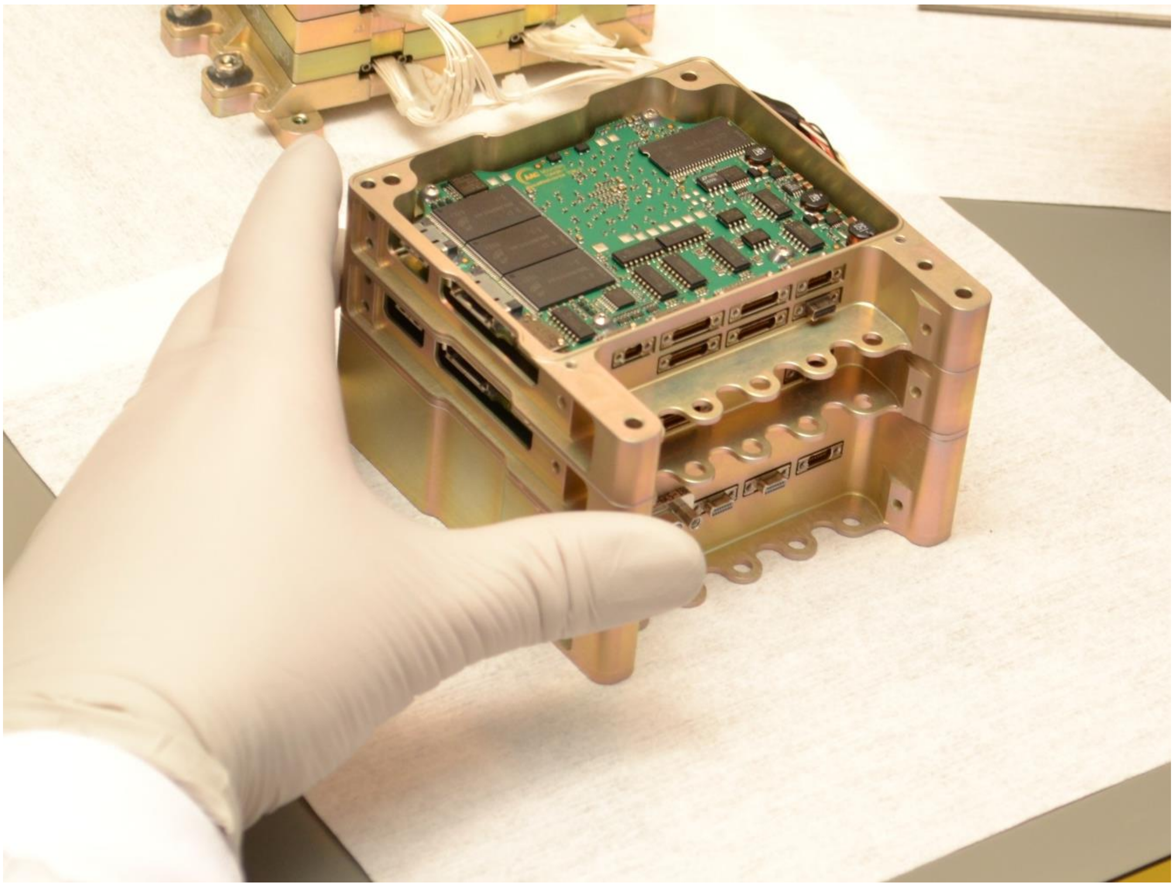
Document number	205065
Version	V
Issue date	2023-10-16

Sirius OBC and TCM User Manual

V



REVISION LOG

Rev	Date	Change description
A	2016-10-25	First release, drafted from 204911 Sirius Breadboard User Manual Rev L
B	2016-12-15	Updated after editorial updates
C	2017-01-03	Release with updates to the following sections: • Massmem (new API with DMA) • Error manager (IOCTL API) • ADC (channel table update, channel limitation) • Sirius TCM (TM/TC defaults, API updates {errno, MMStatus, TMTSStatus}, removed limitations) • Bootrom (extended description) • SCET (extended description, new API) • UART32 (removed) • CCSDS (interrupt API deprecation) • NVRAM (EDAC/non-EDAC modes described)
D	2017-02-01	Release with updates to the following sections: • Sirius TCM (Extra info sections, TMBRSet->TMBRControl) • Mass memory (IOCTL API, error inject info) • SCET (Clarify threshold)
E	2017-03-01	Release with updates to the following sections: • ADC (minor updates to clock div limits) • Setup and operation (find debugger serial, use of multiple debuggers)
F	2017-04-18	Release with updates to the following sections: • CCSDS (new API) • Sirius TCM (new timesync API, NVRAM table updated, new segment sizing for partitions)
G	2017-10-31	Release with updates to the following sections: • Fault tolerant design (new section) • CCSDS (updated API) • Mass memory (updated API) • Sirius TCM (new mass memory partition configuration behaviour & RMAP API) • System flash (new)
H	2018-03-07	Release with updates to the following sections: • Introduction • Equipment information • Sirius TCM (updated API and formatting) • NVRAM (updated API)
I	2018-04-16	Release with updates of the following sections: • Software upload (new) • NVRAM (updated EDAC error reporting API)
J	2018-06-28	Release with updates of the following sections: • SCET, UART, WDT, NVRAM and SpW (updated API) • Mass Memory Handling (auto-padding) • Removed chapter with connector pinout

K	2018-10-26	Release with updates to the following sections: • Re-initialising the NVRAM (new) • Mass Memory (new optional runtime-size API, new chip type support) • System flash (deprecate spare area writes without EDAC/interleaving) • Sirius TCM (config fallback parameters, direct partition limits, new PUS 2.2 service, RMAP transid allocation recommendations, limited direct partition utilisation recommendations)
M	2018-12-04	Release with updates to the following sections: • Software development (how to build silent BSP) • TM/TC-structure and COP-1 (new) • System-on-Chip defitions (system flash bad block table reserved location in NVRAM) • NVRAM (safe/update area address corrections) • Sirius TCM-S (bit error correction information for telecommands)
N	2019-02-01	Release with updates to the following sections: • Sirius TCM (noted possible pointer reset to address 0 on massmem handler recovery)
O	2019-06-28	Release with updates to the following sections: • Clarification and correction of error handling in System Flash driver. • Correction to 10.5.1: RS encoding can only be configured in NVRAM. • Add status code ECANCELED in table 7-27. • Corrected information on SCET event queues.
P	2019-08-27	Release with updates of the following sections: • Updated image 3.1 to include UART5 • Changed text in 5.5.1 to clarify UART-implementation is not fully compliant with 16550D
Q	2019-09-11	Release with updates to the following sections • 1.1: Add LEON3 • 2: Add LEON3 • 3: Add LEON3, minor corrections. • 4: Add LEON3, minor corrections • 8: Add memory mapping and interrupts for LEON3 General formatting clean-up.
R	2020-01-09	Release with following modifications • 5.3 Removed refs to Power Loss • 9.3 Added return codes in PUS service and table with numeric values of error codes • 10.7.9 Added Image of Data Field Header used in TCM-S • 10.7.13 Added description of Idle Data
S	2020-03-05	Release with following modifications • 5.10 Added info about conversion factors for analog input0-7 • 5.5.1.3 Added description of modes of the UART • Updated image 2.1
T	2020-11-05	Release with following modifications • Updated 2.1 System Overview • Added section 4.2 about floating-point • Added RMAP command MMGetPageSize • Updated SoC Info • Minor corrections and clarifications

U	2023-04-18	Release with following modifications • Removed sections about OpenRISC • Added instructions on how to use the GRMON with the FTDI D2XX driver • Various clarifications in chapter 5 • Added Error Manager Reset Cause ioctl • Added Read Timeout functionality on UART and SpW interfaces • Added Backup Routing for TCM TC and UART forwarding • Added RIRP as an alternative way to use the TCM RMAP interface • Added TCStorage service in the TCM • Clarification of TM bitrate configuration in 7.16.5.10 • Added HKResetCause command for the TCM • Broke out unit specific information to a new document, and rearranged some information into new chapters • Added details about the convolutional code implementation in 12.5.1
V	2023-10-16	Release with following modifications • Move out SoC configuration to its own document • Restructure chapters, correct formatting • 2: Add overview of OBC/TCM concept • 2: Add overview of manual • 3: Add info on GRMON issues • 3: Add debugging tips • 3: Update supported OS • 3: Update required software • 5.2: Note that Watchdog is active by default • 5.3: Add missing ioctls • 5.3: Clarify ioctl types • 5.3: Add Boot Status • 5.4: Clarify usage of broadcast queue • 5.5: Add baud rate • 5.9: VC selectable • 5.9: Correct TM bitrate divisor description • 5: Changes to CCSDS_SET_TM_TIMESTAMP to better support implementation of PUS service 9 • 7: Additional configuration for UARTs, add baud rate • 7: Update TC_CONFIG table • 7: TC VC configurable • 7: Clarify RMAP Verify-Data-Before-Write behavior • 7: Add TCQueue • 7: Add GPIO control over RMAP • 7: Update HKResetCause with time stamp • 7: Add HKBootStatus • 7: Add HKDeathReports • 7: Add MMBadBlockCount • 7: Correct TMBRControl description • 7: Change TMTSControl to better support implementation of PUS service 9 • 10: Update error codes for acceptance reports • 11: Add chapter on DeathReports • 12: Update handling of TC Ack flags • 12: Add info on Carrier/Subcarrier lock • 12: Add info on TM channel coding, randomization, and synchronization

TABLE OF CONTENT

1. Introduction

This manual describes the functionality and usage of the AAC Clyde Space Sirius OBC and Sirius TCM products. The Sirius OBC or Sirius TCM differ in certain areas such as the SoC, interfaces etc. but can mostly be described with the same functionality and will throughout this document be referred to as “the Sirius products” when both products are referred at the same time.

1.1. Applicable releases

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

Sirius Leon3 OBC	1.14.1
Sirius Leon3 TCM	1.21.0

1.2. Intended users

This manual is written for software engineers using the AAC Clyde Space Sirius products. The electrical and mechanical interface is described in more detail in the electrical and mechanical ICD documents [RD9] and [RD10].

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

RD#	Document ref	Document name
RD1	ECSS-E-ST-50-12C	SpaceWire – Links, nodes, routers and networks
RD2	ECSS-E-ST-50-52C	SpaceWire – Remote memory access protocol
RD3	ECSS-E-70-41A	Ground systems and operations – Telemetry and telecommand packet utilization
RD4	SNLS378B	PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs
RD5	AD7173-8, Rev. A	Low Power, 8-/16-Channel, 31.25 kSPS, 24-Bit, Highly Integrated Sigma-Delta ADC
RD6	Edition 4.11	RTEMS BSP and Device Driver Development Guide
RD7	CCSDS 132.0-B-4	TM Space Data Link Protocol
RD8	CCSDS 232.0-B-4	TC Space Data Link Protocol
RD9	205088	Sirius OBC electrical and mechanical ICD
RD10	205089	Sirius TCM electrical and mechanical ICD
RD11	SS-EN 61340-5-1	Electrostatics - Part 5-1: Protection of electronic devices from electrostatic phenomena - General requirements
RD12	Edition 4.11	RTEMS POSIX Users Manual
RD13	CCSDS 201.0-B-3	TC Channel Service
RD14	Edition 4.11	RTEMS C User Manual
RD15	GRIP, May 2019, Version 2019.2	GRLIB IP Core User's Manual
RD16	GRMON3-UM, June 2019, Version 3.1.0	GRMON3 User's Manual
RD17	CCSDS 131.0-B-4	TM Synchronization and Channel Coding
RD18	206222	Sirius SoC Configuration Document

RD19	sparcv8, SAV080SI9308	The SPARC Architecture Manual, Version 8

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 [RD15] 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.

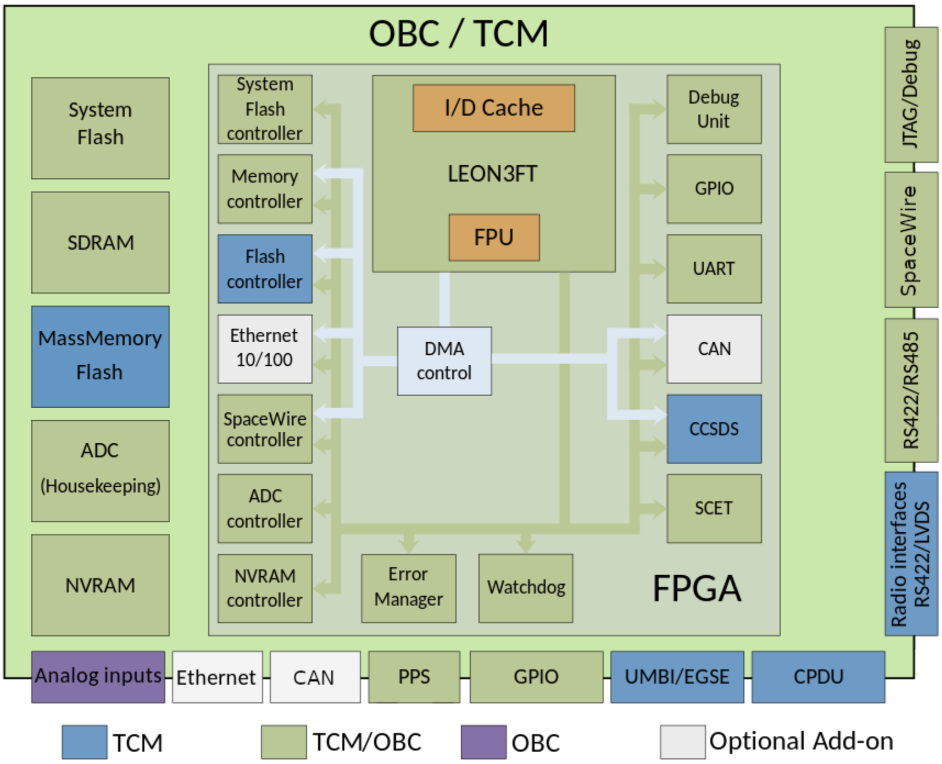


Figure 2-1 - The Sirius OBC / Sirius TCM SoC Overview

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.

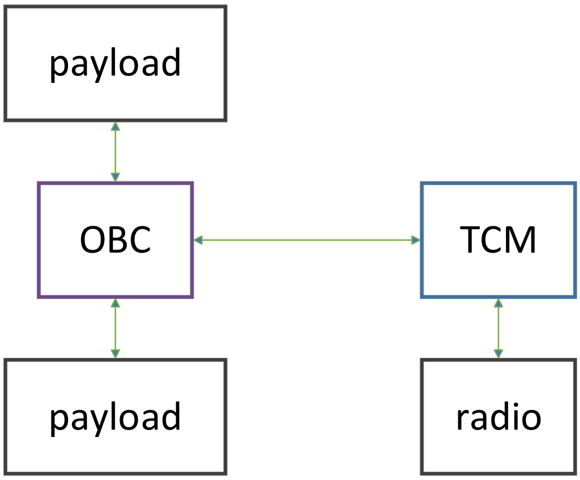


Figure 2-2 Conceptual design of an on-board data handling system

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 section 5) for use when developing its software.

2.4.3. TCM concept

2.4.3.1. Description

The TCM contains pre-programmed flight software (see section 7). 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 (see section 7.4).

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 sections 3.6, 5.12, 9, and 10.

Non-volatile RAM structure and usage is detailed in sections 8 and 5.11 for the OBC and sections 8 and 7.4 for the TCM.

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.

Information on usage of the TCM flight software is mainly in chapter 7, with details of the specific implementation of the CCSDS standards in chapter 12.

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 **http://www.microsemi.com/products/fpga-soc/design-resources/programming/flashpro#software**
- FlashPro5 programmer

3.2. Connecting cables to the Sirius products

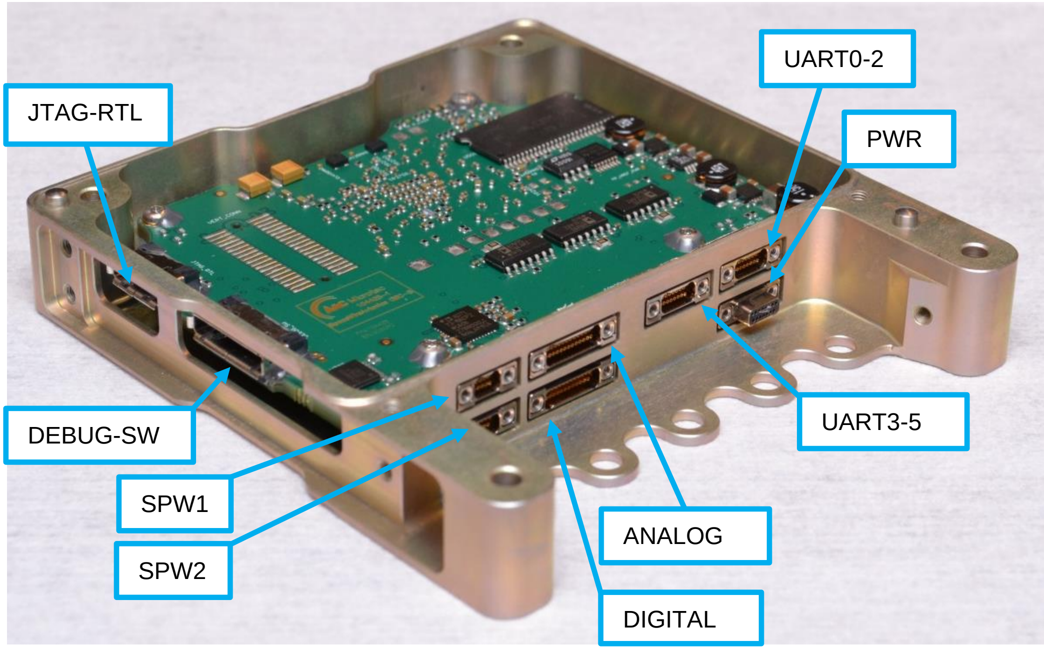


Figure 3-1 – Sirius OBC with connector naming

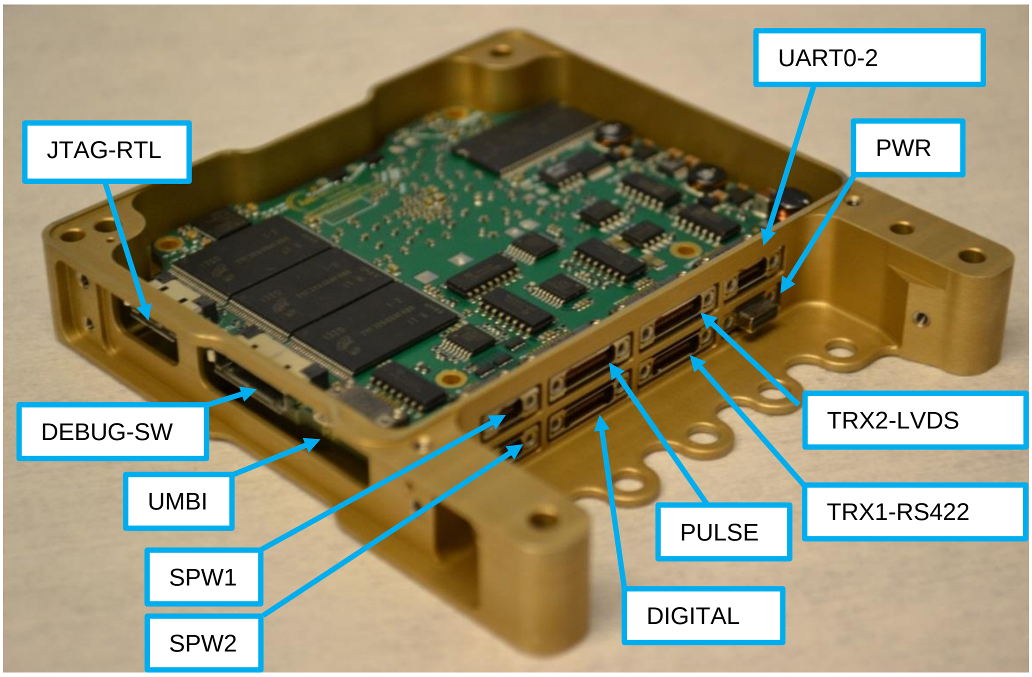


Figure 3-2 - Sirius TCM with connector naming

- 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 [RD11] 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 Chapter 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 13.
- For connecting the SpaceWire interface, connect the nano-D connector to connector SPW1 or SPW2.

For more detailed information about the connectors, see [RD9] and [RD10].

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 https://www.debian.org/releases/jessie/debian-installer/

In order 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. For adding a package server mirror after installation, follow the instructions at https://www.debian.org/doc/manuals/debian-faq/ch-uptodate.en.html

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

1. Add the AAC Package Archive Server

   Open a terminal and execute the following command:

   ```
   sudo gedit /etc/apt/sources.list.d/aac-repo.list
   ```

   This will open a graphical editor; add the following lines to the file and then save and close it:

```
deb http://repo.aacmicrotec.com/archive/ aac/
deb-src http://repo.aacmicrotec.com/archive/ aac/
```

Add the key for the package archive as trusted by issuing the following command:

```
wget -O - http://repo.aacmicrotec.com/archive/key.asc | sudo
apt-key add -
```

The terminal will echo "OK" on success.

2. Install the Toolchain Package

   Update the package cache and install the wanted toolchain by issuing the following commands:

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

3. Setup

   In order to use the toolchain commands, the shell PATH variable needs to be set to include them. This can be done temporarily for the current shell via

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

   To always have the toolchain in the PATH, edit the ~/.bashrc (or equivalent) file

```
gedit ~/.bashrc
```

   and add the following snippet at 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
```

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 https://www.gaisler.com/index.php/downloads/compilers).

3.4. Installing the Board Support Package (BSP)

Board support packages can be found at http://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 chapter 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 https://www.gaisler.com/index.php/downloads/debug-tools. For further instructions please refer to the GRMON3 manual, which is available at https://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:

https://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 [RD16]). 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 (see section 5.2) 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.

In GRMON: `wmem 0xCB000000 0x0`

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

A manual reset can be triggered through the Error Manager (see section 5.3).

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 chapter 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 3.5.5. 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.

The following additional steps are needed to re-initialize the TCM with the TCM core application:

3. Compile and run the `board_initialiser` application using the debugger. This application is located in the `src/nv_config/src/board_initialiser/` directory in the TCM-S BSP; it is compiled as an RTEMS application in a similar fashion as the example applications described in section 4.1.

   This will initialize the mass memory bad block table in the NVRAM.

4. Compile and run the nv_config utility as described in section 7.4.2

   This will initialize the NVRAM configuration parameters.

4. Software development

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

The toolchain (see chapter 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 chapter 3.3 needs to be installed and the BSP unpacked as described in chapter 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 chapter 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 [RD14]. For details about the floating-point unit in the LEON3 systems, see [RD15].

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. The block diagram representing driver functionality access via the RTEMS API is shown in Figure 5-1.

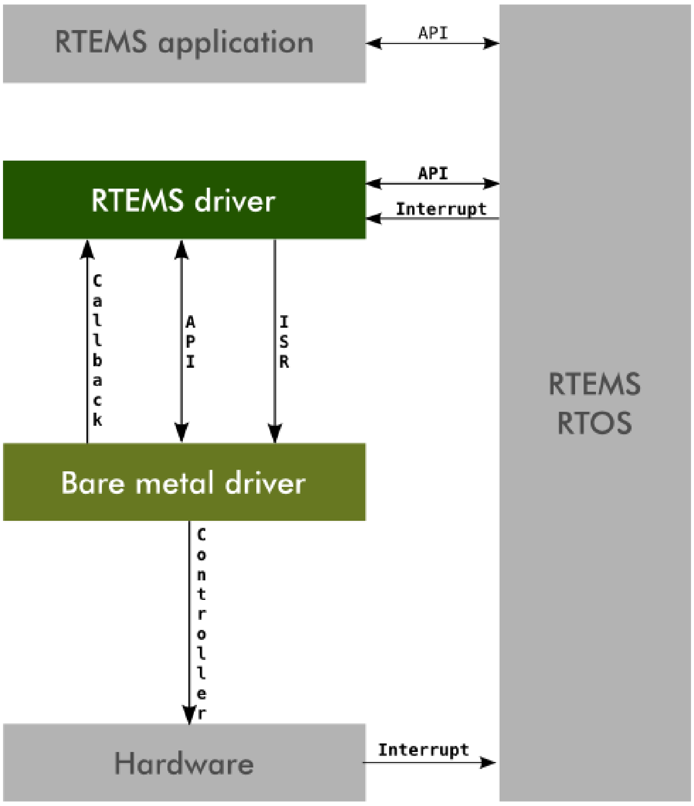


Figure 5-1 - Functionality access via RTEMS API

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. RTEMS 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. int open(…)

Opens access to the bare metal driver. The device can only be opened once at a time.

Argument name	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	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.

5.2.2.2. int close(…)

Closes access to the device.

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**

Return value	Description
0	Device closed successfully
-1	see *errno* values
errno values	
EPERM	Device is not open.

5.2.2.3. ssize_t write(…)

Any data is accepted as a watchdog kick.

Argument name	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
*	nNumber of bytes that were written.
- 1	see *errno* values
errno values	
EPERM	Device was not opened
EBUSY	Device is busy

5.2.2.4. int ioctl(…)

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

Argument name	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) 0 = Disables the watchdog **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	
EINVAL	Invalid data sent

RTEMS_NOT_DEFINED	Invalid I/O command

5.2.3. Usage description

5.2.3.1. RTEMS

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 subchapter 5.2.2. And, if desired, the access can be closed when not needed.

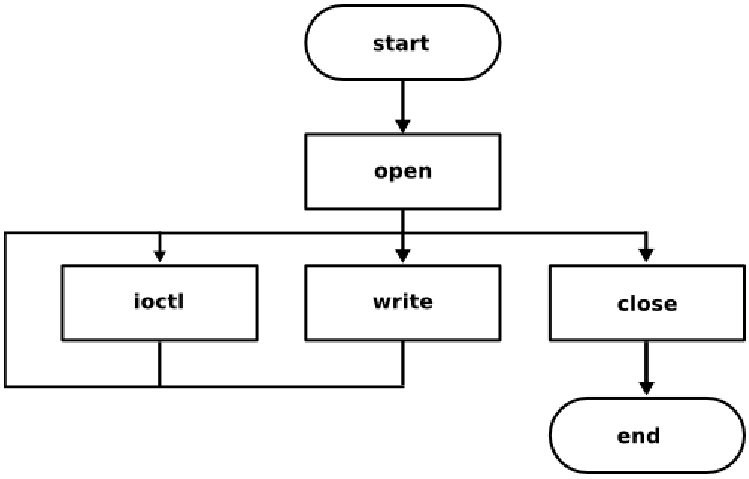


Figure 5-2 – RTEMS driver usage description

All calls to RTEMS driver are blocking calls.

5.2.3.2. RTEMS application example

To use the watchdog driver in the RTEMS environment, the following code structure is suggested:

```
#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 10
#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`.

`CONFIGURE_APPLICATION_NEEDS_WDT_DRIVER` must be defined for using the watchdog driver. By defining this as part of the RTEMS configuration, the driver will automatically be initialised at boot up.

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. RTEMS 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, 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 non-fatal multiple errors generated by the RAM EDAC mechanism.

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	cause	Latest cause of reset, encoded as: 0x0 – Power-Up 0x1 – Watchdog 0x2 – Manual (SW initiated) 0x3 – CPDU (safe image) 0x4 – CPDU (default image) 0x5 – CPU multi-bit error (Uncorrectable) 0x6 – CPU parity error
uint8_t	RESERVED	-

5.3.2.2. int open(…)

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

Argument name	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.

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	
EALREADY	Device already opened

5.3.2.3. int close(…)

Closes access to the device.

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**

Return value	Description
0	Device closed successfully

5.3.2.4. int ioctl(…)

5.3.2.4.1. Description

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

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**
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

5.3.2.4.2. Commands

Command table	Type	Description
ERRMAN_GET_SR_IOCTL	uint32_t *	Get the status register, see 5.3.2.4.3
ERRMAN_GET_CF_IOCTL	uint32_t *	Gets the carry flag register, see 5.3.2.4.4
ERRMAN_GET_SELFW_IOCTL	uint32_t *	Points to which boot firmware that will be loaded and executed upon system reboot. 0x0: Programmable FW from Power on 0x1: Programmable FW, Backup copy 0x2: Programmable FW, Backup copy 0x3: Safe FW 0x4: Safe FW, Backup copy 0x5: Safe FW, Backup copy

ERRMAN_GET_RUNFW_IOCTL	uint32_t *	Gets the currently running firmware 0x0: Programmable FW from Power on 0x1: Programmable FW, Backup copy 0x2: Programmable FW, Backup copy 0x3: Safe FW 0x4: Safe FW, Backup copy 0x5: Safe FW, Backup copy
ERRMAN_GET_SCRUBBER_IOCTL	uint32_t *	Gets the state of the memory scrubber. 0 = Scrubber is disabled 1 = Scrubber is enabled.
~~ERRMAN_GET_RESET_ENABLE_IOCTL~~	~~uint32_t *~~	~~Gets the reset enable state.~~ ~~0 = Soft reset is disabled.~~ ~~1 = Soft reset is enabled~~ The command is deprecated and might be removed in future releases.
ERRMAN_GET_WDT_ERRCNT_IOCTL	uint32_t *	Gets the watchdog error count register. This register can store a value up to 15 and then wraps. After a wrap the WDT carry flag bit is set in the carry flag register. see 5.3.2.4.4
ERRMAN_GET_EDAC_SINGLE_ ERRCNT_IOCTL	uint32_t *	Gets the EDAC single error count. See 5.3.2.4.5 for interpretation of the register. After a wrap the EDAC single error count carry flag bit is set in the carry flag register. See 5.3.2.4.4
ERRMAN_GET_EDAC_MULTI_ ERRCNT_IOCTL	uint32_t *	Gets the EDAC multiple error count. See 5.3.2.4.6 for interpretation of the register. After a wrap the EDAC multiple error count carry flag bit is set in the carry flag register. See 5.3.2.4.4
ERRMAN_GET_CPU_PARITY_ ERRCNT_IOCTL	uint32_t *	Gets the CPU Parity error count register. This register can store a value up to 15 and then wraps. After a wrap the CPU parity error count carry flag bit is set in the carry flag register. See 5.3.2.4.4
ERRMAN_GET_SYS_SINGLE_ ERRCNT_IOCTL	uint32_t *	Gets the system flash single error (correctable) error count. This register is 4 bit wide and will wrap upon overflow.
ERRMAN_GET_SYS_MULTI_ ERRCNT_IOCTL	uint32_t *	Gets the system flash multiple error (un-correctable) error count. This register is 4 bit wide and will wrap upon overflow.
ERRMAN_GET_MMU_SINGLE_ ERRCNT_IOCTL	uint32_t *	Gets the mass memory single error (correctable) error count. This register is 4 bit wide and will wrap upon overflow.
ERRMAN_GET_MMU_MULTI_ ERRCNT_IOCTL	uint32_t *	Gets the mass memory multiple error (un-correctable) error count. This register is 4 bit wide and will wrap upon overflow.
ERRMAN_GET_NVRAM_SINGLE_ ERRCNT_IOCTL	uint32_t*	Gets the nvram single error (correctable) error count. This register is 4 bit wide and will wrap upon overflow

ERRMAN_GET_NVRAM_DOUBLE_ ERRCNT_IOCTL	uint32_t*	Gets the nvram double error (correctable) error count. This register is 4 bit wide and will wrap upon overflow
ERRMAN_GET_NVRAM_MULTI_ ERRCNT_IOCTL	uint32_t*	Gets the nvram multiple error (un-correctable) error count. 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 the corresponding timestamp
ERRMAN_GET_LATEST_BOOT_ STATUS_IOCTL	uint32_t*	Gets the latest boot status. See 5.3.2.4.7 for details.
ERRMAN_SET_SR_IOCTL	uint32_t	Sets the status register, see 5.3.2.4.3
ERRMAN_SET_CF_IOCTL	uint32_t	Sets the carry flag register, see 5.3.2.4.4
ERRMAN_SET_SELFW_IOCTL	uint32_t	Sets the next boot firmware. 0x0: Programmable FW from Power on 0x1: Programmable FW, Backup copy 0x2: Programmable FW, Backup copy 0x3: Safe FW 0x4: Safe FW, Backup copy 0x5: Safe FW, Backup copy
ERRMAN_RESET_SYSTEM_IOCTL	uint32_t	Performs a software reset (value ignored).
ERRMAN_SET_SCRUBBER_IOCTL	uint32_t	Sets the state of the memory scrubber. 1 = On, 0 = Off. 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.~~ ~~0 = Soft reset is disabled.~~ ~~1 = Soft reset is enabled~~ The command is deprecated and might be removed in future releases.
ERRMAN_SET_WDT_ERRCNT_IOCTL	uint32_t	Sets the watchdog error count register. 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 EDAC single error count. See 5.3.2.4.5 for register definition.
ERRMAN_SET_EDAC_MULTI_ ERRCNT_IOCTL	uint32_t	Sets the EDAC multiple error count register. See 5.3.2.4.6 for register definition.
ERRMAN_SET_CPU_PARITY_ ERRCNT_IOCTL	uint32_t	Sets the CPU Parity error count register. The counter width is 4 bits i. e. 15 is the maximum value that can be written.
ERRMAN_SET_SYS_SINGLE_ ERRCNT_IOCTL	uint32_t	Sets the system flash single (correctable) error counter. This register is 4 bit wide.
ERRMAN_SET_SYS_MULTI_ ERRCNT_IOCTL	uint32_t	Sets the system flash multiple (un-correctable) error counter. This register is 4 bit wide.
ERRMAN_SET_MMU_SINGLE_ ERRCNT_IOCTL	uint32_t	Sets the mass memory single (correctable) error counter. This register is 4 bit wide.
ERRMAN_SET_MMU_MULTI_ ERRCNT_IOCTL	uint32_t	Sets the mass memory multiple (un-correctable) error counter. This register is 4 bit wide.
ERRMAN_SET_NVRAM_SINGLE_ ERRCNT_IOCTL	uint32_t	Sets the nvram single (correctable) error counter. This register is 4 bit wide

ERRMAN_SET_NVRAM_DOUBLE_ ERRCNT_IOCTL	uint32_t	Sets the nvram double (correctable) error counter. This register is 4 bit wide
ERRMAN_SET_NVRAM_MULTI_ ERRCNT_IOCTL	uint32_t	Sets the nvram multiple (un-correctable) error counter. 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

5.3.2.4.3. Status register

Bit position	Name	Direction	Description
31:23	RESERVED		
22:20	ERRMAN_RESET_CAUSE	R	Cause of reset encoded as: 0x0 – Power-Up 0x1 – Watchdog 0x2 – Manual (SW initiated) 0x3 – Pulse Command (safe image) 0x4 – Pulse Command (default image) 0x5 – CPU multi-bit error (Un-correctable)
19	RESERVED		
18	ERRMAN_MNVERRFLG	R/W	A previous un-correctable multi error has been detected in the NVRAM. Clear flag by writing a ‘1’.
17	ERRMAN_DNVERRFLG	R/W	A previous correctable double error has been detected in the NVRAM. Clear flag by writing a ‘1’.
16	ERRMAN_SNVERRFLG	R/W	A previous correctable single error has been detected in the NVRAM. Clear flag by writing a ‘1’.
15	ERRMAN_MMMERRFLG	R/W	A previous un-correctable multi error in the mass memory has been detected. Clear flag by writing a ‘1’.
14	ERRMAN_SMMERRFLG	R/W	A previous correctable single error in the mass memory has been detected. Clear flag by writing a ‘1’.
13	ERRMAN_MSYSERRFLG	R/W	A previous un-correctable multi error in the system flash has been detected. Clear flag by writing a ‘1’.
12	ERRMAN_SSYSERRFLG	R/W	A previous correctable single error in the system flash has been detected. Clear flag by writing a ‘1’.
11	ERRMAN_PULSEFLG	R/W	Pulse command flag bit is set. 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		
7	ERRMAN_PARFLG	R/W	A previous CPU register file parity error has been detected. Clear flag by writing a ‘1’
6	ERRMAN_MEOTHFLG	R/W	A previous RAM EDAC un-correctable multiple error has been detected for non-critical data. Clear flag by writing a ‘1’
5	ERRMAN_SEOTHFLG	R/W	A previous RAM EDAC single error has been detected and corrected for non-critical data. Clear flag by writing a ‘1’.
4	ERRMAN_MECRIFLG	R/W	A previous RAM EDAC un-correctable multiple error has been detected for critical data. Clear flag by writing a ‘1’.
3	ERRMAN_SECRIFLG	R/W	A previous RAM EDAC single error has been detected and corrected for critical data. Clear flag by writing a ‘1’

2	ERRMAN_WDTFLAG	R/W	A previous watch dog timer reset has been detected. Clear flag by writing a ‘1’
1	ERRMAN_RFLG	R/W	A previous manual reset has been detected. Clear flag by writing a ‘1’
0	ERRMAN_IFLAG	R/W	Error Manager Interrupt Flag 0 = No interrupt pending 1 = Interrupt pending Clear flag by writing a ‘1’

5.3.2.4.4. Carry flag register

Bit position	Name	Direction	Description
31:19	RESERVED		
18	ERRMAN_MNVERRCFLG	R/W	Carry flag set when NVRAM Multiple error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
17	ERRMAN_DNVERRCFLG	R/W	Carry flag set when NVRAM Double error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
16	ERRMAN_SNVERRCFLG	R/W	Carry flag set when NVRAM Single error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
15	ERRMAN_MMMERRCFLG	R/W	Carry flag set when Mass Memory Multiple error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
14	ERRMAN_SMMERRCFLG	R/W	Carry flag set when Mass Memory Single error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
13	ERRMAN_MSYSERRCFLG	R/W	Carry flag set when Sysflash Multiple error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
12	ERRMAN_SSYSERRCFLG	R/W	Carry flag set when Sysflash Single error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
11:8	RESERVED		
7	ERRMAN_PARCFLG	R/W	Carry flag set when CPU register file parity error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)
6	ERRMAN_MEOFLG	R/W	Carry flag set when RAM EDAC multiple other error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing ‘1’)
5	ERRMAN_SEOFLG	R/W	Carry flag set when RAM EDAC single other error counter overflow has occurred ‘0’ – No CF set ‘1’ – Counter overflow (Cleared by writing a ‘1’)

4	ERRMAN_MECFLG	R/W	Carry flag set when RAM EDAC multiple critical error counter overflow has occurred '0' – No CF set '1' – Counter overflow (Cleared by writing '1')
3	ERRMAN_SECFLG	R/W	Carry flag set when RAM EDAC single critical error counter overflow has occurred '0' – No CF set '1' – Counter overflow (Cleared by writing a '1')
2	ERRMAN_WDTCFLG	R/W	Carry flag set when watch dog reset counter overflow has occurred '0' – No CF set '1' – Counter overflow (Cleared by writing a '1')
1:0	RESERVED	-	

5.3.2.4.5. Single EDAC error register

Bit position	Name	Direction	Description
31:20	RESERVED	-	
19:16	ERRMAN_SENOCNT_SDRAM	R/W	SDRAM EDAC single error counter for non-critical errors
15:4	RESERVED	-	
3:0	ERRMAN_SECRICNT_SDRAM	R/W	SDRAM EDAC single error counter for critical errors

5.3.2.4.6. Multiple EDAC error register

Bit position	Name	Direction	Description
31:20	RESERVED	-	
19:16	ERRMAN_MENOCNT	R/W	SDRAM EDAC multiple error counter for non-critical errors
15:4	RESERVED	-	
3:0	ERRMAN_MECRICNT	R/W	SDRAM EDAC multiple error counter for critical errors

5.3.2.4.7. 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.

Bit position	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. 0x0 – Updated image copy #3 0x1 – Updated image copy #2 0x2 – Updated image copy #1 0x3 – Safe image copy #3 0x4 – Safe image copy #2 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. 0x01 – Init 0x02 – Init timer 0x03 – Init UART 0x04 – Read SoC info 0x05 – Wait for scrubber 0x06 – Read bad-block table 0x07 – Set image 0x08 – Check bad-block table 0x09 – Get SCET before load 0x0A – Init sysflash 0x0B – Load image 0x0C – Compute load time 0x0D – Verify checksum 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

5.3.3.1. RTEMS

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 subchapter 5.3.2. And, if desired, the access can be closed when not needed.

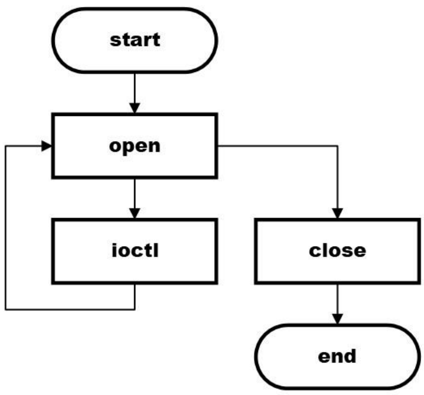


Figure 5-3 - RTEMS driver usage description

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'".
This queue emits messages upon single correctable errors.
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	Multiple EDAC errors that are not critical have been detected

5.3.3.2. RTEMS application example

To use the error manager driver in the RTEMS environment, the following code structure is suggested:

```
#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_rtems.h>` is required for accessing error manager device name `RTEMS_ERROR_MANAGER_DEVICE_NAME`.

`CONFIGURE_APPLICATION_NEEDS_ERROR_MANAGER_DRIVER` must be defined for using the error manager driver. By defining this as part of RTEMS configuration, the driver will automatically be initialised at boot up.

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 main purpose of the SCET IP and driver is to track the time since power on and to act as a source of timestamps. The SCET has also been enhanced with General purpose triggers and PPS signaling.

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

5.4.2. 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 clock for later software readout, matching 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 [RD18].

5.4.3. Pulse-Per-Second (PPS) signals

5.4.3.1. Description

The SCET block is designed to be included in several different units in a system and for time synchronization between these SCETs; each SCET can receive and/or transmit PPS signals using two PPS signals which is intended for off-chip use. The first signal, pps0, is an input only and intended to be used with a time-aware component such as a GPS device for synchronizing the SCET counter to real time. The second signal, pps1, is bidirectional and intended for use in a multi-drop PPS network. One unit in a system can act as master on the multi-drop PPS network with the other units as slaves, with the ability to switch master depending on the redundancy concept used.

When the SCET synchronizes the time counter with a PPS signal, it will also monitor this PPS signal to make sure it arrives as expected within a user set timeframe (PPS threshold). If input PPS is lost, it requires software interaction to resynchronize to the incoming PPS pulse. This is to minimize the risk for sudden glitches in the SCET counter depending on the incoming PPS accuracy and availability. The PPS monitoring will issue interrupts in bare-metal or messages on the SCET message queue in RTEMS to notify the application if the PPS has arrived, been lost or been found.

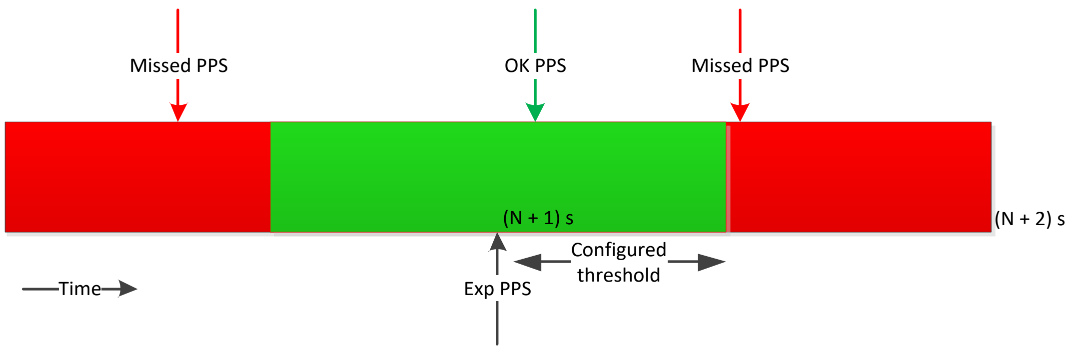


Figure 5-4 PPS Threshold configuration

To differentiate between the uses of the PPS signal synchronization methods, the SCET can be said to operate in a few different modes: Free-running, Master, Master with time synchronization and Slave. Please see the explanations below and 5.4.5.1 for an implementation description.

5.4.3.2. Free-running mode

In this mode, the SCET doesn't use any PPS signals at all. It simply counts the current time since power on without correlation with anyone else.

5.4.3.3. Master mode

In this mode, the SCET is still counting on its own, but now it also emits a pulse on pps1 for every second tick, acting as a master on the bidirectional multi-drop PPS network.

5.4.3.4. Master mode with time synchronization

This mode is the same as the previous master mode, with the addition of also synchronizing the time counter with the incoming pps0 signal. Should the PPS signal on pps0 disappear for some reason, it will revert back to normal master mode and continue issuing PPS signals on pps1.

5.4.3.5. Slave mode

In this mode, the SCET will synchronize the time counter with pps1, using the bidirectional multi-drop PPS network as an input. Should the PPS pulse disappear for some reason, it will revert to free running mode.

5.4.4. RTEMS 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 are not specifically affecting the SCET counter registers.

For event signaling, the SCET driver uses message queues, allowing the application to act upon different events.

5.4.4.1. Function int open(…)

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

Argument name	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

5.4.4.2. Function int close(…)

Closes access to the device.

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**

Return value	Description
0	Device closed successfully

5.4.4.3. 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 name	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

5.4.4.4. Function ssize_t write(…)

Offsets the SCET by an offset specified by buf.

Argument name	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. 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

5.4.4.5. Function int ioctl(…)

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

Note: Please note that the number of available PPS inputs and outputs depend on the hardware configuration.

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

Command table	Description
SCET_SET_PPS_SOURCE_IOCTL	Input value sets the PPS source. 0 = External PPS source 1 = Internal PPS source (default)
SCET_GET_PPS_SOURCE_IOCTL	Returns the current PPS source 0 = External PPS source 1 = Internal PPS source (default)
SCET_SET_PPS_OUTPUT_PORT_IOCTL	Input bit field configures which PPS output drivers to enable. Bit 0 is the output driver of PPS0. Bit N is the output driver of PPSN. Bit 7 is the output driver of PPS7. Bit value 0 = The output driver is disabled Bit value 1 = The output driver is enabled (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. Bit 0 is the output driver of PPS0. Bit N is the output driver of PPSN. Bit 7 is the output driver of PPS7. Bit value 0 = The output driver is disabled Bit value 1 = The output driver is enabled
SCET_SET_PPS_INPUT_PORT_IOCTL	Argument value sets the PPS input port. 0 is PPS0 N is PPSN 7 is PPS7 (1 is default value, PPS1 is default input)
SCET_GET_PPS_INPUT_PORT_IOCTL	Returns the currently used PPS input port. 0 is PPS0 N is PPSN 7 is PPS7
SCET_SET_PPS_THRESHOLD_IOCTL	Input value configures the PPS threshold window where the PPS pulse is allowed to arrive without being deemed lost. Defined in number of subseconds, [0,65535]. (0 is default)
SCET_GET_PPS_THRESHOLD_IOCTL	Returns the currently configured PPS threshold window in subseconds. (0 is default)
SCET_GET_PPS_ARRIVE_COUNTER_IOCTL	Returns 24 bits of the SCET time sampled when PPS arrived. Bit 23:16 contains lower 8 bits of second Bit 15:0 contains subseconds
SCET_SET_GP_TRIGGER_LEVEL_IOCTL	Input bit field configures the trigger level of each trigger: Bit 0 is trigger 0, Bit N is trigger N, Bit 7 is trigger 7. Bit value 0 = trigger activates on 0 to 1 transition (rising edge) Bit value 1 = trigger activates on 1 to 0 transition (falling edge). (0 is default).

SCET_GET_GP_TRIGGER_LEVEL_IOCTL	Returns the currently configured level of the all GP triggers as a bit field: Bit 0 is trigger 0, Bit N is trigger N, Bit 7 is trigger 7. Bit value 0 = trigger activates on 0 to 1 transition (rising edge) Bit value 1 = trigger activates on 1 to 0 transition (falling edge). (0 is default).
SCET_SET_GP_TRIGGER_ENABLE_IOCTL	Input bit field selects which GP trigger(s) to enable: Bit 0 is trigger 0, Bit N is trigger N, Bit 7 is trigger 7. All triggers are disabled by default (0)
SCET_GET_GP_TRIGGER_ENABLE_IOCTL	Returns which GP triggers that are enabled. Bit 0 is trigger 0, Bit N is trigger N, Bit 7 is trigger 7.
SCET_GET_GP_TRIGGER_COUNTER_IOCTL	Input value selects which GP trigger SCET counter sample to read [0,7]. Returns 24 bits of the SCET counter sampled when the GP trigger became active. Bit 23:16 contains lower 8 bits of second Bit 15:0 contains subseconds

5.4.4.1. 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 a PPS input/output configuration other than the one described in the table below (PPS0 input/output, PPS1 input/output) is used, 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. 0 = pps1 is input, no output ports are activated, (default) 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. 0 = pps1 is input, no output ports are activated, (default) 1 = pps0 is input, pps1 is output

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.

5.4.5. Usage description

5.4.5.1. PPS

The four described PPS modes can be obtained by setting the PPS output enable and PPS source according to Table 5-1.

Table 5-1 Mapping between PPS modes and PPS settings

PPS mode	PPS source	PPS output enable
Free-running (default)	Internal	Input
Master	Internal	Output
Master with time synchronization	External	Output
Slave	External	Input

When PPS source is set to external and then lost, it will revert to internal setting.
Slave mode will fall back to Free-running mode and Master mode with time synchronization will revert back to Master mode.
When PPS source is set to internal: If an incoming PPS is detected the PPS found interrupt is asserted. Typically a number of these PPS found interrupts should be investigated by the application and once the PPS is deemed stable enough the PPS source should be set to external (if external synchronization is sought after).
It is up to the application to decide and enforce if and when the external PPS source is to be used again.

5.4.5.2. PPS Threshold

The PPS threshold has a 16 bit resolution and is used to define the subsecond range within which incoming PPS that are deemed acceptable.
The range of acceptability is calculated as >= (65535 – threshold) to <= (65535 + 1 + threshold) subseconds after the previous PPS.

If the PPS threshold is configured to 0 (min value) only incoming PPS that arrive within >= subsecond 65535 of the current second to < subsecond 1 of the next second will be deemed acceptable, (>= 0.65535 to <= 1.0).
If the PPS threshold is configured to 65535 (max value) all incoming PPS are deemed acceptable. Lost events will not be detected at all.

5.4.5.3. Event callback via message queue

The SCET driver exposes message queues for event messaging from the driver to the application. The queues use broadcast, so multiple subscribers are possible, but a subscriber has to be waiting for the message to receive it (see RD14 section 14.4.6.and 14.4.7, polling with RTEMS_NO_WAIT is not possible).

Broadcasting in rtems means that the message will only be copied to listeners if:

- Their task is blocked waiting on the message queue at the time of broadcast.
- There is no other pending message in the message queue.
- Their task has not been made ready by a previous send or broadcast on the queue.

Otherwise, the broadcast message will be discarded and _not_ queued.

For example, if one task is blocked waiting on the message queue, and another task broadcasts two messages without any intervening cpu yield, the receiving task would only see the first message, and the second message would be discarded.

This can also occur more generally based on task priority configuration. If a task, A, was configured with higher priority than the timestamp message listener task, B. It could potentially perform multiple tasks A whilst starving the task B (potentially needing another unrelated high priority task as well), resulting in only seeing the timestamp for the first of many measurements.

The use of broadcast also forces the use of separate tasks for timestamp handling. If using only normal queues, and assuming only one listener, it would be possible to handle the timestamp in the same task as the "generator".

The Scet PPS Perioc task, ‘S’, ‘P’, ‘P’, ‘S’, handles PPS related messages in Table 5-2.

Table 5-2 Driver message queue message types

Event name	Description
SCET_INTERRUPT_STATUS_PPS_ARRIVED	An external PPS signal has arrived
SCET_INTERRUPT_STATUS_PPS_LOST	The external PPS signal is lost
SCET_INTERRUPT_STATUS_PPS_FOUND	The external PPS signal was found

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 5-3.

Table 5-3 General purpose trigger n message queue

Event name	Description
SCET_INTERRUPT_STATUS_TRIGGERn	Trigger n was triggered

5.4.5.4. RTEMS application example

To use the SCET driver in the RTEMS environment, the following code structure is suggested:

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.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));

  result = write(scet_fd, write_buffer, 6);
  assert(result == 6);

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

  memcpy(&new_seconds, read_buffer, sizeof(uint32_t));
  memcpy(&new_subseconds, read_buffer + sizeof(uint32_t),
         sizeof(uint16_t));

  printf("New SCET time is %lu.%u\n",
         new_seconds,  new_subseconds);

  result = close(scet_fd);
  assert(result == 0);

  rtems_task_delete(RTEMS_SELF);
}
```

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.

`CONFIGURE_APPLICATION_NEEDS_SCET_DRIVER` must be defined for using the SCET driver. By defining this as part of RTEMS configuration, the driver will automatically be initialized at boot up.

5.4.6. Limitations

None

5.5. UART

5.5.1. Description

This device is based on the interface for a 16550D UART given in [RD4] and as such has an 8-bit interface, but has been expanded to provide a faster and more delay-tolerant implementation.

5.5.1.1. 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 both RX and TX FIFOs handling in the RTEMS driver.

5.5.1.2. Trigger levels

To be able to utilize the larger FIFOs, the meaning of the trigger levels has been changed. In the specification in [RD4], 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 backwards compatible, since a buffer depth of 16 characters would yield the same trigger levels as those given in [RD4].

5.5.1.3. Modes

The UARTs (0-5) 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. RTEMS 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 [RD18] for the current SoC configuration, including device name and characteristics for each UART device.

Argument name	Type	Direction	Description
pathname	const char *	in	The absolute path to the file that is to be opened, see [RD18].
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	
ENODEV	Device does not exist
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 name	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.

Please note that the read call may return less data than requested.

Argument name	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 name	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 name	Type	Direction	Description
fd	int	in	File descriptor received at **open**
cmd	int	in	Command to send
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: UART_B375000 UART_B347200 UART_B153600 UART_B115200 (default) UART_B76800 UART_B57600 UART_B38400 UART_B19200 UART_B9600 UART_B4800 UART_B2400 UART_B1200
UART_IOCTL_MODE_SELECT	uint32_t	in	Set the mode of the interface. Possible values: UART_RTEMS_MODE_RS422 (default) UART_RTEMS_MODE_RS485 UART_RTEMS_MODE_LOOPBACK (TX connected to RX internally)
UART_IOCTL_RX_FLUSH	uint32_t	in	Flushes the RX software FIFO
UART_IOCTL_SET_PARITY	uint32_t	in	Set parity. Possible values: UART_PARITY_NONE (default) UART_PARITY_ODD UART_PARITY_EVEN
UART_IOCTL_SET_BUFFER_DEPTH	uint32_t	in	Set the FIFO buffer depth. Possible values: UART_BUFFER_DEPTH_16 (default) UART_BUFFER_DEPTH_32 UART_BUFFER_DEPTH_64 UART_BUFFER_DEPTH_128
UART_IOCTL_GET_BUFFER_DEPTH	uint32_t*	out	Get the current buffer depth.
UART_IOCTL_SET_TRIGGER_LEVEL	uint32_t	in	Set the RX FIFO trigger level. 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

5.5.3.1. RTEMS application example

To use the uart driver in the RTEMS environment, the following incomplete code structure is suggested (see Board Support Package for a complete example program):

```
#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.

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. Data Structures

5.6.2.1. Struct massmem_cid_t

This struct is used as the target for reading the mass memory chip IDs.

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

5.6.2.2. 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 XORed with generated EDAC byte
uint32_t	data_error_injection	Bits to be XORed with supplied data

5.6.2.3. Struct massmem_ioctl_spare_area_args_t

This struct is used by the RTEMS API as the target when reading from spare area and data simultaneously.

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.

5.6.2.4. 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 5.6.2.2 for definition

5.6.3. RTEMS API

5.6.3.1. int open(…)

Opens access to the driver. The device can only be opened once at a time.

Argument name	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	Specifies one of the access modes in the following table.

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

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.

5.6.3.2. int close(…)

Closes access to the device.

Argument name	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

5.6.3.3. off_t lseek(…)

Sets page offset for read/ write operations.

Argument name	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**.

5.6.3.4. 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 name	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.

5.6.3.5. 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 name	Type	Direction	Description
fd	int	in	File descriptor received at **open**.
buf	void *	in	Character buffer to read data from.
nbytes	ssize_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.
EAGAIN	Driver failed to write data. Try again.
EIO	Failed to write data. Block should be marked as a bad block.

5.6.3.6. int ioctl(…)

5.6.3.6.1. Description

Additional supported operations via POSIX Input/Output Control API.

Argument name	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 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

5.6.3.6.2. Reset mass memory device

Resets the mass memory device.

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

5.6.3.6.3. 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.

5.6.3.6.4. 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.

5.6.3.6.5. 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.

5.6.3.6.6. 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 5.6.2.1.

5.6.3.6.7. 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

5.6.3.6.8. Read spare area

Reads the spare area.

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, and destination buffers where spare area data is to be stored, see 5.6.2.3

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

5.6.3.6.9. 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 to be written, see 5.6.2.3

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

5.6.3.6.10. Bad block check

Reads the factory bad block status from a block.

Note that 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 page number is out of range, buffers are NULL or not 4-byte aligned.

5.6.3.6.11. 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 5.6.2.4

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

5.6.3.6.12. Get page bytes

Get the available page size in bytes.

- If the BSP is compiled without BSP_AAC_MASSMEM_ENABLE_32GB defined, the value will always be equal to the static define MASSMEM_PAGE_BYTES regardless of the chip type in use.

- If the BSP is compiled with BSP_AAC_MASSMEM_ENABLE_32GB defined, 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.

This is provided in order to support the runtime-determined size usage mode, see 5.6.4.5.

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.

5.6.3.6.13. Get spare area bytes

Get the available spare area size in bytes.

- If the BSP is compiled without BSP_AAC_MASSMEM_ENABLE_32GB defined, the value will always be equal to the static define MASSMEM_SPARE_AREA_BYTES regardless of the chip type in use.
- If the BSP is compiled with BSP_AAC_MASSMEM_ENABLE_32GB defined, 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.

This is provided in order to support the runtime-determined size usage mode, see 5.6.4.5.

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.

5.6.4. Usage description

5.6.4.1. 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.

5.6.4.2. Overview

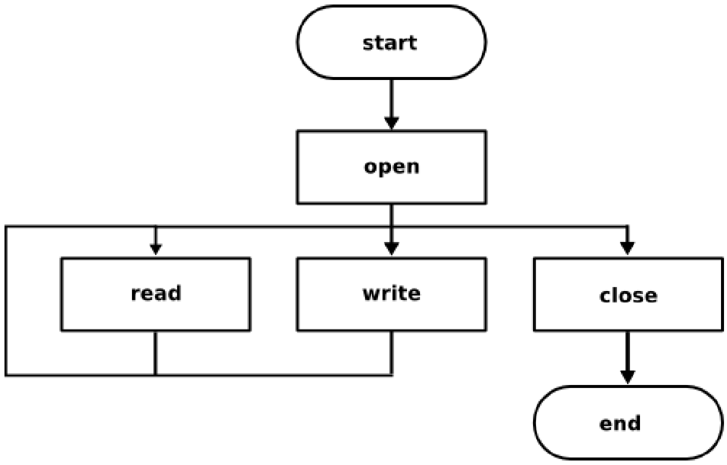
The RTEMS driver accesses the mass memory by the reference a page number.

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.

5.6.4.3. 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 5.5.2. And, if desired, the access can be closed when not needed.



Note! All calls to RTEMS driver are blocking calls.

In order to support different chip types with different size characteristics, two separate modes of usage are available for determining the page size:

5.6.4.4. Same-size usage mode

This usage mode is backwards-compatible, and exposes 16GB of available space regardless of the chip type, it defines MASSMEM_PAGE_BYTES and MASSMEM_SPARE_AREA_BYTES for use by the application at compile-time.

This usage mode is only available if the RTEMS BSP is compiled **without** the BSP_AAC_MASSMEM_ENABLE_32GB define set.

To use the mass memory flash driver in the RTEMS environment with the same-size usage mode, the following incomplete code structure is suggested:

```
#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];

rtems_task Init (rtems_task_argument ignored)
{
  .
  fd = open(MASSMEM_DEVICE_NAME, O_RDWR);
  .
  off = lseek(fd, page_number, SEEK_SET);
  .
  sz = write(fd, buf, MASSMEM_PAGE_BYTES);
  .
  off = lseek(fd, page_number, SEEK_SET)
  .
  sz = read(fd, buf, MASSMEM_PAGE_BYTES);
  .
}
```

5.6.4.5. Runtime-determined size usage mode

This usage mode allows support for differing page sizes at runtime, and defines MASSMEM_PAGE_BYTES_MAX and MASSMEM_SPARE_AREA_BYTES_MAX for use 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.

This usage mode is available both with and without the BSP_AAC_MASSMEM_ENABLE_32GB define set:

- If the driver is compiled **without** the BSP_AAC_MASSMEM_ENABLE_32GB define set, the mass memory (and corresponding available page and spare area size) will always be exposed as 16GB regardless of the chip type at runtime.

  This can be useful for migrating applications to the runtime-determined size usage mode without removing support for the same-size usage mode.

- If the driver is compiled **with** the BSP_AAC_MASSMEM_ENABLE_32GB define set, 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.

  Please note that this disables support for the same-size usage mode.

BSP_AAC_MASSMEM_ENABLE_32GB can be set as an environment or Makefile variable when compiling the RTEMS BSP.

To use the mass memory flash driver in the RTEMS environment with the runtime-determined size usage mode, the following incomplete code structure is suggested:

```
#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);
```

5.6.4.6. 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.

5.6.5. 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 5.6.2.2:
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.

5.6.6. Limitations

The mass memory flash driver may only have one open file descriptor at a time.

5.7. Spacewire

5.7.1. Description

This section describes the SpaceWire driver's design and usability.

5.7.2. RTEMS 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 subchapter 5.7.2.5.

5.7.2.1. 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 of the 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 subchapter 5.7.3.

Argument name	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 5-4.

Return value	Description
>0	A file descriptor for the device.
- 1	see *errno* values
errno values	
EIO	Internal RTEMS resource error.
EALREADY	Device already opened for the requested access mode (read or write).
ENOENT	Invalid filename.

Table 5-4 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. 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 name	Type	Direction	Description
fd	int	in	File descriptor received at **open**.

0	Device name deregistered successfully
-1	see *errno* values
errno values	
EBADF	The file descriptor *fd* is not and open file descriptor.

5.7.2.3. 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 a 32-bit aligned address.

Argument name	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	Packet size in bytes. Must be between 0 and `SPWN_MAX_PACKET_SIZE` bytes.

Return value	Description
>0	Received size of the actual packet. Can be less than *nbytes*.
0	Packet size is 0, or buffer size was lower than received packet size, with errno value is set to EOVERFLOW.
-1	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`, 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.
ETIME	The read operation timed out and no packet was received.

5.7.2.4. 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 name	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. int ioctl(…)

Additional supported operations via POSIX Input/Output Control API.

Argument name	Type	Direction	Description
fd	int	in	A file descriptor received at **open**.
cmd	int	in	Command defined in subchapter 5.7.2.6
value	void *	in	The value relating to command operation as defined in subchapter 5.7.2.6.

5.7.2.6. 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: • SPWN_IOCTL_MODE_OFF: Turns off the node. • SPWN_IOCTL_MODE_LOOPBACK: Internal loopback mode • 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.

5.7.2.7. 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 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 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.

5.7.2.8. 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: 0 – Timing mode disabled 1 – Timing mode Master 2 – Timing mode Slave
SPWN_IOCTL_TIMING_MODE_GET	spwn_timing_mode_t	Out	Gets the current timing mode. Encoded as: 0 – Timing mode disabled 1 – Timing mode Master 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.

5.7.3. Usage description

5.7.3.1. RTEMS

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.

5.7.3.1.1. 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 [RD1] and cannot be registered to.

5.7.3.1.2. Promiscuous Mode

In promiscuous mode only one reader task is needed for reading on all of the 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`

Note! 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];
```

5.7.3.2. 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 subchapter 5.7.2. And, if desired, the access can be closed when not needed.

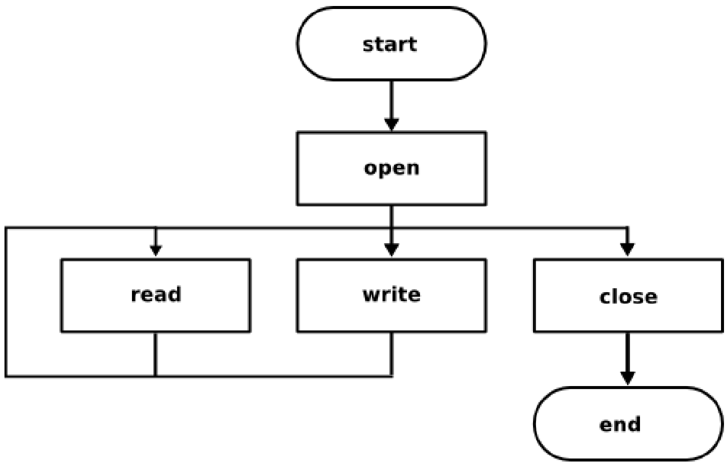


Figure 5-5 – RTEMS driver usage description

Note! All calls to RTEMS driver are blocking calls unless stated otherwise.

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.

5.7.3.3. RTEMS application example

To use the driver in the RTEMS environment, the following incomplete code structure is suggested (see Board Support Package for complete example programs):

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

.
.
#define CONFIGURE_APPLICATION_NEEDS_SPACEWIRE_DRIVER
.
.
#define CONFIGURE_INIT

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

uint8_t __attribute__ ((aligned (4))) buf_rx[SPWN_MAX_PACKET_SIZE];
uint8_t buf_tx[SPWN_MAX_PACKET_SIZE];

rtems_task Init (rtems_task_argument ignored)
{
  .
  fd = open(SPWN_DEVICE_0_NAME_PREFIX”42”, O_RDWR);
  .
}
```

The above code registers the application for using the unique device name with the logical address 42 (`SPWN_DEVICE_0_NAME_PREFIX”42”)` for data transaction.

The reception buffer `buf_rx`, is aligned to a 4-byte boundary in order to correctly handle the DMA access when receiving SpaceWire packets.

Inclusion of `<fcntl.h> and <unistd.h>` are required for using the POSIX functions `open`, `close`, `read` and `write` and `ioctl` functions for accessing the driver.

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

Inclusion of `<bsp/spacewire_node_rtems.h>` is required for driver related definitions.

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

`CONFIGURE_APPLICATION_NEEDS_SPACEWIRE_DRIVER` must be defined for using the driver. This will automatically initialise the driver at boot up.

5.8. GPIO

5.8.1. Description

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 [RD6].

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

5.8.1.1. 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.

5.8.1.2. 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.

5.8.1.3. RTEMS differential mode

In RTEMS, 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. The pins are paired in logical sequence, which means that pin 0 and 1 are paired as are pin 2 and 3 etc. 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 specific pair only, i.e. both normal single ended and differential mode pins can operate simultaneously (though not on the same pins obviously).

5.8.1.4. 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. RTEMS 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 name	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	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
Fildes	A file descriptor for the device on success
-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 name	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 has 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 also 5.8.2.5.

Argument name	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	
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 name	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 name	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. '0' output mode '1' input mode
GPIO_IOCTL_SET_DIRECTION	uint32_t	in	Set input/output direction of the pin. '0' output mode '1' input mode
GPIO_IOCTL_GET_FALL_EDGE_DETECTION	uint32_t	out	Get falling edge detection status of the pin. '0' detection disabled '1' detection enabled
GPIO_IOCTL_SET_FALL_EDGE_DETECTION	uint32_t	in	Set falling edge detection configuration of the pin. '0' detection disabled '1' detection enabled
GPIO_IOCTL_GET_RISE_EDGE_DETECTION	uint32_t	out	Get rising edge detection status of the pin. '0' detection disabled '1' detection enabled
GPIO_IOCTL_SET_RISE_EDGE_DETECTION	uint32_t	in	Set rising edge detection configuration of the pin. '0' detection disabled '1' detection enabled
GPIO_IOCTL_GET_TIMESTAMP_ENABLE	uint32_t	out	Get timestamp enable status of the pin. '0' timestamp disabled '1' timestamp enabled
GPIO_IOCTL_SET_TIMESTAMP_ENABLE	uint32_t	in	Set timestamp enable configuration of the pin. '0' timestamp disabled '1' timestamp enabled
GPIO_IOCTL_GET_DIFF_MODE	uint32_t	out	Get differential mode status of the pin. '0' normal, single ended, mode '1' differential mode
GPIO_IOCTL_SET_DIFF_MODE	uint32_t	in	Set differential mode configuration of the pin. '0' normal, single ended, mode '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.
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

5.8.3.1. RTEMS application example

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

CONFIGURE_APPLICATION_NEEDS_GPIO_DRIVER

```
#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.

See the Board Support Package for a more detailed example.

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.

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 [RD18] 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 [RD18].

5.9.2. 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.

4. 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.

5. 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.

6. 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 5.9.5.3 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.

7. . Steps 4 to 6 are repeated for all VC's signaling an interrupt.

5.9.3. 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.

5.9.4. 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.

5.9.5. RTEMS 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.

5.9.5.1. Device-file names

Access to the CCSDS-driver from an application is provided by different device-files (depending on the used SoC configuration [RD18], 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-tm*N*", 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_VC*N* 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.

5.9.5.2. 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.

5.9.5.3. Data type dma_transfer_cb_t

For TM-devices operated in non-blocking mode (see 5.9.2) a message with the contents below is sent to the message queue “CCSQ” for reporting of transfer status.

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	uint_8	Status of transfer 0 – Not send 1 – Send finished 2 – Send error

5.9.5.4. Data type tm_config_t

This datatype is a struct for configuration of the TM path. The elements of the struct are described below. **Note:** Changing bitrate (clk_divisor) and other settings that affect the TM bitstream requires that TM is first disabled, then reenabled after the change.

Element	Type	Description
clk_divisor	uint16_t	The divisor of the clock

tm_enabled	uint8_t	Enable/disable of telemetry 0 – Disable 1 – Enable
ocf_clcw_enabled	uint8_t	Enable/disable of OCF/CLCW in TM Transfer frames 0 – Disable 1 – Enable
fecf_enabled	uint8_t	Enable/disable of FECF 0 – Disable 1 – Enable
mc_cnt_enabled	uint8_t	Enable/Disable of master channel frame counter 0 – Disable 1 – Enable
idle_frame_enabled	uint8_t	Enable/disable of generation of Idle frames 0 – Disable 1 – Enable
tm_conv_bypassed	uint8_t	Bypassing of the TM convolutional encoder 0 - No bypass 1 - Bypass
tm_pseudo_rand_bypassed	uint8_t	Bypassing of the TM pseudo randomizer encoder 0 - No bypass 1 - Bypass
tm_rs_bypassed	uint8_T	Bypassing of the TM Reed Solomon encoder 0 - No bypass 1 - Bypass

5.9.5.5. 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. 0 - No bypass 1 - Bypass

5.9.5.6. 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 0 - No Error 1 - FIFO Error
tm_busy	uint8_t	Reserved

5.9.5.7. 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 CRC errors in TC path. The counter will wrap around after 2^8-1.

5.9.5.8. 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.

5.9.5.9. 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.

5.9.5.10. 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 RD8 section 4.2.1.8.3
tc_carrier	uint8_t	See RD8 section 4.2.1.8.2

5.9.5.11. 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.

Argument name	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 5.9.5.1.
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, whether it should be cleared when opened, etc). See a list of legal values for this field in the following table.
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 5.9.3. Applies only to devices "/dev/ccsds-tm*N*".

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	
EBUSY	If device already opened

5.9.5.12. int close(…)

Closes access to the device.

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**

Return value	Description
0	Device closed successfully

5.9.5.13. ssize_t write(…)

To send data on a virtual channel *N*, the device descriptor described in 5.9.5.1 ("/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, if data has not been transferred within 1500 ms the write call is aborted and an error is reported. This limits the amount of data that can be written at low bitrates.

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 5.9.2

Argument name	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 or greater	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

5.9.5.14. ssize_t read(…)

To read a Telecommand Transfer frame a read-operation on device "/dev/ccsds-tc" (see section 5.9.5.1) shall be used. This call is blocking until a Telecommand Transfer Frame is received.

Argument name	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 or greater	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.

5.9.5.15. int ioctl(…)

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

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**
cmd	int	in	Command to send
val	void *	in	The parameter to pass is depended on which IOCTL is called. Is described in 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.
CCSDS_GET_TM_CONFIG	/dev/ccsds-tm	tm_config_t *	Returns the configuration of the TM path.
CCSDS_SET_TC_CONFIG	/dev/ccsds-tc	tc_config_t *	Sets a configuration of the TC path.
CCSDS_GET_TC_CONFIG	/dev/ccsds-tc	tc_config_t *	Returns the configuration of the TC path.
CCSDS_GET_RADIO_STATUS	/dev/ccsds	radio_status_t *	Gets radio status.
CCSDS_GET_TM_STATUS	/dev/ccsds-tm	tm_status_t *	Gets status of TM path.
CCSDS_GET_TM_ERR_CNT	/dev/ccsds-tm	tm_error_cnt_t *	Gets the TM error counter.
CCSDS_GET_TC_ERR_CNT	/dev/ccsds-tc	tc_error_cnt_t *	Gets the TC error counter.
CCSDS_GET_TC_STATUS	/dev/ccsds-tc	tc_status_t *	Gets status of TC path.
CCSDS_SET_TC_FRAME_CTRL	/dev/ccsds-tc	uint32_t	Set the TC frame control register. Bit 2-31 unused.

			Bit 1: 0 – No effect 1 – Set to signal the CCSDS IP that a telecommand frame has been read. Bit 0: 0 – No effect 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.	Sets a default configuration of CCSDS IP. See 5.9.1
CCSDS_SET_CLCW	/dev/ccsds-tm	uint32_t	Set the CLCW. See RD8.
CCSDS_GET_CLCW	/dev/ccsds-tm	uint32_t *	Get the CLCW. See RD8.
CCSDS_SET_TM_TIMESTAMP	/dev/ccsds-tm	uint32_t	Set time stamp generation rate. The period of the generation is the power of two with the input as exponent. Allowed Values range from 0 to 8. 0x00 – Take a time stamp every time frame sent 0x01 – Take a time stamp every 2nd time frame sent 0x02 – Take a time stamp every 4th time frame sent … 0x08 – Take a time stamp every 256th time frame sent
CCSDS_GET_TM_TIMESTAMP	/dev/ccsds-tm	uint32_t *	Get period of timestamp generation.

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

5.9.6. Usage description

5.9.6.1. 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.

5.9.6.2. 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.

5.9.6.3. 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.

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.

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) 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]	(ADC_value*2500)/(2^24 – 1)

5.10.2. RTEMS 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	SPS 31250
ADC_IOCTL_SPS_15625	SPS 15625
ADC_IOCTL_SPS_10417	SPS 10417
ADC_IOCTL_SPS_5208	SPS 5208
ADC_IOCTL_SPS_2597	SPS 2597
ADC_IOCTL_SPS_1007	SPS 1007
ADC_IOCTL_SPS_503_8	SPS 503.8
ADC_IOCTL_SPS_381	SPS 381
ADC_IOCTL_SPS_200_3	SPS 200.3
ADC_IOCTL_SPS_100_5	SPS 100.5
ADC_IOCTL_SPS_59_52	SPS 59.52
ADC_IOCTL_SPS_49_68	SPS 49.68
ADC_IOCTL_SPS_20_01	SPS 20.01
ADC_IOCTL_SPS_16_63	SPS 16.63
ADC_IOCTL_SPS_10	SPS 10
ADC_IOCTL_SPS_5	SPS 5
ADC_IOCTL_SPS_2_5	SPS 2.5
ADC_IOCTL_SPS_1_25	SPS 1.25

5.10.2.2. Function int open(…)

Opens access to the ADC. Only one instance can be open at any time, only read access is allowed and only blocking mode is supported.

Argument name	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	
EEXISTS	Device not opened
EALREADY	Device is already open
EINVAL	Invalid options

5.10.2.3. Function int close(…)

Closes access to the device.

Argument name	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.

Note! The size of the given buffer must be a multiple of 32 bits.

Argument name	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
EINVAL	Invalid number of bytes to be read

ADC data buffer bit definition	Description
31:8	ADC value
7:4	ADC status
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.

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. 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. 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 name	Type	Direction	Description
Fd	int	in	File descriptor received at **open**
Cmd	int	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	Type	Direction	Description
ADC_SET_SAMPLE_RATE_IOCTL	uint32_t	in	Set the sample rate of the ADC chip, see [RD5].
ADC_GET_SAMPLE_RATE_IOCTL	uint32_t	out	Get the sample rate of the ADC chip, see [RD5].
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. 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

To use the ADC driver in the RTEMS environment, the following incomplete 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

#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 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);
}
```

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

The NVRAM on the OBC and 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. Description

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.

5.11.2. 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

5.11.3. 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

5.11.4. RTEMS 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.

5.11.4.1. 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.

5.11.4.2. Function int open(...)

Opens access to the requested SPI RAM.

Argument name	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 [RD12]

5.11.4.3. Function int close(...)

Closes access to the device.

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**.

Return value	Description
0	Device closed successfully
-1	See *errno* values in [RD12]

5.11.4.4. Function ssize_t read(...)

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

Argument name	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.

5.11.4.5. Function ssize_t write(...)

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

Argument name	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.

5.11.4.6. Function int lseek(...)

Set the address for the read/write operations.

Argument name	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 [RD12]

5.11.4.7. Function int ioctl(...)

Input/output control for SPI RAM.

Argument name	Type	Direction	Description
fd	Int	in	File descriptor received at **open**.
cmd	uint32_t / uint32_t*	in	Command to send.
val	Int	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 [5.11.4.1.]
SPI_RAM_SET_DIVISOR_IOCTL	uint32_t	in	Configures the serial clock divisor.
SPI_RAM_GET_EDAC_STATUS_IOCTL	uint32_t*	out	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.

5.11.5. Usage description

5.11.5.1. General

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.

5.11.5.2. RTEMS application example

To use the SPI RAM driver in RTEMS the following incomplete 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 section 9. This section details the RTEMS interface to the System flash driver.

5.12.2. Data structure types

5.12.2.1. Type sysflash_cid_t

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

Type	Name	Purpose
Array of 2 uint32_t	chip0	Byte array for chip 0 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	Purpose
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. RTEMS 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 manual 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 name	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.

oflags	int	in	Specifies one of the access modes in the following table.

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

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

5.12.3.2. Function int close(…)

Closes access to the device.

Argument name	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 off_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 name	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 for the System flash.

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 ssize_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 name	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
EINVAL	Page offset set in `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 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 name	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 in **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(…)

5.12.3.6.1. Description

Additional supported operations via POSIX Input/Output Control API.

Argument name	Type	Direction	Description
fd	int	in	File descriptor received at **open**.
cmd	ioctl_command_t	in	Command specifier
value	void *	in/out	The value relating to command operation as defined in 5.12.3.6.2 to 5.12.3.6.9.

The following return and errno values are common for all operations.

Return value	Description
0	Operation successful.
-1	See errno values
errno values	
EBADF	The file descriptor fd is not an open file descriptor.
EINVAL	Invalid command or parameter.
EBUSY	Flash controller busy.

ENODEV	Internal driver error.

5.12.3.6.2. Reset System flash

Resets the system flash chip.

Command	Value type	Direction	Description
SYSFLASH_IO_RESET	n/a	n/a	n/a

5.12.3.6.3. 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.

5.12.3.6.4. 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.

5.12.3.6.5. 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 5.12.2.1.

5.12.3.6.6. 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.

5.12.3.6.7. 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 5.12.2.2.

5.12.3.6.8. 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 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.

5.12.3.6.9. 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

5.12.4.1. 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.

5.12.4.2. Usage

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. And, if desired, the access can be closed when not needed.

NOTE: All calls to the RTEMS driver are blocking calls, though the driver uses interrupts internally to ease processor load.

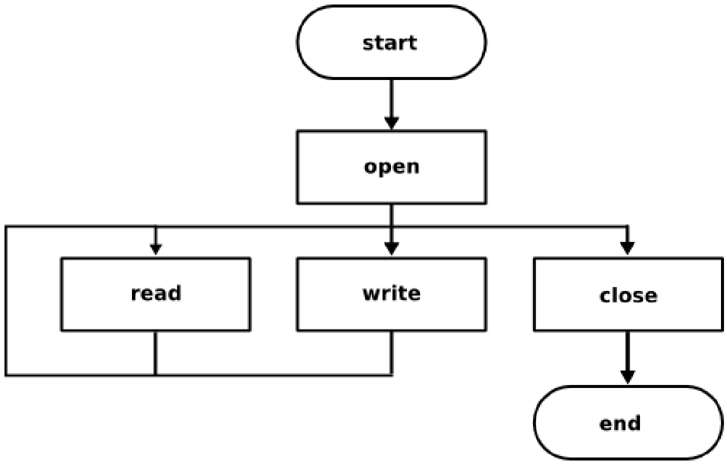


Figure 5-6 - RTEMS driver usage description

5.12.4.3. RTEMS application example

To use the system flash driver in the RTEMS environment, the following incomplete code structure is suggested (see Board Support Package for a full example):

```
#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` functions for accessing 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.

`CONFIGURE_APPLICATION_NEEDS_SYSTEM_FLASH_DRIVER` must be defined for using the driver. This will automatically initialise the driver at boot up.

5.12.5. 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.

5.12.6. 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 [RD1]) and given the topology illustrated in Figure 6-1, the routing addressing can be easily calculated.

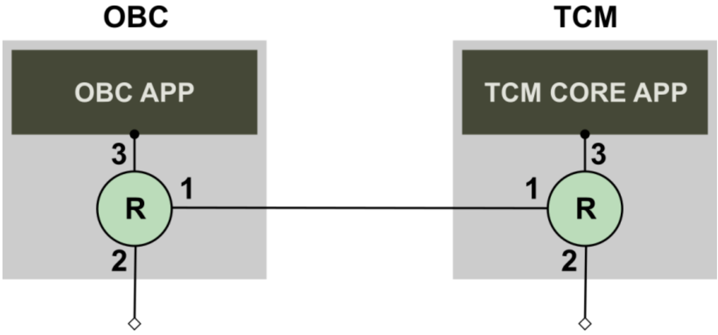


Figure 6-1 Integrated router location

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. Sirius TCM

7.1. Description

The Sirius TCM handles receiving Telecommands (TCs) and sending Telemetry (TM) as well as Spacewire communication using the RMAP protocol.

TC, received from ground, can be of two command types: a pulse command or a Telecommand. A pulse command is decoded directly in the hardware and the hardware then sets an output pin according to the pulse command parameters. All other commands are handled by the Sirius TCM software. Any command not addressing the Sirius TCM will be routed to other nodes on the SpaceWire network according to the current Sirius TCM configuration.

TM is received from other nodes on the SpaceWire network. The Sirius TCM supports both live TM transmissions directly to ground as well as storage of TM to the Mass Memory for later retrieval or download to ground during ground passes.

The Sirius TCM is highly configurable to be adaptable to different customer needs and missions and currently supports SpaceWire (SpW) using the Read Memory Access Protocol (RMAP), UART interfaces, pulse commands as well as Telecommand and Telemetry using CCSDS frame encodings and ECSS PUS packets.

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 250 (giving a bitrate of 100kb/s).
- All available interrupts from the CCSDS IP are enabled.
- Generation of OCF/CLCW in TM Transfer frames is enabled.
- TM is enabled.

The default configuration of the TC uplink is:

- Derandomization of telecommands is disabled.
- Telecommands must include a segment header, see 4.1.3.2.2 in [RD8]

7.2. Block diagram

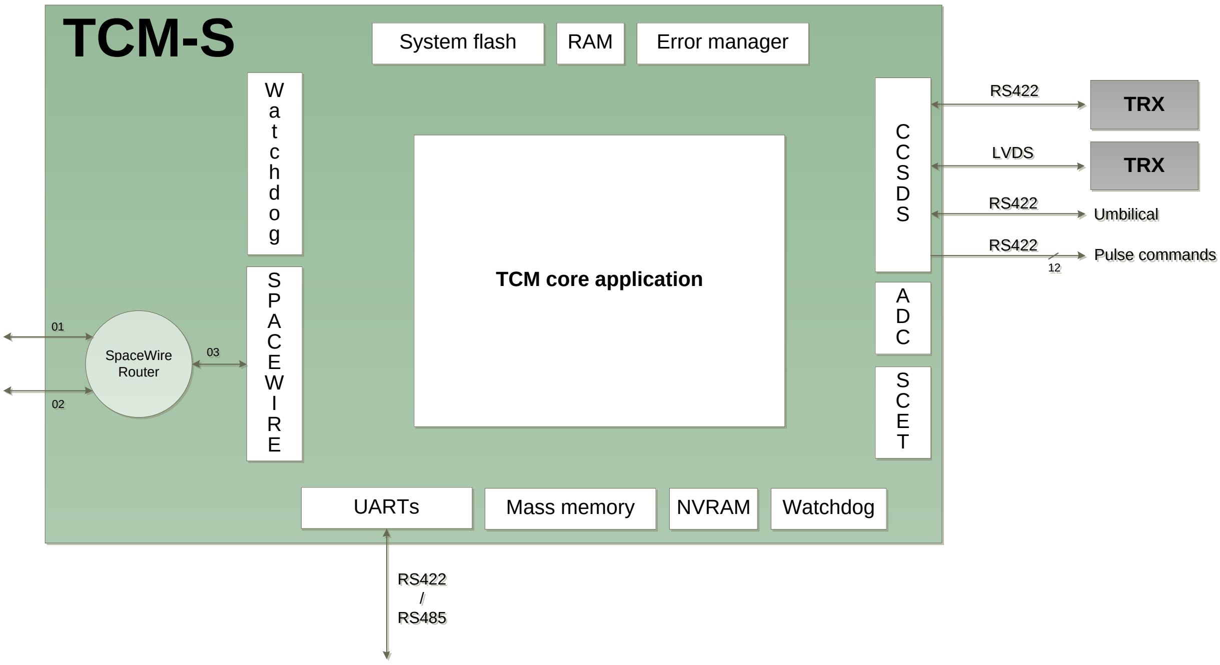


Figure 7-1 – Sirius TCM functionality layout with the external ports depicted

7.3. TCM application overview

The TCM application is partitioned into several software modules; each module handles a specific functional part. An overview of the software architecture of the TCM is presented in Figure 7-2. A main design driver of the TCM software architecture is the ability to pass along data between the different handlers without copying, since that would quickly decrease the performance and throughput of the system. To help with the no-copy policy, each peripheral handling larger amounts of data has DMA functionality, off-loading the CPU from mere data shuffling tasks while at the same time increasing performance by at least an order of magnitude. Data coming in on the SpaceWire interface intended for the mass memory will thus be stored in RAM only once - in the handoff between the SpaceWire and mass memory handlers.

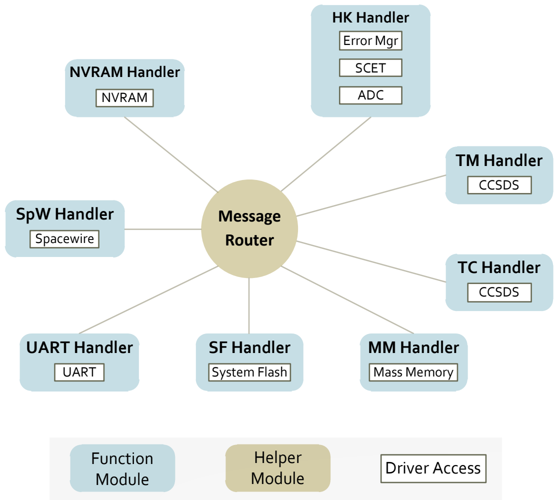


Figure 7-2 TCM software application overview

7.4. Configuration

The TCM can be configured for specific missions by parameters in NVRAM described in chapter 7.4.1. The parameters from NVRAM are read during initialization of the TCM application. Chapter 7.4.2 describes how to write an example configuration to the NVRAM of an actual unit. If reading from NVRAM fails during initialization, a set of fallback parameters are used instead. The fallback parameters are described in chapter 7.4.3.

7.4.1. Configuration parameters

The description and format of the different configuration parameters are detailed in the following tables.

Partition configuration of mass memory is specified in Table 7-1 below.

Table 7-1 PARTITION_CFG

Data	Type	Description
0	UINT32	Starting block number of the partition.
4	UINT32	Ending block number of the partition (exclusive).
8	UINT8	Partition mode. 0 – Direct 1 – Continuous 2 – Circular 3 – Auto-padded Continuous 4 – Auto-padded Circular
9	UINT8	Specifies type of data stored on the partition. 0 – Packets 1 – Raw Data (not supported for download) 2 – TC Storage
10	UINT8	Specifies which virtual channel to be used for downloading of the data in the partition. See [RD18] for VC allocation.
11	UINT8	Segment size for the partition. 1 - 16 kbyte 2 - 32 kbyte 3 - N/A 4 - 64 kbyte
12	UINT32	The data source identifier for the partition. Can be used to set a custom identifier of a data producer to a partition. Setting of this value is not required to successfully configure a partition.

Data from different sources can be routed to the SpW-network. Routing info is set by format specified in Table 7-2

Table 7-2 UART_ROUTING

Data	Type	Description
uart	UINT8	Source of message 0 - UART0 1 - UART1 2 - UART2 3 - UART3 4 - UART4 5 - PSU Ctrl 6 - Safe Bus
address	UINT32	The RMAP-address UART info is routed to
ext address	UINT8	The extended RMAP-address UART info is routed to.
Path	UINT16	The index of the SpW-path for the routing. See Table 7-5.
Backup SpW path	UINT16	The index of the backup SpW-path for UART config. See Table 7-5.
Backup SpW reply path	UINT16	The index of the SpW write reply path for UART config. See Table 7-6.

Configuration of UART-devices is done by Table 7-3 below.

Table 7-3 UART_CONFIG

Data	Type	Description
uart	UINT8	The UART device. 0 - UART0 1 - UART1 2 - UART2 3 - UART3 4 - UART4 5 - PSU Ctrl 6 - Safe Bus
Bitrate	UINT8	UART bitrate: 11 = 375000 baud 10 = 347200 baud 9 = 153600 baud 8 = 115200 baud (default) 7 = 75600 baud 6 = 57600 baud 5 = 38400 baud 4 = 19200 baud 3 = 9600 baud 2 = 4800 baud 1 = 2400 baud 0 = 1200 baud
Mode	UINT8	UART mode: 0 = RS422 mode 1 = RS485 mode 2 = Loopback
UART extended configuration	UINT8	Configuration of UART parity and PUS access block, see Table 7-4 for details

Table 7-4 describes the detailed bit layout of the UART extended configuration.

Table 7-4 - UART extended configuration

Data	Bits	Description
Parity	0-1	0 - no parity 1 - odd parity 2 - even parity
Reserved	2-6	reserved
PUS access blocked	7	0 - no (allowed) 1 - yes (blocked, PUS services cannot access UART)

Paths on SpW-network are specified by table Table 7-5 below. This table can fit 20 different SpW paths, each path can fit 8 bytes.

Note! All SpW paths must contain a terminating null character.

Table 7-5 NVRAM SpW path storage

Data	Type	Description
Path 0	Array of UINT8	A path on SpW network including the logic address of the receiving node.
Path 1	Array of UINT8	A path on SpW network including the logic address of the receiving node.
Path N	Array of UINT8	A path on SpW network including the logic address of the receiving node.

Backup reply paths on the SpW-network are specified by Table 7-6 below. When the TCM SW is requesting a write-reply from an external SpW node, the TCM SW must provide the path for the write reply. Since it is not possible to determine the reply path from the corresponding backup path, the user must also provide one write-reply path for every defined SpW path. This table can fit 20 different paths, each path can fit 8 bytes.

Note! All SpW paths must contain a terminating null character.

Table 7-6 NVRAM SpW Backup Reply Paths

Data	Type	Description
Reply path 0	Array of UINT8	A write-reply path from an external SpW node to the TCM SW.
Reply path 1	Array of UINT8	A write-reply path from an external SpW node to the TCM SW.
Reply path N	Array of UINT8	A write-reply path from an external SpW node to the TCM SW.

Enabling and timeout of the backup SpW paths are specified by Table 7-7 below.
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.

Table 7-7 NVRAM Backup SpW Configuration Storage

Data	Type	Description
SpW backup routing config	UINT32	Bit 0:15 – Sets the timeout in milliseconds. Bit 16 – Enable SpW backup routing. 1 = ENABLE 0 = DISABLE Bit 17:31 - Reserved

RIRP can be enabled/disabled as specified in Table 7-8 below

Table 7-8 RIRP Config

Data	Type	Description
RIRP Config	UINT32	Enabling/Disabling of RIRP 0 = DISABLE 1 = ENABLE

Telecommands can be routed to nodes on the SpW by APID as specified in Table 7-9 and Table 7-10 below.

Table 7-9 NVRAM APID Routing

Byte	Type	Description
0-1	UINT16	APID or lower APID in APID range Bit15 0 = Single APID Routing, 1 = APID range Bit14:13 Routing destination type Bit12:11 Not used Bit10:0 APID
2-3	UINT16	Upper APID in APID range Bit15 0 = Single APID Routing, 1 = APID range Bit14:13 Routing destination type Bit12:11 Not used Bit10:0 APID
4-5	UINT16	The index of the primary SpW-path of the APID. See Table 7-5.
6-7	UINT16	The index of the backup SpW-path of the APID. See Table 7-7.
8-9	UINT16	The index of the SpW write reply path of the APID. See Table 7-6.
10 – 11	-	Reserved for future use

Table 7-10 Routing destination type - mapping

Bit 14:13	Bit 14	Bit 13	Description
0	0	0	Routing via SPW
1	0	1	Reserved
2	1	0	TCM APID
3	1	1	Routing to TC queue

NOTE: If the *Routing destination path* is set to *TCM APID*. Then bit 15 should be set to 0, *Single APID Routing.* This is because the TCM will only handle the APID that it is assigned to from NVRAM configuration and that is a single APID.

Configuration of the TM path is described in Table 7-11 below. **NOTE:** Disabling the RS Encoder will not make the TM frame shorter, the parity bits will still be present in the frame but set to 0 (zero):

Table 7-11 TM_CONFIG

Data	Type	Description
TM Clk divisor	UINT16	The resulting TM bitrate is determined as described in 7.16.7.9.
TM Config	UINT16	Configuration of TM path. Bit6: 0 – Disable RS Encoder, 1 – Enable RS Encoder Bit5: 0 – Disable Conv. Encoder, 1 - Enable Conv. Encoder Bit4: 0 – Disable Randomizer, 1 – Enable Randomizer Bit3: 0 – Disable Idle Frames, 1 – Enable Idle Frames Bit2: 0 – Disable MCFC, 1 – Enable MCFC Bit1: 0 – Disable FECF, 1 – Enable FECF Bit0: 0 – Disable CLCW, 1 – Enable CLCW

Configuration of the TC path is described in Table 7-12 below:

Table 7-12 TC_CONFIG

Data	Type	Description
TC Config	UINT32	Configuration of TC path. Bit0: 0 – Disable Derandomizer, 1 – Enable Derandomizer

Configuration of the TC handler APID described in Table 7-13 below:

Table 7-13 TC_HANDLER_APID

Data	Type	Description
TC Handler APID	UINT32	APID configuration of APID of TC Handler in TCM Core Application

The virtual channel for the TCM to receive telecommands on is configured in NVRAM according to the format given in Table 7-14.

Table 7-14 TC_VC_CONFIG

Data	Type	Description
Telecommand Virtual Channel	UINT32	VC number 0 – 63.

Base configuration of the GPIO pins that can be controlled through the TCM RMAP interface is described in Table 7-15.

Table 7-15 GPIO Configuration

Data	Type	Description
GPIO Configuration	UINT8	Bit 0 – GPIO Value, 0 = Low, 1 = High Bit 1 – GPIO Mode, 0 = Normal (Single Ended), 1 = Differential Bit 2 – GPIO Direction, 0 = Output, 1 = Input Bit 3:7 - Reserved

Note! The TCM SW is limited to only use as many GPIO pins as are configured in NVRAM. Due to the possibility of using 2 GPIOs together for differential mode the amount of configured GPIOs must be an even number, otherwise the behaviour of the TCM SW is undefined.

7.4.2. Creating and writing a new configuration

A modified configuration can be created and written to the NVRAM using the nv_config utility from the TCM BSP.

The recommended way to create a new configuration is:

- Create a copy of the example configuration at `src/nv_config/src/configs/example.h` with a different name located it in the same directory.
- Modify the new file to match the desired configuration. The original example file and the definitions file at `src/nv_config/src/nvram_common.h` are useful references for the format and available parameters.
- Build the nv_config utility by executing the shell command

  ```
  make
  ```

  in the `src/nv_config/src/` directory. This will compile the nv_config utility for each configuration file, with each resulting RTEMS executable located at `src/nv_config/src/nv_config_<config name>.exe`, where `<config name>` is the name of the source configuration file, for example `src/nv_config/src/nv_config_example.exe`.

- Load and run the resulting binary RTEMS application using the debugger unit and GDB. Success is indicated via the output:

```
**************** NVRAM programming finished ***************

**************** System can be power cycled ***************
```

7.4.3. Fallback NVRAM parameters

If reading from NVRAM fails during initialisation of TCM Core Application, a set of fallback-parameters described in the tables below will be used.

Table 7-16 Fallback Partition Configuration

Partition #	Start Block	End Block	Partition Mode	Data Type	Virtual Channel	Segment Size	Data Source
0	0	100	0 (Direct)	0 (Packet)	1	32 kbyte	0
1	101	5000	1 (Cont.)	0 (Packet)	1	32 kbyte	0
2	5001	7500	2 (Circ.)	0 (Packet)	1	32 kbyte	0

Table 7-17 Fallback UART Routing

Uart #	Adress	Extended Adress	SpW Path Index	SpW Backup Path Index	SpW Backup Reply Path Index
0 (UART0)	0x00000000	0xFF	0	0	0
1 (UART1)	0x01000001	0xFF	0	0	0
2 (UART2)	0x02000002	0xFF	0	0	0
3 (UART3)	0x03000003	0xFF	0	0	0
4 (UART4)	0x04000004	0xFF	0	0	0
5 (UART 6)	0x05000005	0xFF	0	0	0
6 (UART 7)	0x06000006	0xFF	0	0	0

Table 7-18 Fallback UART Config

Uart #	Bitrate	Mode	Extended configuration
0 (UART0)	8 (115200 baud)	0 (RS422)	0 (no parity, pus access unblocked)
1 (UART1)	8 (115200 baud)	0 (RS422)	0 (no parity, pus access unblocked)
2 (UART2)	8 (115200 baud)	0 (RS422)	0 (no parity, pus access unblocked)
3 (UART3)	8 (115200 baud)	0 (RS422)	0 (no parity, pus access unblocked)
4 (UART4)	8 (115200 baud)	0 (RS422)	0 (no parity, pus access unblocked)
5 (PSU Ctrl)	8 (115200 baud)	1 (RS485)	0 (no parity, pus access unblocked)
6 (Safe Bus)	8 (115200 baud)	1 (RS485)	0 (no parity, pus access unblocked)

Table 7-19 Fallback SpW Paths

Path #	Path data
0	{0x01, 0x03, 0xFE, '\0'}
1	{0x01, 0x01, 0x03, 0xFE, '\0'}
2	{0x01, 0x02, 0x03, 0xFE, '\0'}
3	{0x02, 0x03, 0xFE, '\0'}
4	{0x02, 0x02, 0x03, 0xFE, '\0'}
5	{0x02, 0x01, 0x03, 0xFE, '\0'}

Table 7-20 Fallback SpW backup reply paths

Reply Path #	Reply Path data
0	{0x01, 0x03, 0x42, '\0'}
1	{0x01, 0x01, 0x03, 0x42, '\0'}
2	{0x02, 0x03, 0x42, '\0'}

Table 7-21 Fallback SpW backup routing configuration

Parameter	Description
SpW backup routing config	Fallback configuration of SpW backup routing: • SpW backup routing is disabled • The RMAP write-reply timeout is 100 ms

Table 7-22 Fallback RIRP Config

Parameter	Description
RIRP Config	Fallback configuration of RIRP • RIRP is disabled

Table 7-23 Fallback APID Routing

APID Routing #	Lower APID	Upper APID	SpW Path Index	SpW backupPath Index	SpW write-reply path Index
0	0xC00A	0xC150	Internal APID for TCM	Internal APID for TCM	Internal APID for TCM
1	0x8151	0x8300	0	3	0
2	0x8301	0x8450	1	4	1
3	0x8451	0x8600	2	5	2

Table 7-24 Fallback TM Configuration

Parameter	Value	Description
TM Clk divisor	250	The resulting TM bitrate will be 100 kbit/s (since convolutional encoding is disabled)
TM Config	0x4F	Configuration of TM path: • RS Encoder Enabled • Conv. Encoder Disabled • Randomizer Disabled • Idle Frames Enabled • MCFC Enabled • FECF Enabled • CLCW Enabled

Table 7-25 Fallback TC Configuration

Parameter	Value	Description
TC Config	0x02	Configuration of TC path. • Derandomizer Disabled

Table 7-26 Fallback TC Handler APID

Parameter	Value	Description
TC Handler APID	0xAF	Fallback APID configuration of APID of TC Handler in TCM Core Application

Table 7-27 Fallback TC VC Configuration

Parameter	Value	Description
Telecommand Virtual Channel	0x00000000	Fallback Telecommand Virtual Channel

Table 7-28: Fallback GPIO configuration

Parameter	Value	Description

GPIO 0 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 1 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 2 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 3 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 4 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 5 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 6 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 7 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 8 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 9 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 10 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0
GPIO 11 Configuration	0x04	Direction – Input Mode – Normal, single ended Value - 0

7.5. Telemetry

Telemetry is simultaneously sent on all the transceiver interfaces, i.e. the RS422 (TRX1), the LVDS (TRX2) and umbilical (UMBI) interfaces. See [RD18] for the VC allocation. The CCSDS IP generates complete TM Transfer Frames from PUS packets. If a PUS packet does not fit in one TM Transfer Frame, the CCSDS module splits the packet into several TM Transfer Frames. If a PUS packet does not fill the whole TM Transfer Frame, an idle packet is added as padding to fill the frame. The following telemetry settings are configurable by RMAP-commands (see 7.16):

- Divisor of TM Clock
- Inclusion of CLCW of TM Transfer Frames
- Inclusion of Frame Error Control Field of TM Transfer Frames
- Updating of Master Channel Frame Counter
- Idle frame generation
- Convolutional encoding
- Pseudo randomization

The TCM supports the format of TM Transfer Frames described in [RD7].

7.6. Telecommands

7.6.1. Description

Telecommands can be received on the RS422 (TRX1), the LVDS (TRX2) or the umbilical (UMBI) interface.

The TCM actively searches for Command Link Transmission Units (CLTU), i.e. telecommands, on all three inputs simultaneously (as long as they are enabled). When a telecommand start sequence is detected, the other inputs are ignored during telecommand reception. The search will restart once the entire telecommand is either received or a reception error is detected. In short, the telecommand reception uses the following reception logic, also illustrated in Figure 7-3:

- All incoming signals on the inputs are synchronized to the system clock domain.
- The BCH decoder is configured in error-correcting mode.
- When the CLTU receptor has detected and decoded a start pattern, it sets an enable signal for the active path, indicating that this CLTU receptor is now active. If one or more bit-errors occur in the start pattern, the CLTU will be rejected, which is not compliant to handling of one-bit errors described in [RD13] since one-bit errors shall be corrected when BCH decoder is configured in error-correcting mode.
- The telecommand path activated is set until the reception status changes, i.e. the current telecommand is finished and a new start pattern is detected correctly on a different CLTU path.
- The selected telecommand clock, data and enable signals are now forwarded through the mux to the BCH decoder, rejecting data and clock on inactive data paths.
- When BCH has decoded the tail in the CLTU, all CLTU receptors are set in search mode again, scanning for the start pattern ready to receive a new telecommand. If one or more bit-errors occur in the tail sequence of the CLTU, the CLTU will be rejected.
- The BCH interface does not “see” the data/clock until the start pattern is decoded correctly and the enable signal is set.

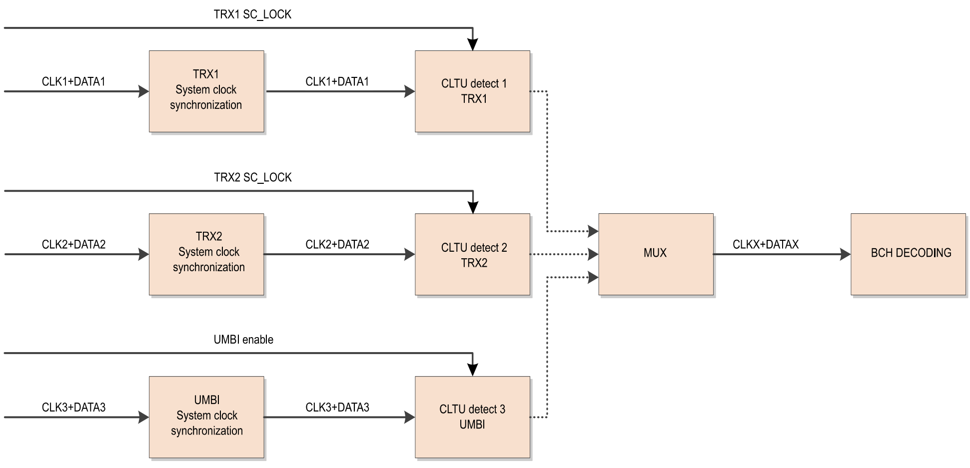


Figure 7-3 – Telecommand Input Multiplexer

Derandomization of TC can be enabled/disabled by RMAP command (see 7.16). Telecommands sent to the TCM must include a segment header, see 4.1.3.2.2 in [RD8]

The TCM supports the format of TC Transfer Frames described in [RD8].

7.6.2. Pulse commands

The CCSDS IP in the TCM has a built-in Command Pulse Distribution Unit (CPDU) execution functionality with the possibility to execute up to twelve CPDUs without interaction from software. A pulse command is decoded directly in hardware, and it sets an output pin according to the pulse command parameters. The CPDU_DURATION_UNIT is defined to 12.5 ms and the output is hence a multiple of this signal length.

The CPDU function can for example be used to reset modules in a spacecraft and choose which software image to boot, an updated version or the safe image. The last executed pulse command can be read from the telecommand status data field.

For details about the format of pulse commands, see 7.14.3

7.6.3. COP-1

The CCSDS COP-1 functionality on the spacecraft is implemented mainly in software where the command link control word (CLCW) is generated based on telecommand status. The CLCW is inserted when the OCF_CLCW flag is set in the control register, otherwise user data will be inserted instead. It will insert four bytes, and the CLCW is also included in the CRC calculation for the master frame on both idle and data frames. The NO RF AVAILABLE flag and NO BIT LOCK flag are set from external pins and will overwrite the respective bits in the CLCW word which hence cannot be controlled by software. The flag NO RF AVAILABLE is set by signal Carrier lock in and the flag NO BIT LOCK is set by signal Sub-carrier lock in.

7.7. Time Management

7.7.1. Description

The TCM has an internal SCET timer that can be synchronised to an external time source. For synchronisation to occur, a stable PPS input must first be provided for at least 7 seconds, after which the PPS will be considered “qualified” and the TCM will automatically sync SCET subseconds to the external PPS arrival time. A received SCETTime write command can then synchronise the seconds value, see 7.16.7.20.

If the PPS is not stable, the TCM will abort synchronisation to the external source and will attempt to re-qualify the PPS. When the PPS is not qualified, neither subseconds nor seconds synchronisation will occur.

The current criteria for stability are set to be extremely generous, and only after a PPS interval of 2 seconds or more will the PPS be considered unstable by the TCM.

7.7.2. TM time stamps

A timestamp can be generated when a TM Transfer Frame is sent on VC0. The rate of timestamp generation is configurable through an RMAP command, and the latest timestamp is readable on the same interface. See 7.16.7.11 and 7.16.7.12 for further info.

7.8. Error Management and System Supervision

The Error Manager in the TCM provides information about different errors and operational status of the system such as:

- EDAC single error count
- EDAC multiple error count
- Watchdog trips
- CPU Parity errors.

Error Manager related information and housekeeping data is available by RMAP. See 7.16.7.19

The status of the TM Downlink and TC Uplink are available through RMAP. See 7.16.7.14 and 7.16.7.1

A watchdog is enabled in the TCM that must be kicked by the TCM Application or a reset will occur. The watchdog is kicked only when all active tasks in the TCM report that they are alive.

7.9. Mass Memory Handling

7.9.1. Description

The mass memory in the TCM is primarily intended for storage of telemetry data while awaiting transfer to ground but can also be used for internal data storage. The mass memory is configurable as described in chapter 7.4

Table 7-29 Mass memory page and block size

Mass Memory size	Page size [byte]	Block size [byte]	Pages per block
16 GB	16 * 1024	2 * 1024 *1024	128
32 GB	32 * 1024	4 * 1024 * 1024	128

The mass memory is accessed through the MM* RMAP commands described in chapter 7.16.7. The mass memory is nandflash-based and that also slightly colours its user interface, even though the detailed handling has been abstracted away. The total amount of mass memory available is 16 or 32 GB, depending on hardware and SW configuration. As shown in Table 7-29 the page size is 16kB and the block size is 2MB for 16 GB of Mass Memory. For a mass memory of 32 GB, the page size is 32kB and the block size is 4MB. The number of pages per block is independent of mass memory size.

Due to the flash nature of the mass memory, each new block will require erasing before accepting writes, but the TCM software will handle this automatically. For each 32-bit word stored in mass memory, there are 8 bits stored as EDAC to be able to detect double errors and correct single errors. During erases or writes the operation may fail, and the software will then mark this block as bad and skip this in all future transactions. The bad block list is stored in NVRAM and will thus survive a reboot and/or power cycling. This graceful degradation behaviour of the mass memory implies that partitions may shrink in size and this phenomenon needs to be considered when planning partition sizes. Another effect of the bad blocks is that available space on a partition may decrease by more than the actual data written and this might need tracking by the user.

To simplify divisions between different types of data with different configurations, the mass memory is divided into logical partitions where each partition is configured by its mode, type, segment size and TM virtual channel for downloading. All partitions have an address space of 4 Gbytes regardless of their physical size and this is also the maximum size of a partition.

Reading and writing to partitions behaves slightly different between different types of partitions, but when a partition is full, it requires a *free* operation to allow for further writes. New space for writing will only become available once a block is completely freed (that is, when a free operation passes over a block boundary).

Figure 7-4 illustrates this with an example two-block partition, showing in the last picture that new data cannot be written until free has reached the block boundary. To simplify operations for the user, free operations can be requested on more data than is available in the mass memory, see 7.16.7.31 for details.

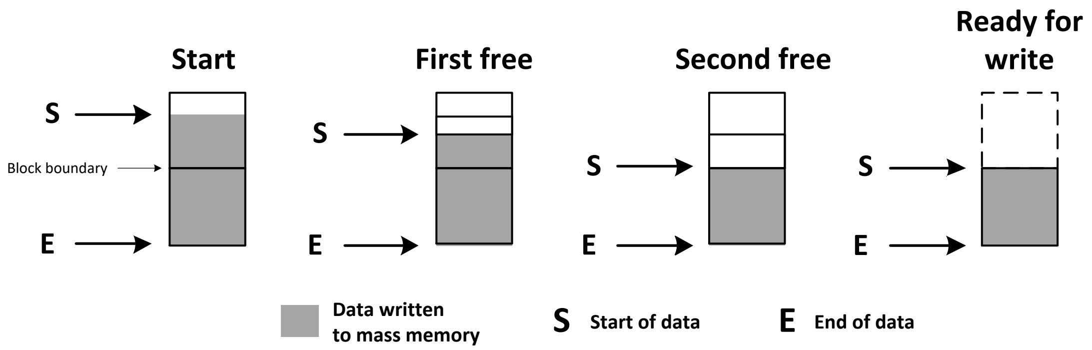


Figure 7-4 Illustration of free behaviour and block boundaries.

7.9.2. Partition configuration

Partitions are configured via the NVRAM configuration tool, according to the format in 7.16.7.28, below follows some detailed information regarding certain configuration items.

7.9.2.1. Partition mode

Each partition can be configured as Continuous, Circular or Direct mode.

In **Continuous mode**, all write accesses are sequential and can be of any size but will return with an error when the partition is full. The MM handler internally implements free and write pointers to keep track of the data in the partition. The write pointer is used as the address for storing the data and is updated after each successful write. The free pointer is used as the address when freeing data and is updated after each successful free. Read access and download of data is available on any arbitrary address within the partition (between the free and write pointer addresses). Obsoleted data need to be freed to enable further writes when the partition is full.

Continuous Auto-padded mode operates in the same way as Continuous mode, with additional automatic segment padding, see 7.9.2.4.

Circular mode operates much in the same way as Continuous mode except that writes will never fail when the partition is full. Instead, it will automatically free one or more blocks used for the oldest written data and update the free pointer accordingly. Thus, data never needs to be freed manually, but the operation is available.

Circular Auto-padded mode operates in the same way as Circular mode, with additional automatic segment padding, see 7.9.2.4.

For both Continuous and Circular mode (with or without automatic padding), an internal cache of one page is used to hold any data that does not fit a page. As soon as the cache is filled, the data is written to physical memory. Any restarts or power cycling will result in loss of any data only written into this cache. If loss of cache data is an issue, ensure that all writes end on a page boundary as this will make sure all data is always written to flash.

In **Direct mode**, a write access can be to any arbitrary address in the address space provided that writing starts at a block boundary and is continuously written within this block. Each access must also be a multiple of the page size and thus keeps no cache of data not stored in physical memory. To determine the actual page size in use, the current page size can be read out using the RMAP command MMGetPageSize described in section 7.16.7.34. Read access and download of data is available from any arbitrary address within a partition, given that it has valid data (previously written). Obsoleted data or data to overwrite need to be freed here as well but can be freed on any valid address in the address space.

Please Note: Due to considerably increased initialisation times when using direct partitions, it is recommended to only allocate a maximum of 200 blocks (400 Mbytes for 16 GB mass memory or 800 Mbytes for 32 GB mass memory) in total to direct partitions. Increasing the amount of direct partition blocks significantly above this limit will cause initialisation failure due to the watchdog timeout being triggered.

The direct partition mode does not utilise free and write pointers.

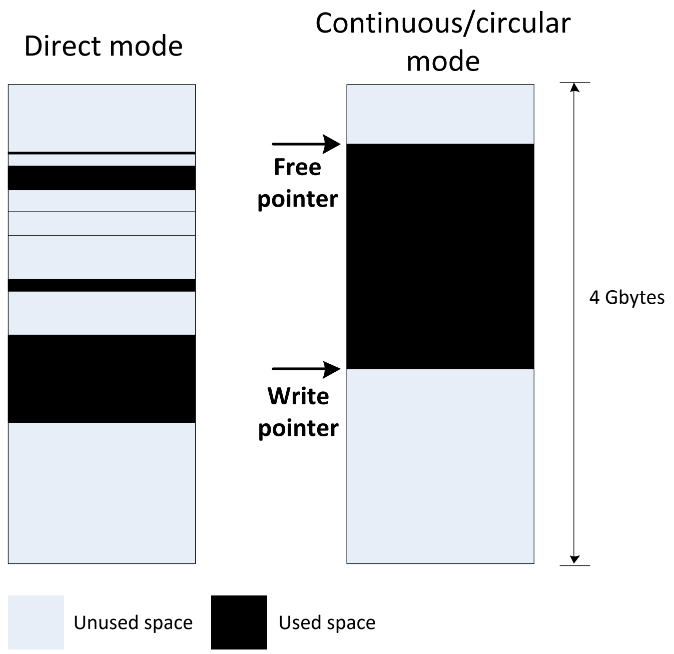


Figure 7-5 Illustration of partition modes and the free/write pointers

7.9.2.2. Partition segment size

The segment size is only applicable for downloading and for partitions of type PUS (see below). The mass memory supports segment sizes of 16, 32, and 64 kbyte.

7.9.2.3. Partition type

Partitions can be of three types, PUS (see[RD3]), raw and TC storage.

Partitions of **type PUS** require that each segment will begin with a PUS packet and unless auto-padding is used, it is up to the software writing into the mass memory to maintain this segmentation. There are no limitations on the number of PUS packets that can be contained in one segment, but if the written data doesn't fit exactly into the segment size it must be padded up to the segment boundary. Padding can be achieved either with a PUS idle packet (which also will be transferred to ground) or with a bit pattern of 0xF5, allowing padding of as little as one byte. During a download operation when the padding bit pattern is discovered, download will skip to the next segment (if available).

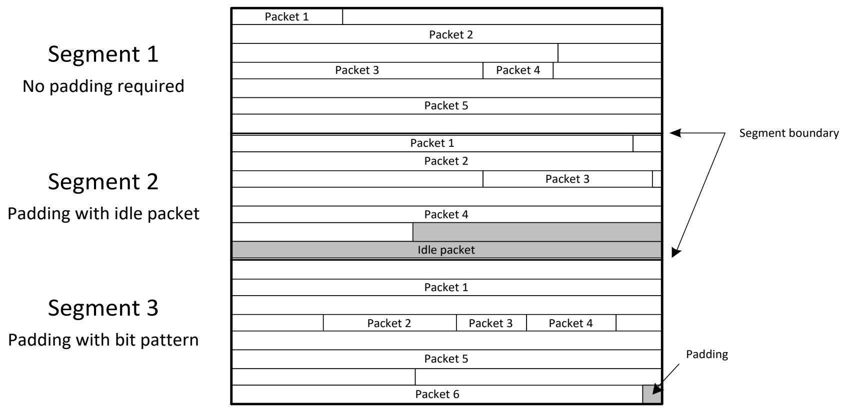


Figure 7-6 Illustration of packet placement inside segments with different padding (marked in grey)

Partitions of **type raw** can be used to store data on-board if that is needed for the mission (to be written/read by other units in the system), but only PUS formatted partitions can be downlinked to ground through the CCSDS block.

Configuring a partition with **type TC storage** dedicates this partition for use by the TC storage, see 7.10 for more info. A partition with this type must use the continuous partition mode (without automatic padding) and no more than one partition may be configured with this type the same time.

7.9.2.4. Automatic padding

Continuous and circular partitions can be configured with automatic padding of segments, which automatically pads data written to the partition with a 0xF5 bit pattern, such that written data never overlaps a segment boundary, and is fit for download.

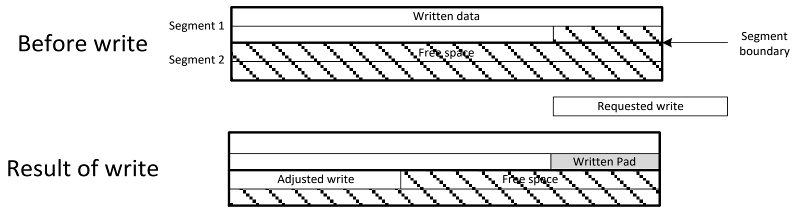


Figure 7-7 Illustration of auto-padding of a requested write.

No examination or validation of data contents are done in the padding process, and if a write command with data containing multiple packets is received, it will be padded as if it was a single large packet.

Auto-padding will never split the data in a received write command, and thus writing with data that is larger than a segment is not supported.

If writing packets to an auto-padded partition, each write should contain data that starts at the beginning of a packet and ends at the end of a packet, in order to ensure that it is possible to download the data correctly.

Reads from an auto-padded partition will return padding and data as it was written to the partition in the auto-padding procedure.

Downloads from an auto-padded partition should consider the additional padding size for written data when calculating the download size. The free and write pointers can be used to determine the total current size of all written data including padding.

Automatic padding limits single writes to the segment size, therefore using segments of 16kB together with pages of 32kB makes it impossible to write a full page immediately to the physical mass memory. In this case some of the data to be written will always be kept in the SDRAM cache. To be able to write a full page of 32kB immediately to the physical mass memory when using automatic padding, the segment size must be equal to or larger than the page size (>=32kB).

7.9.2.5. Partition virtual channel

This specifies which CCSDS virtual channel to be used for downloading of the data in the partition. See [RD18] for the supported channels.

7.9.3. Recovery

The mass memory handler utilises the NVRAM to store on-going operation data, which is used in the initialisation step in order to recover consistency after aborted write or free operations, caused for example by a power failure reset.

If errors or inconsistencies are detected when the stored on-going parameters are read from NVRAM at initialisation, the recovery associated with the unavailable item will be skipped and the initialisation will continue.

The initialisation recovery is aggressive and will prioritise a usable system over data retention; any single block which exhibits metadata inconsistencies that make it impossible to safely add it to the translation table will be erased and considered free.

For continuous and circular partitions, further recovery is performed to ensure that the partition data range is continuous (which is required for the partition to be usable). If a discontinuity is discovered, the recovery process will erase data blocks from the highest logical partition address and downwards, until a continuous range of data is left on the partition. Such discontinuities can for example occur due to corrupt blocks, or if a partition is configured to include blocks with unknown contents (e.g. changing a direct partition into a continuous partition).

Recovery does not take into account the format of the stored data and may for example leave a partition with data that no longer fulfils segmentation requirements for download.

Recovery may cause the free and write pointers of continuous/circular partitions to move.

An empty continuous/circular partition, where the write pointer is located exactly at the start of a block, will have the free and write pointers reset to address 0 if a reset and subsequent re-initialisation occurs.

Recovery will cause rediscovery of previously freed data in a block in the following scenarios:

- If the block was not completely freed.
- If data was freed from the block in a continuous/circular partition and the free did not move past the block boundary.
- If data was freed from the block in a continuous/circular partition and the write pointer was located inside the block.

For continuous/circular partitions, this data rediscovery will only occur in the block where the free pointer was last located. For direct partitions, it will occur in every block which provides one of the scenarios listed above.

7.10. TC Storage

The TCM provides a ”TC storage“ which consists of a non-persistent storage that can be written to directly from the ground segment using PUS service request telecommands and can be read and cleared by the space segment via RMAP.

The TC storage functionality allows burst uploading of data while avoiding directly routing this data as telecommands over the SpaceWire network, which could be used to defer certain processing.

The format of the data stored in the TC storage is not defined by the TCM and instead needs to be coordinated between the ground segment writing into the storage and the space segment reading from the storage. This is especially important if the space segment needs to distinguish the boundary between individual data chunks in the TC storage.

If the existing error correction protections of the mass memory are insufficient for the current mission specification to guarantee that the space segment can always parse the data read from the TC storage, additional synchronization features could be added to the data format to allow discarding invalid data, for example a fixed data chunk size, a synchronization pattern, etc. Such features are the responsibility of the mission-specific ground and space segment.

A suggested use case for the TC storage is to provide deferred telecommand processing, where the data chunks written into the TC storage are in the form of telecommands, which can be read by the space segment when it is ready to process them. In this case the telecommands to be stored need to be embedded in the service data unit field of PUS service request telecommands – one (or more) telecommand(s) wrapped inside another telecommand.

The TC storage is non-persistent, and the stored data along with the status information will be cleared if the TCM is reset, despite using a mass memory partition for its storage. This clear is done in order to eliminate any potential data inconsistency which could occur due to write cache loss on reset or data recovery during initialization.

To enable the TC storage, a single mass memory partition must be configured with the TC storage partition type as described in 7.9.2.3.

The PUS service interface for writing into the TC storage from the ground segment is described in 7.15.2.

The space segment can use the MMData (read only), MMDataRange, MMPartitionConfig and MMPartitionSpace commands to read data from and information about the TC storage partition in the same way as from a standard continuous partition.

In addition, the space segment can use the MMTCStorageStatus command to read specific status information from the TC storage and can use the MMTCStorageClear command to clear the TC storage.

Writing to, freeing from, or downloading from a TC storage partition by the space segment is not supported.

See 7.16 for detailed descriptions of the space segment commands.

7.11. TC Queue

The TCM supports a functionality to route received TC packets to a queue, rather than consuming the TC packets directly, depending on the APID of the TC packet. It also defines an interface to read a packet from this queue, and to remove a previously read packet from the queue.

The TC queue stores individual TC packets and the queue size is 50 packets. The maximum size of TC packets to be stored is 1016 bytes, which represents the maximum size of a transfer frame data field without the segment header (see 12.6). The queue is implemented as a circular queue, meaning that the FIFO (first in, first out) principle applies, and upon a queue overflow, the oldest packet is overwritten.

When a TC packet is added to the TC queue, it is assigned a queue item ID. This is an incremental counter, which is returned as part of the metadata when reading the packet. It can be used to check e.g., whether a queue overflow has occurred.

When successfully adding a TC packet to the TC queue, the TCM will not send any Telecommand Acceptance Success Report. If it fails to add the TC packet to the TC queue (e.g. due to faulty packet CRC) it will send a Telecommand Acceptance Failure report (1,2) (see 7.14.1).

7.12. Spacewire Backup Routing

The TCM provides a “Spacewire Backup Routing” service, this is a service that will resend a command packet over an alternative Spacewire path if the first transmission of the command packet fails.

When SBR is enabled and a command message is redistributed to a SpW node by the TCM SW, the SpW node must send a reply to the TCM SW if the command message was received properly. If the TCM SW has not received a valid reply within the user-specified time period, the TCM SW will switch to using the backup SpW path and try to send the packet once again. After that TCM SW will send command messages using the original routing path to avoid switching paths due to temporal errors. If a write-reply arrives after the user specified time period, or no matching timer is found, the write-reply will be ignored.

When the TCM SW is requesting a write-reply from an external SpW node, the TCM SW must provide the path for the write reply. Since it is not possible to determine the reply path from the corresponding backup path, the user must also provide one write-reply path for every defined SpW path.

Enabling SpW backup routing, setting the primary and backup SpW paths, setting SpW write-reply paths, and configuration of the duration of the timeout can be done by configuring NVRAM. This is described in 7.4. These parameters can also be configured or read out by sending RMAP commands to the TCM, the RMAP commands are described in 7.16.7.35 - 7.16.7.46.These RMAP commands make it possible to configure these parameters during flight, since a debugger must be connected to the TCM when configuring NVRAM.

Altering the SpW routing configuration via RMAP commands does not trigger any write actions to NVRAM, the RMAP commands only alter local copies of the NVRAM parameters in the TCM SW. Upon reboot, all SpW routing configurations (enable/disable, paths and timeout) set by RMAP commands are lost. After reboot, the TCM SW will use the SpW routing configurations and paths set by the NVRAM configuration.

(1) $timeout[s] * packet\ rate\left[\frac{n}{s}\right] \leq \frac{\max buffers}{2}[n]$

The linear relationship in the equation above should be used as a rule of thumb when selecting write reply timeout to avoid running out of resources. A maximum of half the number of available buffers (internal buffers of the TCM SW, here used for holding the data contents of RMAP commands while waiting for a write reply), 64/2=32 buffers, should be allowed to be occupied waiting for timeouts or write replies. Different SpW-networks and different sizes of TC/uart packets require different minimum timeouts therefore care must be taken so that the timeout is set high enough for the packets to be sent properly.

It is allowed to update backup routing parameters via RMAP during ongoing SBR transactions, but updating the SpW reply paths, the enable/disable parameter or the timeout duration parameter will not affect already started transactions. For example if SBR is enabled and an RMAP command requesting a reply is sent to an external node, and SBR is disabled before the TCM SW has received a reply or the timer has timed out, then it is possible that the RMAP command will be resent on its backup path although SBR is disabled. If SBR is enabled and an RMAP command requesting a reply is sent to an external node, and the user updates the SpW routing paths before the TCM SW has received a reply or the timer has fired, the new updated paths will be used for the possible resend of that RMP command packet.

7.13. RIRP RMAP Interface

RIRP is an alternative interface for RMAP command access to the TCM.

Specific RMAP addresses for devices and sub-systems are allocated for RIRP-interface accesses to the TCM, see Table 7-41 for info about addresses.

With RIRP, the reply uses standard RMAP status codes as described in [RD2] and the specific execution status is not generally returned in the reply, but instead stored in a transaction status buffer to be read out separately.

The transaction status buffer is not used in the case of acceptance errors, successful reads, or reads from the transaction status buffer itself.

See 7.16.2 for more information.

7.14. ECSS standard services

The TCM supports a subset of the services described in [RD3]

7.14.1. PUS-1 Telecommand verification service

The TCM performs a verification of APID of the incoming TC. If the verification fails, the telecommand is rejected and a Telecommand Acceptance Failure - report (1,2) is generated as described in [RD3]. On successful verification, the command is routed to the receiving APID. The receiving APID performs further verification of packet length, checksum of packet, packet type, packet subtype and application data and generates reports accordingly [(1,1) or (1,2)]. If specified by the mission, the APID shall implement services for Telecommand Execution Started, Telecommand Execution Progress and Telecommand Execution Complete. Sending these reports can be enabled or disabled by setting the ACK flags of the TC accordingly (see Table 12-4).

Table 7-30 Telecommand Acceptance Report – Success (1,1) data

Packet ID	Packet Sequence Control
UINT16	UINT16

Table 7-31 Telecommand Acceptance Report – Failure (1,2) data

Packet ID	Packet Sequence Control	Code
UINT16	UINT16	UINT8. 0 – Illegal APID 1 – Invalid packet length 2 – Incorrect CRC 3 – Illegal packet type 4 – Illegal packet subtype 5 – Illegal application data 6 – Illegal PUS version

7.14.2. PUS-2 Distributing Register Load Command

By PUS service (2,2) it is possible to write data to devices on the TCM by a telecommand. One register load command per telecommand is supported.

Using this service if the PUS access to the UART is blocked (see Table 7-4) will result in a Telecommand execution completed report - failure.

Table 7-32 Distributing Register Load Command

Register Address	Register Data
0xFF04000100 – UART0 0xFF04000101 – UART1 0xFF04000102 – UART2 (5 octets)	Array of UINT8

7.14.3. PUS-2 Device Command Distribution Service

The TCM supports the command pulse distribution unit (CPDU) pulse commands in hardware as defined in 7.2.2 in [RD3]. The CPDU listens for a specific virtual channel – APID pair, see the configuration document [RD18].
The TCM has 12 controllable (0-11) output lines and can be toggled to supply different pulse lengths according to the following scheme:

Table 7-33 CPDU Command (2, 3)

Output Line ID	Duration
0-11 (1 octet)	0 – 7 (1 octet)

The duration is a multiple of the CPDU_DURATION_UNIT (D), defined to 12.5 ms, as detailed below.

Table 7-34 CPDU Duration

Duration in bits	Duration in time (ms)
000	1 x D = 12.5
001	2 x D = 25
010	4 x D = 50
011	8 x D = 100
100	16 x D = 200
101	32 x D = 400
110	64 x D = 800
111	128 x D = 1600

Note: The APIDs reserved for the CPDU are 1 – 9 for future use.

7.14.4. PUS-2 Distributing Device Command

By PUS service (2,128) it is possible to write a command to devices on the TCM by a telecommand. One device command per telecommand is supported. UART-devices have a fixed configuration of 8 data bits and 1 stop bit.

Using this service if the PUS access to the UART is blocked (see Table 7-4) will result in a Telecommand execution completed report - failure.

Table 7-35 Distributing Device Command

Device Address	Bitrate	Mode	Parity
0xFF04000100 – UART0 0xFF04000101 – UART1 0xFF04000102 – UART2 (5 octets)	11 = 375000 baud 10 = 347200 baud 9 = 153600 baud 8 = 115200 baud (default) 7 = 75600 baud 6 = 57600 baud 5 = 38400 baud 4 = 19200 baud 3 = 9600 baud 2 = 4800 baud 1 = 2400 baud 0 = 1200 baud (1 octet)	0 = RS422 mode (default) 1 = RS485 mode 2 = Loopback (1 octet)	0 = No parity (default) 1 = Odd parity 2 = Even parity (1 octet)

7.15. Custom services

7.15.1. PUS-130 Software upload

During the lifetime of a satellite, the on-board software might need adjustments as bugs are detected or the mission parameters adjusted. This service solves that by providing a means for updating the on-board software in orbit. See chapter 10 for further info.

7.15.2. PUS-131 TC Storage

7.15.2.1. Description

The TC storage service provides the capability to store data into the TC storage for later retrieval by the space segment.

The TC storage service does not provide any capability to read or clear the data stored into the TC storage, this responsibility is delegated completely to the space segment.

The TC storage service provides a storage area into which data chunks can be appended. The storage area is configured via the mass memory partition configuration in the NVRAM, using the special TC storage partition type.

When the TC storage partition becomes full, no more data can be appended into the storage area and attempted stores will be discarded and an execution failure report sent. The space segment is responsible for clearing the storage area for re-use.

The amount of data that can be stored in the TC storage before it becomes full depends on the number of blocks configured for the TC storage partition. The maximum number of data chunks that can be appended into the TC storage before clearing is $2^{32} - 1$, exceeding this limit is not supported (although it is highly unlikely based on the telecommand uplink speed).

The TC storage service maintains status information about the number of bytes used in the storage area, the amount of data chunks currently stored in the storage area and the amount of data chunks which has failed to be stored due to the storage area being full. This status information is only accessible by the space segment.

Requests to the TC storage must use the default TCM core application APID 175.

The TC storage service defines a single service request named “TC storage store data” with service type 131 and service subtype 0.

The service data unit associated with the TC storage store data service request is a single “deduced parameter” in the form of a “fixed-length octet string” which is deduced from the request telecommand packet length, see [RD3] for details. In other words, the payload is treated as raw data bytes and will not be parsed in any way.

When the TC storage service receives a TC storage store data service request it will attempt to append the data in the service data unit of this request into the storage area. The result of the append action will be provided via an execution failure or execution success report (see section 7.14.1).

7.15.2.2. Telecommand Verification Service

A dedicated telecommand verification service is provided in conjunction with the TC storage service.

The verification service defines two acceptance verification reports named “TC storage store data acceptance report - Success” and “TC storage store data acceptance report - Failure”.

The TC storage store data acceptance report – Success contains no custom tailoring compared to the definition in the PUS standard, see [RD3].

The TC storage store data acceptance report – Failure source data format is described in Table 7-36.

Table 7-36 TC Storage store data acceptance completed report - Failure source data

Telecommand Packet ID	Packet Sequence Control	Code
2 octets	2 octets	Enumerated, 1 octet (PFC=1).

The possible values for the code field of the TC storage data acceptance report – Failure is the standard code values defined in the PUS standard (0.5), see [RD3].

The verification service defines two execution verification reports named “TC storage store data execution completed report - Success” and “TC storage store data execution completed report - Failure”.

The TC storage store data execution completed report – Success contains no custom tailoring compared to the definition in the PUS standard, see [RD3].

The TC storage store data execution completed report – Failure source data format is described in Table 7-37.

Table 7-37 TC storage store data execution completed report – Failure source data

Telecommand Packet ID	Packet Sequence Control	Code
2 octets	2 octets	Enumerated, 1 octet (PFC=1).

The possible values for the code field of the TC storage data execution completed report – Failure is described in Table 7-38.

Table 7-38 TC storage store data execution completed report – Failure source data code field details

Value	Description
1	Unable to append due to data store being full.
2	No TC storage partition is configured.

7.16. Spacewire RMAP

A general description of how the Spacewire RMAP is used by the TCM is given in section 7.16.1. Section 7.16.2 describes the alternative RIRP interface. Section 7.16.3 deals with the incoming RMAP commands that the TCM application supports. Any RMAP commands issued by the TCM are described in section 7.16.4. Section 7.16.5 deals with the status codes returned with the replies to incoming commands, and 7.16.6 explains the use of transaction ID's to keep track of where replies shall be sent. Finally, sections 7.16.7 and 7.16.8 provide further details about the incoming and outgoing RMAP commands.

7.16.1. Description

According to [RD2], a 40-bits address consisting of an 8-bit Extended Address field and a 32-bit Address field is used in RMAP. This has been utilized in the TCM according to Table 7-41 to separate between configuration commands and mass memory storage of data (partition handling).

The initiator logic address of output messages from the TCM, and the RMAP key that needs to be used for input messages and should be expected from output messages, are shown in Table 7-39.

Table 7-39 RMAP predefined fields

Field	Value
Initiator Logical Address	0x42
Key	0x30

In addition, target address and reply address must be added to the RMAP header in commands targeting the Sirius TCM to compensate for topology external to the Sirius TCM and the embedded SpaceWire router. As can be seen in Figure 7-1, if the Sirius TCM were to be addressed from SpaceWire port 1, the example addresses below must be added to the routing addresses in the RMAP header.

Table 7-40 RMAP predefined fields for routing

Field	Value
Target Spw Address	0x01, 0x03
Reply Address	0x01, 0x03

Please note that the size requested in RMAP read commands will be ignored and the returned data by the reply will be of a fixed size determined by the TCM. Except for the commands MMData, and RIRPTransactionStatus, where the size requested will be used. Refer to the respective subsection of 7.16.7 for details about the size returned by the individual commands.

In the RMAP header Instruction field there is a Verify-Data-Before-Write bit. In the TCM that is used as follows:

- If an RMAP command is received by the TCM SW with Verify-Data-Before-Write set, the data integrity is verified before the command is processed, according to the RMAP standard.

- If an RMAP command is received by the TCM SW with the Verify-Data-Before-Write bit not set, the data payload integrity is not verified before nor after the command is processed. This is a deliberate deviation from the RMAP standard to allow high throughput writing of data where immediate indications of any data corruption is not critical, and/or will be provided via other means.

7.16.2. RIRP Interface

Specific addresses have been allocated to be used for the RIRP interface as described in Table 7-41.

The command for reading from the transaction status buffer is described in 7.16.7.46.

Limitation:

Using both the standard RMAP interface and the RIRP interface in parallel is not supported. The desired interface must be configured in NVRAM (see Table 7-8) and only the addresses corresponding to the configured interface may be used.

7.16.2.1. Command Acceptance

If an invalid command is received by the TCM it is discarded without sending a reply.

If a command is received by the TCM which contains an invalid extended address, a reply is sent with a status set to 1 (General error code). In this case the command is not stored in the transaction status buffer.

When using RIRP and the RIRP transaction status buffer is full, all incoming RMAP commands will be rejected. When a RIRP command is rejected for this reason, a reply with status 1 (General Error) will be sent to the initiator. Reading from the RIRP transaction buffer must be performed before the TCM-SW can handle new RMAP commands.

Up to 200 transactions can be stored in the transaction status buffer.

7.16.2.2. Write Commands

If a RIRP write command is received and accepted by the TCM, a reply will be sent directly with a status indicating success, the command is then added to the transaction status buffer with an “ongoing” status.

When an accepted write command completes execution either successfully or with an error, the entry in the transaction status buffer is updated with a “finished” status and the specific execution status. The initiator of the write command is expected to read from the transaction status buffer to determine the execution status.

7.16.2.3. Read commands

For RIRP read commands, a reply is sent within 100 ms (excluding any delays due to spacewire network congestion).

Read commands which have been received and accepted will be added to the transaction status buffer with an “ongoing” operation state.

If a read command has been received and accepted and the read execution finishes within 100ms, a reply will be sent with the read data and a status indicating success. The read command is no longer stored in the transaction status buffer after success and the initiator is not expected to read from the transaction status buffer.

If a read command has been received and accepted and the read execution does not finish within 100ms, a reply will be sent with no read data and a status set to 1 (General error code). The entry in the transaction status buffer is updated with a “timed out” operation state (execution status is unspecified in this case). The initiator of the read command is expected to read from the transaction status buffer to determine the timed out status of the read command.

If a read command has been received and accepted and the read execution completes with an error, a reply will be sent with a status set to 1 (General error code) which may include read data, depending on the error. The entry in the transaction status buffer is updated with a "finished" operation state and the specific execution status. The initiator of the read command is expected to read from the transaction status buffer to get the specific execution error.

7.16.2.4. Reading from the Transaction Status Buffer

RIRP read commands which reads from the transaction status buffer is an exception to the general read command handling:

- Transaction status buffers read commands are never added as transaction status buffer entries.
- Transaction status buffer reads cannot time out.
- If a transaction status buffer read fails due to the read length being longer than the transaction status buffer size, a reply will be sent with a status set to 11 (RMW Data Length error), this is a non-standard use of this RMAP status code.
- No other observable execution failures exist for transaction status buffer reads.

Commands with has completed execution or timed out will be cleared from the transaction status buffer once the transaction status entry has been fully read.

7.16.3. Input

The RMAP commands supported by the TCM are specified in table below. See chapter 7.16.7 for details on each specific command.

Table 7-41 RMAP commands TCM

Name	Ext. Addr	Address	Cmd	Description
TMStatus	0x90 - RIRP 0xFF- No RIRP	0x00000000	R	Reads latest telemetry status.
TMConfig	0x90 - RIRP 0xFF- No RIRP	0x00000200	R	Reads telemetry configuration.
TMControl	0x90 - RIRP 0xFF- No RIRP	0x00000300	W	Enable/Disable telemetry.
TMFEControl	0x90 - RIRP 0xFF- No RIRP	0x00000400	W	Enable/Disable Frame Error Control Field for TM Transfer Frames.
TMMCFCControl	0x90 - RIRP 0xFF- No RIRP	0x00000500	W	Enable/Disable Master Channel Frame Counter Control for TM Transfer Frames.
TMIFControl	0x90 - RIRP 0xFF- No RIRP	0x00000600	W	Enable/Disable Idle Frames.
TMPRControl	0x90 - RIRP 0xFF- No RIRP	0x00000700	W	Enable/Disable Pseudo Randomization for telemetry.
TMCEControl	0x90 - RIRP 0xFF- No RIRP	0x00000800	W	Enable/Disable Convolutional Encoding for telemetry.
TMBRControl	0x90 - RIRP 0xFF- No RIRP	0x00000900	W	Sets telemetry clock frequency divisor (bitrate)
TMOCFControl	0x90 - RIRP 0xFF- No RIRP	0x00000A00	W	Enable/Disable inclusion of Operational Control field in TM Transfer Frames.
TMTSControl	0x90 - RIRP 0xFF- No RIRP	0x00000B00	R/W	Configures Timestamp of telemetry.
TMTSStatus	0x90 - RIRP 0xFF- No RIRP	0x00000C00	R	Latest timestamp of telemetry on virtual channel 0.
TMSend	0x90 - RIRP 0xFF- No RIRP	0x0000100N	W	Sends telemetry on virtual channel N.See [RD18] for allowed VCs.
TCStatus	0x90 - RIRP 0xFF- No RIRP	0x01000000	R	Reads latest telecommand status.
TCDRControl	0x90 - RIRP 0xFF- No RIRP	0x01000100	W	Enables/Disables Derandomizer of telecommands.
TCQueueQuery	0x90 - RIRP 0xFF- No RIRP	0x01001000	R	Query the oldest packet from TC queue.

TCQueueRemoveAndQuery	0x90 - RIRP 0xFF- No RIRP	0x01001100	R	Remove packet from TC queue and query next.
TCQueueClear	0x90 - RIRP 0xFF- No RIRP	0x01001200	W	Clear the queue.
HKData	0x90 - RIRP 0xFF- No RIRP	0x02000000	R	Reads housekeeping data.
SCETTime	0x90 - RIRP 0xFF- No RIRP	0x02000100	R/W	Reads/Sets SCET time.
HKResetCause	0x90 - RIRP 0xFF- No RIRP	0x02000500	R	Retrieves the cause of the last TCM reset
HKLastBootStatus	0x90 - RIRP 0xFF- No RIRP	0x02000600	R	Reads out the status of the last failed boot.
HKDeathReports	0x90 - RIRP 0xFF- No RIRP	0x02000700	R	Reads out death reports to allow analysis of resets.
HKClearDeathReports	0x90 - RIRP 0xFF- No RIRP	0x02000800	W	Clears the death report area on NVRAM.

UARTCommand	0x90 - RIRP 0xFF- No RIRP	0x0400010n	W	Sends a command to UART device n. 0 – UART0 1 – UART1 2 – UART2 3 – UART3 4 – UART4 5 – PSU Ctrl 6 – Safe Bus.
MMData	0x80-0x8F- RIRP 0x00-0x0F - No RIRP	0xnnnnnnnn	R/W	Reads/writes data from/to a partition. The extended address field determine the partition number. The address field is used differently on different types of partitions, see command details.
MMDataRange	0x90 - RIRP 0xFF- No RIRP	0x0500010n	R	Address ranges of all stored data in partition n.
MMPartitionConfig	0x90 - RIRP 0xFF- No RIRP	0x0500030n	R	Configuration of partition n.
MMPartitionSpace	0x90 - RIRP 0xFF- No RIRP	0x0500040n	R	Space available in partition n.
MMDownloadPartitionData	0x90 - RIRP 0xFF- No RIRP	0x0500050n	W	Downloads partition n data via telemetry.
MMFree	0x90 - RIRP 0xFF- No RIRP	0x0500060n	W	Frees memory from partition n.
MMDownloadStatus	0x90 - RIRP 0xFF- No RIRP	0x0500070n	R	Amount of data downloaded in partition n.
MMStopDownloadData	0x90 - RIRP 0xFF- No RIRP	0x05000A0n	W	Stops download of data from partition n.
MMGetPageSize	0x90 - RIRP 0xFF- No RIRP	0x05000B00	R	Reads out size of page, block and spare area
MMTCStorageStatus	0x90 - RIRP 0xFF- No RIRP	0x05000C00	R	TC storage status information.
MMTCStorageClear	0x90 - RIRP 0xFF- No RIRP	0x05000D00	W	Clear the TC storage.
MMBadBlockCount	0x90 - RIRP 0xFF- No RIRP	0x05000E00	R	Read out number of bad blocks.
SpwBackupRoutingEnableDisableSet	0x90 - RIRP 0xFF- No RIRP	0x07000200	W	Enables/disables backup SpW routing
SpwBackupRoutingEnableDisableGet	0x90 - RIRP 0xFF- No RIRP	0x07000300	R	Reads out the current SBR configuration
SpwRoutingPathSet	0x90 - RIRP 0xFF- No RIRP	0x07000400	W	Sets the SpW paths
SpwRoutingPathGet	0x90 - RIRP 0xFF- No RIRP	0x07000500	R	Reads out the SpW paths
SpwReplyPathSet	0x90 - RIRP 0xFF- No RIRP	0x07000600	W	Sets the SpW write-reply paths.
SpwReplyPathGet	0x90 - RIRP 0xFF- No RIRP	0x07000700	R	Gets the SpW write-reply paths.
SpwBackupRoutingTimeoutSet	0x90 - RIRP 0xFF- No RIRP	0x07000800	W	Sets the write reply timeout of the TCM in milliseconds
SpwBackupRoutingTimeoutGet	0x90 - RIRP 0xFF- No RIRP	0x07000900	R	Reads out the write reply timeout of the TCM in milliseconds

RIRPTransactionStatus	0x90 - RIRP 0xFF- No RIRP	0x07000A00	R	Reads out the RIRP transaction status
GPIOGetConfig	0x90 - RIRP 0xFF- No RIRP	0x090001nn	R	Get configuration of GPIO pin nn
GPIOSetConfig	0x90 - RIRP 0xFF- No RIRP	0x090002nn	W	Set configuration for GPIO pin nn
GPIOGetValue	0x90 - RIRP 0xFF- No RIRP	0x090003nn	R	Get value of GPIO pin nn
GPIOSetValue	0x90 - RIRP 0xFF- No RIRP	0x090004nn	W	Set output value for GPIO pin nn

7.16.4. Output

The TCM supports routing of data received by some of its interfaces to other Spacewire nodes, according to the address map below:

Note! All outgoing communication will use the Transaction ID range of `0x0000-0x0FFF`.

Table 7-42 Published data from TCM

Name	Ext. Addr.	Address	Cmd	Description
TCCommand	0xFF	0x00000000	W	Routed Telecommands
UARTData	0xFF	0x0400000x	W	Data received on specified UART x. 0 – UART0 1 – UART1 2 – UART2 3 – UART3 4 – UART4 5 – PSU Ctrl 6 – Safe Bus

7.16.5. Status code in reply messages

7.16.5.1. Status field, RIRP Disabled

In the status field of write/read, the values in table below can be returned, this replaces the standard RMAP status codes described in [RD2]. See individual commands for specific status code interpretations.

Table 7-43 Status codes, RIRP disabled

Code	Numeric value
-	0
EIO	5
EAGAIN	11
ENOMEM	12
EEXIST	17
ENODEV	19
EINVAL	22
ENOSPC	28
ENODATA	61
EBADMSG	77
EALREADY	120
ESTALE	133
ENOTSUP	134
ECANCELED	140

7.16.5.2. Status field, RIRP Enabled

In RMAP Write/Read reply messages when using RIRP, the status field of the reply contains values according to [RD2].

The possible status codes are described in Table 7-44.

Table 7-44 RIRP Status Codes

Numeric value	Command type	Meaning
0	read	Success.
0	write	Success.
1	read	One of: • Invalid extended address. • Read execution timed out. • Read execution failed. RIRP transaction status buffer is full
1	write	One of: • Invalid extended address. • Write execution failed
11	read	Length is greater than the maximum when reading from the transaction status buffer (non-standard use of RMAP status code).

If a write command is sent using RIRP without requesting a reply, no reply is returned.

Error code 1 (General error code) can be returned without the RIRP transaction status buffer being updated. This indicates that the RIRP transaction status buffer was full or that the extended address was wrong.

When error code 1 (General error code) is returned in a read reply, more details about the actual error can be obtained by reading from the transaction status buffer, Table 7-45 provides further details of how to distinguish the actual error in this case.

Table 7-45 Determining actual error for read general errors

Actual error	Determined by
Invalid extended address	No command entry with corresponding transaction ID is present in transaction status buffer.
Read execution timed out	Command entry with corresponding transaction ID is present in transaction status buffer and contains a “timed out” operation state.
Read execution failed	Command entry with corresponding transaction ID is present in transaction status buffer and contains a “finished” operation state.
RIRP transaction status buffer is full	No command entry with corresponding transaction ID is present in transaction status buffer.

7.16.6. Transaction ID

The TCM uses the RMAP Transaction ID to separate between outstanding replies to different units. When several nodes are addressing the TCM, they need to be assigned a unique transaction id range to ensure correct system behaviour. To allow for similar transaction identification throughout the system, the TCM uses the Transaction ID range `0x0000-0x0FFF` in all outgoing communication. The transaction id range `0x0000-0x0FDF` is used for normal commands, and the range `0x0FDE-0x0FFF` is used for resends of commands.

A single node addressing the TCM also must make sure that transaction IDs used for commands that can overlap in time are unique, in order to ensure that on-going transactions cannot be mixed up with new transactions. This also applies for commands without a requested reply.

7.16.7. RMAP input address details

The chapters below contain detailed information on the data accesses to the given RMAP addresses.

7.16.7.1. TMStatus

Reads the latest telemetry status.

Table 7-46 TMStatus data

Byte	Type	Description
0	UINT8	0x00 – No Error 0x01 – FIFO error.
1	UINT8	Reserved

RMAP reply status:

Table 7-47: TMStatus reply status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized.
EIO	I/O error. The TM device cannot be accessed

7.16.7.2. TMConfig

Reads the telemetry configuration.

Table 7-48 TMConfig data

Byte	Type	Description
0-1	UINT16	Telemetry clock bitrate divisor value, default 250.
2	UINT8	Telemetry Control 0x00 – Disabled 0x01 – Enabled (default)
3	UINT8	OCF Control 0x00 – Disabled 0x01 – Enabled (default)
4	UINT8	Frame Error Counter Field Control 0x00 – Disabled 0x01 – Enabled (default)
5	UINT8	Master Channel Frame Count Control 0x00 – Disabled 0x01 – Enabled (default)
6	UINT8	Idle Frame Control 0x00 – Disabled 0x01 – Enabled (default)
7	UINT8	Convolutional Encoding Control 0x00 – Disabled (default) 0x01 – Enabled
8	UINT8	Pseudo Randomization Control 0x00 – Disabled (default) 0x01 – Enabled

RMAP reply status:

Table 7-49 TMConfig reply status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized.
EIO	I/O error. The TM device cannot be accessed

7.16.7.3. TMControl

Controls generation of telemetry.

Table 7-50 TMControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled 0x01 – Enabled (default)

RMAP reply status (if a reply is requested):

Table 7-51 TMControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed

7.16.7.4. TMFEControl

Controls Frame Error Control Field inclusion for transfer frames.

Table 7-52 TMFEControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled 0x01 – Enabled (default)

RMAP reply status (if a reply is requested):

Table 7-53 TMFEControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed

7.16.7.5. TMMCFCControl

Controls the Master Channel Frame Counter generation for transfer frames.

Table 7-54 TMMCFCControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled 0x01 – Enabled (default)

RMAP reply status (if a reply is requested):

Table 7-55 TMMCFCControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed

7.16.7.6. TMIFControl

Controls the Idle Frame generation for transfer frames.

Table 7-56 TMIFControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled 0x01 – Enabled (default)

RMAP reply status (if a reply is a requested):

Table 7-57 TMIFControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed

7.16.7.7. TMPRControl

Controls the Pseudo Randomization for transfer frames.

Table 7-58 TMPRControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled (default) 0x01 – Enabled

RMAP reply status (if a reply is requested):

Table 7-59 TMPRControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed

7.16.7.8. TMCEControl

Controls the Convolutional Encoding for transfer frames.

Note! Convolutional encoding **doubles** both the amount of telemetry data sent and also the telemetry clock frequency, keeping the same net datarate as without.

Table 7-60 TMCEControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled (default) 0x01 – Enabled

RMAP reply status (if a reply is requested):

Table 7-61 TMCEControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed

7.16.7.9. TMBRControl

Sets the telemetry clock frequency divisor.

The telemetry clock is fed to the radio and determines the TM output rate. The divisor is defined such that the useful payload bitrate (before possible convolutional encoding) is the same irrespective of whether convolutional encoding is performed or not. The frequency of the telemetry clock thus depends on the divisor and whether convolutional encoding is enabled or disabled according to:

Interface bitrate with convolutional encoding: $TM\_clk = \frac{50e^6}{divisor}$

Interface bitrate without convolutional encoding: $TM\_clk = \frac{50e^6}{2 \cdot divisor}$

Payload bitrate irrespective of convolutional encoding: $TM\_clk = \frac{50e^6}{2 \cdot divisor}$

Note that a 50% duty cycle will not be achieved with an odd divisor and convolutional encoding enabled.

The TM stream will be interrupted while the bitrate change takes place, as TM is disabled before updating the divisor and then reenabled if it was enabled before.

Table 7-62 TMBRControl data

Byte	Type	Description
0–1	UINT16	Bitrate divisor value (default 25). Convolutional encoding: $6 \le divisor \le 12500$ No convolutional encoding: $3 \le divisor \le 6250$

RMAP reply status (if a reply is requested):

Table 7-63 TMBRControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized.
EIO	I/O error. The TM device cannot be accessed

7.16.7.10. TMOCFControl

Controls Operational Control Field inclusion in TM Transfer frames.

Table 7-64 TMOCFControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled 0x01 – Enabled (default)

RMAP reply status (if a reply is requested):

Table 7-65 TMOCFControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed

7.16.7.11. TMTSControl

Configures the timestamping for transfer frames.

Table 7-66 TMTSControl data

Byte	Type	Description
0	UINT8	The period of the generation is the power of two with the input as exponent. 0x00 – Take a timestamp every time frame sent (default) 0x01 – Take a timestamp every 2nd time frame sent 0x02 – Take a timestamp every 4th time frame sent … 0x08 – Take a timestamp every 256th time frame sent

RMAP reply status (if a reply is requested):

Table 7-67 TMTSControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized or the argument is out of range
EIO	I/O error. The TM device cannot be accessed.

7.16.7.12. TMTSStatus

The latest timestamp of telemetry sent on virtual channel 0.

Table 7-68 TMTSStatus data

Byte	Type	Description
0	UINT32	Seconds counter sampled when the frame event triggered
4	UINT16	Subseconds counter sampled when the frame event triggered

RMAP reply status:

Table 7-69 TMTSStatus status codes

Status code	Description
0	Success.
EINVAL	Insufficient command length.
EIO	I/O error. The TM device cannot be accessed

7.16.7.13. TMSend

Sends telemetry to the TM path on virtual channel N. See [RD18] for VC allocation. If RIRP is enabled and Live TM is sent to the TCM at a higher rate than the TCM can push it to the radio, it is indicated by the RMAP reply with status code 1 (General error). The payload of the TMSend command will be rejected. Reading RIRPTransactionStatus gets detailed information about the error that occurred in the TMSend command.

If TMSend commands provide data at a rate higher than the TM downlink can handle (depending on current bitrate configuration and any other ongoing downlink), it will result in the TM live buffer becoming full. If additional TMSend commands are received when the TM live buffer is full, these commands will be rejected with execution status 12 (“ENOMEM”).

The manner in which this execution status is presented depends on which API is used:

- When not using the RIRP API, this execution status will be provided in the error code field of the RMAP reply.
- When using the RIRP API, the RMAP reply error code will be 1 (“general error”) and the execution status will be provided in the RIRP transaction status buffer accessible via the RIRPTransactionStatus command.

Note! The data must contain **at least one** telemetry PUS Packet.

Table 7-70 TMSend data

Byte	Type	Description
0 – nn	Array of UINT8	Data containing PUS packet(s).

RMAP reply status (if a reply is requested):

Table 7-71 TMSend status codes

Status code	Description
0	Success.
EINVAL	The driver for the TM device has not been initialized.
EIO	I/O error. The TM device cannot be accessed
ENOMEM	TM live buffer is full

7.16.7.14. TCStatus

Reads current telecommand status.

Table 7-72 TCStatus data

Byte	Type	Description
0	UINT32	CLCW word of the last received telecommand.
4	UINT8	Number of missed TC frames due to overflow. Wraps after 0xFF.
5	UINT8	Number of rejected CPDU commands. Wraps after 0xFF.
6	UINT8	Number of rejected telecommands. Wraps after 0xFF.
7	UINT8	Number of parity errors generated by checksums in the telecommand path. Wraps after 0xFF.
8	UINT8	Number of received telecommands. Both TC and CPDU are counted. Wraps after 0xFF.
9	UINT16	Last CPDU pulse command. Logic 1 indicates the last activated line. Bit 15:12 – Unused Bit 11:0 – Line 11:0
11	UINT8	Number of accepted CPDU commands. Wraps after 0x0F.
12	UINT8	Derandomizer setting 0x00 – Disabled. 0x01 – Enabled.
13	UINT16	Length of the last received TC frame

RMAP reply status:

Table 7-73 TCStatus status codes

Status code	Description
0	Success.

EINVAL	The driver for the TC device has not been initialized.
EIO	I/O error. The TC device cannot be accessed

7.16.7.15. TCDRControl

Configures derandomization for telecommand frames.

Table 7-74 TCDRControl data

Byte	Type	Description
0	UINT8	0x00 – Disabled (default) 0x01 – Enabled

RMAP reply status (if a reply is requested):

Table 7-75 TCDRControl status codes

Status code	Description
0	Success.
EINVAL	The driver for the TC device has not been initialized.
EIO	I/O error. The TC device cannot be accessed

7.16.7.16. TCQueueQuery

Reads the oldest packet from the TC queue and some metadata.

Table 7-76 TCQueueQuery data

Byte	Type	Description
0	UINT8	Number of packets in queue
1	UINT8	Queue item ID of current TC packet, [1,255]
2	UINT16	Size of TC packet (maximum 1016)
4 to ...	Array	TC packet

RMAP reply status:

Table 7-77 TCQueueQuery status code

Status code	Description
0	Success
EAGAIN	TC queue is empty (no data returned)

7.16.7.17. TCQueueRemoveAndQuery

Remove the oldest packet from the TC queue (supposedly one that was read before) and read the next packet in the queue. Data returned by this command is the same as in Table 7-76.

RMAP reply status:

Table 7-78 TCQueueRemoveAndQuery status code

Status code	Description
0	Success
EAGAIN	Removal succeeded, but there is no available TC packet in the queue to query (no data returned)
ENODATA	No packet to remove in the queue, queue is empty

7.16.7.18. TCQueueClear

This command clears the entire TC queue. This command does not require or provide any data.

RMAP reply status:

Table 7-79 TCQueueClear status code

Status code	Description
0	Success

7.16.7.19. HKData

Reads the housekeeping data.

Table 7-80 HKData data

Byte	Type	Description
0	UINT32	SCET Seconds
4	UINT16	SCET Subseconds
6	UINT16	Input voltage [mV]
8	UINT16	Regulated 3V3 voltage [mV]
10	UINT16	Regulated 2V5 voltage [mV]
12	UINT16	Regulated 1V2 voltage [mV]
14	UINT16	Input current [mA]
16	INT32	Temperature [mºC]
20	UINT8	S/W version 0-padding
21	UINT8	S/W major version
22	UINT8	S/W minor version
23	UINT8	S/W patch version
24	UINT8	CPU Parity Errors
25	UINT8	Watchdog trips
26	UINT8	Critical (CPU) SDRAM EDAC Single Errors
27	UINT8	Other SDRAM EDAC Single Errors
28	UINT8	Critical (CPU) SDRAM EDAC Multiple Errors
29	UINT8	Other SDRAM EDAC Multiple Errors

RMAP reply status:

Table 7-81 HKData status codes

Status code	Description
0	Success.
EINVAL	The driver for the HK device has not been initialized.
EIO	I/O error. The HK device cannot be accessed

7.16.7.20. SCETTime

Reads/sets the SCET time.

Setting the SCET time is only possible when the PPS is considered qualified, see 7.7 for details. If set, the seconds value will be updated at the next PPS, hence the seconds value should normally be the current seconds count + 1.

The subseconds value is ignored for write commands.

Table 7-82 SCETTime data

Byte	Type	Description
0	UINT32	SCETSeconds
4	UINT16	SCETSubSeconds

RMAP reply status (if a reply is requested):

Table 7-83 SCETTime status codes

Status code	Description
0	Success.
EINVAL	Insufficient command length.
EIO	I/O error. Reading from the SCET device failed.

7.16.7.21. HKResetCause

Gets the last cause of system reset.

Table 7-84 HKResetCause data

Byte	Type	Description
0	UINT32	SCET seconds when latest reset was triggered. Zero following a hard reset or power-up.
4	UINT16	SCET subseconds when latest reset was triggered. Zero following a hard reset or power-up.
6	UINT8	Last cause of reset encoded as: 0x0 – Power-Up 0x1 – Watchdog 0x2 – Manual (SW initiated) 0x3 – CPDU (safe image) 0x4 – CPDU (default image) 0x5 – CPU multi-bit error (Uncorrectable) 0x6 – CPU parity error
7	UINT8	RESERVED

RMAP reply status:

Table 7-85 HKData status codes

Status code	Description
0	Success.

7.16.7.22. HKLastBootStatus

Gets status of last failed boot, if any. Otherwise get status of latest successful boot.

Table 7-86: HKLastBootStatus data

Byte	Type	Description
0	UINT8	Steps defined: 1 – Init 2 – Init timer 3 – Init UART 4 – Read SoC info 5 – Wait for scrubber 6 – Read bad-block table 7 – Set image 8 – Check bad-block table 9 – Get SCET before load 10 – Init sysflash 11 – Load image 12 – Compute load time 13 – Verify checksum 14 – Handover to boot image 0x0E thus indicates boot successful

		0x06 indicates an error occurred during read of the bad block table
1	UINT8	The SW image in error (0 to 5)

RMAP reply status (if a reply is requested):

Table 7-87: HKLastBootStatus status codes

Status code	Description
0	Success.

7.16.7.23. HKDeathReports

Gets context of up to 5 anomalous events (A to E) that have led to an unhandled exception. The Trap Type parameter is detailed in Table 7-144.

Table 7-88: 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
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

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

RMAP reply status (if a reply is requested):

Table 7-89: HKDeathReports status codes

Status code	Description
0	Success.
EINVAL	The driver for the HK device has not been initialized
EIO	I/O error. The HK device cannot be accessed

7.16.7.24. HKClearDeathReports

Clears the stored death reports.

Table 7-90: HKClearDeathReports data

Byte	Type	Description
0	UINT8	0x01 – Clear death reports

RMAP reply status (if a reply is requested):

Table 7-91: HKClearDeathReports status codes

Status code	Description
0	Success.
EINVAL	The driver for the HK device has not been initialized or the argument is out of range
EIO	I/O error. The HK device cannot be accessed

7.16.7.25. UARTCommand

Send a command on the specified UART interface.

Table 7-92 UARTCommand data

Byte	Type	Description
0 - nn	Array of UINT8	UART command data

RMAP reply status (if a reply is requested):

Table 7-93 UARTCommand status codes

Status code	Description
0	Success.
ENODEV	This UART device has not been configured/initialized.
EINVAL	The value for the UART device is invalid.
EIO	I/O error. The UARTdevice cannot be accessed

7.16.7.26. MMData

Reads or writes data from/to a partition.

7.16.7.26.1. Read

The address given in the RMAP command defines the starting byte address of the read and the RMAP data size determines the length of the read in bytes.

If no data is available at the starting address an error will be reported. If less than the requested data is available, a short read will be returned with an RMAP error status indication. If read errors occur based on uncorrectable read errors, the data will be returned along with an RMAP error status indication.

Reads which pass the end of the partition logical address space will automatically wrap.

7.16.7.26.2. Write

Writes to direct partitions needs to specify the starting address and the size via the RMAP address and RMAP data size, the size needs to be a multiple of the page size (16 Kbytes for 16 GB mass memory, or 32 Kbytes för 32 GB mass memory). If the write would overwrite existing data or write at an invalid location, an RMAP error status will be reported and no data will be written.

Writes to continuous or circular partitions needs to specify the size via the RMAP data size and must indicate use of the write pointer by setting the address to 0.

Writes which pass the end of the partition logical address space will automatically wrap.

For direct and continuous partitions, if bad blocks occur during a write which causes available blocks to run out, the remainder of the write will be discarded, and a pending copy operation will be set. In order to avoid data loss, freeing of enough data in order to provide two new unused blocks should be performed as soon as possible, which will allow the copy operation to be retried. Confirmation of the success of the copy operation should be done by verifying that the available space is equal to one block, otherwise the freeing and copy success confirmation procedure should be repeated. For circular partitions, the copy retrying is taken care of automatically.

The amount of data that was written and the amount of data that was discarded in case of a write causing available blocks to run out on direct or continuous partitions can be found by examining the data ranges.

Writing to a circular mode partition that is being downloaded is not allowed.

Writing to a TC storage partition via RMAP is not allowed.

The data field of the read/write RMAP message in Table 7-94 contains raw data written to or read from the partition.

Table 7-94 MMData data

Byte	Type	Description
0 - nn	Array of UINT8	Data

RMAP reply status (if a reply is requested):

Table 7-95 MMData data status codes

Status code	Description
0	Success.
ENOSPC	Write: Not enough space on partition (may have been caused by bad blocks, see suggested handling above). Read: Not enough data on partition. **Note!** *It's allowed to ask for more read data than is available on the partition. Available data will be returned (stating the length in the RMAP reply packet) together with this error code.*
EINVAL	Invalid partition number, or Attempt to write partial page to direct mode partition, or Address is not 0 when writing to continuous or circular partition, or Length is greater than INT32_MAX, or Length is greater than segment size when writing to an auto-padded partition.
EEXIST	Write operation to direct mode partition would overwrite existing data.
EALREADY	Write to circular partition that is being downloaded.
ENOTSUP	Write not allowed for TC storage type partition.

7.16.7.27. MMDataRange

This command will return all data address ranges where data is written in this partition, see Table 7-96. The range information should be interpreted differently for different partition modes.

Continuous and circular mode - Only one range will be reported, corresponding to the free and write pointers. Empty and full partitions will show the free and write pointers having the same value, use the MMPartitionSpace command to get size status.

Direct mode - This is a collection of ranges. Empty partitions will return an empty range table (RMAP reply data of length 0). The ranges will represent the start and end of each continuous data segment in the partition.

Ranges will not exactly match the currently unavailable space due to partially freed (but not yet erased) blocks.

The start address of the range is inclusive, the end address of the range is excusive.

Table 7-96 MMDataRange data

Byte	Type	Description
0-3	UINT32	Start address of first data range.
4-7	UINT32	End address of first data range (exclusive).
8-11	*UINT32*	*Start address of second data range (optional).*
12-15	*UINT32*	*End address of second data range (exclusive) (optional).*
. . .	. . .	. . .

RMAP reply status:

Table 7-97 MMDataRange status codes

Status code	Description
0	Success.
EINVAL	Invalid partition number.

7.16.7.28. MMPartitionConfig

Reads the current partition configuration (see 7.9.2), the RMAP reply message data format is described in Table 7-98.

The available blocks in the flash mass memory ranges from 0 to 8191.

Table 7-98 MMPartitionConfig data

Byte	Type	Description
0	UINT32	Starting block number of the partition.
4	UINT32	Ending block number of the partition (inclusive).
8	UINT8	Partition mode. 0 – Direct 1 – Continuous 2 – Circular 3 – Auto-padded Continuous 4 – Auto-padded Circular
9	UINT8	Specifies type of data stored on the partition. 0 – Space Packets 1 – Raw Data (not supported for download) 2 – TC storage
10	UINT8	Specifies which virtual channel to be used for downloading of the data in the partition. See [RD18] for VC allocation.
11	UINT8	Segment size for the partition. 1 - 16 kbyte 2 - 32 kbyte 3 - N/A 4 - 64 kbyte
12	UINT32	The data source identifier for the partition. Can be used to set a custom identifier of a data producer to a partition. Setting of this value is not required to successfully configure a partition.

RMAP reply status:

Table 7-99 MMPartitionConfig data status codes

Status code	Description
0	Success.
EINVAL	Invalid partition number.

7.16.7.29. MMPartitionSpace

Gets the amount of free space in a partition.

Note that due to the nature of the flash memory, as memory is freed, the space will become free for writing only in leaps as the free operation is used up to a block boundary. This means that a partition can have a discrepancy between reported free space and expected free space of maximum one block.

The reported space for direct partitions will correspond to the total space of every available unused page, minus any freed bytes which belongs to a block which has not yet been fully freed.

The reported space for continuous and circular partitions will correspond to the total space of every unused byte, minus the data offset in the initial write block.

For continuous/circular partitions, since the write pointer is never reset it may not be located at the beginning of a block when the initial write occurs or is about to occur, hence the amount of free space may not correspond exactly to the amount of available fully freed blocks. It is possible (but not recommended during normal operation) to re-synchronize the write pointer by writing exactly the amount needed to end up at the start of a block, and then erase up to the write pointer. This will cause the free space to be exactly equal to the amount of available blocks (or the partition maximum logical address space limit).

Table 7-100 MMPartitionSpace data

Byte	Type	Description
0-7	UINT64	Available size in bytes.

RMAP reply status:

Table 7-101 MMPartitionSpace status codes

Status code	Description
0	Success.
EINVAL	Invalid partition number.

7.16.7.30. MMDownloadPartitionData

Downloads data of the requested length from the partition using the virtual channel set in the partition configuration (see 7.9.2.5). Download commands will be processed one at a time and any prioritizations between different partitions must be handled by sending the download commands in priority order. For direct mode, all download data need to be in a continuous address area (i.e. same data range) or the download will stop when reaching the end of a continuous area even though the download ordered is larger.

In case an invalid Space packet length is encountered, or a Space CRC error occurs in a memory segment during download, the rest of the segment will be downloaded with packet errors and the download will re-synchronise at the start of the next segment.

If a download is started at the end of a partition that is simultaneously written to and the amount of data is beyond the current content of the partition from that point, the download will download only the data available at the time that the download command is issued, regardless of the data written to the partition during download.

Data will normally be downloaded in chunks equal to the segment size set for the partition. It's possible to start and end a download on an uneven segment boundary, but then it's the responsibility of the user to make sure it starts and ends on even PUS packet boundaries. See also information in chapter 7.9.2.3 on padding of data.

A download will not automatically free any data.

This command is not allowed on TC storage partitions.

The RMAP write command data format is described in Table 7-102.

Table 7-102 MMDownloadPartitionData data.

Byte	Type	Description
0-3	UINT32	Address of the data to download
4-11	UINT64	Length in bytes to download

The RMAP reply status (if a reply is requested) will be the first error encountered during a single segment download, i.e. all segment downloads must be sent without fault for Success to be returned.

Table 7-103 MMDownloadPartitionData data status codes

Status code	Description
0	Success.
ENOSPC	Not enough data on partition. **Note!** *It's allowed to request download of more data than is available on the partition. This error code will then be returned and to see the actual amount of data downloaded, use the MMDownloadStatus command.*
EINVAL	Invalid partition number.
EIO	I/O error. Failed to access storage or NVRAM.
EALREADY	A download session is already in progress on this partition.
EBADMSG	Data was not successfully downloaded on downlink.
ENOTSUP	Download not allowed for TC storage type partition.

7.16.7.31. MMFree

Frees memory of a partition. The MMFree operation behaves differently depending on the mode of the partition targeted.

Direct mode - The address and length given in the RMAP command together defines which memory area should be freed.

Continuous and circular mode - The free pointer position together with the length given in the RMAP command defines which memory area should be freed and the address field is ignored. This operation will also move the free pointer forward.

Trying to free more memory than is available is a valid use case and can for example. be used to empty a partition by issuing an MMFree call with the maximum partition length.

If a free to a direct partition starts inside used data and not at a block boundary, the operation will free nothing and an RMAP error status will be reported, since such a free could create an illegal address gap. Freeing the whole partition is a special case and still allowed from any starting address.

Frees which pass the end of the partition logical address space will automatically wrap.

Frees may start at unused addresses.

See also 7.9 for an illustration of how free affects the actual amount of memory free for writes.

Note that MMFree on a partition where a download is in progress is not allowed.

This command is not allowed on TC storage partitions.

The RMAP write command data format is described in Table 7-104.

Table 7-104 MMFree data

Byte	Type	Description
0-3	UINT32	Address of memory to free.
4-11	UINT64	Length of memory to free in bytes.

RMAP reply status (if a reply is requested):

Table 7-105 MMFree status codes

Status code	Description
0	Success.
EINVAL	Invalid partition number, or address is not 0 for continuous/circular partition.
EEXIST	Operation could create illegal address gap inside block.
EALREADY	A download is in progress on this partition.
ENOTSUP	Freeing not allowed for TC storage type partition.

7.16.7.32. MMDownloadStatus

Returns the amount of data downloaded for this partition during the last completed download.

This command is not allowed on TC storage partitions.

Table 7-106 MMDownloadStatus data

Byte	Type	Description
0-7	UINT64	Number of bytes downloaded.

RMAP reply status:

Table 7-107 MMFree status codes

Status code	Description
0	Success.
EINVAL	Invalid partition number.
EIO	I/O error. Failed to access storage or NVRAM.
ENOTSUP	Download not allowed for TC storage type partition.

7.16.7.33. MMStopDownloadData

This command can be sent to stop a current download for a partition previously started by the MMDownloadPartitionData command.

This command is not supported on TC storage partitions.

RMAP reply status (if a reply is requested):

Table 7-108 MMStopDownload status codes

Status code	Description
0	Success.
EINVAL	Invalid partition number.
ENOTSUP	Download not allowed for TC storage type partition.

7.16.7.34. MMGetPageSize

This command reads out the available page size and block size of the mass memory.

Table 7-109 MMPartitionSpace data

Byte	Type	Description
0	UINT8	Page size in bytes. 0x00 – 16 * 1024 bytes 0x01 – 32 * 1024 bytes
1	UINT8	Block size in bytes. 0x00 – 2 * 1024 * 1024 bytes 0x01 – 4 * 1024 * 1024 bytes

RMAP reply status:

Table 7-110 MMPartitionSpace status codes

Status code	Description
0	Success.

7.16.7.35. MMTCStorageStatus

Reads the current TC storage status information in the format described in Table 7-111.

Table 7-111 TC Storage status information RMAP address details

Byte	Type	Description
0	UINT8	Bit 7:2 (MSB) – Reserved Bit 1 – Flag indicating if the number of rejected data chunk writes due to storage being full has reached 2^{32} and wrapped since last TCM reset (0 – has not wrapped, 1 – has wrapped). Bit 0 (LSB) – Flag indicating if a TC storage partition is configured (0 – is not configured, 1 – is configured).
1	UINT8	Partition index of TC storage partition.
2 – 3	N/A	Reserved padding.
4 – 7	UINT32	Number of stored data chunks.
8 – 11	UINT32	Number of rejected data chunk writes due to storage being full since last TCM reset.

If the byte 0 - bit 0 flag is not set, indicating that a TC storage partition is not configured, the rest of the status information is invalid/unspecified.

It is not possible to read partial data via this command; the read address must be the base address without any byte offset and the whole status information will be read regardless of the read size specified.

The RMAP reply status for reads via this command can be any of the values described in Table 7-112.

Table 7-112 TC storage status information RMAP read reply status

Status code	Description
0	Success.

7.16.7.36. MMTCStorageClear

Clear all data and the stored data chunk count in the TC storage, the accompanying write data must use the format described in Table 7-113.

Table 7-113 TC storage clear initiation RMAP write format

Bytes	Type	Description
0 – 3	UINT32	Range start address of data on partition.
4 – 7	UINT32	Range end address of data on partition (exclusive).

The clear will be rejected if the range does not match the range of data on the TC storage partition at the point when the clear execution is started. This means that if a new write to the TC storage has occurred, the clear will be rejected, ensuring that it is not possible to silently loose data chunks.

If the clear is accepted, all stored data chunks will be discarded.

The intended use is to first read the current TC storage partition range information via the MMDataRange command, ensure that the range information does not indicate any new data chunks which should not be cleared, and then use this range when sending the clear command.

Clearing can only clear the whole TC storage; no partial clearing is supported.

Clearing does not clear the rejected data chunks count nor the rejected data chunks count wrap flag, these items are only cleared on a TCM reset.

The RMAP reply status for writes via this command can be any of the values described in Table 7-114.

Table 7-114 TC storage clear RMAP write reply status

Status code	Description
0	Success.
19 (ENODEV)	Rejected due to no TC storage being configured.
22 (EINVAL)	Rejected due to size of write data not being 8 bytes.
133 (ESTALE)	Rejected due to range not matching current range of data on partition.

7.16.7.37. MMBadBlockCount

Reads the current number of bad blocks in the Mass Memory.

Table 7-115 - MMBadBlockCount Data

Byte	Type	Description
0	UINT16	Number of Bad Blocks in the Mass Memory

RMAP reply status (if a reply is requested):

Table 7-116: MMBadBlockCount status codes

Status code	Description
0	Success.

7.16.7.38. SpwBackupRoutingEnableDisableSet

Enables/disables backup SpW routing.

Table 7-117 SpwBackupRoutingEnableDisableSet data

Byte	Type	Description
0	UINT8	0x00 – Disabled 0x01 – Enabled

RMAP reply status (if a reply is requested):

Table 7-118 SpwBackupRoutingEnableDisableSet reply status codes

Status code	Description
0	Success.
EINVAL	The argument is out of bounds
EIO	Internal RTEMS error

7.16.7.39. SpwBackupRoutingEnableDisableGet

Reads out the current enable/disable configuration.

Table 7-119 SpwBackupRoutingEnableDisableGet data

Byte	Type	Description
0	UINT8	0x00 – Disabled 0x01 – Enabled

RMAP reply status (if a reply is requested):

Table 7-120 SpwBackupRoutingEnableDisableGet reply status codes

Status code	Description
0	Success.
EINVAL	The argument is out of bounds
EIO	Internal RTEMS error

7.16.7.40. SpwRoutingPathSet

Configures the SpW paths. The maximum size of a path is 8 bytes, and the maximum number of paths is 20. The logic address of the receiving node must be included. It is allowed to send less data than 160 byte, but if the user tries to specify fewer paths than the highest SpW path index configured in nvram, the command will be rejected and EINVAL will be set. The length of the data must be a multiple of 8 bytes, otherwise the command will be rejected and EINVAL will be set.

Note! All SpW paths must contain a terminating null character, otherwise the command will be rejected and EINVAL will be set.

Table 7-121 SpwRoutingSet data

Byte	Type	Description
0 – 7	Array of UINT8	SpW path 0.
8 – 15	Array of UINT8	SpW path 1.

…	…	…
152 –159	Array of UINT8	SpW path 19.

RMAP reply status (if a reply is requested):

Table 7-122 SpwRoutingPathSet reply status codes

Status code	Description
0	Success.
EINVAL	Invalid argument
EIO	Internal RTEMS error

7.16.7.41. SpwRoutingPathGet

Reads out the current SpW paths. The size of a path is 8 bytes, and the maximum number of paths is 20. If a reply is requested, the size of the data returned will always be 160 bytes.

Table 7-123 SpwRoutingPathGet data

Byte	Type	Description
0 – 7	Array of UINT8	SpW path 0.
8 – 15	Array of UINT8	SpW path 1.
…	…	…
152 – 159	Array of UINT8	SpW path 19.

RMAP reply status (if a reply is requested):

Table 7-124 SpwRoutingPathGet reply status codes

Status code	Description
0	Success.
EIO	Internal RTEMS error

7.16.7.42. SpwReplyPathSet

Configures the SpW write-reply paths. The size of a path is 8 bytes, and the maximum number of paths is 20. The logic address of the receiving node must be included. It is allowed to send less data than 160 byte, but if the user tries to specify fewer paths than the highest SpW path index configured in nvram, the command will be rejected and EINVAL will be set. The length of the data must be a multiple of 8 bytes, otherwise the command will be rejected and EINVAL will be set.

Note! All SpW paths must contain a terminating null character, otherwise the command will be rejected and EINVAL will be set.

Table 7-125 SpwReplyPathSet Data

Byte	Type	Description
0 – 7	Array of UINT8	SpW write-reply path 0.
8 – 15	Array of UINT8	SpW write-reply path 1.
…	…	…
152 – 159	Array of UINT8	SpW write-reply path 19.

RMAP reply status (if a reply is requested):

Table 7-126 SpwReplyPathSet Reply Status Codes

Status code	Description
0	Success.
EINVAL	Invalid argument
EIO	Internal RTEMS error

7.16.7.43. SpwReplyPathGet

Reads out the current SpW write-reply paths. The maximum size of a path is 8 bytes, and the maximum number of paths is 20. If a reply is requested, the size of the data returned will always be 160 bytes.

Table 7-127 SpwReplyPathGet Data

Byte	Type	Description
0 – 7	Array of UINT8	SpW write-reply path 0.
8 – 15	Array of UINT8	SpW write-reply path 1.
…	…	…
152 – 159	Array of UINT8	SpW write-reply path 19.

RMAP reply status (if a reply is requested):

Table 7-128 SpwReplyPathGet Reply Status Codes

Status code	Description
0	Success.
EIO	Internal RTEMS error

7.16.7.44. SpwBackupRoutingTimeoutSet

Configures the maximum amount of time the TCM SW will wait for a write-reply from an SpW node. If SpW backup routing is enabled, and an RMAP command has been sent from the TCM SW to a SpW node, and the write-reply does not arrive to the TCM SW before the timeout, the TCM will switch to SpW backup routing and try to send this packet once again.

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.

Table 7-129 SpwBackupRoutingTimeoutSet data

Byte	Type	Description
0 – 1	UINT16	The timeout in milliseconds, max value 65535.

RMAP reply status (if a reply is requested):

Table 7-130 SpwBackupRoutingTImoutSet reply status codes

Status code	Description
0	Success.
EINVAL	The argument is out of bounds.

EIO	Internal RTEMS error

7.16.7.45. SpwBackupRoutingTimeoutGet

Reads out the maximum amount of time the TCM SW will wait for a write-reply from an external SpW node.

Table 7-131 SpwBackupRoutingTimeoutGet data

Byte	Type	Description
0 – 1	UINT16	The timeout in milliseconds, max value 65535.

RMAP reply status (if a reply is requested):

Table 7-132 SpwBackupRoutingTImoutGet reply status codes

Status code	Description
0	Success.
EIO	Internal RTEMS error

7.16.7.46. RIRPTransactionStatus

Read the status of ongoing, finished, and timed out commands from the transaction status buffer.

The RIRPTransactionStatus command will return data in the format described in Table 7-133.

Table 7-133 RIRPTransactionStatus Data

Byte	Type	Description
0 - 3	UINT32	Number of transaction status entries in buffer.
4 - 7	UINT32	Transaction buffer full status. 0x00 – Buffer not full 0x01 – Buffer full
8 - 11	-	Transaction status entry for first command.
12 - 15	-	Transaction status for second command.
..		
NN – (NN+3)	-	Transaction status for last command.

The format of each transaction status entry is described in Table 7-134.

Table 7-134 RIRPTransactionStatus transaction status entry data

Byte	Type	Description
0 - 1	UINT16	Transaction ID
2	UINT8	Operation state: 0x00 - Ongoing 0x01 – Timed out 0x02 - Finished
3	UINT8	Execution status for finished commands. Will contain the same status as non-RIRP replies.

Reading the transaction status entry of a finished or timed out command fully will clear it from the transaction status buffer. When one or more transaction status entries are cleared, the remaining transaction status entries will be shifted towards the beginning of the buffer to remove any gaps.

7.16.7.47. GPIOGetConfig

Gets the configuration of the addressed GPIO pin.

Table 7-135: GPIOGetConfig data

Byte	Type	Description
0	UINT8	Direction 0 – Output 1 – Input
1	UINT8	Mode 0 – Single ended 1 – Differential

RMAP reply status (if a reply is requested):

Table 7-136: GPIOGetConfig status codes

Status code	Description
0	Success.
ENOSPC	The addressed GPIO pin does not exist/is not configured
EIO	I/O error. The GPIO device cannot be accessed

7.16.7.48. GPIOSetConfig

Sets the configuration of the addressed GPIO pin. Differential mode means that a pair of pins is used together for a differential output signal. The pins are paired in sequence, so [0|1], [2|3] and so on, and each pair is controlled by setting the lower numbered pin (i.e. if pin 0 is set to differential output, pin 1 will automatically be set to match). Please note that an RMAP command to change configuration for a lower numbered pin has no effect on the higher numbered pin when both pins are in differential mode. As differential mode is only valid for output, a reply with status code EINVAL will be sent to the initiator if Direction is set to input and Mode to differential.

Note! If a pin pair that shares the same value enters differential mode, the pins will keep their initial values until the lower pin is explicitly set.

Table 7-137: GPIOSetConfig data

Byte	Type	Description
0	UINT8	Direction 0 – Output 1 – Input
1	UINT8	Mode 0 – Single ended 1 – Differential [Note: Differential mode is only valid for output pins]

RMAP reply status (if a reply is requested):

Table 7-138: GPIOSetConfig status codes

Status code	Description
0	Success.
ENOSPC	The addressed GPIO pin does not exist/is not configured
EINVAL	Invalid value or combination of values in configuration
EIO	I/O error. The GPIO device cannot be accessed

7.16.7.49. GPIOGetValue

Gets the value of the addressed GPIO pin. Reading out the value of the higher numbered pin of a differential pair will show the actual value of that pin.

Table 7-139: GPIOGetValue data

Byte	Type	Description
0	UINT8	Value 0 – Pin is low 1 – Pin is high

RMAP reply status (if a reply is requested):

Table 7-140: GPIOGetValue status codes

Status code	Description
0	Success.
ENOSPC	The addressed GPIO pin does not exist/is not configured
EIO	I/O error. The GPIO device cannot be accessed

7.16.7.50. GPIOSetValue

Sets the value of the addressed GPIO pin. In a differential pair it is only valid to set the value of the lower numbered pin.

Table 7-141: GPIOSetValue data

Byte	Type	Description
0	UINT8	Value 0 – Set pin low 1 – Set pin high

RMAP reply status (if a reply is requested):

Table 7-142: GPIOSetValue status codes

Status code	Description
0	Success.
ENOSPC	The addressed GPIO pin does not exist/is not configured
EINVAL	Invalid value **or** Trying to set the higher numbered pin in a differential pair

EIO	I/O error. The GPIO device cannot be accessed

7.16.8. RMAP output address details

7.16.8.1. TCCommand

A fully formed PUS packet according to [RD3] containing a TC packet to be routed.

7.16.8.2. UARTData

Routed data from UART.

Table 7-143 UARTData data

Byte	Type	Description
0 - nn	Array of UINT8	Data received on UART

7.17. Death Report Handling

When an unexpected exception, as defined in Table 7-144, 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. When an unexpected trap has occurred, the watchdog will not be kicked and the TCM will reset. There are five available death report slots in this NVRAM area. If the table is full and a new trap occurs, the death reports handler will not add a new report to the table, it is left unchanged.

Death reports for the TCM can be read via RMAP. TCM death reports can also be cleared via RMAP. FPU traps are disabled in the TCM SW, and thus no death reports will be generated for them.

Table 7-144 - Sirius Trap Allocation

Sirius Trap Allocation					
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
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	

Table 1471 in RD15 describes the implemented traps for LEON3FT. Table 7-144 shows the implementation for Sirius.

7.18. FPU Traps

Table 7-145 - 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*	

There are six subcategories of floating-point exceptions according to Table 4-4 in RD19. Table 7-145 shows the implementation for Sirius. According to section 49.2.3 in RD15 all five floating point exceptions defined by the IEEE-754 standard can be detected (ftt=1).

Floating point traps are disabled by default. Information on how to enable FPU traps is available in section 11.2.

7.19. Limitations

For performance reasons, the current TCM release calculates checksums on neither the incoming nor the outgoing RMAP/SpaceWire packets.

The mass memory maximum partition size is 4 Gbytes. However, there is no limit on the number of blocks assigned for a specific partition, allowing a configuration to compensate for any possible loss in size due to bad blocks.

The mass memory doesn't support download of data from partitions of type raw.

8. NVRAM areas

This chapter is an extension of the RTEMS NVRAM API in 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.

The TCM SW configuration described in 7.4 is stored in two copies, one in the safe area for the safe SW images to use and one copy in the update area for the update images to use. The boot procedure is described further in section 9. When configuring NVRAM with the nv_config library, EDAC mode (described further in 5.11) is used. Therefore Table 8-1 lists addresses as how they are used when EDAC is enabled.

The mass memory bad block table is used by the TCM SW and it is updated during runtime when new bad blocks are discovered. The TCM SW has a reserved area for storing operation markers during runtime.

Table 8-1 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 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.

9. Boot procedure

9.1. 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 9-1.

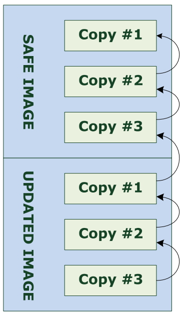


Figure 9-1 Software images in flash

9.2. Usage description

The locations in the system flash where the bootrom looks for software images are given in Table 9-1. The first two 32-bit words of the image are expected to be a header with image size and an XOR checksum, see Table 9-2. 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 8-1. 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 5.3.2.4.7). Reading out that register in orbit requires a subsequent successful boot, so if multiple image copies fail to boot the register information that is saved will be from the first failed attempt.

9.3. 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 9-1 Software image locations

Image	Flash page number
Safe copy #1	0x00000
Safe copy #2	0x20000
Safe copy #3	0x40000
Updated copy #1	0x80000
Updated copy #2	0xA0000
Updated copy #3	0xC0000

Table 9-2 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.

9.4. Cause of last reset

The Error Manager RTEMS driver supports reading out the last reset cause, see 5.3.2.4 for details. There is also an RMAP command for reading out the cause of last reset from the TCM, see 7.16.7.21 for details.

9.5. 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.

10. Software upload

10.1. 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.

Updating a flight image entails four types of operations. First the actual data transfer and commanding from earth, which requires the software upload mechanism to be compliant with the CCSDS standard for TC and where the principal recipient would be the TCM, regardless of the end target. The TCM simply acts as a router in this case, routing the PUS command to the intended source based on the PUS APID and the TCM routing table. Second would be the mechanism for distributing the image upload data to different recipients in a data handling system (i.e. also the TCM itself) using the PUS extension of the CCSDS standard (see [RD3]). Third would be the assembly of all telecommands, with a data fragment each, into a full or partial image for update with verification. Finally, the fourth would be the actual update of the physical flash image.

The descriptions in sections 10.3 and 10.4 will cover the two middle operations. The first (inital CCSDS handling) and the last (flash operations) are covered in 7.6 and 5.12. The picture in Figure 10-1 shows the intended control flow when commanding the software update from ground.

10.2. Block diagram

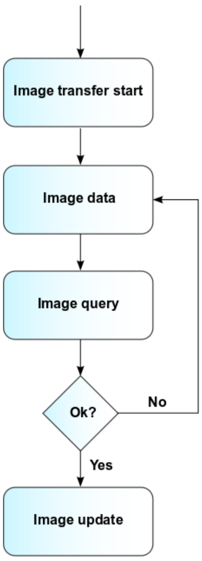


Figure 10-1 The intended software upload command flow

10.3. CCSDS API – custom PUS service 130

10.3.1. 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 [RD3]) if requested in the telecommand PUS header. See [RD18] 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 10-1 without any parameters (see [RD3]).

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 10-1 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 10-2 below.

Table 10-2 Error code numerical values

Error code	Numeric value
ENOENT	2
EIO	5
EBUSY	16
EINVAL	22
ENOSPC	28
ENODATA	61
EALREADY	120

10.3.2. 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 10-3 below.
Minimum image size is currently 272 bytes including header, and maximum image size is 16 Mbyte.

Table 10-3 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 10-4 in case of a failure.

Table 10-4 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

10.3.3. Subtype 2 – Image data

This subtype transports data segments of the actual flight software image. Each segment can be maximum 1000 bytes long (to avoid splitting packets over several frames), and all segments except the last shall be of maximum length. The data format is specified in Table 10-5 below, with the data length given in the PUS header.

Table 10-5 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 10-6 in case of a failure.

Table 10-6 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)
ENOENT	UINT8	No upload in progress

10.3.4. 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 10-7

Table 10-7 Verify uploaded image argument

Checksum
UINT32

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 10-8 in case of a failure.

Table 10-8 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

10.3.5. 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 10-9.

Table 10-9 Write image command argument

Checksum
UINT32

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 10-10 in case of a failure.

Table 10-10 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)

10.3.6. 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 10-11. Image copy numbers 1 – 3 are for the (non-updateable) safe image and 4 – 6 cover the updated image copies.

Table 10-11 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 10-12 in case of a failure.

Table 10-12 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
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 10-13.

Table 10-13 Calculated flash CRC report

Image copy number	Checksum
UINT8	UINT32

10.4. 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.

10.4.1. 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 name	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

10.4.2. 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 name	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)
-ENOSPC	Out of non-flash blocks in intermediate storage area of slash (critical)
-ENOENT	No upload in progress

10.4.3. 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 name	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

10.4.4. 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 name	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

10.4.5. 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 name	Type	Direction	Decription
image_number	uint8_t	in	Image number in flash to calculate the checksum of
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

10.5. Usage description

A user of the software upload module can either let the module handle all PUS commanding through the PUS API (see section 10.3) or handle all PUS packetizing and reporting internally and only hook into the functional interface described in section 10.4. A code example is provided in the directory `src\example`.

10.6. Limitations

The maximum size of an image for upload is 16 Mbytes.

11. Death Reports

11.1. Format

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/`. When using this library, the format of death reports saved in the NVRAM is shown in Table 11-1.

Table 11-1: 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
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
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 11-1. 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 [RD19], 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.

11.2. Reports for FPU 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.

11.3. 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.

11.4. 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 `src/death_reports/examples/exception_handler.c`.

An example application for reading out and parsing death reports from NVRAM is available in `src/death_reports/examples/read_nvram_death_report_area.c`. An example application that clears the death reports area on NVRAM is available in `src/death_reports/examples/clear_nvram_death_report_area.c`.

Note! Please note that 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.

12. TM/TC-structure and COP-1

12.1. SCID

For commanding the spacecraft, a 10-bit Spacecraft Identifier is needed. For every mission, a mission specific SCID is configured in the TCM.

12.2. APID

The application running on the TCM has a unique identifier, Application Process Identifier, that is configurable for every mission by a parameter stored in the NVRAM on the TCM.

12.3. Virtual Channel Allocation

See [RD18] for VC allocation.

12.4. Uplink Channel Coding, Randomization and Synchronization

12.4.1. Channel Coding

The Telecommand Code Block is BCH (63, 56) and supports Single Error Correction mode.

12.4.2. Randomization

Derandomization of telecommands can be enabled/disabled by a configuration in NVRAM or by a RMAP command.

12.4.3. Channel Synchronization

The 2-byte start sequence of Telecommands is 0xEB90. The 8-byte tail sequence of Telecommands is 0xC5C5C5C5C5C5C579.

12.5. Downlink Channel Coding, Randomization and Synchronization

12.5.1. 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. RS encoding can be enabled/disabled by a configuration in NVRAM.

Convolutional encoding 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) can be enabled/disabled by a configuration in NVRAM or by a RMAP command.

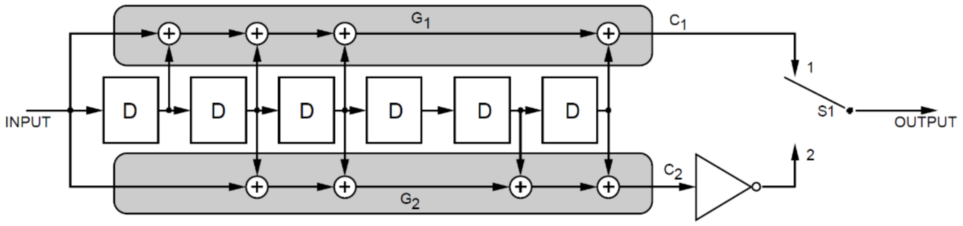


Figure 12-1 Convolutional Encoder Block Diagram

12.5.2. Randomization

Randomization of Telemetry Transfer Frames can be enabled/disabled by a configuration in NVRAM or by a RMAP command.

12.5.3. Synchronization

The 4-byte synchronization pattern prepended to the Reed-Solomon code block is 0x1ACFFC1D.

12.6. Telecommand format

This chapter describes the format of the TC Transfer frames and TC Packets the TCM handles.

12.6.1. Telecommand Transfer Frame

The Telecommand Transfer Frame conforms to the format described in [RD8] and shown below.

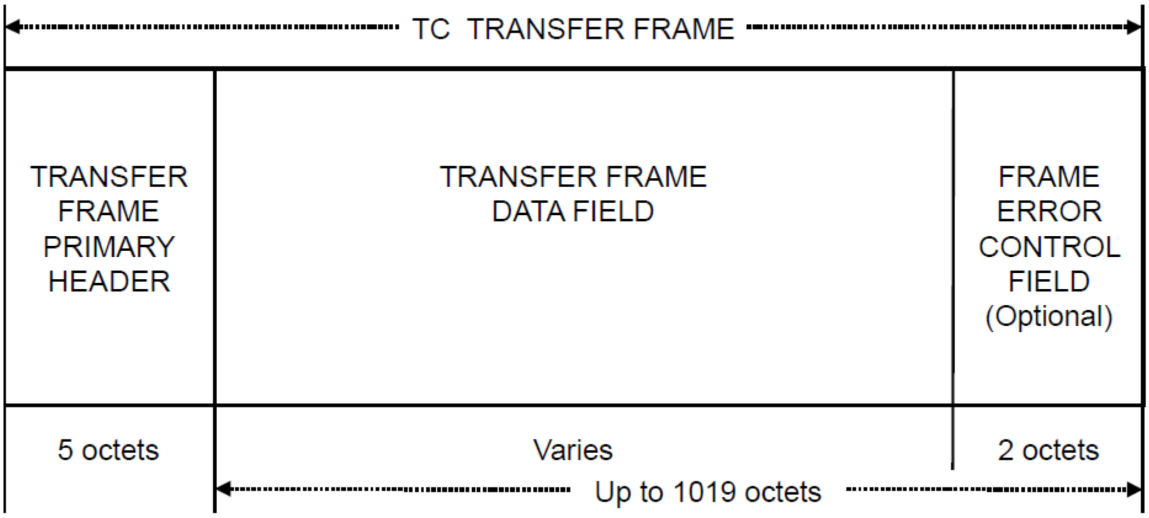


Figure 12-2 TC Transfer Frame

12.6.2. Transfer Frame Header

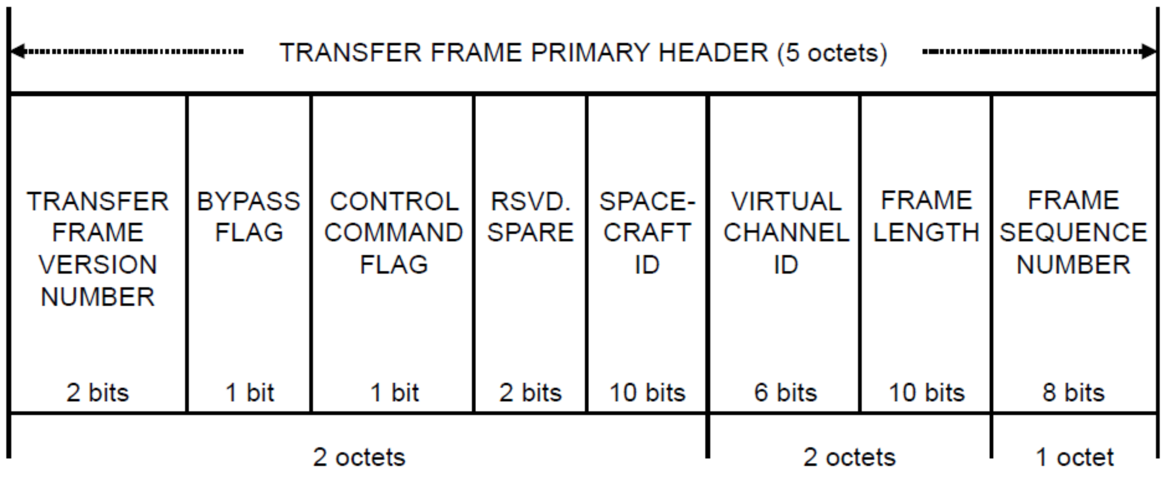


Figure 12-3 Transfer Frame Header

Table 12-1 Transfer Frame Header

Field	Description	Comment
VERSION NUMBER	Shall be set to ‘00’	
BYPASS FLAG	Set to ‘0’ to set Type-A of frame. Set to ‘1’ to set Type-B of frame.	When this flag is set to ‘0’, the frame will be subject to Frame Acceptance Check of the FARM on TCM. When this flag is set to ‘1’, the Frame Acceptance Check will be bypassed on the TCM.
CONTROL COMMAND FLAG	Set to ‘0’ to indicate the Transfer Frame Data Field contains a Frame Data Unit (Type-D) Ser to ‘1’ to indicate the Transfer Frame Data field contains control information (Type-C)	In conjunction with BYPASS FLAG, the frame types Type-AD, Type-BD and Type-BC are supported by the TCM.
RESERVED SPARE		
SPACECRAFT ID	Contains the mission-specific spacecraft identifier (SCID)	If the SCID of the TC Transfer Frame is not same as the SCID configured on the TCM, the TC Transfer Frame will be rejected.
Virtual Channel ID	Virtual channel ID of Telecommand	See [RD18] for VC allocation.
FRAME LENGTH	Shall be set to total number of octets in the TC Transfer Frame - 1	The maximum number of octets in the TC Transfer Frame is 1024.
FRAME SEQUENCE NUMBER	The number of the TC Transfer Frame	The Frame sequence number enables the FARM to check sequence of incoming Type-A transfer frames

12.6.3. Transfer Frame Data Field

TC Transfer Frames sent to the TCM are expected to contain the Frame Error Control Field, which results in a maximum length of 1017 octets of the Transfer Frame Data Field. The Transfer Frame Data Field shall contain either a Frame Data Unit (for Type-D Transfer Frame) or a control command (for Type-C Transfer Frames).

For Transfer Frames carrying a Frame Data Unit, a Segment Header follows the Transfer Frame Primary Header, see Figure 12-4. For Frame Data Units, the user data shall contain a complete packet, see 12.6.5

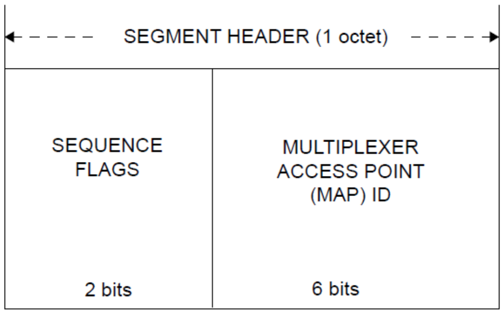


Figure 12-4 Segment Header

Table 12-2 Segment Header

Field	Description	Comment
SEQUENCE FLAGS	Shall be set to ‘11’ since no segmentation is supported on TCM	
MAP ID	Shall be set to 0.	Only MAP ID 0 is supported on TCM

Two control commands are supported by TCM: Unlock and Set V(R). The Unlock Control Command consists of a single octet containing “all zeroes”. The Set V(R) Control Command shall consist of three octets with the following values:

10000010 00000000 XXXXXXXX

where XXXXXXXX is the value the FARM shall set the Receiver Frame Sequence Number, V(R).

12.6.4. Frame Error Control Field

The Frame Error Control holding an error detection code (checksum) shall always be included in the telecommand transfer frames, which allows the receiving application to verify the integrity of the telecommand frame data.

The checksum shall be calculated using CRC with polynomial 0x8408, LSB first (reverse of 0x1021, MSB first); and initial value 0xFFFF over the whole TC Transfer Frame except the two last octets.

12.6.5. Telecommand Packet

All Telecommand Packets in Frame Data Units shall be Space Packets as used in the ECSS Packet Utilization Standard [RD3], with the format given in Figure 12-5 below.

Packet Header (48 Bits)							Packet Data Field (Variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional) (see Note 1)	Application Data	Spare	Packet Error Control (see Note 2)
Version Number (=0)	Type (=1)	Data Field Header Flag	Application Process ID	Sequence Flags	Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	16

Figure 12-5 Telecommand Packet

Table 12-3 Packet Header

Field	Description	Comment
Version number	Packet structure version number. Shall be set to 0	
Type	Distinguishes telecommand packets and telemetry packets. For telecommands, the type shall be set to 1	
Data Field Header Flag	With exceptions of CPDU telecommands, all telecommand packets shall have a data field header so this bit shall be set to 1	
Application Process ID	Sets the destination on-board application for the telecommand packet.	
Sequence Flags	The TCM does only support stand-alone packets, so this field shall be set to ‘11’	
Sequence Count	Identifier provided to be able to track a specific packet.	
Packet Length	Specifies number of octets within the packet data field. The number shall be number of octets in packet data field - 1.	

CCSDS Secondary Header Flag	TC Packet PUS Version Number	Ack	Service Type	Service Subtype	Source ID	Spare
Boolean (1 bit)	Enumerated (3 bits)	Enumerated(4 bits)	Enumerated (8 bits)	Enumerated (8 bits)	Enumerated (n bits)	Fixed BitString (n bits)
					Optional	Optional

Figure 12-6 Data Field Header

Table 12-4 Data Field Header

Field	Description	Comment
CCSDS Secondary Header Flag	Shall be set to ‘0’	
TC PUS Packet PUS Version Number	Shall be set to ‘001’	
Ack	Specifies level of reporting to ground by the receiving Application Process. The TCM sends acceptance success report and execution completion success report based on the ack flags.	See section 5.3.3 in [RD3].
Service Type	Indicates the service to which the packet relates	
Service Subtype	Indicates the subtype of the service the packet relates to	
Source ID	Not used	
Spare	Not used	

Application data holds the data elements of the command.

Spare may be used to do padding of TC to achieve an integral number of words.

The checksum of the packet 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 TC Packet except the two last octets.

12.6.6. 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 12.7.5.

12.7. Telemetry Format

This chapter describes the format of TM Transfer Frames and TM Packets sent from the TCM to ground.

12.7.1. Telemetry Transfer Frame

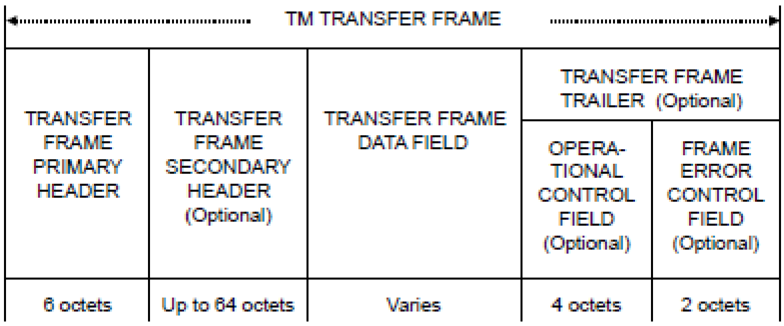


Figure 12-7 Telemetry Transfer Frame

12.7.2. Transfer Frame Primary Header

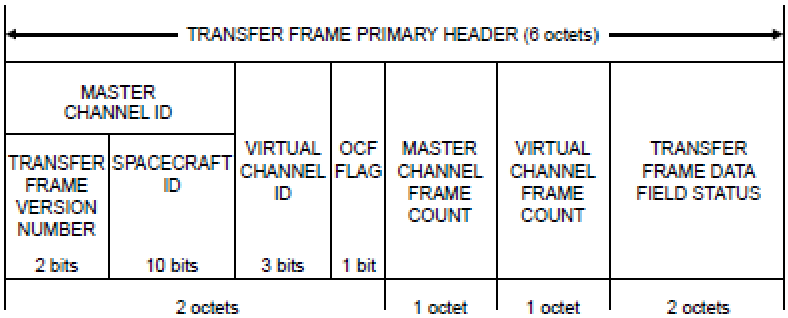


Figure 12-8 Telemetry Transfer Primary Header

Table 12-5 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 [RD18] 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	

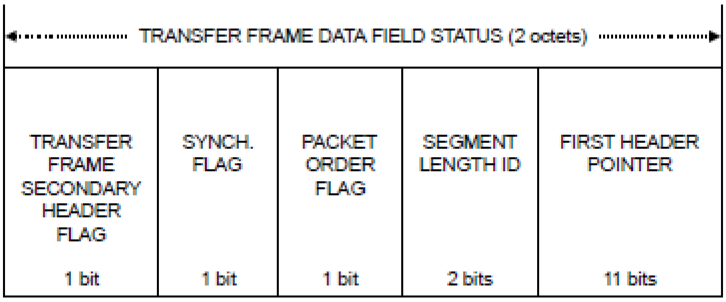


Figure 12-9 Transfer Frame Data Field Status

Table 12-6 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’.
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’.	

12.7.3. Transfer Frame Secondary Header

The Transfer Frame Secondary Header is not used by the TCM.

12.7.4. Transfer Frame Data Field

The Transfer Frame Data Field contains an integral number of octets of data formatted as TM Packets, see 12.7.7. 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)

12.7.5. Operational control field

The Operational Control Field contains a Communications Link Control Word as described in RD8 section 4.2.

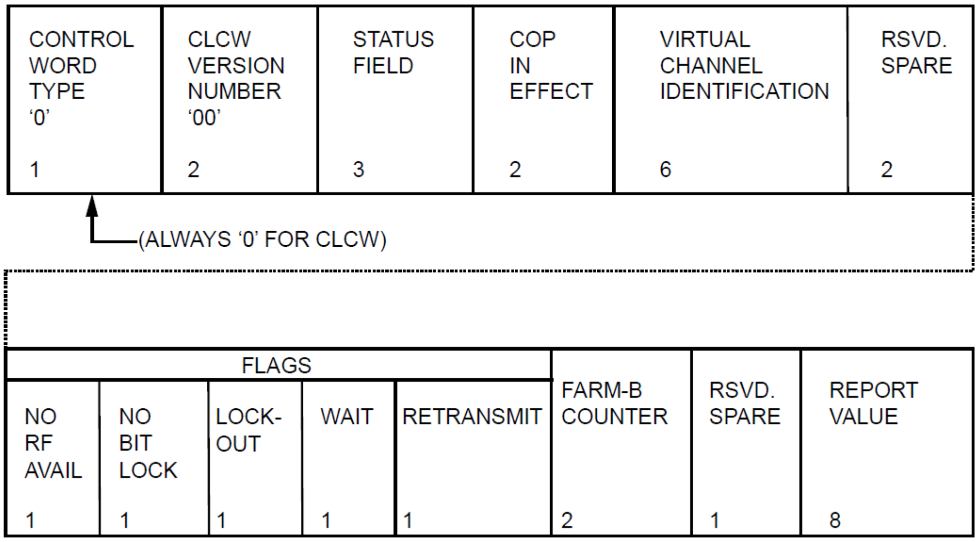


Figure 12-10 Command Link Control Word

Table 12-7 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. Set to '1' if Physical Layer is not available.	Controlled by physical input signal, see 12.6.6.
NO BIT LOCK	Set to '0' when bit lock has been achieved. Set to '1' when bit lock has not been achieved.	Controlled by physical input signal, see 12.6.6.
LOCK-OUT	Shows Lockout status of the FARM. Set to '0' when FARM is not in Lockout. Set to '1' when FARM is in Lockout.	
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. 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. 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).	

12.7.6. 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.

12.7.7. Telemetry Packet

Packet Header (48 Bits)							Packet Data Field (Variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional) (see Note 1)	Source Data	Spare (Optional)	Packet Error Control (Optional)
Version Number (=0)	Type (=0)	Data Field Header Flag	Applica-tion Process ID	Grouping Flags	Source Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	(see Note 2)

Figure 12-11 Telemetry Packet

12.7.8. Telemetry Packet Header

Table 12-8 Telemetry Packet Header

Field	Description	Comment
Version Number	Set to ‘000’.	
Type	Set to ‘0’.	
Data Field Header Flag	Set to ‘1’ to indicate presence of Data Field Header. Set to ‘’0’ to indicate absence of Data Field Header.	All TM Packet generated by TCM use Data Field Header.
Application Process ID	Indicates application process that is the source of the packet.	
Grouping Flags	Set to ‘11’ to indicate “stand-alone” packet.	The TCM generates “stand-alone” packet only.
Source Sequence Flags	Source sequence counter.	
Packet Length	Number of octets in packet data field - 1.	

12.7.9. Data Field Header

The data field header from the Telemetry and Telecommand packet utilization standard [RD3], is depicted in Figure 12-12.

Spare	TM Source Packet PUS Version Number	Spare	Service Type	Service Subtype	Packet Sub-counter	Destination ID	Time	Spare
Fixed BitString (1 bit)	Enumerated (3 bits)	Fixed BitString (4 bits)	Enumerated (8 bits)	Enumerated (8 bits)	Unsigned Integer (8 bits)	Enumerated	Absolute Time	Fixed BitString (n bits)
					Optional	Optional	Optional	Optional

Figure 12-12 Data Field Header

Table 12-9 Data Field Header

Field	Description	Comment
Spare		
TM Source Packet PUS Version Numbers	Set to ‘001’	
Spare		
Service Type	Indicates the service this source packet relates to	
Service Subtype	Together with Service Type, this field indicates the subtype this source packet relates to.	
Packet Subcounter	Counter related to a specific service and subservice	Not used by TCM
Destination ID	Can be used for destination of a TM Packet	Not used by TCM
Time	On-board reference time	On TCM, the time field consist of CUC Time Seconds (32-bit), followed by CUC Time Fractions (16-bit)
Spare		Not used by TCM

The format of the Data Field Header used in TCM is shown below:

Spare (1 bit)	TM Source Packet PUS Version Number (3 bits)	Spare (4 bits)	Service Type (8 bits)	Service Subtype (8 bits)	Time (48 bits)

Figure 12-13 Data Field Header supported by TCM

12.7.10. Source Data

The packet source data of the Telemetry Packets sent to ground.

12.7.11. Spare

Spare may be used to pad a packet to an integral number of words if needed.

12.7.12. Packet Error Control

Packet Error Control is used by TCM and the checksum of the Packet 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 Packet except the two last octets.

12.7.13. Idle Data

In the TCM, 0x5A is the data sent for Idle Frames and Idle Packets.

12.8. FARM-parameters

COP-1 is supported on the TCM.

12.8.1. FARM_Sliding_Window_Width(W)

In the TCM, the parameter W is fixed to 128.

12.8.2. FARM_Positive_Window_Width(PW)

In the TCM, the parameter PW is fixed to 64.

12.8.3. FARM_Negative_Window_Width(NW)

In the TCM, the parameter NW is fixed to 64.

13. Updating the Sirius FPGA

To be able to update the SoC on the Sirius OBC and Sirius TCM you need the following items.

13.1. Prerequisite hardware

- Microsemi FlashPro5 unit
- 104470 FPGA programming cable assembly

13.2. Prerequisite software

- Microsemi FlashPro Express v11.8 or later
- The updated FPGA firmware

13.3. 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.

13.4. 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 3-1
2. Connect the power cables according to Figure 3-1
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 13-1) and select the supplied .pro file.

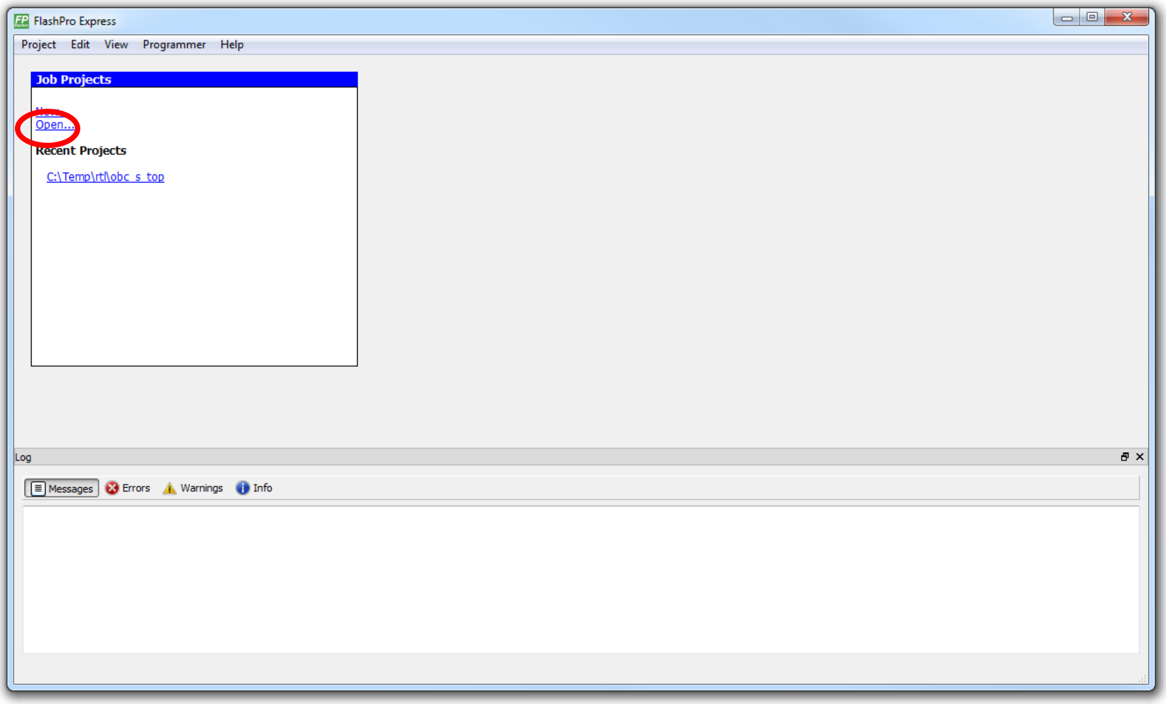


Figure 13-1 - Startup view of FlashPro Express

5. Once the file has loaded (warnings might appear), click RUN (see Figure 13-2). Please note that the connected FlashPro5 programmed ID should be shown.

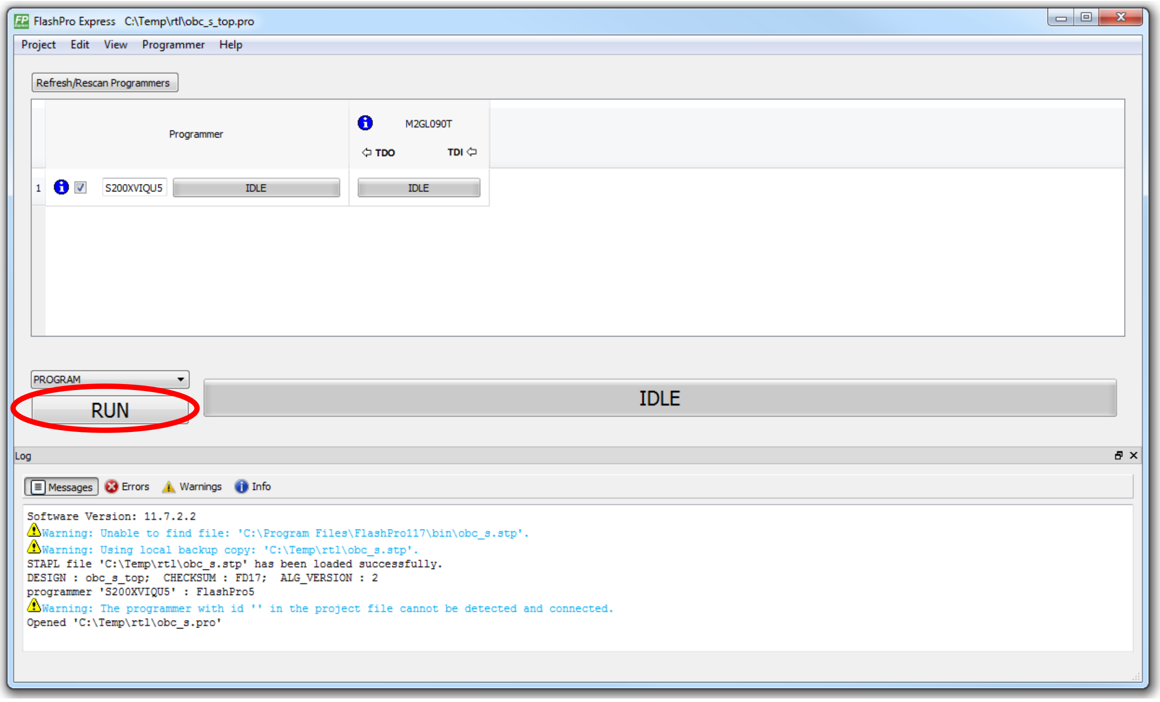


Figure 13-2 - View of FlashPro Express with project loaded.

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 13-3.

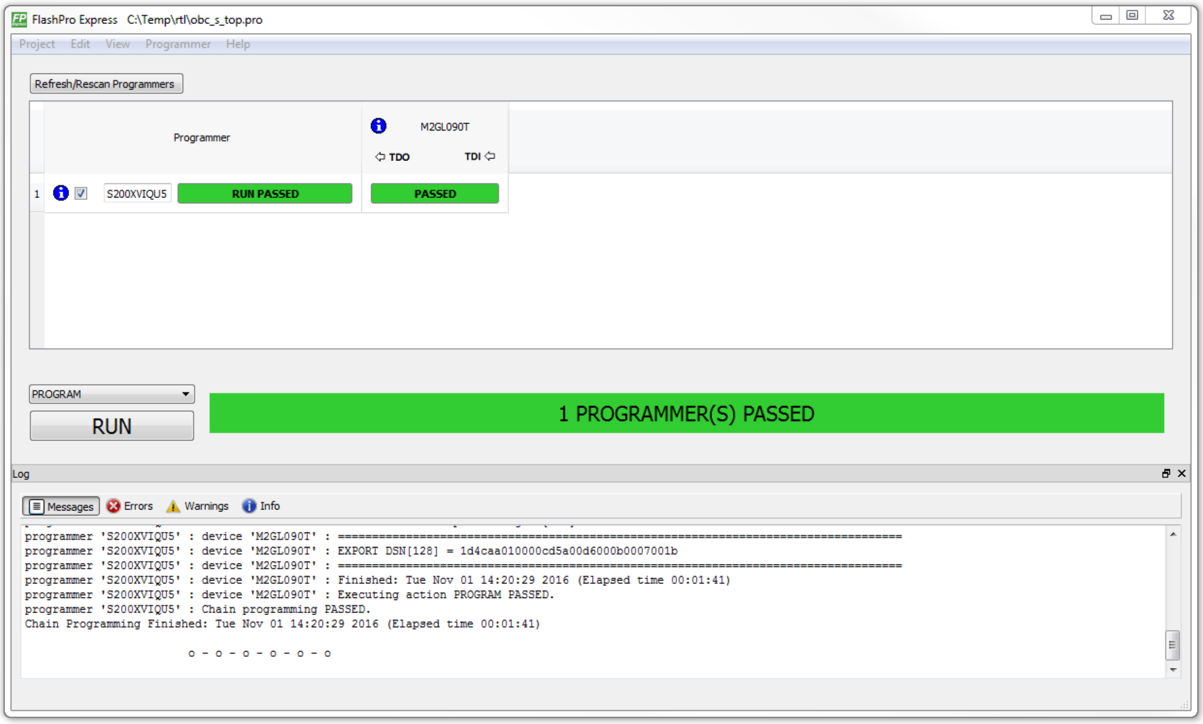


Figure 13-3 - View of FlashPro Express after program passed.

The Sirius FPGA image is now updated.

14. Mechanical data

Please refer to the Mechanical and Electrical ICDs (RD9, RD10).

15. Glossary

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 [RD7] and [RD8]
COP	Communications Operation Procedure, see [RD7] and [RD8]
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 [RD8]
FECF	Frame Error Control Field, see [RD7] and [RD8]
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
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 [RD7] 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
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

AAC CLYDE SPACE