek is out of range or *nbytes* is too large and reaches a page that is out of range. |
| `EBUSY` | Device is busy with previous read/write operation. |

#### 5.6.2.5. Function: `ssize_t write(…)`

Writes requested size of bytes to the device starting from the offset set in lseek.

**NOTE** For iterative write operations, lseek must be called to set page offset before each write operation.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `buf` | `void *` | in | Character buffer to read data from |
| `nbytes` | `size_t` | in | Number of bytes to write from *buf*. |

| Return value | Description |
|---|---|
| `>0` | Number of bytes that were written. |
| `-1` | see *errno* values |
| | **errno values** |
| `EBADF` | The file descriptor *fd* is not an open file descriptor |
| `EINVAL` | Page offset set in **lseek** is out of range or *nbytes* is too large and reaches a page that is out of range. |
| `EIO` | Failed to write data. Block should be marked as a bad block. |

#### 5.6.2.6. Function: `int ioctl(…)`

Additional supported operations via POSIX Input/Output Control API.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| cmd | ioctl_command_t | in | Command specifier |
| (varies) | (varies) | (varies) | Command-specific argument |

The following return and errno values are common for all commands except MASSMEM_IO_BAD_BLOCK_CHECK.

| Return value | Description |
|---|---|
| 0 | Operation successful (or block is marked ok in case of bad block check) |
| -EBUSY | Device is busy with previous read/write operation. |
| -1 | See errno values |
| **errno values** | |
| ENODEV | Internal RTEMS error |
| EIO | Internal RTEMS error |

#### 5.6.2.6.1. Reset mass memory device

Resets the mass memory device.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_RESET | n/a | n/a | n/a |

#### 5.6.2.6.2. Read status data

Reads the status register value.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_READ_DATA_STATUS | uint32_t* | out | Pointer to variable in which status data is to be stored. |

#### 5.6.2.6.3. Read control status data

Reads the control status register value.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_READ_CTRL_STATUS | uint8_t* | out | Pointer to variable in which control status data is to be stored. |

#### 5.6.2.6.4. Read EDAC register data

Reads the EDAC register value.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_READ_EDAC_STATUS | uint8_t* | out | Pointer to variable in which control status data is to be stored. |

#### 5.6.2.6.5. Read ID

Reads the chip IDs

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_READ_ID | massmem_cid_t.* | out | Pointer to struct in which ID is to be stored, see Section 5.6.1.2.1 |

#### 5.6.2.6.6. Erase block

Erases a block

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_ERASE_BLOCK | uint32_t | in | Block number |

| Return value | Description |
|---|---|
| -EINVAL | The block number is out of range |
| -EIO | Failed to erase block. Block should be marked as a bad block |

#### 5.6.2.6.7. Read spare area

Reads the spare area with given data.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_READ_SPARE_AREA | massmem_ioctl_spare_area_args_t* | in/out | Pointer to struct with input page number specifier, byte offset, and destination buffers where spare area data is to be stored, see Section 5.6.1.2.3 |

| Return value | Description |
|---|---|
| -EINVAL | Indicates one or more of:<br>• The page number is out of range<br>• Size is 0<br>• Size + offset is larger than spare area size<br>• Size is not a multiple of 4<br>• The data buffer is NULL<br>• The data or EDAC buffer is not 4-byte aligned |
| -EIO | Reading timed out or read status indicated failure. |

#### 5.6.2.6.8. Write spare area

Writes the given data to the spare area.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_WRITE_SPARE_AREA | massmem_ioctl_spare_area_args_t* | in/out | Pointer to struct with page number specifier, byte offset and pointer to data which is to be written, see Section 5.6.1.2.3 |

| Return value | Description |
|---|---|
| -EINVAL | Indicates one or more of:<br>• The page number is out of range<br>• Size is 0<br>• Size + offset is larger than spare area size<br>• Size is not a multiple of 4<br>• The data buffer is NULL<br>• The data buffer is not 4-byte aligned |
| -EIO | Failed to write data. Block should be marked as a bad block. |

#### 5.6.2.6.9. Bad block check

Reads the factory bad block status from a block.

**NOTE** This only gives information about factory bad blocks; subsequent bad block status is not included in this information.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_BAD_BLOCK_CHECK | uint32_t | in | Block number. |

| Return value | Description |
|---|---|
| 0 | Block is marked ok. |
| 1 | Block is marked as bad. |
| -EINVAL | The block number is out of range |

#### 5.6.2.6.10. Error Injection

Injects errors in page write command call. The purpose is to test error corrections (EDAC).

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_ERROR_INJECTION | massmem_ioctl_error_injection_args_t* | in | Pointer to struct with program page arguments as defined in Section 5.6.1.2.4 |

| Return value | Description |
|---|---|
| -EINVAL | Indicates one or more of:<br>• The page number is out of range<br>• Size is 0<br>• Size is larger than page size<br>• Size is not a multiple of 4<br>• The data or EDAC buffer is NULL<br>• The data buffer is not 4-byte aligned |
| -EIO | The mass memory write operation failed, the block should be marked as a bad block |

#### 5.6.2.6.11. Get page bytes

Get the available page size in bytes. The value will differ based on the chip type in use, but will always be less or equal to the static define MASSMEM_PAGE_BYTES_MAX.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_GET_PAGE_BYTES | uint32_t* | out | Pointer to variable in which the available page size in bytes is to be stored. |

#### 5.6.2.6.12. Get spare area bytes

Get the available spare area size in bytes. The value will differ based on the chip type in use, but will always be less or equal to the static define MASSMEM_SPARE_AREA_BYTES_MAX.

| Command | Value type | Direction | Description |
|---|---|---|---|
| MASSMEM_IO_GET_SPARE_AREA_BYTES | uint32_t* | out | Pointer to variable in which the available spare area size in bytes is to be stored. |

### 5.6.3. Usage description

#### 5.6.3.1. Overview

The RTEMS driver accesses the mass memory by the reference a page number.

When writing new data into a page, the memory area must be in its reset value. If there is data that was previously written to a page, the block where the page resides must first be erased in order to clear the page to its reset value. **Note** that the whole block is erased, not only the page.

It is the user application's responsibility to make sure any data the needs to be preserved after the erase block operation must first be read and rewritten after the erase block operation, with the new page information.

#### 5.6.3.2. Usage

The RTEMS driver must be opened before it can access the mass memory flash device. Once opened, all provided operations can be used as described in the subchapter Section 5.6.2. And, if desired, the access can be closed when not needed.

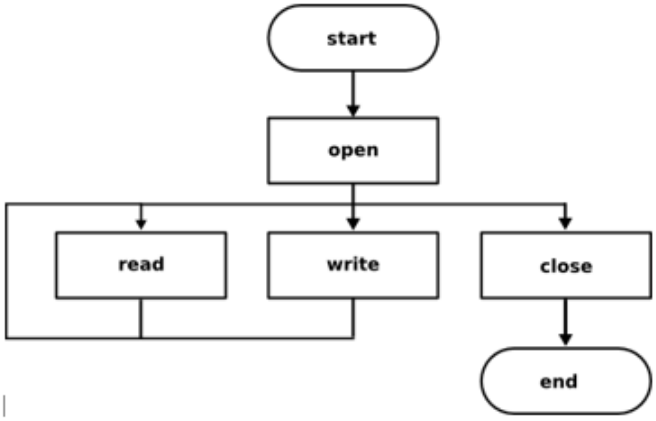



Figure 5.13 - RTEMS driver usage description

**NOTE** | All calls to RTEMS driver are blocking calls.

The mass memory (and corresponding available page and spare area size) will vary between exposing 16GB and 32GB depending on the chip type at runtime.
MASSMEM_PAGE_BYTES_MAX` and `MASSMEM_SPARE_AREA_BYTES_MAX` can be used at compile time, when the sizes are not yet known. At runtime, the available page and spare area sizes will be accessible via the `MASSMEM_IO_GET_PAGE_BYTES` and `MASSMEM_IO_GET_SPARE_AREA_BYTES` *ioctl()* commands.

In order to use the memory flash driver in RTEMS environment with runtime-determined page size, the following code structure is suggested to be used:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
```

```
#include <bsp/massmem_flash_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_MASSMEM_FLASH_DRIVER
#define CONFIGURE_INIT
rtems_task Init(rtems_task_argument argument);

#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

static uint8_t buf[MASSMEM_PAGE_BYTES_MAX];

rtems_task Init(rtems_task_argument ignored)
{
  fd = open(MASSMEM_DEVICE_NAME, O_RDWR);
  s = ioctl(fd, MASSMEM_IO_GET_PAGE_BYTES, &page_bytes)
      off = lseek(fd, page_number, SEEK_SET);
  sz = write(fd, buf, page_bytes);
  off = lseek(fd, page_number, SEEK_SET);
  sz = read(fd, buf, page_bytes);
}
```

#### 5.6.3.3. Defines and includes

Inclusion of <fcntl.h> and <unistd.h> are required for using the POSIX functions open, close, lseek, read and write to access the mass memory bare metal driver.

Inclusion of <errno.h> is required for retrieving error values on failures.

Inclusion of <bsp/massmem_flash_rtems.h> is required for mass memory flash related definitions.

Inclusion of <bsp/bsp_confdefs.h> is required to initialise the driver at boot up.

CONFIGURE_APPLICATION_NEEDS_MASSMEM_FLASH_DRIVER must be defined for using the mass memory driver. This will automatically initialise the driver at boot up.

#### 5.6.3.4. Error injection

Error injection is used to verify the EDAC capabilities of the IP. The IP always writes/reads 8 32-bit data words. If less or an uneven amount of data is requested from the application the drivers pads this internally.

To ensure that the memory can withstand a full byte corruption of data the 8 words of data are interleaved over the mass memory chips. This is done transparently from the user perspective except when writing the error injection vector. Looking at the `massmem_error_injection_t` struct defined in Section 5.6.1.2: the `data_error_injection` member is an `uint32_t`.

Bit 0 of byte 0, 1, 2, 3 affects the first data word.
Bit 1 of byte 0, 1, 2, 3 affects the second data word.
...
Bit 7 of byte 0, 1, 2, 3 affects the eight data word.

To inject a correctible error in the third data word flip either bit 2, 10, 18 or 26.
To inject an uncorrectible in the third data word flip two bits of either 2, 10, 18, 26.

## 5.7. SpaceWire

### 5.7.1. Description

This section describes the driver as one utility for accessing the SpaceWire device.

### 5.7.2. API

This API represents the driver interface from a user application's perspective for the RTEMS driver.

The driver functionality is accessed through RTEMS POSIX API for ease of use. In case of failure on a function call, *errno* value is set for determining the cause. Additional functionalities are supported via POSIX Input/Output Control API as described in Section 5.7.2.5.

#### 5.7.2.1. Function: `int open(…)`

Opens a file descriptor associated with the named device, and, for normal operation, open() registers with the corresponding logic address. It is also possible to open a SpaceWire device in promiscuous mode, where only one reader task is needed for reading on all logical addresses. Each unique device may only be opened once for read-only and once for write-only at the same time, or alternatively opened only once for read-write at the same time. If a SpaceWire device is opened in promiscuous mode, it is not possible to open a device with a specific logical address. If a device is opened for a specific logical address, it is not possible to open another device in promiscuous mode.

The device name must be set as described in the usage description in Section 5.7.3

| Argument | Type | Direction | Description |
|---|---|---|---|
| `filename` | `char *` | in | Device name to register to for data transaction. |
| `oflags` | `int` | in | Device must be opened by exactly one of the symbols defined in Table 5.10 |

| Return value | Description |
|---|---|
| `>0` | A file descriptor for the device. |
| `-1` | see *errno* values |
| **errno values** | |
| `EIO` | Internal RTEMS resource error. |

| Return value | Description |
|---|---|
| EALREADY | Device already opened for the requested access mode (read or write). |
| ENOENT | Invalid filename. |

Table 5.10 - Open flag symbols

| Symbol | Description |
|---|---|
| O_RDONLY | Open for reading only |
| O_WRONLY | Open writing only |
| O_RDWR | Open for reading and writing |

#### 5.7.2.2. Function: `int close(…)`

Deregisters the device name from data transactions.

**NOTE** Closing a file descriptor that has ongoing read, write or ioctl processes is not supported. The application must guarantee that all accesses has completed (returned) before closing the descriptor.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |

| Return value | Description |
|---|---|
| 0 | Device name deregistered successfully |
| -1 | see *errno* values |
| **errno values** | |
| EBADF | The file descriptor `fd` is not an open file descriptor. |

#### 5.7.2.3. Function: `ssize_t read(…)`

Reads a packet when available.

**NOTE** This call is blocked until a packet is received, unless Spacewire read timeout is enabled. In addition, only **one** task must access one file descriptor at a time. Multiple task accessing the same file descriptor is not allowed.

Spacewire read timeout can be enabled with the ioctl `SPWN_IOCTL_READ_TIMEOUT_ENABLE`. The duration of the timeout can be set by the ioctl `SPWN_IOCTL_READ_TIMEOUT_DURATION_SET`. If Spacewire read timeout is enabled, and `read()` has not received any data before the timer fires, `read()` will return minus one and set the status code `ETIME`. If the reception of a packet has been started, the configurable timeout has no effect anymore. Though there is no risk of blocking indefinitely if the whole packet is not received as there is a fixed timeout of 1 second implemented in the SpaceWire router. If the rest of the packet does not arrive within 1 second, `read()` will return minus one and set the status code `ETIMEDOUT`.

If a packet with an EEP (Error End of Packet) is received, `read()` will return minus one and set the status code `ETIMEDOUT`. If a SpW packet is terminated by an EEP character, this means that the packet has been truncated somewhere along its path due to SpW link failure.

**NOTE** | **Argument** `buf` **must** be aligned to a 32 bit aligned address.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `buf` | `void *` | in | Character buffer where to store the packet |
| `nbytes` | `size_t` | in | Maximum number of bytes available for storage in `buf`. Must be between `0` and `SPWN_MAX_PACKET_SIZE`. |

| Return value | Description |
|---|---|
| `>=0` | Received size of the actual packet. Can be less than *nbytes*. |
| `-1` | see *errno* values |
| **errno values** | |
| `EBADF` | The file descriptor `fd` is not an open file descriptor. |
| `EINVAL` | *nbytes* is larger than `SPWN_MAX_PACKET_SIZE`, or buffer is NULL. |
| `EIO` | Internal RTEMS resource error. |
| `EBUSY` | Receive descriptor not currently available. |
| `EOVERFLOW` | Packet size overflow occurred on reception. |
| `ETIMEDOUT` | EEP received. Received packet is incomplete. |

| Return value | Description |
|---|---|
| ETIME | The read operation timed out and no packet was received. |

#### 5.7.2.4. Function: `ssize_t write(…)`

Transmits a packet.

> **NOTE** This call is blocked until the packet is transmitted, unless write timeout is enabled.

Spacewire write timeout can be enabled with the ioctl command `SPWN_IOCTL_WRITE_TIMEOUT_ENABLE`. The duration of the timeout can be set by the ioctl `SPWN_IOCTL_WRITE_TIMEOUT_DURATION_SET`. If write timeout is enabled, and the user application tries to write on Spacewire, if the Spacewire driver does not get an interrupt saying that the transmission was successful before the timer fires, `write()` will return minus one and set the status code `ETIME`. If a timeout occurs during an ongoing transmission of a packet, the packet will be truncated, terminated by an EEP and sent.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| buf | void * | in | Character buffer containing the packet. |
| nbytes | size_t | in | Packet size in bytes. Must be between 0 and SPWN_MAX_PACKET_SIZE bytes. |

| Return value | Description |
|---|---|
| >=0 | Number of bytes that were transmitted. |
| <0 | see *errno* values |
| **errno values** | |
| EBADF | The file descriptor fd is not an open file descriptor. |
| EINVAL | Packet size is larger than SPWN_MAX_PACKET_SIZE. |
| EBUSY | Transmission already in progress. |
| EIO | Internal RTEMS resource error, or internal transmission error. |
| ETIME | The write operation timed out. |

#### 5.7.2.5. Function: `int ioctl(…)`

Additional supported operations via POSIX Input/Output Control API.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | A file descriptor received at open. |
| `cmd` | `int` | in | Command identifier. |
| `value` | `void *` | in | Command-specific parameter. |

The individual command identifiers and command-specific parameters are described in the subsections below.

##### 5.7.2.5.1. Mode setting

Sets the device into the given mode.

**NOTE** The mode setting affects the SpaceWire device and therefore all file descriptors registered to it.

| Command | Value type | Direction | Description |
|---|---|---|---|
| `SPWN_IOCTL_MODE_SET` | `uint32_t` | in | Modes available:<br><br>`SPWN_IOCTL_MODE_OFF`: Turns off the node.<br>`SPWN_IOCTL_MODE_LOOPBACK`: Internal loopback mode<br>`SPWN_IOCTL_MODE_NORMAL`: Normal mode. |

| Return value | Description |
|---|---|
| `0` | Given mode was set |
| `-1` | see *errno* values |
| **errno values** | |
| `EBADF` | The file descriptor `fd` is not an open file descriptor. |
| `EINVAL` | Invalid command, or invalid mode value. |

##### 5.7.2.5.2. Spacewire timeout

**NOTE** Since the granularity of the system is 10ms, values not divisible by

> 10 ms will be truncated to the nearest multiple if 10ms. Setting a timeout less than 10 ms will result in a timeout of 0 ms.

The timeout configuration applies to all open file descriptors. If more than one Spacewire device is opened, it is not possible to control the timeout configuration for a specific file descriptor.

| Command | Value type | Direction | Description |
|---|---|---|---|
| SPWN_IOCTL_READ_TIMEOUT_ENABLE | uint32_t | in | 1 = Enables SpW read timeout<br>0 = Disables SpW read timeout (default) |
| SPWN_IOCTL_READ_TIMEOUT_DURATION_SET | uint32_t | in | Sets the duration of the read timeout in milliseconds. Default is 1000 ms. |
| SPWN_IOCTL_WRITE_TIMEOUT_ENABLE | uint32_t | in | 1 = Enables SpW write timeout<br>0 = Disables SpW write timeout (default) |
| SPWN_IOCTL_WRITE_TIMEOUT_DURATION_SET | uint32_t | in | Sets the duration of the write timeout in milliseconds. Default is 1000 ms. |

| Return value | Description |
|---|---|
| 0 | Given mode was set |
| -1 | see *errno* values |
| **errno values** | |
| EBADF | The file descriptor fd is not an open file descriptor. |
| EINVAL | Invalid command, or invalid mode value. |

#### 5.7.2.5.3. Timing mode and Timecodes

| Command | Value type | Direction | Description |
|---|---|---|---|
| SPWN_IOCTL_TIMING_MODE_SET | spwn_timing_mode_t | In | Sets the timing mode. Encoded as:<br><br>0 – Timing mode disabled<br>1 – Timing mode Master<br>2 – Timing mode Slave |

| Command | Value type | Direction | Description |
|---|---|---|---|
| SPWN_IOCTL_TIMING_MODE_GET | spwn_timing_mode_t | Out | Gets the current timing mode. Encoded as:<br><br>0 - Timing mode disabled<br>1 - Timing mode Master<br>2 - Timing mode Slave |
| SPWN_IOCTL_TIMECODE_GET | uint8_t | Out | Gets the current time code. |

| Return value | Description |
|---|---|
| 0 | Success |
| -1 | Failure, see *errno* values |
| **errno values** | |
| EBADF | The file descriptor fd is not an open file descriptor. |
| EINVAL | Invalid command, or invalid timing mode. |

#### 5.7.2.5.4. Write with Hardware RMAP CRC Support

Depending on the SoC configuration, the spacewire driver can support hardware-accelerated CRC calculation for transmission of RMAP packets.

The SoC support status for this feature is specified in [RD8]. It is also possible to dynamically detect the support status of this feature using the SPWN_IOCTL_SUPPORT_INFO_GET command described in Section 5.7.2.5.6.

If this feature is used on a supported SoC, the CRC fields will be replaced with calculated values during the transmission of the RMAP packet.

In order to make use of this feature, writing must be performed via the SPWN_IOCTL_WRITE_WITH_RMAP_CONTROL ioctl command.

ioctl calls using this command are blocked until the packet is transmitted or a configured timeout has elapsed; the timeout behaviour is identical to the write function. See Section 5.7.2.4 for details regarding the write timeout behaviour.

The details of the SPWN_IOCTL_WRITE_WITH_RMAP_CONTROL ioctl command are shown below.

| Command | Value type | Direction | Description |
|---|---|---|---|
| SPWN_IOCTL_WRITE_WITH_RMAP_CONTROL | spwn_write_with_rmap_control_t | In | Write with hardware RMAP CRC support. |

The fields in the spwn_write_with_rmap_control_t struct are described below.

| Field | Type | Direction | Description |
|---|---|---|---|
| rmap | spwn_rmap_tx_control_t | out | RMAP transmit control. |
| buf | const void * | out | Write buffer. |
| size | size_t | out | Number of bytes to transmit from buf. Must not be larger than SPWN_MAX_PACKET_SIZE. |

The fields in the spwn_rmap_tx_control_t struct are described below.

| Field | Type | Direction | Description |
|---|---|---|---|
| do_replace_crc | 1-bit unsigned int bit-field member | in | Flag indicating if hardware should calculate and replace the RMAP CRC field(s) in the transmitted packet.<br><br>For hardware RMAP CRC calculation to function correctly, the write buffer must contain an RMAP packet which is valid except for the CRC field(s); the CRC fields(s) can be set to any value.<br><br>For RMAP packet types which do not contain a data field, only the header CRC will be calculated and replaced. For RMAP packet types which do contain a data field, both the header CRC and data CRC will be calculated and replaced.<br><br>If this flag is not set, no RMAP CRC replacement will occur and the value of header_offset will be ignored. |

| Field | Type | Direction | Description |
|---|---|---|---|
| `header_offset` | 8-bit `unsigned int` bit-field member | in | Offset in bytes to start of RMAP header from start of write buffer.<br><br>For hardware RMAP CRC calculation to function correctly, this value must be set to indicate the start of the RMAP header in the write buffer, excluding the spacewire path address.<br><br>For RMAP commands, this must indicate the location of the target logical address field and must exclude the target spacewire address field.<br><br>For RMAP replies, this must indicate the location of the initiator logical address field and must exclude the reply spacewire address field. |

The possible return and errno values for the `SPWN_IOCTL_WRITE_WITH_RMAP_CONTROL` command are shown below.

| Return value | Description |
|---|---|
| `0` | Success |
| `-1` | Failure, see *errno* values |
| **errno values** | |
| `EBADF` | The file descriptor `fd` is not a valid file descriptor for writing. |
| `EINVAL` | Invalid command, or packet size is larger than `SPWN_MAX_PACKET_SIZE`. |
| `ENOSYS` | Hardware-accelerated CRC calculation for RMAP transmission is not supported. |
| `EBUSY` | Transmission already in progress. |
| `EIO` | Internal RTEMS resource error, or internal transmission error. |
| `ETIME` | The write operation timed out. |

#### 5.7.2.5.5. Read with Hardware RMAP CRC Support

Depending on the SoC configuration, the spacewire driver can support hardware-accelerated CRC calculation for reception of RMAP packets.

The SoC support status for this feature is specified in [RD8]. It is also possible to dynamically detect the support status of this feature using the `SPWN_IOCTL_SUPPORT_INFO_GET` command described in Section 5.7.2.5.6.

If this feature is used on a supported SoC, the CRCs will be calculated during the reception of the RMAP packet and the result will be made available when the reception has finished.

In order to make use of this feature, reading must be performed via the `SPWN_IOCTL_READ_WITH_RMAP_INFO` ioctl command.

`ioctl` calls using this command are blocked until a packet is received or a configured timeout has elapsed; the timeout behaviour is identical to the `read` function. See Section 5.7.2.3 for details regarding the read timeout behaviour.

Similar to the `read` function, only one task may read from a file descriptor at a time.

The details of the `SPWN_IOCTL_READ_WITH_RMAP_INFO` ioctl command are shown below.

| Command | Value type | Direction | Description |
|---|---|---|---|
| `SPWN_IOCTL_READ_WITH_RMAP_INFO` | `spwn_read_with_rmap_info_t` | Out | Read with hardware RMAP CRC support. |

The fields in the `spwn_read_with_rmap_info_t` struct are described below.

| Field | Type | Direction | Description |
|---|---|---|---|
| `rmap` | `spwn_rmap_rx_info_t` | out | RMAP receive information. |
| `buf` | `const void *` | out | Read buffer. |

| Field | Type | Direction | Description |
|---|---|---|---|
| `size` | `size_t` | in/out | Maximum and actual number of bytes in read buffer.<br><br>As input, this indicates the maximum number of bytes to read into `buf`, it must be set to a value between `0` and `SPWN_MAX_PACKET_SIZE`.<br><br>As output, this indicates the actual number of bytes received into `buf` on success. |

The fields in the `spwn_rmap_rx_info_t` struct are described below.

| Field | Type | Direction | Description |
|---|---|---|---|
| `has_protocol_rmap` | 1-bit `unsigned int` bit-field member | out | Flag indicating if the received packet is an RMAP packet according to its protocol field.<br><br>If this flag is not set, the values of all other fields in this struct are unspecified. |
| `has_packet_with_data` | 1-bit `unsigned int` bit-field member | out | Flag indicating if the received packet is an RMAP packet that contains data according to its instruction field.<br><br>If this flag is not set, the values of the `has_data_crc_error`, `has_data_crc_match` and `data_crc` fields are unspecified. |
| `has_header_crc_error` | 1-bit `unsigned int` bit-field member | out | Flag indicating if an error occurred during hardware calculation of the RMAP header CRC in the received packet.<br><br>If this flag is set, it likely indicates an RMAP format error in the received packet.<br><br>If this flag is set, the value of the `has_header_crc_match` and `header_crc` fields are unspecified. |

| Field | Type | Direction | Description |
|---|---|---|---|
| has_data_crc_error | 1-bit unsigned int bit-field member | out | Flag indicating if an error occurred during hardware calculation of the RMAP data CRC in the received packet.<br><br>If this flag is set, it likely indicates an RMAP format error in the received packet.<br><br>If this flag is set, the value of the has_data_crc_match and data_crc fields are unspecified. |
| has_header_crc_match | 1-bit unsigned int bit-field member | out | Flag indicating if the hardware-calculated header CRC matched the header CRC in the packet. |
| has_data_crc_match | 1-bit unsigned int bit-field member | out | Flag indicating if the hardware-calculated data CRC matched the data CRC in the packet. |
| header_crc | 8-bit unsigned int bit-field member | out | Hardware-calculated header CRC. |
| data_crc | 8-bit unsigned int bit-field member | out | Hardware-calculated data CRC. |

The possible return and errno values for the SPWN_IOCTL_READ_WITH_RMAP_INFO command are shown below.

| Return value | Description |
|---|---|
| 0 | Success |
| -1 | Failure, see *errno* values |
| **errno values** | |
| EBADF | The file descriptor fd is not a valid file descriptor for reading. |
| EINVAL | Invalid command, or the size parameter is larger than SPWN_MAX_PACKET_SIZE. |
| ENOSYS | Hardware-accelerated CRC calculation for RMAP reception is not supported. |

| Return value | Description |
|---|---|
| EIO | Internal RTEMS resource error. |
| EBUSY | Receive descriptor not currently available. |
| EOVERFLOW | Packet size overflow occurred on reception, no packet data is available. |
| ETIMEDOUT | EEP received. Received packet is incomplete, no packet data is available. |
| ETIME | The read operation timed out and no packet was received. |

#### 5.7.2.5.6. Supported Features Information

Depending on the SoC configuration, the spacewire driver may or may not support certain features. The support status of these features can be dynamically obtained using the SPWN_IOCTL_SUPPORT_INFO_GET ioctl command. The details of this command are shown below.

| Command | Value type | Direction | Description |
|---|---|---|---|
| SPWN_IOCTL_SUPPORT_INFO_GET | spwn_support_info_t | Out | Get SoC feature support information. |

The fields in the spwn_support_info_t struct are described below.

| Field | Type | Direction | Description |
|---|---|---|---|
| has_rx_rmap_hardware_crc | 1-bit unsigned int bit-field member | out | Flag indicating if hardware RMAP CRC is supported for reception. |
| has_tx_rmap_hardware_crc | 1-bit unsigned int bit-field member | out | Flag indicating if hardware RMAP CRC is supported for transmission. |

The possible return and errno values for the SPWN_IOCTL_SUPPORT_INFO_GET command are shown below.

| Return value | Description |
|---|---|
| 0 | Success |
| -1 | Failure, see *errno* values |
| **errno values** | |

| Return value | Description |
|---|---|
| EBADF | The file descriptor fd is not a valid file descriptor for reading. |
| EINVAL | Invalid command. |

#### 5.7.3. Usage description

The following #define needs to be set by the user application to be able to use the spacewire:

- CONFIGURE_APPLICATION_NEEDS_SPACEWIRE_DRIVER

This will automatically initialise the driver at boot up.

The driver provides SpaceWire link setup and data transaction via the SpaceWire device. Each application that wants to communicate via the SpaceWire device must register for operation in normal or promiscuous mode.

##### 5.7.3.1. Normal Operation

Registration to a logical address is performed by calling open with a device name consisting of the predefined string SPWN_DEVICE_0_NAME_PREFIX concatenated with a string corresponding to the chosen logical address number.

Deregistration is performed via close().

Multiple logic addresses may be registered at the same time. But each individual logic address may only be registered for read and write once at the same time.

Logical addresses between 0 - 31 and 255 are reserved by the ESA's ECSS SpaceWire standard [RD10] and cannot be registered to.

##### 5.7.3.2. Promiscuous Mode

In promiscuous mode only one reader task is needed for reading on all logical addresses. All the received SpaceWire packets are passed to the calling application, allowing the application to handle the received packets and perform additional routing if required. The write operation is unchanged; it has the same functionality as in normal operation.

For opening a spacewire device in promiscuous mode, open shall be called with the device name SPWN_PROMISCUOUS_DEVICE_NAME.

Deregistration is performed via close().

#### 5.7.3.3. Buffer Alignment

A reception packet buffer must be aligned to 4 bytes in order to handle the packet's reception correctly. It is therefore recommended to assign the reception buffer in the following way:

```
uint8_t __attribute__ ((aligned (4)) buf_rx[PACKET_SIZE];
```

#### 5.7.3.4. Usage

The RTEMS driver must be opened before it can be used to access the SpaceWire device. Once opened, all provided RTEMS API operations can be used as described in Section 5.7.2 and, if desired, the access can be closed when not needed.

**NOTE** The data rate depends on the packet size and the transmission rate of the SpaceWire IP core. The larger the packet size, the higher the data rate.

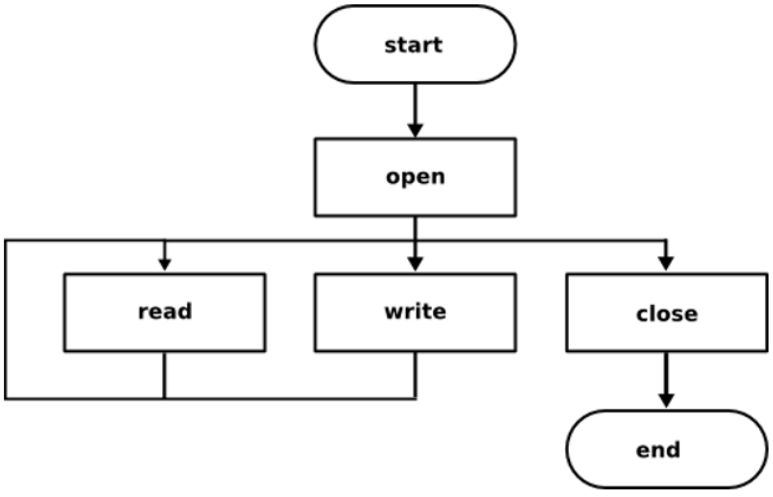



Figure 5.14 - RTEMS driver usage description

**NOTE** All calls to RTEMS driver are blocking calls.

#### 5.7.3.5. Application Usage Example

The following code shows an example of using the driver to transmit and receive a SpaceWire packet.

```
#include <stdlib.h> /* For exit(). */
#include <stdint.h> /* For uint8_t. */
```

```
#include <stdio.h> /* For printf() and perror(). */
#include <unistd.h> /* For write() and read(). */
#include <fcntl.h> /* For open(). */
#include <rtems.h> /* For Init() task. */
#include <bsp/spacewire_node_rtems.h> /* For SpaceWire driver specifics. */

/* Aligned for reception via DMA. */
uint8_t __attribute__((aligned(4))) read_buf[SPWN_MAX_PACKET_SIZE];

rtems_task Init(rtems_task_argument ignored)
{
  (void)ignored;
  /* Receive packets directed to logical address 254. */
  int fd = open(SPWN_DEVICE_0_NAME_PREFIX "254", O_RDWR);
  if (fd < 0) {
    perror("Failed to open");
    exit(EXIT_FAILURE);
  }
  uint8_t write_buf[] = {
    0x03, /* SpaceWire address. */
    0xFE, 0xAA, 0xBB, 0xCC, 0xDD,
  };
  ssize_t size = write(fd, write_buf, sizeof(write_buf));
  if (size < 0) {
    perror("Failed to write");
    exit(EXIT_FAILURE);
  }
  size = read(fd, &read_buf, sizeof(read_buf));
  if (size < 0) {
    perror("Failed to read");
    exit(EXIT_FAILURE);
  }
  printf("Received packet with size %zd\n", size);
  exit(EXIT_SUCCESS);
}

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_SPACEWIRE_DRIVER
/* Strict object limits omitted. */
#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_UNIFIED_WORK_AREAS
#define CONFIGURE_MAXIMUM_DRIVERS 2
/* stdin, stdout, stderr, and descriptor opened by application. */
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS (3 + 1)
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT
#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>
```

#### 5.7.3.6. Hardware RMAP CRC Examples

The following code shows an example of using hardware-accelerated CRC calculation for transmission of an RMAP packet.

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <rtems.h>
#include <spacewire_node_rtems.h>

rtems_task Init(rtems_task_argument ignored)
{
  (void)ignored;
  int fd = open(SPWN_DEVICE_0_NAME_PREFIX "254", O_RDWR);
  if (fd < 0) {
    perror("Failed to open");
    exit(EXIT_FAILURE);
  }
  uint8_t buf[] = {
    0x01, 0x03, /* Target SpaceWire address. */
    /* RMAP read command. */
    0xFE, 0x01, 0x4C, 0x00, 0x67, 0x00, 0x01, 0x00, 0xA0, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x10,
    0x00, /* Header CRC, will be replaced by hardware. */
  };
  size_t target_address_size = 2;
  spwn_write_with_rmap_control_t param;
  param.rmap.header_offset = target_address_size;
  param.rmap.do_replace_crc = 1;
  param.buf = buf;
  param.size = sizeof(buf);
  int status = ioctl(fd, SPWN_IOCTL_WRITE_WITH_RMAP_CONTROL, &param);
  if (status < 0) {
    perror("Failed to write");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_SPACEWIRE_DRIVER
#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_UNIFIED_WORK_AREAS
#define CONFIGURE_MAXIMUM_DRIVERS 2
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS (3 + 1)
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
```

```
#define CONFIGURE_INIT
#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>
```

The following code shows an example of using hardware-accelerated CRC calculation for reception of an RMAP packet.

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <rtems.h>
#include <spacewire_node_rtems.h>

uint8_t __attribute__((aligned(4))) buf[SPWN_MAX_PACKET_SIZE];

rtems_task Init(rtems_task_argument ignored)
{
  (void)ignored;
  int fd = open(SPWN_DEVICE_0_NAME_PREFIX "254", O_RDWR);
  if (fd < 0) {
    perror("Failed to open");
    exit(EXIT_FAILURE);
  }
  spwn_read_with_rmap_info_t param;
  param.buf = buf;
  param.size = sizeof(buf);
  int status = ioctl(fd, SPWN_IOCTL_READ_WITH_RMAP_INFO, &param);
  if (status < 0) {
    perror("Failed to read");
    exit(EXIT_FAILURE);
  }
  spwn_rmap_rx_info_t info = param.rmap;
  if (info.has_protocol_rmap &&
      !info.has_header_crc_error &&
      info.has_header_crc_match) {
    printf("Received RMAP packet with valid header CRC.\n");
  }
  exit(EXIT_SUCCESS);
}

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_SPACEWIRE_DRIVER
#define CONFIGURE_UNLIMITED_OBJECTS
#define CONFIGURE_UNIFIED_WORK_AREAS
#define CONFIGURE_MAXIMUM_DRIVERS 2
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS (3 + 1)
```

```
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT
#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>
```

## 5.8. GPIO

### 5.8.1. Description

This section describes the driver as one utility for accessing the GPIO device.

#### 5.8.1.1. Driver

This driver software for the GPIO IP, handles the setting and reading of general purpose input/output pins. It implements the standard set of device file operations according to [RD11].

The GPIO IP has, apart from logical pin and input/output operations, also a number of other features.

#### 5.8.1.2. Falling and rising edge detection

Once configured, the GPIO IP can detect rising or falling edges on a pin and alert the driver software by the means of an interrupt. In order to not get false edge-detections the procedure to turn on edge detection is.

1. Enable edge detection.
2. Set direction to input.
3. Call read() function.

#### 5.8.1.3. Time stamping in SCET

Instead, or in addition to the interrupt, the GPIO IP can also signal the SCET to sample the current timer when a rising or falling edge is detected on a pin. Reading the time of the timestamp requires interaction with the SCET and exact register address depends on the current board configuration. One SCET sample register is shared by all GPIOs.

#### 5.8.1.4. RTEMS differential mode

In RTEMS finally, a GPIO pin can also be set to operate in differential mode on output only. This requires two pins working in tandem and if this functionality is enabled, the driver will automatically adjust the setting of the paired pin to output mode as well. When in differntial mode the pins are paired in logical sequence, which means that pin 0 and 1 are paired, pin 2 and 3 are paired etc. If setting the first pin in the pair to a value, the second pin in the pair will have the inverted value. If setting the second pin in the pair to a value, the first pin in the pair will have the inverted value. Thus, in differential mode it is recommended to operate on the lower numbered pin only to avoid confusion. Pins can be set in differential mode on a specific pair only, i.e. both normal single ended and differential mode pins can be combined in a

configuration and operate simultaneously (though not within a pin pair).

| Pin that is set | Value | Pin 0 Value | Pin 1 Value |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

#### 5.8.1.5. Operating on pins with pull-up or pull-down

For scenarios when one or multiple pins are connected to a pull-up or pull-down (for e.g. open-drain operation), it's recommended that the output value of such a pin should always be set to 1 for pull-down or 0 for pull-up mode. The actual pin value should then be selected by switching between input or output mode on the pin to comply with the external pull feature.

### 5.8.2. API

This API represents the driver interface of the module from an RTEMS user application's perspective.

The driver functionality is accessed through the RTEMS POSIX API for ease of use. In case of a failure on a function call, the *errno* value is set for determining the cause.

#### 5.8.2.1. Function: `int open(…)`

Opens access to the specified GPIO pin, but do not reset the pin interface and instead retains the settings from any previous access.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `pathname` | `const char*` | in | The absolute path to the GPIO pin to be opened. All possible paths are given by "/dev/gpioX" where X matches 0-31. The actual number of devices available depends on the current hardware configuration. |
| `flags` | `int` | in | Access mode flag: `O_RDONLY`, `O_WRONLY` or `O_RDWR` |

| Return value | Description |
|---|---|
| `fd` | A file descriptor for the device on success |

| Return value | Description |
|---|---|
| `-1` | See *errno* values |
| **errno values** | |
| `EALREADY` | Device is already open |
| `EINVAL` | Invalid options |

#### 5.8.2.2. Function: `int close(…)`

Closes access to the GPIO pin.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |
| `-1` | See *errno* values |
| **errno values** | |
| `EINVAL` | Invalid options |

#### 5.8.2.3. Function: `ssize_t read(…)`

Reads the current value of the specified GPIO pin. If no edge detection have been enabled, this call will return immediately. With edge detection enabled, this call will block with a timeout until the pin changes status such that it triggers the edge detection. The timeout can be adjusted using an ioctl command, but defaults to zero - blocking indefinitely, see Section 5.8.2.5.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `buf` | `void*` | in | Pointer to character buffer to put the read data in. |
| `count` | `size_t` | in | Number of bytes to read, must be set to 1. |

| Return value | Description |
|---|---|
| `>=0` | Number of bytes that were read. |
| `-1` | See *errno* values |
| **errno values** | |

| Return value | Description |
|---|---|
| EINVAL | Invalid options |
| ETIMEDOUT | Driver timed out waiting for the edge detection to trigger |

#### 5.8.2.4. Function: `ssize_t write(…)`

Sets the output value of the specified GPIO pin. If the pin is in input mode, the write is allowed, but its value will not be reflected on the pin until it is set in output mode.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| buf | const void* | in | Pointer to character buffer to get the write data from. |
| count | size_t | in | Number of bytes to write, must be set to 1. |

| Return value | Description |
|---|---|
| >=0 | Number of bytes that were written. |
| -1 | See *errno* values |
| **errno values** | |
| EINVAL | Invalid options |

#### 5.8.2.5. Function: `int ioctl(…)`

The input/output control function can be used to configure the GPIO pin as a complement to the simple data settings using the read/write file operations.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| cmd | int | in | Command to send. |
| val | void* | in/out | Data according to the specific command. |

| Command table | Type | Direction | Description |
|---|---|---|---|
| GPIO_IOCTL_GET_DIRECTION | uint32_t* | out | Get input/output direction of the pin.<br><br>0 output mode<br>1 input mode |
| GPIO_IOCTL_SET_DIRECTION | uint32_t* | in | Set input/output direction of the pin.<br><br>0 output mode<br>1 input mode |
| GPIO_IOCTL_GET_FALL_EDGE_DETECTION | uint32_t* | out | Get falling edge detection status of the pin.<br><br>0 detection disabled<br>1 detection enabled |
| GPIO_IOCTL_SET_FALL_EDGE_DETECTION | uint32_t* | in | Set falling edge detection configuration of the pin.<br><br>0 detection disabled<br>1 detection enabled |
| GPIO_IOCTL_GET_RISE_EDGE_DETECTION | uint32_t* | out | Get rising edge detection status of the pin.<br><br>0 detection disabled<br>1 detection enabled |
| GPIO_IOCTL_SET_RISE_EDGE_DETECTION | uint32_t* | in | Set rising edge detection configuration of the pin.<br><br>0 detection disabled<br>1 detection enabled |
| GPIO_IOCTL_GET_TIMESTAMP_ENABLE | uint32_t* | out | Get timestamp enable status of the pin.<br><br>0 timestamp disabled<br>1 timestamp enabled |
| GPIO_IOCTL_SET_TIMESTAMP_ENABLE | uint32_t* | in | Set timestamp enable configuration of the pin.<br><br>0 timestamp disabled<br>1 timestamp enabled |

| Command table | Type | Direction | Description |
|---|---|---|---|
| GPIO_IOCTL_GET_DIFF_MODE | uint32_t* | out | Get differential mode status of the pin.<br><br>0 normal, single ended, mode<br>1 differential mode |
| GPIO_IOCTL_SET_DIFF_MODE | uint32_t* | in | Set differential mode configuration of the pin.<br><br>0 normal, single ended, mode<br>1 differential mode |
| GPIO_IOCTL_GET_EDGE_TIMEOUT | uint32_t* | out | Get the edge trigger timeout value in ticks. Defaults to zero which means wait indefinitely.<br><br>` |
| GPIO_IOCTL_SET_EDGE_TIMEOUT | uint32_t* | in | Set the edge trigger timeout value in ticks. Zero means wait indefinitely. |

| Return value | Description |
|---|---|
| 0 | Command executed successfully |
| -1 | See *errno* values |
| **errno values** | |
| EINVAL | Invalid options |

### 5.8.3. Usage description

The following `#define` needs to be set by the user application to be able to use the GPIO:

- `CONFIGURE_APPLICATION_NEEDS_GPIO_DRIVER`

#### 5.8.3.1. RTEMS application example

In order to use the GPIO driver in the RTEMS environment, the following code structure is suggested to be used:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <errno.h>
#include <bsp/gpio_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_GPIO_DRIVER

#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM

#define CONFIGURE_MAXIMUM_DRIVERS                15
#define CONFIGURE_MAXIMUM_SEMAPHORES             20
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 30

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 20

#define CONFIGURE_INIT

#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

rtems_task Init(rtems_task_argument argument)
{
  rtems_status_code status;
  int gpio_fd;
  uint32_t buffer;
  uint32_t config;
  ssize_t size;

  gpio_fd = open("/dev/gpio0", O_RDWR);
  config = GPIO_DIRECTION_IN;
  status = ioctl(gpio_fd, GPIO_IOCTL_SET_DIRECTION, &config);
  size = read(gpio_fd, &buffer, 1);
  status = close(gpio_fd);
}
```

- Inclusion of `<fcntl.h>` and `<unistd.h>` are required for using the POSIX functions: open, close, read, write and ioctl.
- Inclusion of `<errno.h>` is required for retrieving error values on failures.
- Inclusion of `<bsp/gpio_rtems.h>` is required for accessing the GPIO.

#### 5.8.4. Limitations

Differential mode works on output only.

## 5.9. CCSDS

### 5.9.1. Description

This section describes the driver as a utility for accessing the CCSDS IP.

#### 5.9.1.1. Driver

On the telemetry, the frames are encoded with Reed Solomon encoding that conforms to the CCSDS standard with an RS(255,223) encoder implementation and an interleaving depth of 5. That makes a total frame length of 1115 bytes. The standard RS polynomial is used.

On the telecommands the BCH decoder (63,56) supports the error correcting mode. The BCH decoder cannot be disabled.

The driver can be configured to handle all available interrupts from the CCSDS IP:

- Pulse commands (CPDU)
- Timestamping of telemetry, see [RD8] for details.
- DMA transfer finished.
- Telemetry transfer frame error.
- Telecommand rejection due to error in the incoming telecommand.
- Telecommand frame buffer errors.
- Telecommand frame buffer overflow.
- Telecommand successfully received.

Telemetry is sent as blocks of TM Space packets of maximum block size of $2^{17}$ bytes. When using the RTEMS driver, Telemetry is sent by writing to a writable device. The device can be opened in non-blocking or blocking mode described chapters below. Up to 8 virtual channels for telemetry are supported by the CCSDS IP and driver. For telecommands, 64 virtual channels are supported.

The actual allocation and availability of virtual channels is described in [RD8].

#### 5.9.1.2. Non-blocking

In non-blocking mode for the RTEMS driver, a write access is done without waiting for a response from the IP before returning from the write-call. During non-blocking transfer of a chunk of data with a maximum size of four times the maximum descriptor length, the sequence below is executed:

1. The address DMA transfer of next available descriptor is set.

2. DESC LENGTH, TM PRESENT, IRQ EN, WRAP is set of next available descriptor.
3. If the data to send needs several descriptors, steps 1 and 2 are repeated until all data in the data-chunk has been transferred. . When a DMA transfer is finished, an interrupt is generated, and the interrupt status indicates which VC`s that were involved in the DMA transfers.
4. The TM Status of the actual VC is read, which will get the last descriptor for the last DMA transfer of that VC. When the TM Status is read, the interrupt is cleared.
5. The driver reads status of the descriptor transfers since the last DMA transfers on the actual virtual channel and prepares messages of the type described in Section 5.9.2.3 and sent to a message queue, named “CCSQ”, provided by the driver. The user-application of the ccsds-driver must implement a listener of the message queue and take actions if an error occurred during transfer.
6. Steps 4 to 6 are repeated for all VC`s signaling an interrupt.

#### 5.9.1.3. Blocking

In blocking mode for the RTEMS driver, a DMA finished interrupt must occur before the write call is returned. The user of the driver does not need to prepare any transfer list or implement a listener of the message queue.

#### 5.9.1.4. Buffer data containing TM Space packets

TM Space packets can be packed within the same buffer, but a TM Space packet must not be split over two different buffers. The first byte of the buffer must always start with a TM Space packet. Data can be padded at the end, with padding byte value of 0xF5. The padding data will not be sent to ground.

### 5.9.2. API

This API represents the driver interface from a user application’s perspective for the RTEMS driver.

The driver functionality is accessed through the RTEMS POSIX API for ease of use. In case of failure on a function call, `errno` value is set for determining the cause.

#### 5.9.2.1. Device-file names

Access to the CCSDS-driver from an application is provided by different device-files (depending on the used SoC configuration [RD8], some devices might not be available):

- `/dev/ccsds` that is used for configuration and status for common TM and TC functionality in the IP. Is defined as `CCSDS_NAME` in RTEMS driver interface file.

- `/dev/ccsds-tm` that is used for configuration and status of the TM path common for all virtual channels. Is defined as `CCSDS_NAME_TM` in RTEMS driver interface file.
- `/dev/ccsds-tmN`, where `N` is to be replaced by the VC number in the range [0..6] (if supported according to the SoC configuration [RD18]). Used for sending telemetry on virtual channel N. The names are defined as `CCSDS_NAME_TM_VCN` in the RTEMS driver interface file. (TM VC7 is reserved for idle frames generated in hardware.)
- `/dev/ccsds-tc` is used for configuration and status of the TC path common for all virtual channels. It is also used for reading on all TC virtual channels. Is defined as `CCSDS_NAME_TC`.
- `/dev/ccsds-tc0` and `CCSDS_NAME_TC_VC0` are deprecated aliases for `/dev/ccsds-tc`, they may be removed in future releases.

#### 5.9.2.2. Default configuration

The default configuration of the TM downlink is:

- FECF is included in TM transfer frames.
- Master Channel Frame counter is enabled for telemetry.
- Generation of Idle frames is enabled.
- Pseudo randomization of telemetry is disabled.
- Reed Solomon encoding of telemetry is enabled.
- Convolutional encoding of telemetry is disabled.
- The divisor of the TM clock is set to 25 (giving a bitrate of 1 Mb/s).
- All available interrupts from the CCSDS IP are enabled.
- Generation of OCF/CLCW in TM Transfer frames is enabled.
- TM is disabled.

The default configuration of the TC uplink is:

- Derandomization of telecommands is disabled.

All available interrupts are enabled.

#### 5.9.2.3. Data type `dma_transfer_cb_t`

For TM-devices operated in non-blocking mode (see Section 5.9.1.2) a message with the content below is sent via a message queue for reporting the transfer status.

| **NOTE** | The “adress” element uses a non-standard spelling. |
|---|---|

| Element | Type | Description |
|---|---|---|
| adress | uint32_t | The start address in SDRAM that is fetched during transfer |
| length | uint16_t | The length of the transfer. Can be maximum 65535. |
| vc | uint8_t | The virtual channel of the transfer. |
| status | uint8_t | Status of transfer<br>0 - Not sent<br>1 - Send finished<br>2 - Send error |

#### 5.9.2.4. Data type `tm_config_t`

This datatype is a struct for configuration of the TM path. The elements of the struct are described below:

| Element | Type | Description |
|---|---|---|
| clk_divisor | uint8_t | The divisor of the clock |
| tm_enabled | uint8_t | Enable/disable of telemetry<br>0 - Disable<br>1 - Enable |
| ocf_clcw_enabled | uint8_t | Enable/disable of OCF/CLCW in TM Transfer frames<br>0 - Disable<br>1 - Enable |
| fecf_enabled | uint8_t | Enable/disable of FECF<br>0 - Disable<br>1 - Enable |
| mc_cnt_enabled | uint8_t | Enable/Disable of master channel frame counter<br>0 - Disable<br>1 - Enable |
| idle_frame_enabled | uint8_t | Enable/disable of generation of Idle frames<br>0 - Disable<br>1 - Enable |

| Element | Type | Description |
|---|---|---|
| tm_conv_bypassed | uint8_t | Bypassing of the TM convolutional encoder<br>0 - No bypass<br>1 - Bypass |
| tm_pseudo_rand_bypassed | uint8_t | Bypassing of the TM pseudo randomizer encoder<br>0 - No bypass<br>1 - Bypass |

#### 5.9.2.5. Data type `tc_config_t`

This datatype is a struct for configuration of the TC path. The elements of the struct are described below:

| Element | Type | Description |
|---|---|---|
| tc_derandomizer_bypassed | uint8_t | Bypassing of TC derandomizer.<br>0 - No bypass<br>1 - Bypass |

#### 5.9.2.6. Data type `tm_status_t`

This datatype is a struct to store status parameters of the TM. The elements of the struct are described below:

| Element | Type | Description |
|---|---|---|
| dma_desc_addr | uint8_t | The LSB of the descriptor address giving the DMA Finished interrupt |
| tm_fifo_err | uint8_t | Reports if a FIFO error occurred during transmission of data<br>0 - No Error<br>1 - FIFO Error |
| tm_busy | uint8_t | Reports if a transfer is in progress.<br>0 - No transfer<br>1 - A transfer is in progress |

#### 5.9.2.7. Data type `tc_error_cnt_t`

This datatype is a struct to store error counters of the TC path. The elements of the struct are described below:

| Element | Type | Description |
|---|---|---|
| tc_overflow_cnt | uint8_t | Indicates number of missed TC frames due to overflow in TC Buffers. The counter will wrap around after 255. |
| tc_cpdu_rej_cnt | uint8_t | Indicates number of rejected CPDU commands. The counter will wrap around after 255. |
| tc_buf_rej_cnt | uint8_t | Indicates number of rejected TC commands. The counter will wrap around after 255. |
| tc_par_err_cnt | uint8_t | Indicates number of CRC errors in TC path. The counter will wrap around after 255. |

#### 5.9.2.8. Data type `tm_error_cnt_t`

This datatype is a struct to store error counters of the TM path. The elements of the struct are described below:

| Element | Type | Description |
|---|---|---|
| tm_par_err_cnt | uint8_t | Indicates number of FIFO parity errors in TM path. The counter will wrap around after 255. |

#### 5.9.2.9. Data type `tc_status_t`

This datatype is a struct to store status parameters of the TC path. The elements of the struct are described below:

| Element | Type | Description |
|---|---|---|
| tc_frame_cnt | uint8_t | Number of received TC frames. The counter will wrap around after 255. |
| tc_buffer_cnt | uint16_t | Actual length on the read TC buffer data in bytes. MAX val 1024 bytes. |
| cpdu_line_status | uint16_t | Bits 0-11 show if the corresponding pulse command line was activated by the last command. |
| cpdu_bypass_cnt | uint8_t | Indicates the number of accepted commands. Wraps at 15. |

#### 5.9.2.10. Data type `radio_status_t`

This datatype is a struct to hold radio status. The elements of the struct are described below:

| Element | Type | Description |
|---|---|---|
| tc_sub_carrier | uint8_t | See [RD12] section 4.2.1.8.3 |
| tc_carrier | uint8_t | See [RD12] section 4.2.1.8.2 |

#### 5.9.2.11. Function: `int open(…)`

Opens the devices provided by the CCSDS RTEMS driver. Only one instance of every device can be opened.

**NOTE** Since `/dev/ccsds-tc0` is an alias of `/dev/ccsds-tc` they cannot be opened at the same time.

**NOTE** Non-blocking is activated using the `LIBIO_FLAGS_NO_DELAY` in the mode argument instead of the standard O_NONBLOCK access flag.

| Argument | Type | Direction | Description |
|---|---|---|---|
| filename | char * | in | The absolute path to the file that is to be opened. The name of the descriptor is described in Section 5.9.2 |
| oflags | int | in | Access flags, may be any of:<br>O_RDONLY<br>O_WRONLY<br>O_RDWR |
| mode | int | in | A bitwise 'or' separated list of values that determine the mode of the opened device. If the flag `LIBIO_FLAGS_NO_DELAY` is set, the device is opened in non-blocking mode. Otherwise it is opened in blocking mode. For further info see Section 5.9.1.2. Applies only to devices /dev/ccsds-tm*N*. |

| Return value | Description |
|---|---|
| >=0 | A file descriptor for the device on success |
| -1 | see *errno* values |
| | **errno values** |
| EBUSY | If device already opened |

#### 5.9.2.12. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |

#### 5.9.2.13. Function: `ssize_t write(…)`

To send data on virtual channel `N`, the device descriptor described in Section 5.9.2 ("/dev/ccsds-tm*N*") shall be used. TM needs to be enabled to successfully send telemetry. If the device is opened in blocking mode, the write operation will wait until all data has been transferred before returning. For devices opened in blocking mode and data has not been transferred within 1500 msec, the write call is aborted, and an error is reported. The timeout value is based on expected time of writing $2^{17}$ bytes at lowest TM Bitrate. For devices opened in non-blocking mode, the write call returns immediately, and the status of the transfer is returned by a message available in a message queue of the driver. See Section 5.9.1.2.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `Int` | in | File descriptor received at **open** |
| `buf` | `void *` | in | Character buffer to read data from |
| `nbytes` | `size_t` | in | Number of bytes (0-65535) to write to the device. |

| Return value | Description |
|---|---|
| `>=0` | number of bytes that were written. |
| `-1` | see *errno* values |
| | **errno values** |
| `EIO` | Device not ready for write or write operation is not supported on device |
| `ETIMEDOUT` | A write to a device in blocking mode did not get a response from IP within expected time. |
| `ENOSYS` | TM is not enabled |

#### 5.9.2.14. Function: `ssize_t read(…)`

To read a Telecommand Transfer frame a read-operation on device “/dev/ccsds-tc” shall be used. This call is blocking until a Telecommand Transfer Frame is received.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `buf` | `void *` | in | Character buffer where read data is returned. The max TC frame size, 1024 byte, must be able to fit in the buffer. |
| `nbytes` | `size_t` | in | Maximum number of bytes to read, must be at least 1024 bytes |

| Return value | Description |
|---|---|
| `>=0` | Number of bytes that were read. |
| `-1` | see *errno* values |
| | **errno values** |
| `EINVAL` | Invalid value of nbytes |
| `EIO` | A read operation is not supported on the device. |

#### 5.9.2.15. Function: `int ioctl(…)`

The devices provided by the CCSDS driver support different IOCTL’s.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at **open** |
| `cmd` | `ioctl_command_t` | in | Command to execute. |
| `val` | `void *` | in | The parameter to pass depends on which IOCTL is called. It is described in the table below. |

| Command table | Device | Parameter type | Description |
|---|---|---|---|
| `CCSDS_SET_TM_CONFIG` | `/dev/ccsds-tm` | `tm_config_t *` | Sets a configuration of the TM path. See Section 5.9.2.4 |
| `CCSDS_GET_TM_CONFIG` | `/dev/ccsds-tm` | `tm_config_t *` | Returns the configuration of the TM path. See Section 5.9.2.4 |

| Command table | Device | Parameter type | Description |
|---|---|---|---|
| CCSDS_SET_TC_CONFIG | /dev/ccsds-tc | tc_config_t * | Sets a configuration of the TC path. See Section 5.9.2.5 |
| CCSDS_GET_TC_CONFIG | /dev/ccsds-tc | tc_config_t * | Returns the configuration of the TC path. See Section 5.9.2.5 |
| CCSDS_GET_RADIO_STATUS | /dev/ccsds | radio_status_t * | Gets radio status. See Section 5.9.2.10 |
| CCSDS_GET_TM_STATUS | /dev/ccsds-tm | tm_status_t* | Gets status of TM path. See Section 5.9.2.6 |
| CCSDS_GET_TM_ERR_CNT | /dev/ccsds-tm | tm_error_cnt_t* | Gets the TM error counter. See Section 5.9.2.8 |
| CCSDS_GET_TC_ERR_CNT | /dev/ccsds-tc | tc_error_cnt_t* | Gets the TC error counter. See Section 5.9.2.7 |
| CCSDS_GET_TC_STATUS | /dev/ccsds-tc | tc_status_t* | Gets status of TC path. See Section 5.9.2.9 |
| CCSDS_SET_TC_FRAME_CTRL | /dev/ccsds-tc | uint32_t | Set the TC frame control register.<br>Bit 2-31 unused.<br>Bit 1:<br>0 - No effect<br>1 - Set to signal for the CCSDS IP a telecommand frame has been read.<br>Bit 0:<br>0 - No effect<br>1 - Reset the buffer function in the CCSDS IP |
| CCSDS_ENABLE_TM | /dev/ccsds-tm | N.A | Enable TM. |
| CCSDS_DISABLE_TM | /dev/ccsds-tm | N.A | Disable TM. |
| CCSDS_INIT | /dev/ccsds | N.A. | Reset the configuration of the TM and TC paths to defaults.See Section 5.9.2.2 |
| CCSDS_SET_CLCW | /dev/ccsds-tm | uint32_t | Set the CLCW. See [RD13].<br>Carrier and subcarrier lock bits always reflect the SoC state and will be ignored when set via this command. |
| CCSDS_GET_CLCW | /dev/ccsds-tm | uint32_t* | Get the CLCW. See [RD13] |

| Command table | Device | Parameter type | Description |
|---|---|---|---|
| CCSDS_SET_TM_TIMESTAMP | /dev/ccsds-tm | uint32_t | Set timestamp generation rate for VC 0 frames. The period of the generation is a power of 2 with the input as exponent. Allowed values range from 0 to 8.<br>0 - Take a timestamp for every sent frame.<br>1 - Take a timestamp for every 2nd sent frame.<br>2 - Take a timestamp for every 4th sent frame.<br>...<br>8 - Take a timestamp for every 256th ($2^8$) sent frame.<br>Reading of timestamps is available via the SCET driver interface. |
| CCSDS_GET_TM_TIMESTAMP | /dev/ccsds-tm | uint32_t * | Get period of timestamp generation. See CCSDS_SCET_TM_TIMESTAMP |

| Return value | Description |
|---|---|
| 0 | Command executed successfully |
| -1 | see *errno* values |
| **errno values** | |
| EIO | Unknown IOCTL for device. |
| EINVAL | Invalid input value. |

### 5.9.3. Usage description

#### 5.9.3.1. RTEMS - Send Telemetry

1. Open the devices /dev/ccsds-tm`N (with N=VC), `/dev/ccsds-tm and /dev/ccsds. Set up the TM path by ioctl-call CCSDS_SET_TM_CONFIG on device /dev/ccsds-tm or ioctl CCSDS_INIT on device /dev/ccsds.
2. Prepare the content in SDRAM that will be fetched by DMA-transfer.
3. Write the SDRAM content to the device for the virtual channel to use.

#### 5.9.3.2. RTEMS - Receive Telecommands

1. Open the device `/dev/ccsds-tc` and `/dev/ccsds`. Set up the TC path by ioctl-call `CCSDS_SET_TC_CONFIG` on device `/dev/ccsds-tc` or ioctl `CCSDS_INIT` on device `/dev/ccsds`.
2. Do a read from `/dev/ccsds-tc`, this call will block until a new TC has been received.

#### 5.9.3.3. RTEMS - Application configuration

Inclusion of `<fcntl.h>` and `<unistd.h>` are required for using the POSIX functions `open()`, `close()`, `read()`, `write()` and `ioctl()` to access the CCSDS device.

Inclusion of `<errno.h>` is required for retrieving error values on failures.

Inclusion of `<bsp/ccsds_rtems.h>` is required for datatypes, definitions of IOCTL of device CCSDS.

The define `CONFIGURE_APPLICATION_NEEDS_CCSDS_DRIVER` must be defined to use the CCSDS driver from the application.

See the Board Support Package for example code.

## 5.10. ADC

### 5.10.1. Description

This section describes the driver for accessing the ADC device.

#### 5.10.1.1. Channels

The following ADC channels are available for the Sirius OBC:

| Parameter | Abbreviation | ADC channel |
|---|---|---|
| Analog input | ADC in 0 | 0 |
| Analog input | ADC in 1 | 1 |
| Analog input | ADC in 2 | 2 |
| Analog input | ADC in 3 | 3 |
| Analog input | ADC in 4 | 4 |
| Analog input | ADC in 5 | 5 |
| Analog input | ADC in 6 | 6 |
| Analog input | ADC in 7 | 7 |
| Regulated 1.2V | 1V2 | 8 |
| Regulated 2.5V | 2V5 | 9 |
| Regulated 3.3V | 3V3 | 10 |
| Input voltage | Vin | 11 |
| Input current | Iin | 12 |
| Temperature | Temp | 13 |

The following ADC channels are available for the Sirius TCM:

| Parameter | Abbreviation | ADC channel |
|---|---|---|
| Regulated 1.2V | 1V2 | 8 |
| Regulated 2.5V | 2V5 | 9 |
| Regulated 3.3V | 3V3 | 10 |
| Input voltage | Vin | 11 |
| Input current | Iin | 12 |
| Temperature | Temp | 13 |

The TCM board does not contain any input ADC channels.

#### 5.10.1.2. Data format

When data is read from a channel, the lower 8 bits contains the channel status information, and the upper 24 bits contains the raw ADC data.

To convert the ADC value into mV, mA or m°C, the formulas specified in the table below shall be used. Note that this assumes a 24-bit ADC value which is what the ADC IP returns on read. Should the raw bit value be truncated or scaled down, the scale factor ($2^{24}$-1) in the equations need to be adjusted as well. Note also that the temperature equation requires the 3V3 [mV] value.

| HK channel | Formula |
|---|---|
| Temp [m°C] | `Temp_mV = (ADC_value*2500)/(2^24 - 1)`<br><br>`Temp_mC = (1000*(3V3_mV - Temp_mV) - Temp_mV*1210) / 0.00385*(Temp_mV - 3300)` |
| Iin [mA] | `Iin_mA = (ADC_value*5000)/(2^24 - 1)` |
| Vin [mV] | `Vin_mV = (ADC_value*20575)/(2^24 - 1)` |
| 3V3 [mV] | `3V3_mV = (ADC_value*5000)/(2^24 - 1)` |
| 2V5 [mV] | `2V5_mV = (ADC_value*5000)/(2^24 - 1)` |
| 1V2 [mV] | `1V2_mV =(ADC_value*2525)/(2^24 - 1)` |
| Analog input0 - Analog input 7 [mV] | `Value_mV = (ADC_value*2500)/(2^24 - 1)` |

### 5.10.2. API

This API represents the driver interface of the module from an RTEMS user application's perspective.

The driver functionality is accessed through the RTEMS POSIX API for ease of usage. In case of a failure on a function call, the *errno* value is set for determining the cause.

#### 5.10.2.1. Enum: `adc_ioctl_sample_rate_e`

Enumerator for the ADC sample rate.

| Enumerator | Description |
|---|---|
| `ADC_IOCTL_SPS_31250` | 31250 SPS |

| Enumerator | Description |
|---|---|
| ADC_IOCTL_SPS_15625 | 15625 SPS |
| ADC_IOCTL_SPS_10417 | 10417 SPS |
| ADC_IOCTL_SPS_5208 | 5208 SPS |
| ADC_IOCTL_SPS_2597 | 2597 SPS |
| ADC_IOCTL_SPS_1007 | 1007 SPS |
| ADC_IOCTL_SPS_503_8 | 503.8 SPS |
| ADC_IOCTL_SPS_381 | 381 SPS |
| ADC_IOCTL_SPS_200_3 | 200.8 SPS |
| ADC_IOCTL_SPS_100_5 | 100.5 SPS |
| ADC_IOCTL_SPS_59_52 | 59.52 SPS |
| ADC_IOCTL_SPS_49_68 | 49.68 SPS |
| ADC_IOCTL_SPS_20_01 | 20.01 SPS |
| ADC_IOCTL_SPS_16_63 | 16.63 SPS |
| ADC_IOCTL_SPS_10 | 10 SPS |
| ADC_IOCTL_SPS_5 | 5 SPS |
| ADC_IOCTL_SPS_2_5 | 2.5 SPS |
| ADC_IOCTL_SPS_1_25 | 1.25 SPS |

#### 5.10.2.2. Function: `int open(…)`

Opens access to the ADC. Only read access is allowed and only blocking mode is supported.

| Argument | Type | Direction | Description |
|---|---|---|---|
| pathname | const char * | in | The absolute path to the ADC to be opened. ADC device is defined as ADC_DEVICE_NAME. |
| flags | int | in | Access mode flag, only O_RDONLY is supported. |

| Return value | Description |
|---|---|
| fd | A file descriptor for the device on success |
| -1 | See *errno* values |
| **errno values** | |

| Return value | Description |
|---|---|
| EEXISTS | Device already exists |
| EALREADY | Device is already open |
| EINVAL | Invalid options |

#### 5.10.2.3. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |

| Return value | Description |
|---|---|
| 0 | Device closed successfully |
| -1 | See *errno* values |
| **errno values** | |
| EFAULT | Device not opened |

#### 5.10.2.4. Function: `ssize_t read(…)`

This is a blocking call to read data from the ADC, see Table 5.11 for definition of the data.

**NOTE** The size of the given buffer must be a multiple of 32 bit.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| buf | void* | in | Pointer to buffer to write data into. |
| count | size_t | in | Number of bytes to read. Only 4 bytes is supported in this implementation. |

| Return value | Description |
|---|---|
| >=0 | Number of bytes that were read. |
| -1 | see *errno* values |
| **errno values** | |
| EPERM | Device not open |

| Return value | Description |
|---|---|
| EINVAL | Invalid number of bytes to be read |

Table 5.11 - ADC data buffer bit definition

| Bits | Description |
|---|---|
| 31:8 | ADC value |
| 7:4 | ADC status, see Table 5.12 |
| 3:0 | Channel number |

The ADC status field holds error flags from the ADC chip that can be used to determine the validity of the conversion.

Table 5.12 - ADC Status bits definition

| Bit | Name | Description |
|---|---|---|
| 3 | RDY | The RDY flag goes low when a conversion is finished and is set high when a conversion is started or the data register is read. |
| 2 | ADC_ERROR | The ADC_ERROR bit in the status register flags any errors that occur during the conversion process.<br><br>The flag is set when an overrange or underrange occurs at the output of the ADC. When an underrange or overrange occurs, the ADC also outputs all 0s or all 1s, respectively.<br><br>This flag is reset only when the underrange or overrange is removed. It is not reset by a read of the data register. |
| 1 | CRC_ERROR | If the CRC value that accompanies a write operation does not correspond with the information sent, the CRC_ERROR flag is set. The flag is reset as soon as the status register is explicitly read. |
| 0 | REG_ERROR | The ADC chip calculates a checksum of the on-chip registers. If one of the register values has changed, the REG_ERROR bit is set. |

#### 5.10.2.5. Function: `int ioctl(…)`

Ioctl allows for more in-depth control of the ADC IP like setting the sample mode, clock divisor etc.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open |
| cmd | int | in | Command to send |

| Argument | Type | Direction | Description |
|---|---|---|---|
| `val` | `uint32_t/ uint32_t*` | in/out | Value to write or a pointer to a buffer where data will be written. |

| Command table | Type | Direction | Description |
|---|---|---|---|
| `ADC_SET_SAMPLE_RATE_IOCTL` | `uint32_t` | in | Set the sample rate of the ADC chip, see Section 5.10.2.1. |
| `ADC_GET_SAMPLE_RATE_IOCTL` | `uint32_t` | out | Get the sample rate of the ADC chip, see Section 5.10.2.1. |
| `ADC_SET_CLOCK_DIVISOR` | `uint32_t` | in | Set the clock divisor of the clock used for communication with the ADC chip. Minimum 4 and maximum 255.<br><br>Default is 255. |
| `ADC_GET_CLOCK_DIVISOR` | `uint32_t` | out | Get the clock divisor of the clock used for communication with the ADC chip. |
| `ADC_ENABLE_CHANNEL` | `uint32_t` | in | Enable specified channel number to be included when sampling. Minimum 0 and maximum 15. |
| `ADC_DISABLE_CHANNEL` | `uint32_t` | in | Disable specified channel number to be included when sampling. Minimum 0 and maximum 15. |

| Return value | Description |
|---|---|
| `0` | Command executed successfully |
| `-1` | see *errno* values |
| **errno values** | |
| `RTEMS_NOT_DEFINED` | Invalid IOCTL |
| `EINVAL` | Invalid value supplied to IOCTL |

### 5.10.3. Usage description

The following `#define` needs to be set by the user application to be able to use the ADC:

- CONFIGURE_APPLICATION_NEEDS_ADC_DRIVER

#### 5.10.3.1. RTEMS application example

In order to use the ADC driver on RTEMS environment, the following code structure is suggested to be used:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <bsp/adc_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_ADC_DRIVER

#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

#define CONFIGURE_INIT
rtems_task Init(rtems_task_argument argument);

rtems_task Init(rtems_task_argument argument)
{
  rtems_status_code status;
  int read_fd;
  uint32_t buffer;
  ssize_t size;

  read_fd = open(ADC_DEVICE_NAME, O_RDONLY);
  status = ioctl(read_fd, ADC_ENABLE_CHANNEL_IOCTL, 4);
  size = read(read_fd, &buffer, 4);
  status = ioctl(read_fd, ADC_DISABLE_CHANNEL_IOCTL, 4);
  status = close(read_fd);
}
```

- Inclusion of <fcntl.h> and <unistd.h> are required for using the POSIX functions: open, close, ioctl.
- Inclusion of <errno.h> is required for retrieving error values on failures.
- Inclusion of <bsp/adc_rtems.h> is required for accessing the ADC.

### 5.10.4. Limitations

Only one ADC channel can be enabled at a time. To switch channels, disabling the old and enabling the new channel is required.

Setting the clk divisor to something else than the default (255) might yield that some

ADC reads returns 0.

## 5.11. NVRAM

### 5.11.1. Description

This section describes the driver as one utility for accessing the NVRAM device.

The NVRAM on the Sirius Leon3 TCM is a 262,144-bit magnetoresistive random access memory (MRAM) device organized as 32,768 bytes of 8 bits. EDAC is implemented on a byte basis meaning that half the address space is filled with checksums for correction. It is a strong correction which corrects 1 or 2 bit errors on a byte and detects multiple. The table below presents the address space defined as words (16,384 bytes can be used). The address space is divided into two subgroups as product- and user address space.

#### 5.11.1.1. Driver

This driver software for the SPI RAM IP, handles the initialization, configuration and access of the NVRAM.

The SPI RAM is divided into an in-flight protected “safe” area and an in-flight programmable “update” area. The in-flight protected area must be unlocked by physically connecting the debugger unit before writing.

##### 5.11.1.1.1. EDAC mode

When in EDAC mode, which is the normal mode of operation, all write and read transactions are protected by EDAC algorithms. All NVRAM addresses containing EDAC are hidden by the IP. The address space is given by the table below:

| Area | Range start | Range end |
|---|---|---|
| Safe | `0x0000` | `0x0FFF` |
| Update | `0x1000` | `0x3FFF` |

##### 5.11.1.1.2. Non-EDAC mode

Non-EDAC mode is a debug mode that allows the user to examine the EDAC bytes. The purpose of this mode is to be able to insert errors into the memory for testing of the EDAC algorithm. When in Non-EDAC mode net data and EDAC data is interleaved on an 8 bit basis, i.e. when reading a 32 bit word byte, 0, 2 contains the net data and byte 1, 3 contains EDAC data. The address space is doubled when compared to EDAC mode, as is shown with the table below:

| Area | Range start | Range end |
|---|---|---|
| Safe | 0x0000 | 0x1FFF |
| Update | 0x2000 | 0x7FFF |

#### 5.11.2. API

This API represents the driver interface of the module from an RTEMS user application's perspective.

The driver functionality is accessed through the RTEMS POSIX API for ease of usage. In case of a failure on a function call, the *errno* value is set for determining the cause.

##### 5.11.2.1. Enum: `rtems_spi_ram_edac_e`

Enumerator for the error correction and detection of the SPI RAM.

| Enumerator | Description |
|---|---|
| SPI_RAM_IOCTL_EDAC_ENABLE | Error Correction and Detection enabled. |
| SPI_RAM_IOCTL_EDAC_DISABLE | Error Correction and Detection disabled. |

##### 5.11.2.2. Function: `int open(…)`

Opens access to the requested SPI RAM.

| Argument | Type | Direction | Description |
|---|---|---|---|
| pathname | const char * | in | The absolute path to the SPI RAM to be opened. SPI RAM device is defined as SPI_RAM_DEVICE_NAME. |
| flags | int | in | Specifies one of the access modes in the following table. |

| Flags | Description |
|---|---|
| O_RDONLY | Open for reading only. |
| O_WRONLY | Open for writing only. |
| O_RDWR | Open for reading and writing. |

| Return value | Description |
|---|---|
| fd | A file descriptor for the device on success |
| -1 | See *errno* values in [RD14] |

#### 5.11.2.3. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |

| Return value | Description |
|---|---|
| `0` | Device closed successfully |
| `-1` | See *errno* values in [RD14] |

#### 5.11.2.4. Function: `ssize_t read(…)`

Read data from the SPI RAM. The call block until all data has been received from the SPI RAM.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `buf` | `void*` | in | Pointer to character buffer to write data into. |
| `count` | `size_t` | in | Number of bytes to read. Must be a multiple of 4. |

| Return value | Description |
|---|---|
| `>=0` | Number of bytes that were read. May also set errno `EIO`. |
| `-1` | See *errno* values |
| **errno values** | |
| `EINVAL` | Invalid options |
| `ENODEV` | Internal RTEMS resource error. |
| `EIO` and `>=0` return value | Read was successful and a single or double-bit error was corrected using EDAC. The corrected value has **NOT** been re-written. |
| `EIO` and `-1` return value | Multi-bit uncorrectable read error. |

#### 5.11.2.5. Function: `ssize_t write(…)`

Write data into the SPI RAM. The call block until all data has been written into the SPI

RAM.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `buf` | `void*` | in | Pointer to character buffer to read data from. |
| `count` | `size_t` | in | Number of bytes to write. Must be a multiple of 4. |

| Return value | Description |
|---|---|
| `>=0` | Number of bytes that were written. |
| `-1` | See *errno* values |
| **errno values** | |
| `EINVAL` | Invalid options |
| `ENODEV` | Internal RTEMS resource error. |

#### 5.11.2.6. Function: `int lseek(…)`

Set the address for the read/write operations.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `offset` | `void*` | in | SPI RAM read/write byte offset. Must be a multiple of 4. |
| `whence` | `int` | in | `SEEK_SET` and `SEEK_CUR` are supported. |

| Return value | Description |
|---|---|
| `>=0` | Byte offset |
| `-1` | See *errno* values in [RD14] |

#### 5.11.2.7. Function: `int ioctl(…)`

Input/output control for SPI RAM.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `cmd` | `int` | in | Command to send. |

| Argument | Type | Direction | Description |
|---|---|---|---|
| `val` | `uint32_t/ uint32_t*` | in/out | Value to write or a pointer to a buffer where data will be written. |

| Command table | Type | Direction | Description |
|---|---|---|---|
| `SPI_RAM_SET_EDAC_IOCTL` | `uint32_t` | in | Configures the error correction and detection for the SPI RAM, see Section 5.11.2.1. |
| `SPI_RAM_SET_DIVISOR_IOCTL` | `uint32_t` | in | Configures the serial clock divisor. |
| `SPI_RAM_GET_EDAC_STATUS_IOCTL` | `uint32_t*` | out | Deprecated. Get EDAC status for previous read operations. |
| `SPI_RAM_GET_DEBUG_DETECT_IOCTL` | `uint32_t*` | out | Get Debug detect status. |

| EDAC Status | Description |
|---|---|
| `SPI_RAM_EDAC_STATUS_MULT_ERROR` | Multiple errors detected. |
| `SPI_RAM_EDAC_STATUS_DOUBLE_ERROR` | Double error corrected. |
| `SPI_RAM_EDAC_STATUS_SINGLE_ERROR` | Single error corrected. |

| Debug Detect Status | Description |
|---|---|
| `SPI_RAM_DEBUG_DETECT_TRUE` | Debugger detected. |
| `SPI_RAM_DEBUG_DETECT_FALSE` | Debugger not detected. |

| Return value | Description |
|---|---|
| `0` | Command executed successfully |
| `-1` | See *errno* values |
| **errno values** | |
| `EINVAL` | Invalid options |
| `ENODEV` | Internal RTEMS resource error. |

### 5.11.3. Usage description

The following `#define` needs to be set by the user application to be able to use the SPI RAM:

- `CONFIGURE_APPLICATION_NEEDS_SPI_RAM_DRIVER`

The SPI RAM RTEMS driver supports multiple file descriptors opened simultaneously.

EDAC error information is reported via errors in the read operation, which is the recommended way to obtain this information.

The `SPI_RAM_GET_EDAC_STATUS_IOCTL` command is deprecated and may be removed in future versions.

#### 5.11.3.1. RTEMS Example

In order to use the SPI RAM driver on RTEMS environment, the following code structure is suggested to be used (see Board Support Package for a full example):

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <bsp/spi_ram_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_SPI_RAM_DRIVER

#define CONFIGURE_INIT

#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

rtems_task Init(rtems_task_argument argument);

rtems_task Init(rtems_task_argument argument)
{
  rtems_status_code status;
  int dsc;
  uint8_t buf[8];
  ssize_t cnt;
  off_t offset;

  dsc = open(SPI_RAM_DEVICE_NAME, O_RDWR);
  offset = lseek(dsc, 0x200, SEEK_SET);
  cnt = write(dsc, &buf[0], sizeof(buf));
  offset = lseek(dsc, 0x200, SEEK_SET);
  cnt = read(dsc, &buf[0], sizeof(buf));
  status = close(dsc);
}
```

- Inclusion of `<fcntl.h>` and `<unistd.h>` are required for using the POSIX functions: open, close, ioctl.
- Inclusion of `<errno.h>` is required for retrieving error values on failures.

- Inclusion of <bsp/spi_ram_rtems.h> is required for accessing the SPI_RAM.

## 5.12. System flash

### 5.12.1. Description

The System flash holds the software images for the system as described in Chapter 8. This section details the RTEMS interface to the System flash driver.

#### 5.12.1.1. Overview

In NAND flash the memory area is divided into *pages* that have a data area and a spare area. The pages are grouped into *blocks*. Before data can be programmed to a page it must be erased (all bytes are `0xFF`). The smallest area to erase is a block consisting of a number of pages, so if the block contains any data that needs to be preserved this must first be read out. The driver defines some constants for the application software to use when handling blocks and pages. There are `SYSFLASH_BLOCKS` blocks starting from block number 0 and `SYSFLASH_PAGES_PER_BLOCK` pages within each block starting from page 0. Each page data area is `SYSFLASH_PAGE_SIZE` bytes. Each page also has a spare area that is `SYSFLASH_PAGE_SPARE_AREA_SIZE` bytes. Partial pages can be read/programmed, but reading/programming always starts at the beginning of the page (or spare area). Pages (including spare area) must be programmed in sequence within a block.

With NAND flash memory technology some blocks will be bad from the factory, and more bad blocks will appear due to wear. The driver itself does not manage bad blocks, but it will supply the information needed for the application software to implement a system to keep track of them. A common use for the page spare area is to hold ECC information. However, this system has a more comprehensive EDAC solution, so the main use for the spare area is to hold the factory bad block markers (first byte of the first page spare area is `0x00`). Bad blocks should never be erased or programmed.

#### 5.12.1.2. Debug detect

Erasing blocks/programming pages to the first half of the flash memory (lower addresses) only works when the debug detect signal is high (indicating debugger is connected). If erase/program operations to that area are attempted when the debug detect signal is low they will appear to succeed from a software perspective but the controller will not pass them on to the flash chip.

### 5.12.2. Data Structures

#### 5.12.2.1. Type: `sysflash_cid_t`

This struct type holds the result of reading the system flash chip ID.

| Type | Name | Description |
|---|---|---|
| `uint32_t[2]` | `chip0` | Byte array for chip0 ID |

#### 5.12.2.2. Type: `sysflash_ioctl_spare_area_args_t`

This struct is used by the RTEMS API as the target when reading or writing the spare area.

| Type | Name | Description |
|---|---|---|
| `uint32_t` | `page_num` | What page to read/write. Values: [`0` - `(SYSFLASH_MAX_NO_PAGES-1)`] |
| `uint32_t` | `raw` | Ignored when writing (programming is always done with EDAC and interleaving active). On read, set to 0 to do deinterleaving and EDAC checking, set to 1 to read raw interleaved data without EDAC checking. |
| `uint8_t *` | `data_buf` | Pointer to buffer in which the data is to be stored, or to the data that is to be written. |
| `uint32_t` | `size` | Size to read/write in bytes. Values: [1 - `SYSFLASH_PAGE_SPARE_AREA_SIZE`] |

### 5.12.3. API

This API represents the driver interface from a user application's perspective for the RTEMS driver.

The driver functionality is accessed through RTEMS POSIX API for ease of use. In case of failure on a function call, the `errno` value is set for determining the cause.

> **NOTE** This documentation only lists the most likely errno values and those that have special meaning for this driver. For an exhaustive list please see the Open Group POSIX specification documentation.

#### 5.12.3.1. Function: `int open(…)`

Opens access to the driver. The device can only be opened by one user at a time.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `filename` | `char *` | in | The absolute path to the file that is to be opened. System flash device is defined as `SYSFLASH_DEVICE_NAME`. |

| Argument | Type | Direction | Description |
|---|---|---|---|
| `oflags` | `int` | in | Access mode flags, see Table 5.13. |

| Return Value | Description |
|---|---|
| `>0` | A file descriptor for the device. |
| `-1` | see **errno values** |
| **errno values** | |
| `EBUSY` | Device already opened. |
| `ENODEV` | Internal driver error |

Table 5.13 - open flag symbols

| Access mode | Description |
|---|---|
| `O_RDONLY` | Open for reading only |
| `O_WRONLY` | Open writing only |
| `O_RDWR` | Open for reading and writing |

#### 5.12.3.2. Function: `int close(…)`

Closes access to the device.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |

| Return Value | Description |
|---|---|
| `0` | Device closed successfully |
| `-1` | see **errno values** |
| **errno values** | |
| `EBADF` | The file descriptor `fd` is not an open file descriptor. |

#### 5.12.3.3. Function: `size_t lseek(…)`

Sets page offset for read/write operations.

| NOTE | The interface is not strictly POSIX, as the offset argument is expected to be given in pages and not bytes. |
|---|---|

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| offset | off_t | in | Page number. (**NOTE:** Not bytes!) |
| whence | int | in | Must be set to SEEK_SET. |

| Return Value | Description |
|---|---|
| offset | Page number |
| -1 | see **errno values** |
| **errno values** | |
| EBADF | The file descriptor fd is not an open file descriptor |
| EINVAL | whence is not a proper value. |
| EOVERFLOW | The resulting file offset would overflow off_t. |

#### 5.12.3.4. Function: size_t read(…)

Reads requested size of bytes from the device starting from the offset set using lseek.

| NOTE | For iterative read operations, lseek must be called to set page offset **before** each read operation. |
|---|---|

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| buf | void * | in | Character buffer where to store the data (should be 32-bit aligned for most efficient read). |
| nbytes | size_t | in | Number of bytes to read into buf (should be a multiple of 4 for most efficient read). |

| Return Value | Description |
|---|---|
| >0 | Number of bytes that were read. |
| -1 | see **errno values** |
| **errno values** | |
| EBADF | The file descriptor fd is not an open file descriptor |

| Return Value | Description |
|---|---|
| EINVAL | Page offset set by lseek is out of range or nbytes is too large and reaches a page that is out of range. |
| ENODEV | Internal driver error. |
| EBUSY | Flash controller busy. |

#### 5.12.3.5. Function: size_t write(…)

Writes requested size of bytes to the device starting from the offset set using lseek.

**NOTE** For iterative write operations, lseek must be called to set page offset before each write operation.

| Argument | Type | Direction | Description |
|---|---|---|---|
| fd | int | in | File descriptor received at open. |
| buf | void * | in | Character buffer to write data from (should be 32-bit aligned for most efficient write). |
| nbytes | size_t | in | Number of bytes to write from buf (should be a multiple of 4 for most efficient write). |

| Return Value | Description |
|---|---|
| >0 | Number of bytes that were written. |
| -1 | see **errno values** |
| **errno values** | |
| EBADF | The file descriptor fd is not an open file descriptor |
| EINVAL | Page offset set by lseek is out of range or nbytes is too large and reaches a page that is out of range. |
| ENODEV | Internal driver error. |
| EBUSY | Flash controller busy. |
| EIO | Program failed at chip level, block should be considered bad (double check chip status FAIL flag using SYSFLASH_IO_READ_CHIP_STATUS). |

#### 5.12.3.6. Function: `int ioctl(…)`

Additional supported operations via POSIX Input/Output Control API.

| Argument | Type | Direction | Description |
|---|---|---|---|
| `fd` | `int` | in | File descriptor received at open. |
| `cmd` | `int` | in | Command defined in sections below. |
| `value` | `void *` | in | The value relating to command operation as defined in sections below. |

##### 5.12.3.6.1. Reset System flash

Resets the system flash chip.

| Command | Value type | Direction | Description |
|---|---|---|---|
| `SYSFLASH_IO_RESET` | n/a | n/a | n/a |

##### 5.12.3.6.2. Read chip status

Reads the chip status register.

| Command | Value type | Direction | Description |
|---|---|---|---|
| `SYSFLASH_IO_READ_CHIP_STATUS` | `uint8_t *` | out | Pointer to variable in which status data is to be stored. |

##### 5.12.3.6.3. Read controller status

Reads the controller status register.

| Command | Value type | Direction | Description |
|---|---|---|---|
| `SYSFLASH_IO_READ_CTRL_STATUS` | `uint16_t *` | out | Pointer to variable in which controller status data is to be stored. |

##### 5.12.3.6.4. Read ID

Reads the flash chip ID.

| Command | Value type | Direction | Description |
|---|---|---|---|
| SYSFLASH_IO_READ_ID | sysflash_cid_t * | out | Pointer to struct in which ID is to be stored, see Section 5.12.2.1. |

#### 5.12.3.6.5. Erase block

Erases a block.

| Command | Value type | Direction | Description |
|---|---|---|---|
| SYSFLASH_IO_ERASE_BLOCK | uint32_t | in | Block number to erase. |

| Return value | Description |
|---|---|
| 0 | Operation successful. |
| -1 | See errno values. |
| **errno values** | |
| EIO | Erase failed on chip level; block should be considered bad. |

#### 5.12.3.6.6. Read spare area

Reads the spare area for a given page.

| Command | Value type | Direction | Description |
|---|---|---|---|
| SYSFLASH_IO_READ_SPARE_AREA | sysflash_ioctl_spare_area_args_t * | in | Pointer to struct with page number specifier, and destination buffers where spare area data is to be stored, see Section 5.12.2.2. |

#### 5.12.3.6.7. Write spare area

Writes the data to the given page spare area.

| Command | Value type | Direction | Description |
|---|---|---|---|
| SYSFLASH_IO_WRITE_SPARE_AREA | sysflash_ioctl_spare_area_args_t * | in | Pointer to struct with page number specifier, and source buffer with data to be written, see Section 5.12.2.2. |

| Return value | Description |
|---|---|
| `0` | Operation successful. |
| `-1` | See errno values. |
| **errno values** | |
| `EIO` | Program failed on chip level; block should be considered bad. |

#### 5.12.3.6.8. Factory bad block check

Reads the factory bad block marker from a block and reports status.

**NOTE** This only gives information about factory marked bad blocks. Bad blocks that arise during use need to be handled by the application software.

| Command | Value type | Direction | Description |
|---|---|---|---|
| `SYSFLASH_IO_BAD_BLOCK_CHECK` | `uint32_t` | in | Block number. |

| Return value | Description |
|---|---|
| `SYSFLASH_FACTORY_BAD_BLOCK_CLEARED` | Block is OK. |
| `SYSFLASH_FACTORY_BAD_BLOCK_MARKED` | Block is marked bad. |
| **errno values** | |
| `ETIMEDOUT` | Polled read of spare area timed out. |

### 5.12.4. Usage Description

The RTEMS driver provides the application software with a POSIX file interface for accessing the functionality of the bare-metal driver. However, unlike the POSIX calls where the offset is given in bytes, the Sysflash driver expects the offset to be in pages. The read and write calls provide an abstraction to the page-by-page access in the bare-metal driver, so multiple pages can be read/written with one call, but the application will still need to make sure that pages are erased before they are written.

In RTEMS the device file must be opened to grant access to the system flash device. Once opened, all provided operations can be used as described in Section 5.12.3. If desired, the access can be closed when not needed.

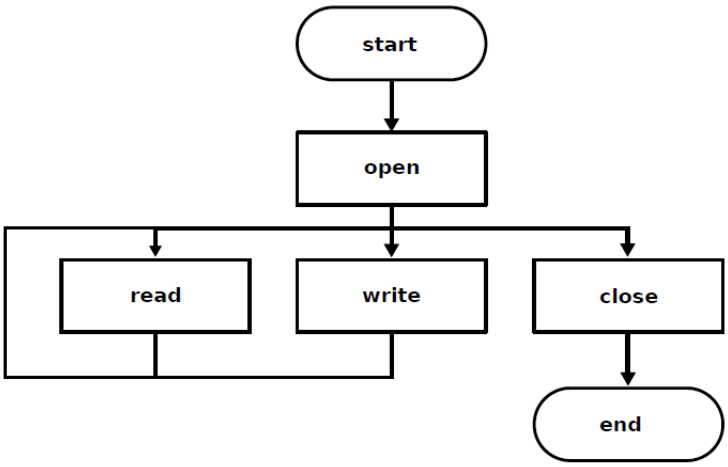



Figure 5.15 - RTEMS driver usage description.

The following `#define` needs to be set by the user application to be able to use the system flash driver:

- `CONFIGURE_APPLICATION_NEEDS_SYSTEM_FLASH_DRIVER`

By defining this as part of RTEMS configuration, the driver will automatically be initialized at boot up.

| NOTE | All calls to the RTEMS driver are blocking calls, though the driver uses interrupts internally to ease processor load. |
|---|---|

#### 5.12.4.1. RTEMS application example

In order to use the system flash driver in the RTEMS environment, the following code structure is suggested to be used:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <bsp/system_flash_rtems.h>

#define CONFIGURE_APPLICATION_NEEDS_SYSTEM_FLASH_DRIVER
/*...*/
#define CONFIGURE_INIT
rtems_task Init(rtems_task_argument argument);

#include <bsp/bsp_confdefs.h>
#include <rtems/confdefs.h>

rtems_task Init(rtems_task_argument ignored)
```

```
{
  /*...*/
  fd = open(SYSFLASH_DEVICE_NAME, O_RDWR);
  /*...*/
}
```

- Inclusion of <fcntl.h> and <unistd.h> are required for using the POSIX functions open, close, lseek, read, write, and ioctl for accessing the driver.
- Inclusion of <errno.h> is required for retrieving error values on failures.
- Inclusion of <bsp/system_flash_rtems.h> is required for driver related definitions.
- Inclusion of <bsp/bsp_confdefs.h> is required to initialise the driver at boot up.

#### 5.12.5. Limitations

The system flash driver may only have one open file descriptor at a time.

The POSIX interface is modified to use an offset in pages instead of bytes.

# 6. SpaceWire router

In both Sirius OBC and Sirius TCM products, a small router is integrated in the SoCs. The routers use path addressing (see [RD10]) and given the topology illustrated in Figure 6.1, the routing addressing can be easily calculated.

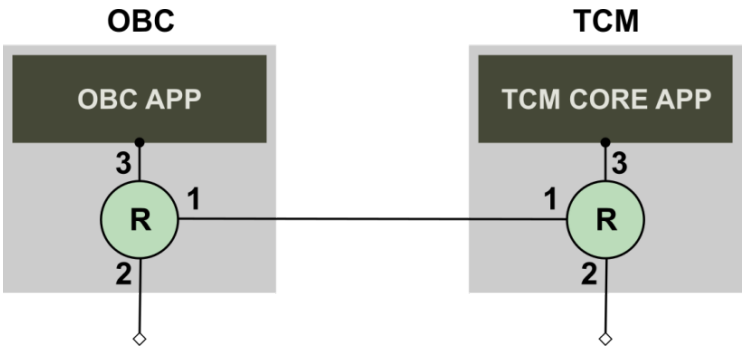



Figure 6.1 - Integrated router location

In the topology above, sending a package from the OBC to the TCM or vice versa, the routing address will be 1-3. Each end node, Sirius OBC or Sirius TCM, also has one or more logical address(es) to help distinguish between different applications or services running on the same node. The logical address complements the path address and must be included in a SpaceWire packet.

Example: If a packet is to be sent from Sirius OBC to the Sirius TCM it needs to be prepended with 0x01 0x03 XX:

- 0x01 routes the packet to port 1 of the Sirius OBC router.
- 0x03 routes the packet to port 3 of the Sirius TCM router.
- XX is the logical address (0x20 – 0xFE) of the recipient application/service on the Sirius TCM.

# 7. NVRAM areas

This chapter is an extension of the RTEMS NVRAM API in Section 5.11 to show how the different areas on NVRAM are used by the Sirius products. The system flash bad block table located at 0x0E00 – 0x11FF is used by the bootrom, the Software upload library and nandflash program.

Table 7.1 - NVRAM Areas

| Area | Area type | Board type | Range | Description |
|---|---|---|---|---|
| TCM SW Configuration | Safe | TCM | `0x0000 - 0x0DFF` | nv_config: Configuration parameters for TCM SW. |
| SF_BAD_BLOCKS | Safe | OBC and TCM | `0x0E00 - 0x0FFF` | Bad-block information for System Flash |
| SF_BAD_BLOCKS | Update | OBC and TCM | `0x1000 - 0x11FF` | Bad-block information for System Flash. |
| TCM SW Configuration | Update | TCM | `0x1200 - 0x1FFF` | nv_config: Configuration parameters for TCM-S SW. |
| MM_BAD_BLOCKS | Update | TCM | `0x2000 - 0x23FF` | Bad-block information for Mass Memory. |
| TCM SW Parameters | Update | TCM | `0x2400 - 0x25FF` | Reserved area for operation markers of the TCM SW. |
| Free space | Update | | `0x2600 - 0x3FFF` | Currently unused area. |

# 8. Boot procedure

## 8.1. Description

The bootrom is a small piece of software built into a read-only memory inside the SoC. Its main function is to load a software image from the system flash to RAM and start it by jumping to the reset vector. To make the system fault tolerant, there are two logical images of the main software, designated Updated and Safe. Each logical image is stored in three physical copies distributed over the system flash. By default, the bootrom will first try to load the Updated image and if that fails fall back to the Safe image. The image to load can also be selected by setting the *Next FW* register in the Error Manager and doing a soft reset (see Section 5.3 for more details). Boot order of the logical images and their physical copies is shown in Figure 8.1.

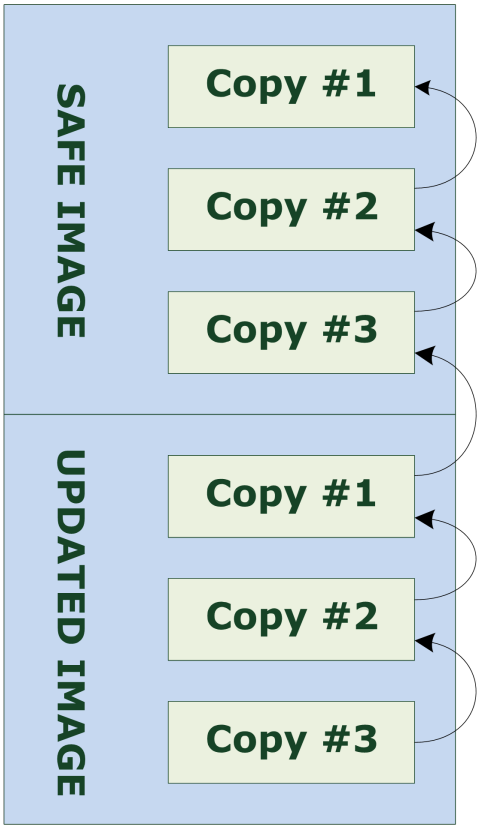



Figure 8.1 - Software images in flash

## 8.2. Usage description

The locations in the system flash where the bootrom looks for software images are given in Table 8.1. The first two 32-bit words of the image are expected to be a header

with image size and an XOR checksum, see Table 8.2. If the size falls within the accepted range, the bootrom loads the image to RAM while verifying the checksum. Both the image size check and the checksum verification are performed in addition to the EDAC built into the System Flash. The System Flash EDAC is handled by hardware and calculates one extra byte of redundancy data for each true data byte written to flash.

The bootrom loads the system flash bad-block table from an NVRAM offset described in Table 7.1. If a flash block within the range to load from is marked as bad in the table, that block is assumed to have been skipped when the image was programmed, so the bootrom continues reading from the next block. If the image could be loaded from flash without error and its checksum is correct, the bootrom jumps to the reset vector in RAM. If there is a flash error when loading, if the checksum is incorrect, or if the image has an invalid size, the bootrom steps to the next image by changing the *Next FW* field in the Error Manager and doing a soft reset. If the image being loaded is the last available the bootrom will ignore errors and attempt to start it anyway, in order to always have a chance of a working system. To indicate to the software which image and copy is loaded, the *Running FW* field in the Error Manager is updated before handing over execution. The boot loader will also update the Error Manager Latest Boot Status register to indicate where it is in the boot process, so that more information can be retrieved in case of a failed boot, see Section 5.3.2.4.5.

Reading out that register in orbit requires a subsequent successful boot. Therefore, if multiple image copies fail to boot, the register information that is saved will be from the first failed attempt.

## 8.3. Limitations

If the image size is out of range for Safe image copy #1 (the final fallback image), the bootrom will not be able to load it and the fallback option of handing execution to a damaged software image if no other is available cannot be used.

Table 8.1 - Software image locations

| Image number | Description | Flash page number |
|---|---|---|
| 0x0 | Updated copy #3 | `0xC0000` |
| 0x1 | Updated copy #2 | `0xA0000` |
| 0x2 | Updated copy #1 | `0x80000` |
| 0x3 | Safe copy #3 | `0x40000` |
| 0x4 | Safe copy #2 | `0x20000` |
| 0x5 | Safe copy #1 | `0x00000` |

Table 8.2 - Software image header

| Field | Size | Description |
|---|---|---|
| Image size | 32 bits | The size in bytes of the software image, not including the header, stored as a 32-bit unsigned integer. A software image can be 264 Bytes - 63 MB. |
| Checksum | 32 bits | A cumulative XOR of all 32-bit words in the image including the size, so that a cumulative XOR of the whole image and header (including checksum) shall evaluate to 0. |

## 8.4. Cause of last reset

The Error Manager RTEMS driver supports reading out the last reset cause, see Section 5.3.2.4 for details.

## 8.5. Pulse commands

The pulse command inputs to the Sirius products can be used to force a board to reboot from a specific image. Paired with the ability of the Sirius TCM to decode PUS CPDU telecommands without software interaction and issue pulse commands, this provides a means to reset malfunctioning boards by direct telecommand from ground as a last resort.

Each board has two pulse command inputs. Input 0 resets the board and loads the updated image while input 1 resets the board and loads the safe image. Both require an active-high pulse length between 20 - 40 ms to be valid. If, for some reason, both pulse command inputs would be active at the same time, the pulse on input 0 takes precedence.

# 9. Software Upload

There is a software upload library available in the BSP that can be used when writing custom applications. This library is located under `src/software_upload/`.

## 9.1. Description

During the lifetime of a satellite, the on-board software might need adjustments as bugs are detected or the mission parameters adjusted. This module tries to solve that by providing a means for updating the on-board software in orbit. The OBC and the TCM are both prepared for this functionality by having two software images, where writing to the first one requires the debugger to be connected, thus making only the second one available for updates in orbit.

The process for updating a flight software image is described below:

- The actual data transfer and commanding from earth performing the software upload needs to be compliant with the CCSDS standard for TC. In this description, it is assumed that the TCM is the initial recipient of TC, regardless of the end target.
- All the individual telecommand frames, containing one data fragment each, need to be assembled into a full or partial image for update with verification.
- Finally, the actual update of the physical flash image can take place, where the uploaded image is written to the system flash.

The API described in the following chapter takes care of the last two steps.

The picture in Figure 9.1 shows the intended control flow when commanding the software update from ground.

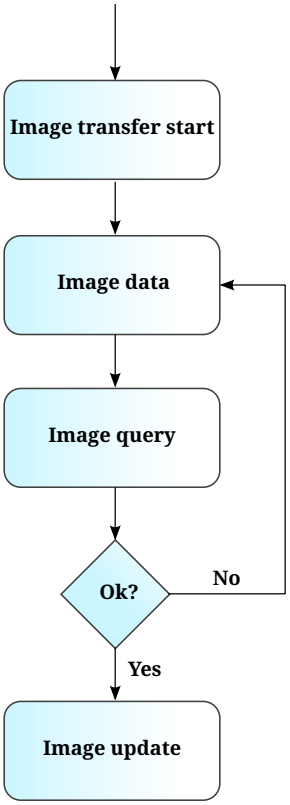



Figure 9.1 - The intended software upload command flow

## 9.2. CCSDS API – custom PUS service 130

| | |
|---|---|
| **WARNING** | The CCSDS API of the Software upload library is marked as deprecated and will be removed in future BSP releases. Please use the Software API instead (Section 9.3). |

### 9.2.1. Description

This service is provided to allow updates to the flight software on a node in a data handling system using Sirius components, but can be used for any type of on-board computer. The subtypes consist of a set of commands.

All service subtypes will report telecommand acceptance as PUS service [1,1] / [1,2] and telecommand execution complete as PUS services [1,7] / [1,8] (see [RD15]) if requested in the telecommand PUS header. See [RD8] for information on the allocated virtual channel for sending PUS reports. Recommended usage is to always request acceptance and execution complete reports so that the Ground Segment can keep track of the upload process.

All checksum parameters in the service are CRC32 with polynomial `0x04C11DB7` and

seed value `0`.

*The Telecommand Acceptance Report - Failure* will use the standard error codes according to Table 9.1 without any parameters (see [RD15]).

*Telecommand Execution Completed Report - Failure* values are listed under each subtype heading. Errors noted as 'critical' will cause the whole software upload process to be aborted.

Table 9.1 - Telecommand acceptance failure error types

| Error code | Data type | Error description |
|---|---|---|
| 0 | UINT8 | Illegal APID (PAC error) |
| 1 | UINT8 | Incomplete or invalid length packet |
| 2 | UINT8 | Incorrect checksum |
| 3 | UINT8 | Illegal packet type |
| 4 | UINT8 | Illegal packet subtype |
| 5 | UINT8 | Illegal or inconsistent application data |
| 6 | UINT8 | Illegal PUS version |

The numerical values of error codes returned in execution failure report are shown in Table 9.2 below.

Table 9.2 - Error code numerical values

| Error code | Numeric value |
|---|---|
| ENOENT | 2 |
| EIO | 5 |
| EBUSY | 16 |
| EINVAL | 22 |
| ENOSPC | 28 |
| ENODATA | 61 |
| EALREADY | 120 |

### 9.2.2. Subtype 1 – Image transfer start

A telecommand using this subtype must be sent first before sending any image data and will set up for a new image upload. It can also be used to abort an existing upload transaction during the data transfer phase, by simply initializing a new one. The data format is specified in Table 9.3 below.

Minimum image size is currently 272 bytes including header, and maximum image

size is 16 Mbyte.

Table 9.3 - Image transfer start command data structure

| Total number of bytes in image | Reserved (zero) | Reserved (zero) |
|---|---|---|
| UINT32 | UINT32 | UINT32 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.4 in case of a failure.

Table 9.4 - Image transfer start telecommand execution failure codes

| Error code | Data type | Error description |
|---|---|---|
| EINVAL | UINT8 | Invalid image size |
| EBUSY | UINT8 | Unable to open System Flash for writing or processing queue for requests is full. |

### 9.2.3. Subtype 2 – Image data

This subtype transports data segments of the actual flight software image. Each segment can carry data with a maximum length of 900 bytes (to avoid splitting packets over several frames) and all segments except the last must have data of maximum length. The data in each segment is preceded by a 2-byte segment number and a 2-byte segment length, see Table 9.5 below.

Table 9.5 - Image data command structure

| Segment number | Segment length | Segment data | | | |
|---|---|---|---|---|---|
| UINT16 | UINT16 | UINT8 | UINT8 | UINT8 | ... |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.6 in case of a failure.

Table 9.6 - Image data telecommand execution failure codes

| Error code | Data type | Error description |
|---|---|---|
| EALREADY | UINT8 | This segment number has already been added. |
| EINVAL | UINT8 | Segment number or segment length is out of bounds. |
| EIO | UINT8 | Read/write error in intermediate storage area of flash (critical.) |
| ENOSPC | UINT8 | Out of non-bad blocks in intermediate storage area of flash (critical.) |

| Error code | Data type | Error description |
|---|---|---|
| ENOENT | UINT8 | No upload in progress |
| EBUSY | UINT8 | Processing queue for requests is full. |

#### 9.2.4. Subtype 3 – Verify uploaded image

This subtype calculates and compares the checksum of the uploaded software image with the checksum set in the command's payload data, see Table 9.7.

Table 9.7 - Verify uploaded image argument

| Checksum |
|---|
| UINT32 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.8 in case of a failure.

Table 9.8 - Verify uploaded image telecommand execution failure codes

| Error code | Data type | Error description |
|---|---|---|
| EINVAL | UINT8 | Checksum argument doesn't match image checksum. |
| ENOENT | UINT8 | No upload in progress. |
| ENODATA | UINT8 | Segments missing. |
| EBUSY | UINT8 | Processing queue for requests is full. |

#### 9.2.5. Subtype 4 – Write uploaded image

To do the updating of the flight image, this command is sent to the service provider which will then write the image to flash. To safeguard against accidental update commanding, a correct CRC is required as input argument for this command, see Table 9.9.

Table 9.9 - Write image command argument

| Checksum |
|---|
| UINT32 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.10 in case of a failure.

Table 9.10 - Write image telecommand execution failure codes

| Error code | Data type | Error description |
|---|---|---|
| EINVAL | UINT8 | Checksum argument doesn't match image checksum. |
| ENOSPC | UINT8 | Out of non-bad blocks in flash(critical.) |
| ENOENT | UINT8 | No upload in progress |
| EIO | UINT8 | Read/write error in intermediate storage area of flash (critical.) |
| EBUSY | UINT8 | Processing queue for requests is full. |

### 9.2.6. Subtype 5 – Calculate CRC in flash

This command allows the CRC calculation of an image copy stored in flash. This can be used for extra verification after update of an image, or whenever the flight image copies need verification. The telecommand takes the image copy number as argument (max value 6), see Table 9.11. Image copy numbers 1 - 3 are for the (non-updateable) safe image and 4 - 6 cover the updated image copies.

Table 9.11 - Calculate CRC in flash command argument

| Image copy number |
|---|
| UINT8 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.12 in case of a failure.

Table 9.12 - Calculate flash CRC telecommand execution failure codes

| Error code | Data type | Error description |
|---|---|---|
| EINVAL | UINT8 | Image number too high (maximum 6). |
| EBUSY | UINT8 | Unable to open System Flash device or processing queue for requests is full. |
| EIO | UINT8 | Read/write error in intermediate storage area of flash (critical.) |

Furthermore, upon execution completed, a report will be generated using the same type and subtype as for the telecommand. This report will contain the calculated checksum, see Table 9.13.

Table 9.13 - Calculated flash CRC report

| Image copy number | Checksum |
|---|---|
| UINT8 | UINT32 |

## 9.3. Software API

This API depicts the functions available on the level below the PUS API and share many similarities with these. In many cases, the PUS API simply handle the PUS packaging and validation and maps almost directly into the software API functions.

### 9.3.1. Function: `int32_t swu_init(…)`

This function initializes all internal parameters for a new image upload. Calling init again while an upload is in progress will cause the existing upload to be aborted. A valid image must be at least `272` bytes and at most `16777216` bytes including header; but setting the argument to 0 is also allowed in order to abort an upload without starting a new one.

| Argument | Type | Direction | Decription |
|---|---|---|---|
| `total` | `uint32_t` | In | Total size of the uploaded image |

| Return value | Description |
|---|---|
| `0` | Success |
| `-EINVAL` | Invalid image size |
| `-EBUSY` | Unable to open System Flash for writing |

### 9.3.2. Function: `int32_t swu_segment_add(…)`

This function is used for putting together data segments into a full image. Use the function swu check to get current upload status.

| Argument | Type | Direction | Decription |
|---|---|---|---|
| `seg_num` | `uint16_t` | in | Number of this data segment |
| `length` | `uint16_t` | in | Length of this data segment |
| `data` | `uint8_t *` | in | Data of the added segment |

| Return value | Description |
|---|---|
| `0` | Success |
| `-EALREADY` | This segment has already been added. |
| `-EINVAL` | Segment number or segment length is invalid, or data is a NULL pointer. |
| `-EIO` | Read/write error in intermediate storage area of flash (critical.) |

| Return value | Description |
|---|---|
| -ENOSPC | Out of non-flash blocks in intermediate storage area of slash (critical.) |
| -ENOENT | No upload in progress. |

#### 9.3.3. Function: `int32_t swu_check(…)`

This function can be used to check the status of a current image upload. If all segments have been added, it will calculate the checksum of the entire image. If all segments have not been added, it will instead return an error code and an array of the ten first missing segments (maximum).

| Argument | Type | Direction | Decription |
|---|---|---|---|
| Checksum | uint32_t * | out | Data checksum if the image is complete, 0 otherwise. |
| Mlist | uint16_t * | out | An array of the first 10 missing segments. If the image is complete, no data will be entered into this variable. If only the checksum is of interest this may be a NULL pointer. |
| Mlength | uint16_t * | out | The number of elements in the missing segment array. If only the checksum is of interest this may be a NULL pointer. |

| Return value | Description |
|---|---|
| 0 | Success |
| -ENODATA | Not enough data - some data segments missing |
| -ENOENT | No upload in progress |
| -EINVAL | NULL pointer in arguments |

#### 9.3.4. Function: `int32_t swu_update(…)`

This function will perform the actual write of the image to flash. If one or more of the boot image areas in flash is out of space due to too many bad blocks an error will be returned, but the copies with enough space will still be written.

| Argument | Type | Direction | Decription |
|---|---|---|---|
| Checksum | uint32_t | in | Externally calculated checksum (checked against an internal calculation before update.) |

| Return value | Description |
|---|---|
| 0 | Success |
| -EINVAL | Checksum argument doesn't match image checksum. |
| -EIO | Error when accessing flash. |
| -ENOSPC | Out of non-bad blocks in one or more of the boot image areas in flash. |
| -ENOENT | No upload in progress |

#### 9.3.5. Function: `int32_t swu_flash_check(…)`

This function will calculate the checksum of an image in flash for specific verification purposes. The maximum image number is 6 and number 1 - 3 maps to the safe image copies and number 4 - 6 maps to the updated image copies. If the argument is out of bounds of the number of images, an error return code will be returned instead.

| Argument | Type | Direction | Decription |
|---|---|---|---|
| image_number | uint8_t | in | Image number in flash to calculate the checksum on. |
| checksum | uint32_t * | out | The calculated checksum. |

| Return value | Description |
|---|---|
| 0 | Success |
| -EINVAL | Image number is too small or large, or checksum is a NULL pointer |
| -EIO | Read error in image |
| -EBUSY | Unable to open flash device file |

## 9.4. Usage description

A user of the software upload module can either let the module handle all PUS commanding through the PUS API (see Section 9.2) or handle all PUS packetizing and reporting internally and only hook into the functional interface described in Section 9.3. A code example is provided in the directory `software_upload/src/example`.

## 9.5. Limitations

The maximum size of an image for upload is 16 Mbytes.

# 10. Death Reports

There is a death reports library available in the BSP that can be used when writing custom applications. This library is located under `src/death_reports/`, with examples for how to use it located in `src/death_reports/examples`.

## 10.1. Description

When an exception occurs, a death report consisting of a SCET timestamp, relevant process registers and further information about the trap is written to the death report area on persistent NVRAM. There are five available death report slots in the NVRAM. If the table is full and a new trap occurs, no new report will be added to the table, it is left unchanged.

## 10.2. Trap types

Table 7-1 in [RD16] describes the implemented traps for LEON3FT. Table 10.1 shows the implementation for Sirius and which unexpected traps that will/will not result in Death Reports. When an exception has occurred, the trap type can be determined by reading the tt-field in the death report entry field. See Table 10.3 and Table 10.1 for details.

Table 10.1 - Sirius Trap Types

| Trap | tt-value | Pri | Description | Class | Comment |
|---|---|---|---|---|---|
| ~~**reset**~~ | ~~**00**~~ | ~~**1**~~ | ~~**Power-on reset**~~ | ~~**Interrupting**~~ | Expected trap |
| data store error | 0x2b | 2 | Write buffer error during data store | Interrupting | |
| instruction access exception | 0x01 | 3 | Error or MMU page fault during instruction fetch | Precise | |
| privileged instruction | 0x03 | 4 | Execution of privileged instruction in user mode | Precise | |
| illegal instruction | 0x02 | 5 | UNIMP or other un-implemented instruction | Precise | |
| fp disabled | 0x04 | 6 | FP instruction while FPU disabled | Precise | |
| ~~**cp disabled**~~ | ~~**0x24**~~ | ~~**6**~~ | ~~**CP instruction while Co-processor disabled**~~ | ~~**Precise**~~ | No co-processor in current implementation |

| Trap | tt-value | Pri | Description | Class | Comment |
|---|---|---|---|---|---|
| ~~**watchpoint detected**~~ | ~~**0x0B**~~ | ~~**7**~~ | ~~**Hardware breakpoint match**~~ | ~~**Precise**~~ | Expected trap |
| window overflow | 0x05 | 8 | SAVE into invalid window | Precise | |
| window underflow | 0x06 | 8 | RESTORE into invalid window | Precise | |
| ~~**r register access error**~~ | ~~**0x20**~~ | ~~**9**~~ | ~~**Register file EDAC error (LEON3FT only)**~~ | ~~**Interrupting**~~ | Not present in current implementation |
| mem address not aligned | 0x07 | 10 | Memory access to un-aligned address | Precise | |
| fp exception | 0x08 | 11 | FPU Exception | Deferred | |
| ~~**cp exception**~~ | ~~**0x28**~~ | ~~**11**~~ | ~~**Co-processor exception**~~ | ~~**Deferred**~~ | No co-processor in current implementation |
| data access exception | 0x09 | 13 | Access error during data load, MMU page fault | Precise | |
| tag overflow | 0x0A | 14 | Tagged arithmetic overflow | Precise | |
| division by zero | 0x2A | 15 | Divide by zero | Precise | |

### 10.2.1. Floating point traps

FPU traps are disabled by default. The helper function `aac_enable_floating_point_traps()` via `<bsp/trap.h>` can be used to enable FP traps when writing custom applications. See the examples `fp_exception_div_by_zero.c` and `fp_exception_subnormal_number.c` in `bsp/src/death_reports/examples/`. If FPU traps are enabled, death reports will also be generated for this type of traps when using the death reports library.

There are six subcategories of floating-point exceptions according to Table 4-4 in [RD16]. According to section 47.2.3 in [RD3] all five floating point exceptions defined by the IEEE-754 standard can be detected (ftt=1). See section 4.4 in [RD16] for information on how to enable and detect floating point exceptions of type IEEE-754.

Table 10.2 - Sirius Floating-point Trap Types

| Floating-point Trap Type (ftt) Field of FSR | | |
|---|---|---|
| **ftt** | **Trap Type** | **Comment** |
| 0 | None | |
| 1 | IEEE_754_exception | |
| ~~**2**~~ | ~~**unfinished_FPop**~~ | Not used in GRFPU Lite |
| ~~**3**~~ | ~~**unimplemented_FPop**~~ | Not used in GRFPU Lite |
| 4 | sequence_error | |
| ~~**5**~~ | ~~**hardware_error**~~ | Not used in current implementation |
| ~~**6**~~ | ~~**invalid_fp_register**~~ | Not used in GRFPU Lite |
| 7 | *reserved* | |

When a trap of type floating point has occurred, information about the actual instruction that triggered the trap can be obtained from the Death Report Table. The floating point trap type (ftt) can be obtained by reading the FSR from the Death Report Table, detailed information on the contents of the FSR is in [RD16], section 4.4. When the trap is of the type floating point, the fields for direct traps in the Death Report Table are undefined and vice versa.

## 10.3. Format

When using this library, the format of death reports saved in the NVRAM is shown in Table 10.3.

Table 10.3 - HKDeathReports data

| Byte | Type | Description | Trap category |
|---|---|---|---|
| 0 | `UINT32` | Number of death reports currently in table | - |
| 4 | `UINT32` | A: SCET Seconds | All |
| 8 | `UINT32` | A: SCET Subseconds | All |
| 12 | `UINT32` | A: Processor Status Register (PSR) | All |
| 16 | `UINT32` | A: Trap Type | All |
| 20 | `UINT32` | A: Program Counter (PC) | Direct |
| 24 | `UINT32` | A: next Program Counter (nPC) | Direct |
| 28 | `UINT32` | A: Stack Pointer | Direct |

| Byte | Type | Description | Trap category |
|---|---|---|---|
| 32 | UINT32 | A: FPU Control/Status Register (FSR) | Floating point |
| 36 | UINT32 | A: Instruction address (Deferred traps) | Floating point |
| 40 | UINT32 | A: Instruction code (Deferred traps) | Floating point |
| 44 | UINT32 | B: SCET Seconds | All |
| 48 | UINT32 | B: SCET Subseconds | All |
| 52 | UINT32 | B: Processor Status Register (PSR) | All |
| 56 | UINT32 | B: Trap Type | All |
| 60 | UINT32 | B: Program Counter (PC) | Direct |
| 64 | UINT32 | B: next Program Counter (nPC) | Direct |
| 68 | UINT32 | B: Stack Pointer | Direct |
| 72 | UINT32 | B: FPU Control/Status Register (FSR) | Floating point |
| 76 | UINT32 | B: Instruction address | Floating point |
| 80 | UINT32 | B: Instruction code | Floating point |
| 84 | UINT32 | C: SCET Seconds | All |
| 88 | UINT32 | C: SCET Subseconds | All |
| 92 | UINT32 | C: Processor Status Register (PSR) | All |
| 96 | UINT32 | C: Trap Type | All |
| 100 | UINT32 | C: Program Counter (PC) | Direct |
| 104 | UINT32 | C: next Program Counter (nPC) | Direct |
| 108 | UINT32 | C: Stack Pointer | Direct |
| 112 | UINT32 | C: FPU Control/Status Register (FSR) | Floating point |
| 116 | UINT32 | C: Instruction address | Floating point |
| 120 | UINT32 | C: Instruction code | Floating point |
| 124 | UINT32 | D: SCET Seconds | All |
| 128 | UINT32 | D: SCET Subseconds | All |
| 132 | UINT32 | D: Processor Status Register (PSR) | All |
| 136 | UINT32 | D: Trap Type | All |
| 140 | UINT32 | D: Program Counter (PC) | Direct |
| 144 | UINT32 | D: next Program Counter (nPC) | Direct |
| 148 | UINT32 | D: Stack Pointer | Direct |

| Byte | Type | Description | Trap category |
|---|---|---|---|
| 152 | UINT32 | D: FPU Control/Status Register (FSR) | Floating point |
| 156 | UINT32 | D: Instruction address | Floating point |
| 160 | UINT32 | D: Instruction code | Floating point |
| 164 | UINT32 | E: SCET Seconds | All |
| 158 | UINT32 | E: SCET Subseconds | All |
| 162 | UINT32 | E: Processor Status Register (PSR) | All |
| 166 | UINT32 | E: Trap Type | All |
| 170 | UINT32 | E: Program Counter (PC) | Direct |
| 174 | UINT32 | E: next Program Counter (nPC) | Direct |
| 178 | UINT32 | E: Stack Pointer | Direct |
| 182 | UINT32 | E: FPU Control/Status Register (FSR) | Floating point |
| 186 | UINT32 | E: Instruction address | Floating point |
| 200 | UINT32 | E: Instruction code | Floating point |

When an exception has occurred, the trap type can be determined by reading the Trap Type-field in the death reports table.

For direct traps the address of the trap inducing instruction can be determined from the program counter PC. The trap inducing instruction is then PC-1.

The stack and frame pointers are always 16 registers (64 byte) apart in the frame windows.

When a trap of type floating point has occurred, information about the actual instruction that triggered the trap can be obtained from the death reports table, see Instruction address and Instruction code in Table 10.3. The floating-point trap type (ftt) can be obtained by reading the FSR from the death reports table, detailed information on the contents of the FSR is in [RD16], section 4.4. When the trap is of the type floating point, the fields for direct traps in the death reports table are undefined and vice versa.

## 10.4. NVRAM

The table is located on NVRAM at offset `0x1F34 - 0x1FFF`. The table can contain up to five death reports, and it is updated A → E. If the table is full and a new trap occurs, the death reports module will not add a new report to the table, it is left unchanged.

When clearing the table, the counter at offset 0 shall also be updated by the custom

application. The death reports module cannot handle gaps in the table.

## 10.5. Usage Description

A custom application which wants to generate death reports needs to:

- `#include "death_reports.h"`
- Link with `libdeath_reports.a`
- Add the library-provided function in an RTEMS fatal handler registered as a user extension.

To install the death reports handler into a custom RTEMS application, an RTEMS user extension fatal handler has to be added to the application. Helper functions for obtaining LEON3 architecture specific SW trap information are available in `<bsp/traps.h>`. An example RTEMS application with an installed death reports handler is available in the subfolder `death_reports/examples/exception_handler.c`.

An example application for reading out and parsing death reports from NVRAM is available in the subfolder `death_reports/examples/read_nvram_death_report_area.c`. An example application that clears the death reports area on NVRAM is available in the subfolder `death_reports/examples/clear_nvram_death_report_area.c`.

| | |
|---|---|
| **IMPORTANT** | Fatal handlers do not support normal use of RTEMS POSIX API, therefore this library is provided to allow for (otherwise unsupported) use of the AAC bare metal drivers. Modifying this library (except for the examples) is not recommended nor supported. |

# 11. TM/TC-structure and COP-1

## 11.1. SCID

For commanding the spacecraft, a 10-bit Spacecraft Identifier is needed. For every mission, a mission specific SCID is configured in the TCM.

## 11.2. Virtual Channel Allocation

See [RD8] for VC allocation.

## 11.3. Uplink Channel Coding, Randomization and Synchronization

### 11.3.1. Channel Coding

The Telecommand Code Block is BCH (63, 56) and supports Single Error Correction mode.

### 11.3.2. Randomization

Derandomization of telecommands can be enabled/disabled by an ioctl command (see Section 5.9.2.5).

### 11.3.3. Channel Synchronization

The 2-byte start sequence of Telecommands is `0xEB90`. The 8-byte tail sequence of Telecommands is `0xC5C5C5C5C5C5C579`.

## 11.4. Downlink Channel Coding, Randomization and Synchronization

### 11.4.1. Channel Coding

Reed-Solomon encoding by a RS (255, 223) encoder with an interleaving depth of 5, resulting in a Telemetry Transfer Frame length of 1115 octets.

Convolutional encoding is according to [RD17] section 3.3 (code rate 1/2 bit per symbol; constraint length 7 bits; polynomial generators G1=171 octal and G2=133 octal; inversion on G2).

It can be enabled/disabled by an ioctl command (see Section 5.9.2.4).

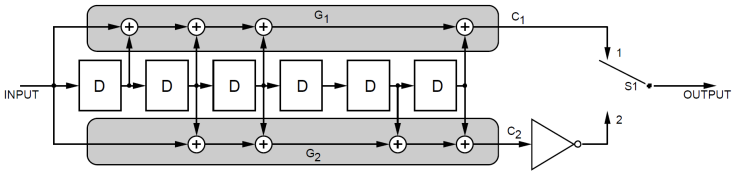



Figure 11.1 - Convolutional Encoder Block Diagram

### 11.4.2. Randomization

Randomization of Telemetry Transfer Frames can be enabled/disabled by an ioctl command (see Section 5.9.2.4).

### 11.4.3. Synchronization

The 4-byte synchronization pattern prepended to the Reed-Solomon code block is `0x1ACFFC1D`.

## 11.5. Telecommand format

This chapter describes the format of the TC Transfer frames and TC Packets the TCM handles.

### 11.5.1. Telecommand Transfer Frame

The Telecommand Transfer Frame conforms to the format described in [RD13] and shown below.

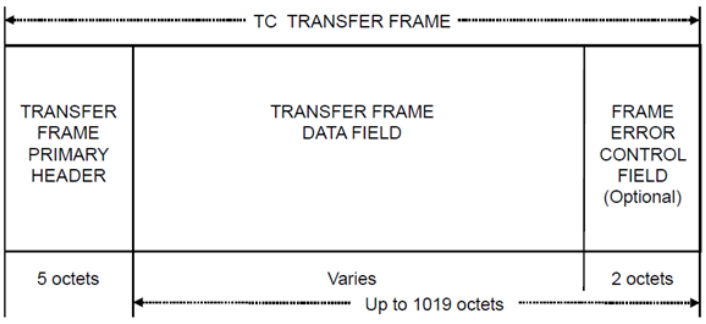



Figure 11.2 - TC Transfer Frame

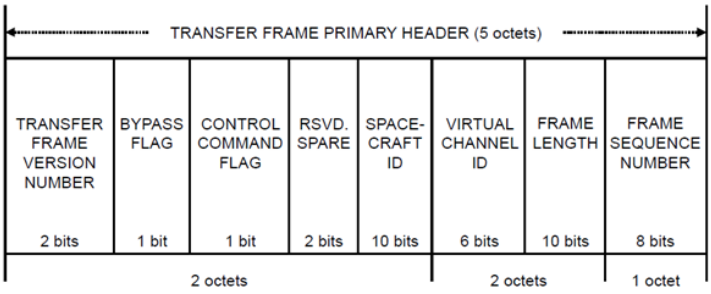



Figure 11.3 - Transfer Frame Header

### 11.5.2. Carrier Lock and Subcarrier Lock

In the radio interface connectors on the TCM there are two input signals called Carrier Lock and Subcarrier Lock. These need to be active for the TCM to process the incoming telecommand data. The state of the signals is reflected in the CLCW flags “No RF Available” and “No Bit Lock”, see Section 11.6.4.

## 11.6. Telemetry Format

This chapter describes the format of TM Transfer Frames and TM Packets sent from the TCM to ground.

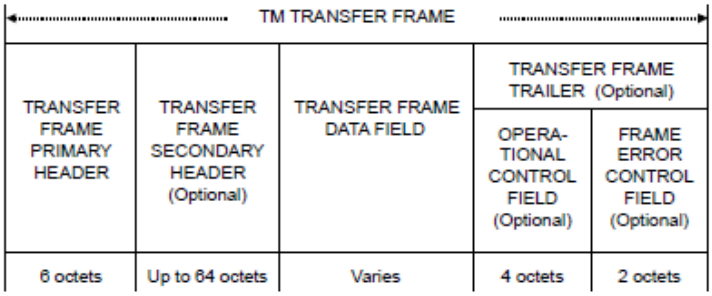



Figure 11.4 - Telemetry Transfer Frame

### 11.6.1. Transfer Frame Primary Header

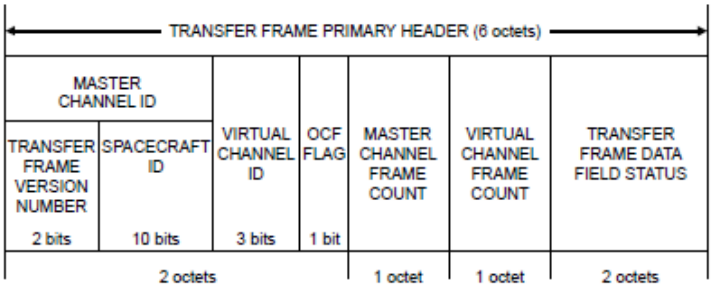



Figure 11.5 - Telemetry Transfer Frame Primary Header

Table 11.1 - Telemetry Transfer Frame Primary Header

| Field | Description | Comment |
|---|---|---|
| TRANSFER FRAME VERSION NUMBER | Set to `00`. | |
| SPACECRAFT ID | Mission specific identifier of the spacecraft. | |
| VIRTUAL CHANNEL ID | See [RD8] for VC allocation. | |
| OCF FLAG | Indicates presence of Operation Control Field (OCF) in TM Transfer Frames. It shall be 1 if the OCF is present. It shall be `0` if the OCF is not present. | This is configurable by a setting in NVRAM for the TCM. It can also be set by a RMAP-command. |
| MASTER CHANNEL FRAME COUNT | An 8-bit sequential binary count (modulo 256). | |
| VIRTUAL CHANNEL FRAME COUNT | An 8-bit sequential binary count (modulo 256). | |
| TRANSFER FRAME DATA FIELD STATUS | See below | |

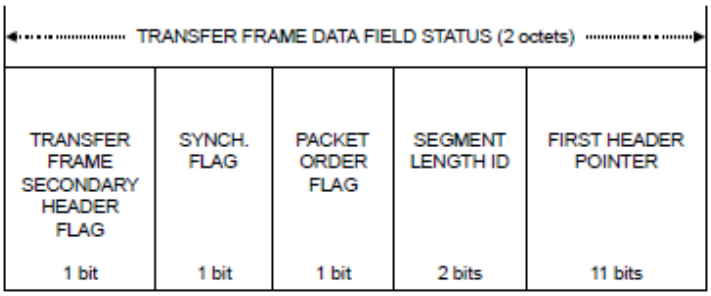



Figure 11.6 - Transfer Frame Data Field Status

Table 11.2 - Transfer Frame Data Field Status

| Field | Description | Comment |
|---|---|---|
| TRANSFER FRAME SECONDARY HEADER FLAG | Shall be 1 if Transfer Frame Secondary Header is present. Shall be `0` if Transfer Frame Secondary Header is not present. | In the TCM, the Transfer Frame Secondary Header is not used, so this field is always set to `0`. |

| Field | Description | Comment |
|---|---|---|
| SYNCHRONIZATION FLAG | Indicates type of data inserted in the Transfer Frame Data Field. It shall be ´0´ if octet-synchronized, 1 otherwise. | In the TCM, data is always inserted octet-synchronized, so this field is always set to `0`. |
| PACKET ORDER FLAG | Packet Order Flag. | Always set to `0` in TCM. |
| SEGMENT LENGTH ID | Shall be set to 11 if Synchronization Flag is set to `0`. | Set to 11 in TCM. |
| FIRST HEADER POINTER | If the Synchronization Flag is set to `0`, the First Header Pointer shall contain the position of the first octet of the first Packet that starts in the Transfer Frame Data Field. When valid data exist in frame, but no packet/segment header is present the First Header Pointer is set to 11111111111. If the frame contains only idle data, the First Header Pointer is set to 11111111110. | |

#### 11.6.2. Transfer Frame Secondary Header

The Transfer Frame Secondary Header is not used by the TCM.

#### 11.6.3. Transfer Frame Data Field

The Transfer Frame Data Field contains an integral number of octets of data formatted as TM Packets. The length of this field is fixed but can be different for different configurations depending on inclusion of OCF and FECF. The maximum length of this field is 1109 octets (1115 – 6), and the minimum length is 1103 octets (1115 - 6 - 4 - 2).

#### 11.6.4. Operational control field

The Operational Control Field contains a Communications Link Control Word as described in [RD13] section 4.2.

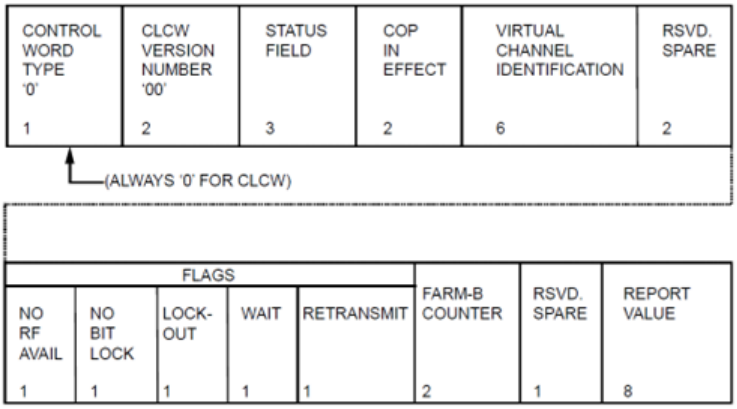



Figure 11.7 - Command Link Control Word

Table 11.3 - Command Link Control Word

| Field | Description | Comment |
|---|---|---|
| CONTROL WORD TYPE | Is set to `0`. | |
| CLCW VERSION NUMBER | Is set to `00`. | |
| STATUS FIELD | Can be used for Mission-specific status. | No specific setting by TCM. |
| COP IN EFFECT | Set to `01`. | |
| VIRTUAL CHANNEL IDENTIFICATION | Virtual Channel Identifier. | |
| RESERVED SPARE | Set to `00`. | |
| NO RF AVAIL | Set to `0` if Physical Layer Available.<br><br>Set to 1 if Physical Layer is not available. | Controlled by physical input signal, see Section 11.5.2 |
| NO BIT LOCK | Set to `0` when bit lock has been achieved.<br><br>Set to 1 when bit lock has not been achieved. | Controlled by physical input signal, see Section 11.5.2 |
| LOCK-OUT | Shows Lockout status of the FARM.<br><br>Set to `0` when FARM is not in Lockout.<br><br>Set to 1 when FARM is in Lockout. | |

| Field | Description | Comment |
|---|---|---|
| WAIT | Set to 1 (Wait) indicates that all further Type-A Transfer Frames on that virtual channel will be rejected by FARM until the condition cleared.<br><br>Set to `0` indicates TCM is able to accept and process incoming Type-A Transfer Frames. | |
| RETRANSMIT | Set to 1 indicates that one or more Type-A Transfer Frames have been rejected.<br><br>Set to `0` indicates no outstanding Type-A Transfer Frame rejections so far. | |
| FARM-B COUNTER | Contains two least significant bits of FARM-B Counter. | |
| RESERVED SPARE | Set to `0`. | |
| REPORT VALUE | Contains the value of the Next Expected Frame Sequence Number, N(R). | |

#### 11.6.5. Frame Error Control Field

If used, the checksum of the Frame Error Control Field shall be calculated using CRC with polynomial `0x8408`, LSB first (reverse of `0x1021`, MSB first); and initial value `0xFFFF` over the whole TM Transfer Frame except the two last octets.

#### 11.6.6. Idle Data

In the TCM, `0x5A` is the data sent for Idle Frames and Idle Packets.

## 11.7. FARM-parameters

COP-1 is supported on the TCM.

#### 11.7.1. FARM_Sliding_Window_Width(W)

In the TCM, the parameter W is fixed to 128.

#### 11.7.2. FARM_Positive_Window_Width(PW)

In the TCM, the parameter PW is fixed to 64.

#### 11.7.3. FARM_Negative_Window_Width(NW)

In the TCM, the parameter NW is fixed to 64.

# 12. Updating the Sirius FPGA

To be able to update the SoC on the Sirius OBC and Sirius TCM you need the following items:

**Prerequisite hardware:**

- Microsemi FlashPro5 unit
- 104470 FPGA programming cable assembly

**Prerequisite software:**

- Microsemi FlashPro Express v11.8 or later
- The updated FPGA firmware

## 12.1. Generation of encryption key

When AAC Clyde Space is supporting a customer, files with sensitive data to be transferred between AAC and customers can be encrypted/decrypted by GPG.

1. Generate a key by:
   `gpg --gen-key`
2. Select option “DSA and Elgamal” and a keysize of 2048 bits
3. After successful generation of the key, export the key by:
   `gpg --export -a -o your_pub.key`
4. The generated key, `your_pub.key`, in example above is to be sent to AAC if needed.

## 12.2. Step-by-step guide

The following instructions show the necessary steps that need to be taken in order to upgrade the FPGA firmware:

1. Connect the FlashPro5 programmer via the 104470 FPGA programming cable assembly to the JTAG-RTL connector in Figure 12.1.
2. Connect the power cables according to Figure 12.1.
3. The updated FPGA firmware delivery from AAC should contain at least two files:
   a. The actual FPGA file with an .stp file ending
   b. The programmer file with a .pro file ending
4. Start the FlashPro Express application, click “Open…” in the “Job Projects” box (see Figure 12.1) and select the supplied .pro file.

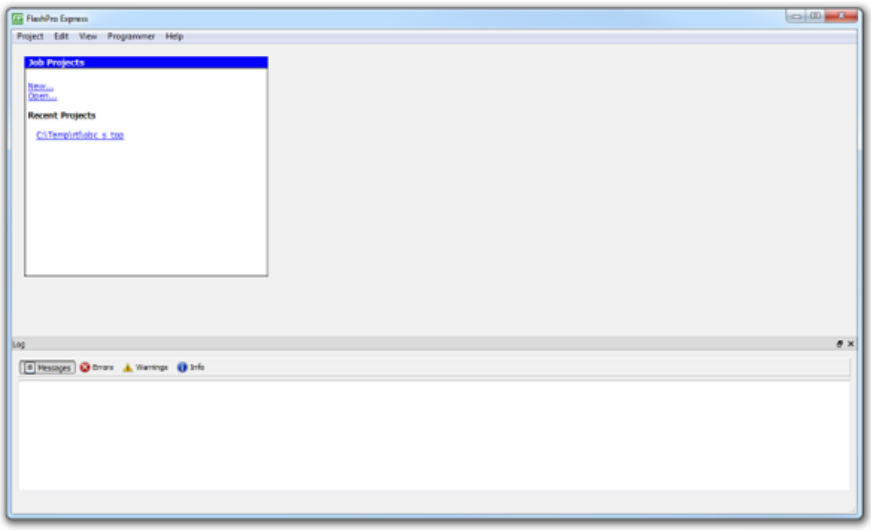



Figure 12.1 - Startup view of FlashPro Express

5. Once the file has loaded (warnings might appear), click RUN (see Figure 12.2). Please note that the connected FlashPro5 programmed ID should be shown.

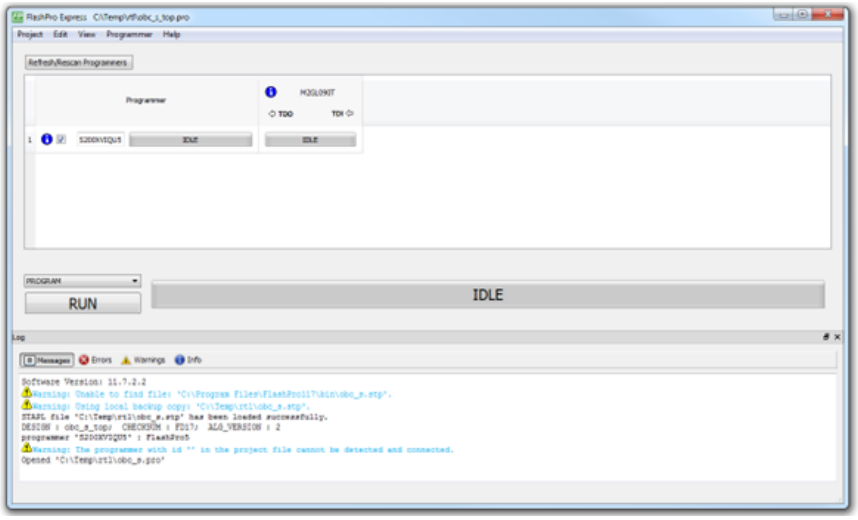



Figure 12.2 - View of FlashPro Express with project loaded.

6. The FPGA should now be loaded with the new firmware, which might take a few minutes. Once it is finalized the second last message should be "Chain programming PASSED", see Figure 12.3.

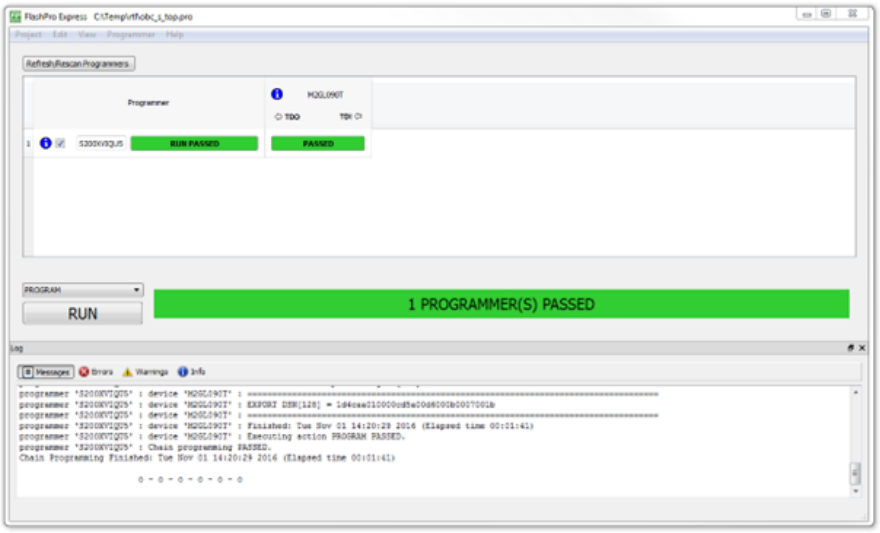



Figure 12.3 - View of FlashPro Express after program passed.

The Sirius FPGA image is now updated.

# 13. Mechanical data

Please refer to the mechanical and Electrical ICD [RD2].

# 14. Glossary

| Acronym | Description |
|---|---|
| ABI | Application Binary Interface |
| ADC | Analog Digital Converter |
| API | Application Programming Interface |
| APID | Application Process ID |
| BCH | Bose-Chaudhuri-Hocquenghem code, a type of error correction code |
| BSP | Board Support Package |
| CCSDS | The Consultative Committee for Space Data Systems |
| CLCW | Command Link Control Word, see [RD12] and [RD13] |
| COP | Communications Operation Procedure, see [RD12] and [RD13] |
| CPDU | Command Pulse Distribution Unit |
| CRC | Cyclic Redundancy Check |
| DMA | Direct Memory Access |
| ECC | Error Correction Code |
| EDAC | Error Detection and Correction |
| EM | Engineering model |
| ESD | Electrostatic Discharge |
| FARM | Frame Acceptance and Reporting Mechanism, see [RD13] |
| FECF | Frame Error Control Field, see [RD12] and [RD13] |
| FIFO | First In First Out |
| FLASH | Flash memory is a non-volatile computer storage chip that can be electrically erased and reprogrammed |
| FPGA | Field Programmable Gate Array |
| FW | Firmware |
| GCC | GNU Compiler Collection program (type of standard in Unix) |
| GDB | GNU Debugger |
| GPIO | General Purpose Input/Output |
| Gtkterm | A terminal emulator that drives serial ports |

| Acronym | Description |
|---|---|
| $I^2C$ | Inter-Integrated Circuit, generally referred as “two-wire interface” is a multi-master serial single-ended computer bus invented by Philips. |
| IP (core) | Intellectual property core, reusable functional logic block used e.g. in a FPGA |
| JTAG | Joint Test Action Group, interface for debugging the PCBs |
| LVTTL | Low-Voltage TTL |
| LSB | Least significant bit/byte |
| MCFC | Master Channel Frame Counter |
| Minicom | Is a text based modem control and terminal emulation program |
| MSB | Most significant bit/byte |
| NA | Not Applicable |
| NVRAM | Non Volatile Random Access Memory |
| OBC | On Board Computer |
| OCF | Operational Control Field, see [RD12] and [RD8 |
| OS | Operating System |
| PCB | Printed Circuit Board |
| PCBA | Printed Circuit Board Assembly |
| POSIX | Portable Operating System Interface |
| PPS | Pulse-Per-Second |
| PSU | Power Supply Unit |
| PUS | Packet Utilization Standard |
| RAM | Random Access Memory, however modern DRAM has not random access. It is often associated with volatile types of memory |
| RMAP | Remote Memory Access Protocol |
| ROM | Read Only Memory |
| RTEMS | Real-Time Executive for Multiprocessor Systems |
| SCET | SpaceCraft Elapsed Timer |
| SCID | SpaceCraft ID |
| SDRAM | Synchronous Dynamic Random Access Memory |

| Acronym | Description |
|---|---|
| SoC | System-on-Chip |
| SPI | Serial Peripheral Interface Bus is a synchronous serial data link which sometimes is called a 4-wire serial bus. |
| SpW | SpaceWire |
| SW | Software |
| TC | Telecommand |
| TCL | Tool Command Language, a script language |
| TCM | Telemetry, Tracking and Command Control Module |
| TM | Telemetry |
| TMR | Triple Modular Redundancy |
| TTL | Transistor Transistor Logic, digital signal levels used by IC components |
| UART | Universal Asynchonous Receiver Transmitter that translates data between parallel and serial forms. |
| USB | Universal Serial Bus, bus connection for both power and data |
| VC | Virtual Channel |
| WDT | WatchDog Timer |