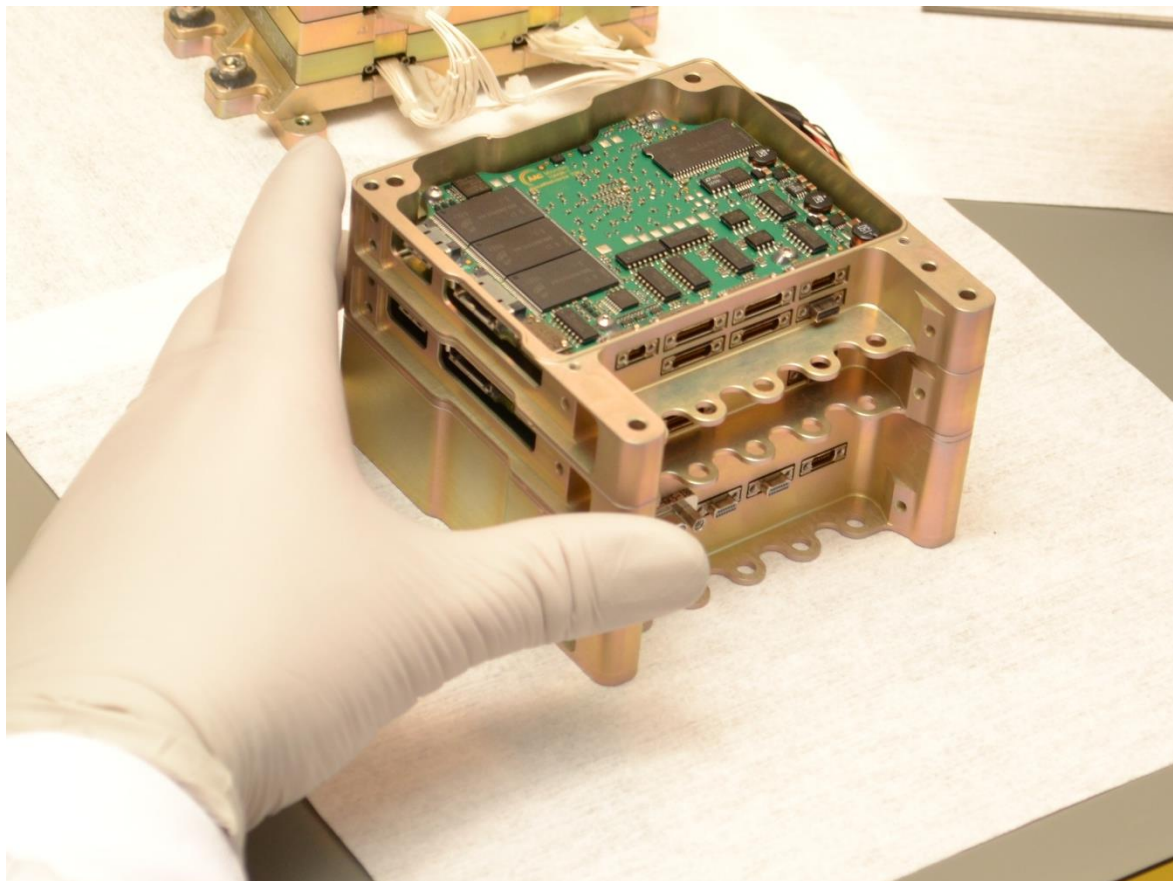


Sirius OBC and TCM User Manual

Rev. J



© AAC Microtec 2016-2018

AAC Microtec AB owns the copyright of this document which is supplied in confidence and which shall not be used for any purpose other than for which it is supplied and shall not in whole or in part be reproduced, copied, or communicated to any person without written permission from the owner.

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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • Introduction • Equipment information • Sirius TCM (updated API and formatting) • NVRAM (updated API) |
| I | 2018-04-16 | Release with updates of the following sections: <ul style="list-style-type: none"> • Software upload (new) • NVRAM (updated EDAC error reporting API) |
| J | 2018-06-28 | Release with updates of the following sections: <ul style="list-style-type: none"> • SCET, UART, WDT, NVRAM and SpW (updated API) • Mass Memory Handling (auto-padding) • Removed chapter with connector pinout |

TABLE OF CONTENT

| | |
|--|-----------|
| 1. INTRODUCTION | 7 |
| 1.1. Applicable releases | 7 |
| 1.2. Intended users | 7 |
| 1.3. Getting support..... | 7 |
| 1.4. Reference documents | 8 |
| 2. EQUIPMENT INFORMATION | 9 |
| 2.1. System Overview with peripherals | 9 |
| 2.2. Fault tolerant design | 10 |
| 3. SETUP AND OPERATION..... | 12 |
| 3.1. User prerequisites | 12 |
| 3.2. Connecting cables to the Sirius products..... | 13 |
| 3.3. Installation of toolchain | 14 |
| 3.3.1. Supported Operating Systems | 14 |
| 3.3.2. Installation Steps..... | 14 |
| 3.4. Installing the Board Support Package (BSP) | 15 |
| 3.5. Deploying a Sirius application | 15 |
| 3.5.1. Establish a debugger connection to the Sirius products..... | 15 |
| 3.5.2. Setup a serial terminal to the device debug UART..... | 16 |
| 3.5.3. Loading an application | 17 |
| 3.5.4. Using multiple debuggers on the same PC | 17 |
| 3.6. Programming an application (boot image) to system flash | 18 |
| 4. SOFTWARE DEVELOPMENT | 19 |
| 4.1. RTEMS step-by-step compilation..... | 19 |
| 4.2. Software disclaimer of warranty | 19 |
| 5. RTEMS..... | 20 |
| 5.1. Introduction..... | 20 |
| 5.2. Watchdog | 21 |
| 5.2.1. Description | 21 |
| 5.2.2. RTEMS API..... | 21 |
| Usage description | 22 |
| 5.2.3. | 22 |
| 5.3. Error Manager | 25 |
| 5.3.1. Description | 25 |
| 5.3.2. RTEMS API..... | 25 |
| 5.3.3. Usage..... | 30 |
| 5.3.4. Limitations | 31 |
| 5.4. SCET | 32 |
| 5.4.1. Description | 32 |
| 5.4.2. General purpose triggers | 32 |
| 5.4.3. Pulse-Per-Second (PPS) signals | 32 |
| 5.4.4. RTEMS API..... | 33 |
| 5.4.5. Usage description | 38 |
| 5.4.6. Limitations | 41 |
| 5.5. UART..... | 42 |
| 5.5.1. Description | 42 |

| | |
|---|------------|
| 5.5.2. RTEMS API..... | 42 |
| 5.5.3. Usage description | 46 |
| 5.5.4. Limitations | 47 |
| 5.6. Mass memory..... | 47 |
| 5.6.1. Description | 47 |
| 5.6.2. Data Structures | 48 |
| 5.6.3. RTEMS API..... | 49 |
| 5.6.4. Usage..... | 56 |
| 5.6.5. Error injection | 59 |
| 5.6.6. Limitations | 59 |
| 5.7. Spacewire..... | 60 |
| 5.7.1. Description | 60 |
| 5.7.2. RTEMS API..... | 60 |
| 5.7.3. Usage description | 65 |
| 5.8. GPIO..... | 67 |
| 5.8.1. Description | 67 |
| 5.8.2. RTEMS API..... | 67 |
| 5.8.3. Usage description | 70 |
| 5.8.4. Limitations | 71 |
| 5.9. CCSDS | 72 |
| 5.9.1. Description | 72 |
| 5.9.2. Non-blocking | 72 |
| 5.9.3. Blocking | 73 |
| 5.9.4. Buffer data containing TM Space packets..... | 73 |
| 5.9.5. RTEMS API..... | 73 |
| 5.9.6. Usage description | 80 |
| 5.10. ADC..... | 82 |
| 5.10.1. Description | 82 |
| 5.10.2. RTEMS API..... | 83 |
| 5.10.3. Usage description | 86 |
| 5.10.4. Limitations | 87 |
| 5.11. NVRAM | 88 |
| 5.11.1. Description | 88 |
| 5.11.2. EDAC mode | 88 |
| 5.11.3. Non-EDAC mode | 88 |
| 5.11.4. RTEMS API..... | 88 |
| 5.11.5. Usage description | 91 |
| 5.12. System flash | 93 |
| 5.12.1. Description | 93 |
| 5.12.2. Data structure types | 93 |
| 5.12.3. RTEMS API..... | 93 |
| 5.12.4. Usage description | 98 |
| 5.12.5. Debug detect..... | 100 |
| 5.12.6. Limitations | 100 |
| 6. SPACEWIRE ROUTER..... | 101 |
| 7. SIRIUS TCM..... | 102 |
| 7.1. Description..... | 102 |
| 7.2. Block diagram..... | 103 |
| 7.3. TCM-S application overview | 103 |
| 7.4. Configuration | 104 |

| | |
|---|----------------|
| 7.4.1. Creating and writing a new configuration | 108 |
| 7.5. Telemetry..... | 109 |
| 7.6. Telecommands | 109 |
| 7.6.1. Pulse commands..... | 110 |
| 7.6.2. COP-1 | 111 |
| 7.7. Time Management | 111 |
| 7.7.1. TM time stamps..... | 111 |
| 7.8. Error Management and System Supervision | 111 |
| 7.9. Mass Memory Handling..... | 112 |
| 7.9.1. Partition configuration | 113 |
| 7.9.2. Recovery..... | 116 |
| 7.10. ECSS standard services | 117 |
| 7.10.1. PUS-1 Telecommand verification service | 117 |
| 7.10.2. PUS-2 Device Command Distribution Service | 117 |
| 7.11. Custom services | 118 |
| 7.11.1. PUS-130 Software upload..... | 118 |
| 7.12. Spacewire RMAP | 118 |
| 7.12.1. Input..... | 118 |
| 7.12.2. Output | 119 |
| 7.12.3. Status code in reply messages | 120 |
| 7.12.4. RMAP input address details..... | 120 |
| 7.12.5. RMAP output address details..... | 135 |
| 7.13. Limitations | 135 |
| 8. SYSTEM-ON-CHIP DEFINITIONS | 136 |
| 8.1. Memory mapping..... | 136 |
| 8.2. Interrupt sources | 137 |
| 8.3. SCET timestamp trigger sources | 137 |
| 8.4. Boot images and boot procedure..... | 138 |
| 8.4.1. Description | 138 |
| 8.4.2. Block diagram | 138 |
| 8.4.3. Usage description | 138 |
| 8.4.4. Limitations | 139 |
| 8.5. Reset behaviour..... | 139 |
| 8.6. General synchronize method | 139 |
| 8.7. Pulse command inputs | 139 |
| 8.8. SoC information map | 140 |
| 9. SOFTWARE UPLOAD | 141 |
| 9.1. Description..... | 141 |
| 9.2. Block diagram..... | 141 |
| 9.3. CCSDS API – custom PUS service 130..... | 142 |
| 9.3.1. Subtype 1 – Image transfer start..... | 143 |
| 9.3.2. Subtype 2 – Image data | 143 |
| 9.3.3. Subtype 3 – Verify uploaded image | 144 |
| 9.3.4. Subtype 4 – Write uploaded image | 144 |
| 9.3.5. Subtype 5 – Calculate CRC in flash | 145 |
| 9.4. Software API | 146 |
| 9.4.1. int32_t swu_init(...) | 146 |
| 9.4.2. int32_t swu_segment_add(...) | 146 |
| 9.4.3. int32_t swu_check(...)..... | 147 |
| 9.4.4. int32_t swu_update(...)..... | 147 |

| | |
|-------------------------------------|------------|
| 9.4.5. int32_t swu_flash_check(...) | 148 |
| 9.5. Usage description | 148 |
| 9.6. Limitations | 148 |
| | |
| 10. UPDATING THE SIRIUS FPGA | 149 |
| 10.1. Prerequisite hardware | 149 |
| 10.2. Prerequisite software | 149 |
| 10.3. Generation of encryption key | 149 |
| 10.4. Step by step guide | 149 |
| | |
| 11. MECHANICAL DATA | 151 |
| | |
| 12. GLOSSARY | 152 |

1. Introduction

This manual describes the functionality and usage of the AAC 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 OBC | 1.3.0 |
| Sirius TCM | 1.3.0 |

1.2. Intended users

This manual is written for software engineers using the AAC Sirius products. The electrical and mechanical interface is described in more detail in the electrical and mechanical ICD documents [RD10] and [RD11].

1.3. Getting support

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

Email: support@aacmicrotec.com

1.4. Reference documents

| RD# | Document ref | Document name |
|------|---|---|
| RD1 | https://openrisc.io/architecture | OpenRISC 1000 Architecture Manual |
| RD2 | ECSS-E-ST-50-12C | SpaceWire – Links, nodes, routers and networks |
| RD3 | ECSS-E-ST-50-52C | SpaceWire – Remote memory access protocol |
| RD4 | ECSS-E-70-41A | Ground systems and operations – Telemetry and telecommand packet utilization |
| RD5 | SNLS378B | PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs |
| RD6 | AD7173-8, Rev. A | Low Power, 8-/16-Channel, 31.25 kSPS, 24-Bit, Highly Integrated Sigma-Delta ADC |
| RD7 | Edition 4.11 | RTEMS BSP and Device Driver Development Guide |
| RD8 | CCSDS 132.0-B-2 | TM Space Data Link Protocol |
| RD9 | CCSDS 232.0-B-2 | TC Space Data Link Protocol |
| RD10 | 205088 | Sirius OBC electrical and mechanical ICD |
| RD11 | 205089 | Sirius TCM electrical and mechanical ICD |
| RD12 | SS-EN 61340-5-1 | Electrostatics - Part 5-1: Protection of electronic devices from electrostatic phenomena - General requirements |
| RD13 | Edition 4.11 | RTEMS POSIX Users Manual |

2. Equipment information

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 centered around a 32 bit OpenRISC Fault Tolerant processor [RD1] 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 μ s
- 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 - 16GB 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. The power consumption is highly dependent on peripheral loads and ranges from 0.8 W to 2 W.

2.1. System Overview with peripherals

Figure 2-1 depicts a System-on-Chip (SoC) overview including the related peripherals of the Sirius OBC and Sirius TCM products. The figure shows what parts that are included for which products.

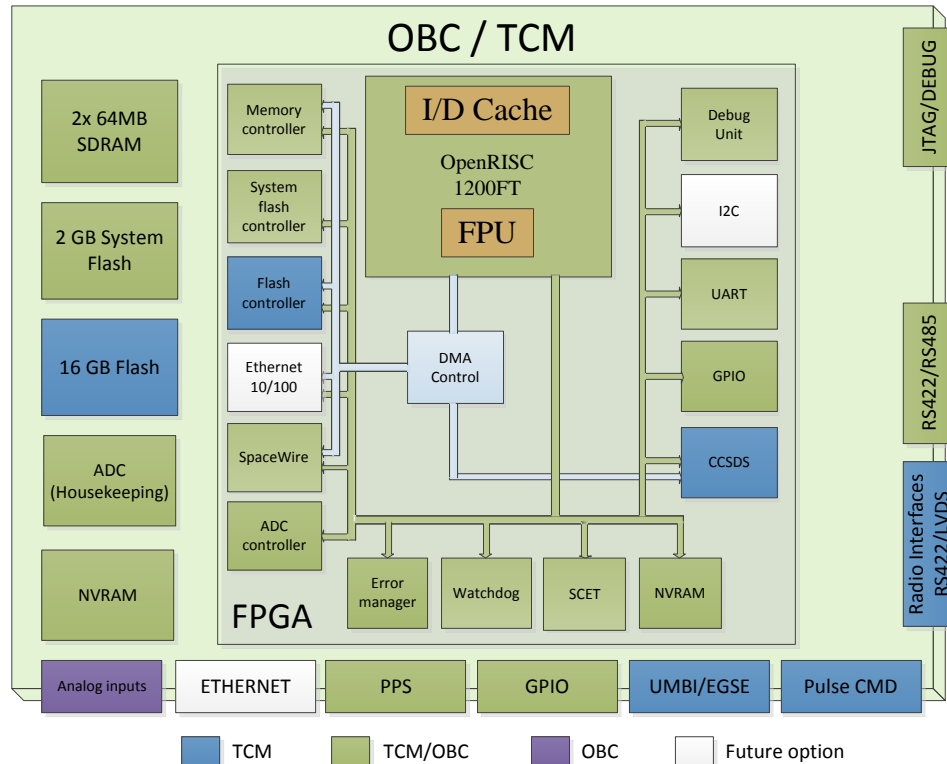


Figure 2-1 - The Sirius OBC / Sirius TCM SoC Overview

2.2. 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.

3. Setup and operation

3.1. User prerequisites

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

PC computer

- 1 GB free space for installation (minimum)
- Debian 8 64-bit with super user rights
- USB 2.0

JTAG debugger

- AAC JTAG debugger hardware including harness

Recommended applications and software

- 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
- The following software is installed by the AAC toolchain package
 - GCC, C compiler for OpenRISC
 - GCC, C++ compiler for OpenRISC
 - GNU binutils and linker for OpenRISC
 - Custom openocd binary designed for OpenRISC

For FPGA update capabilities

- Microsemi FlashPro Express v11.8, <http://www.microsemi.com/products/fpga-soc/design-resources/programming/flashpro#software>
- FlashPro5 programmer

3.2. Connecting cables to the Sirius products

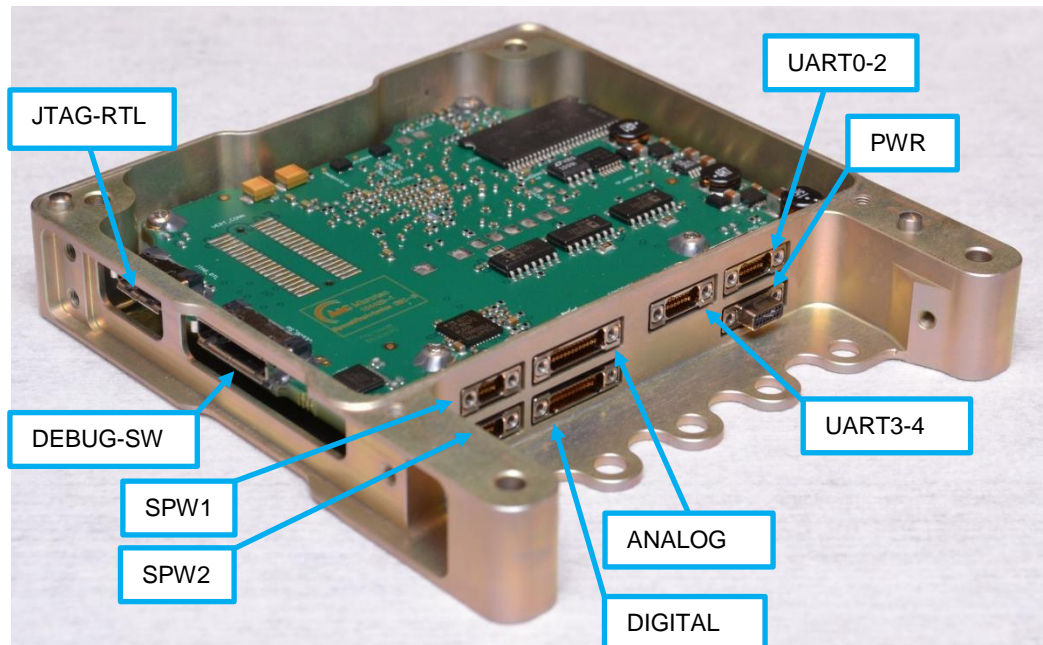


Figure 3-1 – AAC Sirius OBC with connector naming

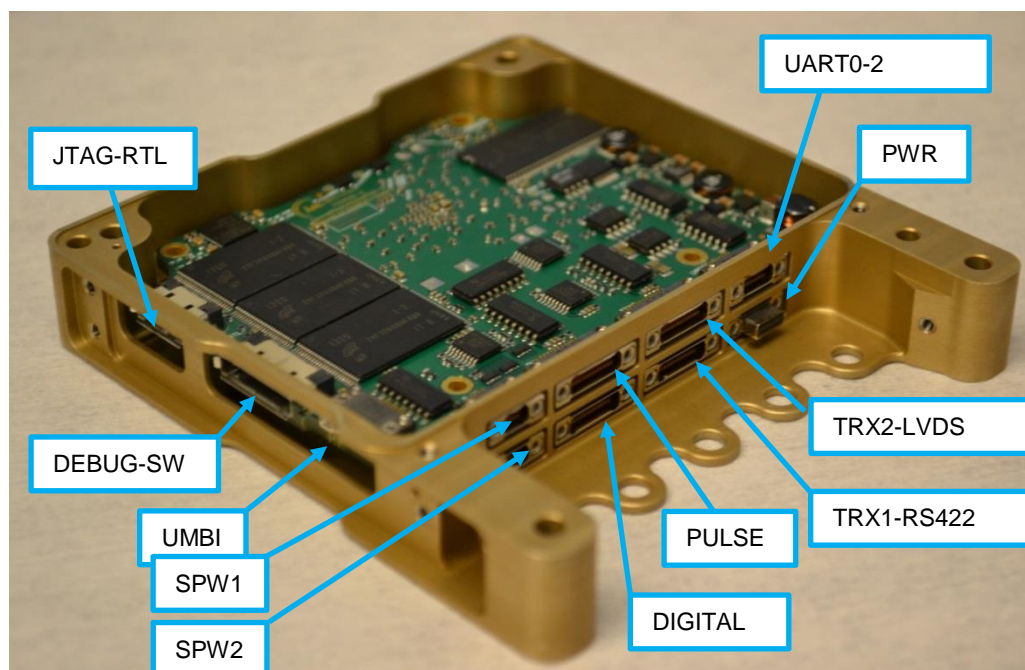


Figure 3-2 - AAC Sirius TCM with connector naming

- 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 [RD12] 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 10.
- For connecting the SpaceWire interface, connect the nano-D connector to connector SPW1 or SPW2.

For more detailed information about the connectors, see [RD10] and [RD11].

3.3. Installation of toolchain

This chapter describes instructions for installing the aac-or1k-toolchain.

3.3.1. Supported Operating Systems

- Debian 8 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>

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 toolchain by issuing the following commands:

```
sudo apt-get update  
sudo apt-get install aac-or1k-toolchain
```

Note: The toolchain package is roughly 1GB uncompressed, downloading/installing it will take some time.

3. Setup

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

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

or permanently by editing the ~/.bashrc (or equivalent) file

```
gedit ~/.bashrc
```

and adding the following snippet at the end of the file, and then saving and closing it:

```
# AAC OR1k toolchain PATH setup  
if [ -f /opt/aac/aac-path.sh ]; then  
    . /opt/aac/aac-path.sh >/dev/null  
fi
```

3.4. Installing the Board Support Package (BSP)

The BSP can be downloaded from <http://repo.aacmicrotec.com/bsp>. Simply extract the tarball [aac-or1k-xxx-x-bsp-y.tar.bz2](#) to a directory of your choice (xxx-x depends on your intended hardware target - Sirius OBC or Sirius TCM and y matches the current version number of that BSP).

The newly created directory [aac-or1k-xxx-x-bsp](#) now contains the drivers for both bare-metal applications and RTEMS. See the included README and chapter 4.1 for build instructions.

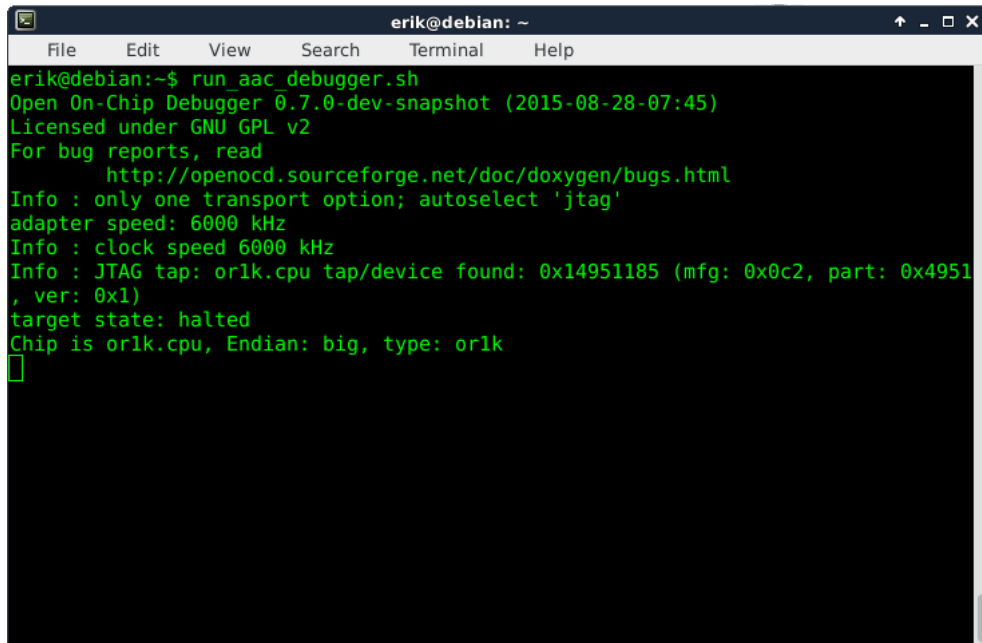
3.5. Deploying a Sirius application

3.5.1. Establish a debugger connection to the Sirius products

The Sirius products are shipped with debuggers who connect to a PC via USB. To interface the Sirius products, the Open On-Chip Debugger (OpenOCD) software is used. A script called `run_aac_debugger.sh` is shipped with the toolchain package which starts an OpenOCD server for gdb to connect to.

1. Connect the Sirius products according to section 3.2 and switch on the power supply.

2. Start the `run_aac_debugger.sh` script from a terminal.
3. If the printed message is according to Figure 3-3, the connection is working.



```
erik@debian: ~  
File Edit View Search Terminal Help  
erik@debian:~$ run_aac_debugger.sh  
Open On-Chip Debugger 0.7.0-dev-snapshot (2015-08-28-07:45)  
Licensed under GNU GPL v2  
For bug reports, read  
  http://openocd.sourceforge.net/doc/doxygen/bugs.html  
Info : only one transport option; autoselect 'jtag'  
adapter speed: 6000 kHz  
Info : clock speed 6000 kHz  
Info : JTAG tap: or1k.cpu tap/device found: 0x14951185 (mfg: 0x0c2, part: 0x4951  
      , ver: 0x1)  
target state: halted  
Chip is or1k.cpu, Endian: big, type: or1k  
█
```

Figure 3-3 - Successful OpenOCD connection to the Sirius products

The line

```
target state: halted
```

must be displayed in the output, otherwise the OpenOCD connection has failed and the board must be power-cycled.

3.5.2. 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 you have to make sure you're using the correct device number to communicate to the board's debug UART.

On Debian 8, a more foolproof way of identifying the terminal to use is the by-id mechanism. Once you've identified the serial number of your debugger (see 3.5.4.), you can connect to it

using the autocreated path at `/dev/serial/by-id/`. The debug UART is identified as `usb-AAC_Microtec_JTAG_Debugger_FTZ7QCMF-if01-port0`, where `FTZ7QCMF` is the serial number in this case. Make sure you use the `if01` number and not `if00` as this is consumed by the OpenOCD server later.

3.5.3. Loading an application

An application can either be loaded only to the volatile memory, which is easier and typically used during the development stages, or to NAND flash (see section 3.6). This is done using `gdb`.

- 1.a) Start `gdb` with the following command from a shell for a bare-metal environment
`or1k-aac-elf-gdb`

or

- 1.b) Start `gdb` with the following command from a shell for an RTEMS environment
`or1k-aac-rtems4.11-gdb`

2. When `gdb` has opened successfully, connect to the hardware through the OpenOCD server using the `gdb` command
`target remote localhost:50001`
3. To run an executable program in hardware, first specify its name using the `gdb` command file. Make sure the application is in ELF format.
`file path/to/binary_to_execute`
4. Now it needs to be uploaded onto the target RAM
`load`
5. In the `gdb` prompt, type `c` to start to run the application

3.5.4. Using multiple debuggers on the same PC

In order to use multiple debuggers connected to the same PC, each instance of OpenOCD must be configured to connect to the specific debugger serial number and to use unique ports. Support for this is included in the `run_aac_debugger.sh` script.

In order to determine the serial number for a specific device, run the following command before connecting the debugger

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

which 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.

The GDB, telnet and TCL ports must be set to a unique value in the Linux user-available range 1025-65535, the defaults are GDB: 50001, telnet: 4444, TCL: 6666.

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

```
run_aac_debugger.sh -s FTZ7QCMF -g 50001 -t 4444 -p 6666  
run_aac_debugger.sh -s FTZ7IB10 -g 50002 -t 4445 -p 6667
```

Two instances of GDB can then be opened, and connected to the different debuggers via

```
target remote localhost:50001
```

and

```
target remote localhost:50002
```

respectively. Only the GDB port is used when connecting from GDB.

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

This chapter describes how to program the NAND flash memory with a selected boot image. To achieve this, the boot image binary is bundled together with the NAND flash programming application during the latter's compilation. The NAND flash programming application is then uploaded to the target and started just as an ordinary application using gdb. The maximum allowed size for the boot image for 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:
Note that `X` is to be replaced according to what your application has been compiled against, either `elf` for a bare-metal application or `rtems4.11` for the RTEMS variant.

```
or1k-aac-X-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-or1k-xxx-x-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 3.5.3. Follow the instructions on screen and when it's ready, reboot the board by a reset or power cycle.

4. Software development

Applications to be deployed on the Sirius products can either use a bare-metal approach or use the RTEMS OS. This corresponds to the two toolchain prefixes available: or1k-aac-elf-* or or1k-aac-rtems4.11-*

Drivers for both are available in the BSP, see chapter 3.4 and the BSP README for more information. However, the RTEMS OS is the recommended way and documentation for the bare-metal layer is not included in this manual.

4.1. RTEMS step-by-step compilation

The BSP is supplied with an application 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 have been 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-or1k-xxx-x-bsp/src/`
2. Type `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
`export RTEMS_MAKEFILE_PATH=path/to/librtems/or1k-aac-rtems4.11/or1k-aac`
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.

4.2. 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.

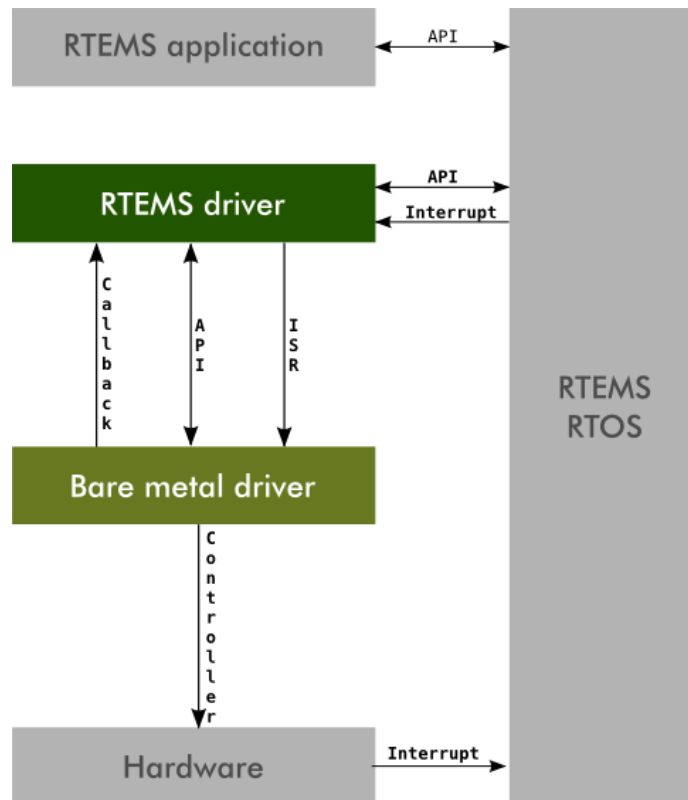


Figure 5-1 - Functionality access via RTEMS API

5.2. Watchdog

5.2.1. Description

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

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.

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 | A bitwise 'or' separated list of values that determine the method in which the file is to be opened (whether it should be read only, read/write). |

| Return value | Description |
|---------------------|---|
| > 0 | A file descriptor for the device on success |
| - 1 | see <i>errno</i> 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 <i>errno</i> values |
| errno values | |
| EPERM | Device is not open. |

5.2.2.3. size_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 <i>errno</i> 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 0 = Disables the watchdog |
| WATCHDOG_SET_TIMEOUT_IOCTL | 0 – 255 = Number of seconds until the watchdog barks |

| Return value | Description |
|---------------------|-------------------------------|
| 0 | Command executed successfully |
| -1 | see <i>errno</i> 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.

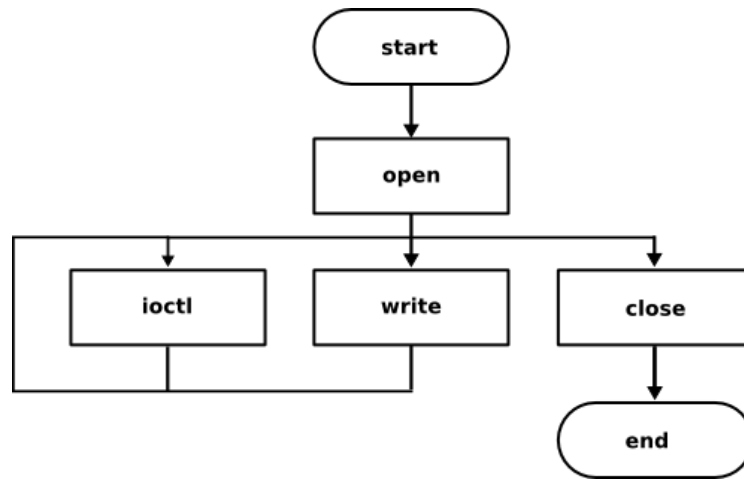


Figure 2 – RTEMS driver usage description

All calls to RTEMS driver are blocking calls.

5.2.3.2. RTEMS application example

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

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

#define CONFIGURE_APPLICATION_NEEDS_WDT_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MAXIMUM_DRIVERS
#define CONFIGURE_MAXIMUM_TASKS 2 /* Idle & Init */
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTOR 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 power loss, non-fatal multiple errors generated by the RAM EDAC mechanism.

5.3.2.1. int open(...)

Opens access to the device, it 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. Error manager device is defined as RTEMS_ERRMAN_DEVICE_NAME. |
| 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 at the end. |

| Return value | Description |
|---------------------|---|
| fd | A file descriptor for the device on success |
| -1 | see <i>errno</i> values |
| errno values | |
| EALREADY | Device already opened |

5.3.2.2. 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 |

5.3.2.3. int ioctl(...)

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 |

| Command table | Description |
|-------------------------------------|--|
| ERRMAN_GET_SR_IOCTL | Get the status register, see 5.3.2.3.1 |
| ERRMAN_GET_CF_IOCTL | Gets the carry flag register, see 5.3.2.3.2 |
| ERRMAN_GET_SELFV_IOCTL | 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 | 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 | Gets the state of the memory scrubber. 0 = Scrubber is disabled 1 = Scrubber is enabled. |
| ERRMAN_GET_RESET_ENABLE_IOCTL | Gets the reset enable state. 0 = Soft reset is disabled. 1 = Soft reset is enabled |
| ERRMAN_GET_WDT_ERRCNT_IOCTL | 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 5.3.2.3.2 |
| ERRMAN_GET_EDAC_SINGLE_ERRCNT_IOCTL | Gets the EDAC single error count. See 5.3.2.3.3 for interpretation of the register. After a wrap the EDAC single error count carry flag bit is set in the carry flag register. See 5.3.2.3.2 |
| ERRMAN_GET_EDAC_MULTI_ERRCNT_IOCTL | Gets the EDAC multiple error count. See 5.3.2.3.4 for interpretation of the register. After a wrap the EDAC multiple error count carry flag bit is set in the carry flag register. See 5.3.2.3.2 |
| ERRMAN_GET_CPU_PARITY_ERRCNT_IOCTL | 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 5.3.2.3.2 |

| | |
|-------------------------------------|--|
| ERRMAN_GET_SYS_SINGLE_ERRCNT_IOCTL | 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 | 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 | 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 | Gets the mass memory multiple error (un-correctable) error count. This register is 4 bit wide and will wrap upon overflow. |
| ERRMAN_GET_POWER_LOSS_ENABLE_IOCTL | Gets the power loss detection enable state. 0 = Power loss detection disabled. 1 = Power loss detection enabled |
| ERRMAN_SET_SR_IOCTL | Sets the status register, see 5.3.2.3.1 |
| ERRMAN_SET_CF_IOCTL | Sets the carry flag register, see 5.3.2.3.2 |
| ERRMAN_SET_SELFV_IOCTL | 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 | Performs a software reset. The reset enable state is required to be 1 (On). |
| ERRMAN_SET_SCRUBBER_IOCTL | 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 | Sets the reset enable state. 0 = Soft reset is disabled. 1 = Soft reset is enabled |
| ERRMAN_SET_WDT_ERRCNT_IOCTL | 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 | Sets the EDAC single error count. See 5.3.2.3.3 for register definition. |
| ERRMAN_SET_EDAC_MULTI_ERRCNT_IOCTL | Sets the EDAC multiple error count register. See 5.3.2.3.4 for register definition. |
| ERRMAN_SET_CPU_PARITY_ERRCNT_IOCTL | 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 | Sets the system flash single (correctable) error counter. This register is 4 bit wide. |
| ERRMAN_SET_SYS_MULTI_ERRCNT_IOCTL | Sets the system flash multiple (un-correctable) error counter. This register is 4 bit wide. |
| ERRMAN_SET_MMU_SINGLE_ERRCNT_IOCTL | Sets the mass memory single (correctable) error counter. This register is 4 bit wide. |
| ERRMAN_SET_MMU_MULTI_ERRCNT_IOCTL | Sets the mass memory multiple (un-correctable) error counter. This register is 4 bit wide. |
| ERRMAN_SET_POWER_LOSS_ENABLE_IOCTL | Sets the power loss enable state. 0 = Power loss detection disabled. 1 = Power loss detection enabled |

5.3.2.3.1. Status register

| Bit position | Name | Direction | Description |
|--------------|-----------------|-----------|---|
| 31:12 | RESERVED | | |
| 11 | ERRMAN_PULSEFLG | R/W | Pulse command flag bit is set. Clear flag by write a '1' |
| 10 | ERRMAN_POWFLG | R/W | The power loss signal has been set. |
| 9 | ERRMAN_MEMCLR | R | The memory cleared signal is set from the scrubber unit function from the memory controller. Set when the memory has been cleared and read by the bootrom to wait for image. |
| 8 | RESERVED | | |
| 7 | ERRMAN_PARFLG | R/W | A previous CPU Register File Parity Error Reset has been detected Clear flag by write a '1' |
| 6 | ERRMAN_MEOTHFLG | R/W | A previous RAM EDAC Multiple Error Reset has been detected for non-critical data Clear flag by write a '1' |
| 5 | ERRMAN_SEOTHFLG | R/W | A previous RAM EDAC Single Error Reset has been detected for critical data Clear flag by write a '1' |
| 4 | ERRMAN_MECRIFLG | R/W | A previous RAM EDAC Multiple Error Reset has been detected for non-critical data Clear flag by write a '1' |
| 3 | ERRMAN_SECRIFLG | R/W | A previous RAM EDAC Single Error Reset has been detected for critical data Clear flag by write a '1' |
| 2 | ERRMAN_WDTFLG | R/W | A previous Watch Dog Timer Reset has been detected Clear flag by write a '1' |
| 1 | ERRMAN_RFLG | R/W | A previous Manual Reset has been detected Clear flag by write a '1' |
| 0 | ERRMAN_IFLAG | R/W | Error Manager Interrupt Flag (multiple sources i.e. read the whole status register) Read: '0' – No interrupt pending '1' – Interrupt pending Write: '0' – Ignored '1' – Clear |

5.3.2.3.2. Carry flag register

| Bit position | Name | Direction | Description |
|--------------|----------------|-----------|---|
| 31:8 | 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 write '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 write '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 write '1') |
| 4 | ERRMAN_MECFLG | R/W | Carry flag set when RAM EDAC Multiple Error counter overflow has occurred '0' – No CF set '1' – Counter overflow (Cleared by write '1') |
| 3 | ERRMAN_SECFLG | R/W | Carry flag set when RAM EDAC Single Error counter overflow has occurred '0' – No CF set '1' – Counter overflow (Cleared by write '1') |
| 2 | ERRMAN_WDTCFLG | R/W | Carry flag set when Watch Dog Timer counter overflow has occurred '0' – No CF set '1' – Counter overflow (Cleared by write '1') |
| 1 | ERRMAN_RFCFLG | R/W | Carry flag set when Manual Reset counter overflow has occurred '0' – No CF set '1' – Counter overflow (Cleared by write '1') |
| 0 | RESERVED | - | |

5.3.2.3.3. 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 |

5.3.2.3.4. 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 |
|-----|-----------------|-----|---|

| Return value | Description |
|---------------------|---------------------------------|
| 0 | Command executed successfully |
| -1 | See <i>errno</i> values |
| errno values | |
| RTEMS_NOT_DEFINED | Invalid IOCTL |
| INVAL | Invalid value supplied to IOCTL |

5.3.3. Usage

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

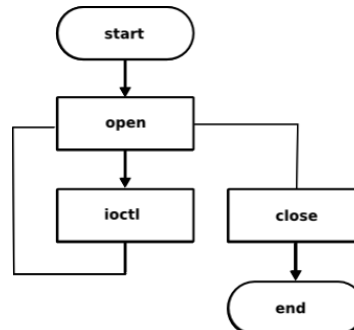


Figure 5-3 - RTEMS driver usage description

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 power loss and 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_POWER_LOSS | A power loss has been detected |
| ERRMAN_IRQ_EDAC_MULTIPLE_ERR_OTHER | Multiple EDAC errors that are not critical have been detected |

5.3.3.2. RTEMS application example

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

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

#define
CONFIGURE_APPLICATION_NEEDS_ERROR_MANAGER_DRIVER

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

#define CONFIGURE_INIT
rtems_task Init (rtems_task_argument argument);

rtems_task Init (rtems_task_argument 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/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 μ s.

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).

5.4.3. Pulse-Per-Second (PPS) signals

The SCET block is designed to be included in several different units in a system and for time synchronization between these SCETs; each SCET has the ability to 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.

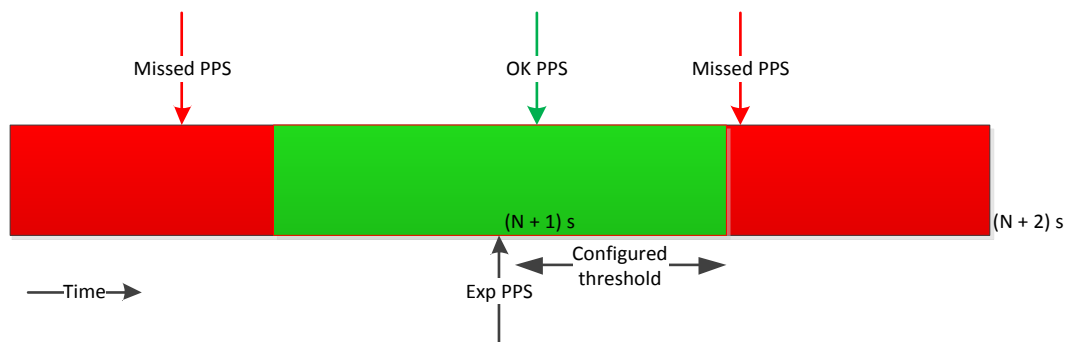


Figure 4 PPS Threshold configuration

To differentiate between the uses of the PPS signal synchronization methods, the SCET can be said to operate in a number of different modes: Free-running, Master, Master with time synchronization and Slave. Please see the explanations below and 0 for an implementation description.

5.4.3.1. 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.

5.4.3.2. 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.

5.4.3.3. 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.

5.4.3.4. 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 isn't specifically affecting the SCET counter registers.

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

5.4.4.1. 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 | 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, etc.). |

| Return value | Description |
|---------------------|---|
| >0 | A file descriptor for the device on success |
| -1 | see <i>errno</i> 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 <i>errno</i> 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(...)`

Adjusts the SCET value's second and subsecond counters using two's complement difference values.

| 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 adjustment difference values are stored. The first four bytes are the difference value for the seconds and the last two bytes are the difference value for the subseconds. |
| 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 <i>errno</i> 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 isn't specifically aimed at reading and/or writing the SCET time value.

| 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_O_EN_IOCTL | Input value configures if pps0 or pps1 is input and if pps1 is input or output. 0 = pps1 is input (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 (default) 1 = pps0 is input, pps1 is output |
| 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 | <p>Returns the currently configured level of the all GP triggers as a bit field:</p> <p>Bit 0 is trigger 0, Bit N is trigger N, Bit 7 is trigger 7.</p> <p>Bit value 0 = trigger activates on 0 to 1 transition (rising edge)</p> <p>Bit value 1 = trigger activates on 1 to 0 transition (falling edge).</p> <p>(0 is default).</p> |
| SCET_SET_GP_TRIGGER_ENABLE_IOCTL | <p>Input bit field selects which GP trigger(s) to enable:</p> <p>Bit 0 is trigger 0, Bit N is trigger N, Bit 7 is trigger 7. All triggers are disabled by default (0)</p> |
| SCET_GET_GP_TRIGGER_ENABLE_IOCTL | <p>Returns which GP triggers that are enabled.</p> <p>Bit 0 is trigger 0, Bit N is trigger N, Bit 7 is trigger 7.</p> |
| SCET_GET_GP_TRIGGER_COUNTER_IOCTL | <p>Input value selects which GP trigger SCET counter sample to read [0,7].</p> <p>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</p> |

| Return value | Description |
|---------------------|--|
| ≥ 0 | Data returned from get commands, or 0 for success in other cases |
| -1 | See <i>errno</i> values |
| errno values | |
| EBADF | File descriptor not opened for writing |
| EINVAL | Invalid value for command, or invalid command. |

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 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 in 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 $\geq (65535 - \text{threshold})$ to $\leq (65535 + 1 + \text{threshold})$ subseconds after the previous PPS.

If the PPS threshold is configured to 0 (min value) only incoming PPS that arrive within \geq 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. A single subscriber is allowed for each queue.

'S', 'P', 'P', 'S' handles PPS related messages with a prefix of:
SCET_INTERRUPT_STATUS_*

Table 5-2 Driver message queue message types

| Event name | Description |
|-------------|------------------------------------|
| PPS_ARRIVED | An external PPS signal has arrived |
| PPS_LOST | The external PPS signal is lost |
| PPS_FOUND | The external PPS signal was found |

'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 General purpose trigger n message queue

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

5.4.5.4. RTEMS application example

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

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

#define CONFIGURE_APPLICATION_NEEDS_SCET_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER

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

#define CONFIGURE_INIT

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

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

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

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

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

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

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



```
secs_to_adjust, subsecs_to_adjust);

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

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 driver is using the de facto standard interface for a 16550D UART given in [RD5] 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 have been changed. In the specification in [RD5], it defines the trigger levels as 1 character, 4 characters, 8 characters and 14 characters. This has now been changed to instead mean 1 character, 1/4 of the FIFO is full, 1/2 of the FIFO is full and the FIFO is 2 characters from the given buffer depth top. This results in the IP being fully backwards compatible, since a buffer depth of 16 characters would yield the same trigger levels as those given in [RD5].

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.

| Argument name | Type | Direction | Description |
|---------------|--------------|-----------|--|
| pathname | const char * | in | The absolute path to the file that is to be opened. See table below for uart naming. |
| flags | 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 etc). See below. |

| 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 <i>errno</i> values |
| errno values | |
| ENODEV | Device does not exist |
| EALREADY | Device is already open |
| EIO | Failed to obtain internal resource |

| Device name | Description |
|-----------------------|-----------------------------------|
| /dev/uart0 | Ordinary UART, default mode RS422 |
| /dev/uart1 | Ordinary UART, default mode RS422 |
| /dev/uart2 | Ordinary UART, default mode RS422 |
| /dev/uart3 | Ordinary UART, default mode RS422 |
| /dev/uart4 | Ordinary UART, default mode RS422 |
| /dev/uart5 | Ordinary UART, default mode RS422 |
| /dev/uart_psu_control | PSU Control, RS485 only |
| /dev/uart_safe_bus | Safe bus, RS485 only |

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.
Please note that it is not uncommon for the read call to 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 / overflow error occurred. The RX data path has been flushed. Data was lost. |
| - 1 | see <i>errno</i> values |
| errno values | |
| EIO | Failed to get internal resource |

5.5.2.4. Function ssize_t write(...)

Write data to the UART. The write call is blocking until all data have been transmitted.

| 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 <i>errno</i> values |
| errno values | |
| EIO | Failed to get internal resource |

5.5.2.5. Function int ioctl(...)

ioctl allows for toggling the RS422/RS485/Loopback mode and setting the baud rate.
RS422/RS485 mode selection is not applicable for safe bus and power ctrl UARTs.

| 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_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 |

| Return value | Description |
|---------------------|--|
| 0 | Command executed successfully |
| -1 | see <i>errno</i> 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

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

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

#define CONFIGURE_APPLICATION_NEEDS_UART_DRIVER
#define CONFIGURE_SEMAPHORES 40

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

#define CONFIGURE_INIT
rtems_task Init (rtems_task_argument argument);

rtems_task Init (rtems_task_argument ignored){}
```

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

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

Inclusion of `<bsp/uart_rtems.h>` is required for accessing the uarts.

5.5.3.2. Parity, framing and overrun error notification

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

5.5.4. Limitations

8 data bits only.

1 stop bit only.

No hardware flow control support.

5.6. Mass memory

5.6.1. Description

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

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 mass memory.

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

5.6.2.3. 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 | Pointer to buffer in which the EDAC data is to be stored (or NULL if EDAC data is not desired). Not used for write operations. |
| 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 | 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 <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <i>fd</i> is not an open file descriptor |
| ENOENT | Invalid filename |
| EEXIST | Device already opened. |

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 |

5.6.3.2. int close(...)

Closes access to the device.

| Argument name | Type | Direction | Description |
|---------------|------|-----------|---|
| fd | int | in | File descriptor received at <code>open</code> . |

| Return value | Description |
|---------------------|--|
| 0 | Device closed successfully |
| -1 | see <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <i>fd</i> 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 <code>open</code> . |
| offset | off_t | in | Page number. |
| whence | int | in | Must be set to <code>SEEK_SET</code> . |

| Return value | Description |
|---------------------|--|
| offset | Page number |
| - 1 | see <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <i>fd</i> is not an open file descriptor |
| ESPIPE | <i>fd</i> is associated with a pipe, socket or FIFO. |
| EINVAL | <i>whence</i> 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 |
|---------------------|---------------------|-----------|--|
| <code>fd</code> | <code>int</code> | in | File descriptor received at <code>open</code> . |
| <code>buf</code> | <code>void *</code> | in | Character buffer where to store the data |
| <code>nbytes</code> | <code>size_t</code> | in | Number of bytes to read into <u><code>buf</code></u> . |

| Return value | Description |
|---------------------|--|
| <code>>0</code> | Number of bytes that were read. |
| <code>-1</code> | see <i>errno</i> values |
| errno values | |
| <code>EBADF</code> | The file descriptor <code>fd</code> is not an open file descriptor |
| <code>EINVAL</code> | Page offset set in <code>lseek</code> is out of range or <code>nbytes</code> is too large and reaches a page that is out of range. |
| <code>EBUSY</code> | 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 |
|---------------------|----------------------|-----------|---|
| <code>fd</code> | <code>int</code> | in | File descriptor received at <code>open</code> . |
| <code>buf</code> | <code>void *</code> | in | Character buffer to read data from |
| <code>nbytes</code> | <code>ssize_t</code> | in | Number of bytes to write from <u><code>buf</code></u> . |

| Return value | Description |
|---------------------|--|
| >0 | Number of bytes that were written. |
| - 1 | see <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <i>fd</i> is not an open file descriptor |
| EINVAL | Page offset set in lseek is out of range or <i>nbytes</i> 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(...)

Additional supported operations via POSIX Input/Output Control API.

| Argument name | Type | Direction | Description |
|---------------|-----------------|-----------|---|
| fd | int | in | File descriptor received at <code>open</code> . |
| 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.

| 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 <i>errno</i> values |
| errno values | |
| ENODEV | Internal RTEMS error |
| EIO | Internal RTEMS error |

5.6.3.6.1. Reset mass memory device

Resets the mass memory device.

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

5.6.3.6.2. Read status data

Reads the status register value.

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

5.6.3.6.3. Read control status data

Reads the control status register value.

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

5.6.3.6.4. Read EDAC register data

Reads the EDAC register value.

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

5.6.3.6.5. Read ID

Reads the chip IDs

| Command | Value type | Direction | Description |
|--------------------|-----------------|-----------|---|
| MASSMEM_IO_READ_ID | massmem_cid_t.* | out | Pointer to struct in which ID is to be stored, see 5.6.2.1. |

5.6.3.6.6. Erase block

Erases a block

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

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

5.6.3.6.7. Read spare area

Reads the spare area with given data.

| Command | Value type | Direction | Description |
|----------------------------|----------------------------------|-----------|--|
| MASSMEM_IO_READ_SPARE_AREA | massmem_ioctl_spare_area_args_t* | in/out | Pointer to struct with input page number specifier, 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: <ul style="list-style-type: none"> 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. |

5.6.3.7. Write spare area

Writes the given data to the spare area.

| Command | Value type | Direction | Description |
|-----------------------------|----------------------------------|-----------|---|
| MASSMEM_IO_WRITE_SPARE_AREA | massmem_ioctl_spare_area_args_t* | in/out | Pointer to struct with page number specifier, byte offset and pointer to data which is to be written, see 5.6.2.3 |

| Return value | Description |
|--------------|---|
| -EINVAL | Indicates one or more of: <ul style="list-style-type: none"> 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. |

5.6.3.7.1.

5.6.3.7.2. 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. |

5.6.3.7.3. 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: <ul style="list-style-type: none"> 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 |

5.6.4. Usage

5.6.4.1. RTEMS

5.6.4.1.1. Overview

The RTEMS driver accesses the mass memory by the reference a page number. There are MASSMEM_BLOCKS blocks starting from block number 0 and MASSMEM_PAGES_PER_BLOCK pages within each block starting from page 0. Each page is of size MASSMEM_PAGE_SIZE bytes.

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.

5.6.4.1.2. Usage

The RTEMS driver must be opened before it can access the mass memory flash device. Once opened, all provided operations can be used as described in the subchapters 5.6.3.3. to 5.6.3.6. And, if desired, the access can be closed when not needed.

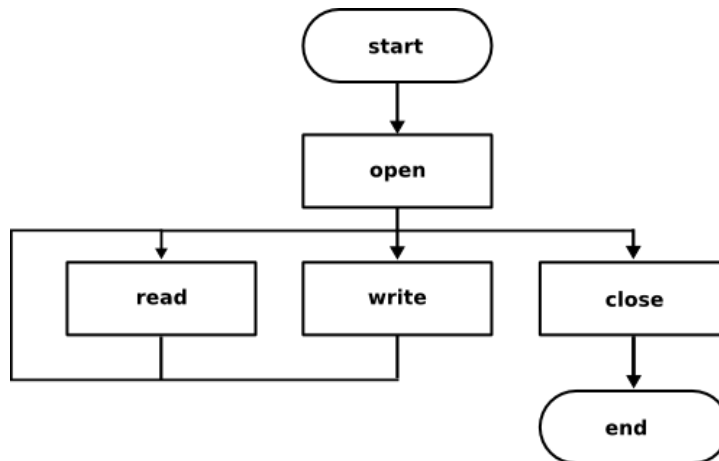


Figure 5-5 - RTEMS driver usage description

Note! All calls to RTEMS driver are blocking calls.

5.6.4.2. RTEMS application example

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

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

#define CONFIGURE_APPLICATION_NEEDS_MASS_MEMORY_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_SIZE];

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

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

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

Inclusion of `<bsp/massmem_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_MASSMEM_FLASH_DRIVER` must be defined for using the 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 eighth 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 registers with the corresponding logic address. 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. The device name must be set with a logical address number 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 <i>errno</i> values |
| errno values | |
| EIO | Internal RTEMS resource error. |
| EALREADY | Device already opened for the requested access mode (read or write). |
| ENOENT | Invalid filename. |

Table 5-5 - 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.

| Argument name | Type | Direction | Description |
|---------------|------|-----------|---|
| fd | int | in | File descriptor received at <code>open</code> . |

| | |
|---------------------|--|
| 0 | Device name deregistered successfully |
| -1 | see <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <i>fd</i> is not and open file descriptor. |

5.7.2.3. size_t read(...)

Reads a packet when available.

Note! This call is blocked until a package for the logic address is received. In addition, only **one** task must access one file descriptor at a time. Multiple task accessing the same file descriptor is not allowed.

Note! buf reference **must** be aligned to a 32 bit aligned address.

| Argument name | Type | Direction | Description |
|---------------|--------|-----------|--|
| fd | int | in | File descriptor received at <code>open</code> . |
| buf | void * | in | Character buffer where to store the packet |
| nbytes | size_t | in | Packet size in bytes. Must be between 0 and <code>SPWN_MAX_PACKET_SIZE</code> bytes. |

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

5.7.2.4. size_t write(...)

Transmits a packet.

Note! This call is blocked till the package is transmitted.

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

| Return value | Description |
|--------------|--|
| >=0 | Number of bytes that were transmitted. |
| <0 | see <i>errno</i> values |

| errno values | |
|--------------|--|
| EBADF | The file descriptor <i>fd</i> is not an open file descriptor. |
| EINVAL | Packet size is larger than <code>SPWN_MAX_PACKET_SIZE</code> . |
| EBUSY | Transmission already in progress. |
| ETIMEDOUT | Failed to transmit the complete packet. |
| EIO | Internal RTEMS resource error, or internal transmission error. |

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 <code>open</code> . |
| cmd | int | in | Command defined in subchapter 0 |
| value | void * | in | The value relating to command operation as defined in subchapter 0. |

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: <ul style="list-style-type: none"> 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 <i>errno</i> values |
| errno values | |

| | |
|--------|---|
| EBADF | The file descriptor <i>fd</i> is not an open file descriptor. |
| EINVAL | Invalid command, or invalid mode value. |

5.7.3. Usage description

5.7.3.1. RTEMS

5.7.3.2. Overview

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 with a logical address.

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 [RD2] and cannot be registered to.

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];
```

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

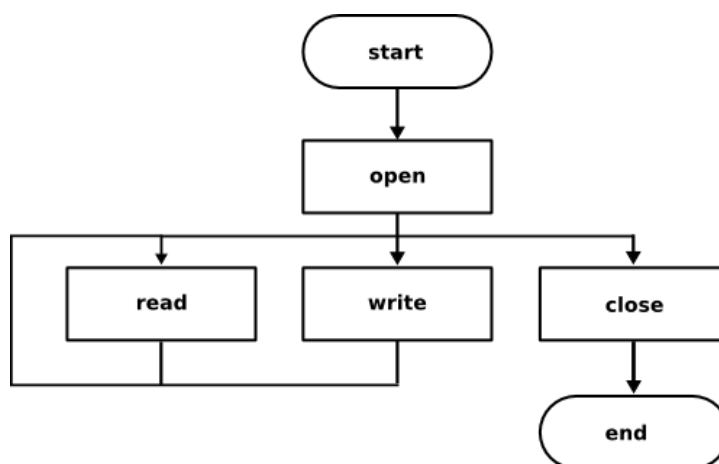


Figure 5-6 – RTEMS driver usage description

Note! All calls to RTEMS driver are blocking calls.

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.

5.7.3.4. RTEMS application example

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

```
#include <bsp.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <bsp/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 [RD7].

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 | Access mode flag, O_RDONLY, O_WRONLY or O_RDWR. |

| Return value | Description |
|---------------------|---|
| Fildes | A file descriptor for the device on success |
| -1 | See <i>errno</i> 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 <i>errno</i> values |
| errno values | |
| EINVAL | Invalid options |

5.8.2.3. Function ssize_t read(...)

Reads the current value of the specified GPIO pin. If no edge detection have been enabled, this call will return immediately. With edge detection enabled, this call will block with a timeout until the pin changes status such that it triggers the edge detection. The timeout can be adjusted using an ioctl command, but defaults to zero - blocking indefinitely, see 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 <i>errno</i> 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 <i>errno</i> 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 <i>errno</i> 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_DESCRIPTOR 30

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 20

#define CONFIGURE_INIT

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

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

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

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

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

Inclusion of `<bsp/gpio_rtems.h>` is required for accessing the GPIO.

5.8.4. Limitations

Differential mode works on output only.

5.9. CCSDS

5.9.1. Description

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

On the telemetry, the frames are encoded with Reed Solomon encoding that conforms to the CCSDS standard with a (255-223) RS 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 driver can be configured to handle all available interrupts from the CCSDS IP:

- Pulse commands (CPDU)
- Timestamping of telemetry sent on virtual channel 0
- 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. In the current configuration, two virtual channels VC0 and VC1 are supported. For telecommands, 64 virtual channels are supported.

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 5.9.5.1 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.

Access to the CCSDS-driver from an application is provided by different device-files:

- “/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-tm0”, “/dev/ccsds-tm1”, ..., “/dev/ccsds-tm6” that are used for sending telemetry on virtual channel 0-6. The names are defined as CCSDS_NAME_TM_VC0, CCSDS_NAME_TM_VC1, ..., CCSDS_NAME_TM_VC6 in RTEMS driver interface file.
- “/dev/ccsds-tc” that is used for configuration and status of the TC path common for all virtual channels. Is defined as CCSDS_NAME_TC in RTEMS driver interface file.
- “/dev/ccsds-tc0” that is used for functions related to handling of Telecommands. Is defined as CCSDS_NAME_TC_VC0 in RTEMS driver interface file.

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.
- Generation of telemetry is disabled.
- The divisor of the TM clock is set to 25.
- 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.

5.9.5.1. Datatype dma_transfer_cb_t

For TM-devices operated in non-blocking mode (see 5.9.2) a message with content below are send 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. Can be 0,...,6 |
| status | uint_8 | Status of transfer 0 – Not send 1 – Send finished 2 – Send error |

5.9.5.2. Data type tm_config_t

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

| Element | Type | Description |
|------------------|----------|---|
| clk_divisor | 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 |

5.9.5.3. 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 |

5.9.5.4. 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 an FIFO error occurred during transmission of data 0 - No Error 1 - FIFO Error |
| tm_busy | uint8_t | Reports if a transfer is in progress. 0 – No transfer 1 – A transfer is in progress |

5.9.5.5. 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 . |

5.9.5.6. 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. |

5.9.5.7. Data type tc_error_cnt_t

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

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

5.9.5.8. 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 |
| tc_carrier | uint16_t | See RD8 |

5.9.5.9. int open(...)

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

| 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 |
| 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 at the end. |
| 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-tm0,..., /dev/ccsds-tm6. |

| Return value | Description |
|---------------------|---|
| ≥0 | A file descriptor for the device on success |
| - 1 | see <i>errno</i> values |
| errno values | |
| EBUSY | If device already opened |

5.9.5.10. 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 |

5.9.5.11. size_t write(...)

To send data on virtual channel 0-6, the device descriptor described 5.9.5 shall be used. TM needs to be enabled to successfully send telemetry. If the device is opened in blocking mode, the write operation will wait until all data has been transferred before returning. For devices opened in blocking mode and data has not been transferred within 1500 msec, the write call is aborted and an error is reported. The timeout value is based on expected time of writing 2^{17} bytes at lowest TM Bitrate. For devices opened in non-blocking mode, the write call returns immediately and the status of the transfer is returned by a message available in a message queue of the driver. See 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 <i>errno</i> 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 |

5.9.5.12. size_t read(...)

To read a Telecommand Transfer frame a read-operation on device "/dev/ccsds-tc0" shall be used. The read Telecommand Transfer frame is passed as a pointer to a variable of type `tc_frame_t`. 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 |
| nbytes | size_t | in | Number of bytes to write from the |

| Return value | Description |
|---------------------|--|
| 0 or greater | number of bytes that were read. |
| - 1 | see <i>errno</i> values |
| errno values | |
| EIO | A read operation is not supported on the device. |

5.9.5.13. 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 for the CCSDS IP 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. |

Sirius OBC and TCM User Manual

| | | | |
|------------------------|---------------|------------|---|
| 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 period of timestamp generation. 0x00 – No time stamping 0x01 – Take a time stamp every time frame sent 0x02 – Take a time stamp every 2 nd time frame sent ... 0xFF – Take a time stamp every 255 th time frame sent |
| CCSDS_GET_TM_TIMESTAMP | /dev/ccsds-tm | uint32_t * | Get period of timestamp generation. |

| Return value | Description |
|--------------|-------------------------------|
| 0 | Command executed successfully |
| -EIO | Unknown IOCTL for device. |

5.9.6. Usage description

5.9.6.1. RTEMS – Send Telemetry

1. Open the device "/dev/ccsds-tm0", "/dev/ccsds-tm1", ..., "/dev/ccsds-tm6", "/dev/ccsds-tc0" 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-tm", "/dev/ccsds-tc0" and "/dev/ccsds". Set up the TC path by ioctl-call CCSDS_SET_TC_CONFIG on device "/dev/ccsds-tc" or or ioctl CCSDS_INIT on device "/dev/ccsds"
2. Do a read from "/dev/ccsds-tc0". 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 data-types, definitions of IOCTL of device CCSDS.

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

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-S FM 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}) in the equations need to be adjusted as well. Note also that the temperature equation require the 3V3 [mV] value.

| HK channel | Formula |
|------------|--|
| Temp [m°C] | Temp_mV = (ADC_value*2500)/(2^24) Temp_mC = (1000*(3V3_mV - Temp_mV) - Temp_mV*1210) / 0.00385*(Temp_mV - 3300) |
| lin [mA] | lin_mA = (ADC_value*5000)/(2^24) |
| Vin [mV] | Vin_mV = (ADC_value*20575)/(2^24) |
| 3V3 [mV] | 3V3_mV = (ADC_value*5000)/(2^24) |
| 2V5 [mV] | 2V5_mV = (ADC_value*5000)/(2^24) |
| 1V2 [mV] | 1V2_mV = (ADC_value*2525)/(2^24) |

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 <i>errno</i> 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 <i>errno</i> 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 bit.

| 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 <i>errno</i> 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 |

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 [RD6]. |
| ADC_GET_SAMPLE_RATE_IOCTL | uint32_t | out | Get the sample rate of the ADC chip, see [RD6]. |
| 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 <i>errno</i> values |

| errno values | |
|-------------------|---------------------------------|
| RTEMS_NOT_DEFINED | Invalid IOCTL |
| EINVAL | Invalid value supplied to IOCTL |

5.10.3. Usage description

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

CONFIGURE_APPLICATION_NEEDS_ADC_DRIVER

5.10.3.1. RTEMS application example

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

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

#define CONFIGURE_APPLICATION_NEEDS_ADC_DRIVER

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

#define CONFIGURE_INIT
rtems_task Init (rtems_task_argument argument);

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

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

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's 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 sub groups 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 NVRAM is divided into a system memory area and a user memory area. System memory area is protected and 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 |
|--------|-------------|-----------|
| System | 0x100 | 0x1FC |
| User | 0x200 | 0x7FFC |

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:

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 <code>SPI_RAM_DEVICE_NAME</code> . |
| flags | int | in | Access mode flag. |

| Return value | Description |
|--------------|---|
| fd | A file descriptor for the device on success |
| -1 | See <i>errno</i> values in [RD13] |

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 <i>errno</i> values in [RD13] |

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 <i>errno</i> EIO. |
| -1 | See <i>errno</i> 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 <i>errno</i> 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 <i>errno</i> values in [RD13] |

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 <i>errno</i> values |
| errno values | |
| EINVAL | Invalid options |
| ENODEV | Internal RTEMS resource error. |

5.11.5. Usage description

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

```
CONFIGURE_APPLICATION_NEEDS_SPI_RAM_DRIVER
```

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

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

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

5.11.5.1. RTEMS application example

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

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

#define CONFIGURE_APPLICATION_NEEDS_SPI_RAM_DRIVER

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

#define CONFIGURE_INIT
rtems_task Init (rtems_task_argument argument);

rtems_task Init (rtems_task_argument argument){
    rtems_status_code status;
    int 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 8.4. 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 | 0: Raw reading off (EDAC and Interleaving active) 1: Raw reading on (EDAC and Interleaving disabled) |
| 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.

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 | 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 <i>errno</i> values |

| errno values | |
|--------------|------------------------|
| ENOENT | Invalid filename |
| EEXIST | Device already opened. |

Table 5-6 - open flag symbols

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

5.12.3.2. Function `int close(...)`

Closes access to the device.

| Argument name | Type | Direction | Description |
|-----------------|------------------|-----------|---|
| <code>fd</code> | <code>int</code> | in | File descriptor received at <code>open</code> . |

| Return value | Description |
|--------------|--|
| 0 | Device closed successfully |
| -1 | see <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <i>fd</i> 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 |
|---------------------|--------------------|-----------|---|
| <code>fd</code> | <code>int</code> | in | File descriptor received at <code>open</code> . |
| <code>offset</code> | <code>off_t</code> | in | Page number. (NOTE: Not bytes!) |
| <code>whence</code> | <code>int</code> | in | Must be set to <code>SEEK_SET</code> . |

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

5.12.3.4. 13Function 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 <code>open</code> . |
| 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 <code>buf</code> (should be a multiple of 4 for most efficient read). |

| Return value | Description |
|--------------|--|
| >0 | Number of bytes that were read. |
| - 1 | see <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <code>fd</code> is not an open file descriptor |
| EINVAL | Page offset set in <code>lseek</code> is out of range or <code>nbytes</code> is too large and reaches a page that is out of range. |
| ENODEV | Semaphore not available. |
| EBUSY | Flash controller busy. |
| EIO | Semaphore error or internal driver error. |

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 <code>open</code> . |
| 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 <code>buf</code> (should be a multiple of 4 for most efficient write). |

| Return value | Description |
|--------------|--|
| >0 | Number of bytes that were written. |
| - 1 | see <i>errno</i> values |
| errno values | |
| EBADF | The file descriptor <code>fd</code> is not an open file descriptor |
| EINVAL | Page offset set in <code>lseek</code> is out of range or <code>nbytes</code> is too large and reaches a page that is out of range. |
| ENODEV | Semaphore not available. |
| EBUSY | Flash controller busy. |

| | |
|-----|---|
| EIO | Semaphore error, internal driver error, or program failed at chip level, block should be considered bad (double check chip status FAIL flag using SYSFLASH_IO_READ_CHIP_STATUS). |
|-----|---|

5.12.3.6. Function int ioctl(...)

Additional supported operations via POSIX Input/Output Control API.

| Argument name | Type | Direction | Description |
|---------------|-----------------|-----------|---|
| fd | int | in | File descriptor received at <code>open</code> . |
| cmd | ioctl_command_t | in | Command specifier |
| value | void * | in | The value relating to command operation as defined in 5.6.3.6.1 to . |

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

| 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. |
| EIO | Internal driver semaphore error. |

5.12.3.6.1. Reset System flash

Resets the system flash chip.

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

5.12.3.6.2. Read chip status

Reads the chip status register.

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

5.12.3.6.3. Read controller status

Reads the controller status register.

| Command | Value type | Direction | Description |
|------------------------------|------------|-----------|---|
| SYSFLASH_IO_READ_CTRL_STATUS | uint16_t * | out | Pointer to variable in which controller status data is to be stored. |

5.12.3.6.4. Read ID

Reads the flash chip ID.

| Command | Value type | Direction | Description |
|---------------------|------------------|-----------|---|
| SYSFLASH_IO_READ_ID | sysflash_cid_t * | out | Pointer to struct in which ID is to be stored, see 5.6.2.1. |

5.12.3.6.5. Erase block

Erases a block.

| Command | Value type | Direction | Description |
|-------------------------|------------|-----------|------------------------|
| SYSFLASH_IO_ERASE_BLOCK | uint32_t | in | Block number to erase. |

| Return value | Description |
|--------------|---|
| 0 | Operation successful. |
| -1 | See errno values. |
| errno values | |
| EINVAL | The block number is out of range |
| EIO | Erase failed on chip level, block should be considered bad. |

5.12.3.6.6. Read spare area

Reads the spare area for a given page.

| Command | Value type | Direction | Description |
|-----------------------------|------------------------------------|-----------|--|
| SYSFLASH_IO_READ_SPARE_AREA | sysflash_ioctl_spare_area_args_t * | in | Pointer to struct with page number specifier, and destination buffers where spare area data is to be stored, see 5.6.2.3 |

| Return value | Description |
|--------------|---|
| 0 | Operation successful. |
| -1 | See errno values. |
| errno values | |
| EINVAL | Page number is out of range or Buffer is NULL |

5.12.3.6.7. Write spare area

Writes the data to the given page spare area.

| Command | Value type | Direction | Description |
|---------|------------|-----------|-------------|
|---------|------------|-----------|-------------|

| | | | |
|------------------------------|------------------------------------|----|---|
| SYSFLASH_IO_WRITE_SPARE_AREA | sysflash_ioctl_spare_area_args_t * | in | Pointer to struct with page number specifier, and source buffer with data which is to be written, see 5.6.2.3 |
|------------------------------|------------------------------------|----|---|

| Return value | Description |
|--------------|---|
| 0 | Operation successful. |
| -1 | See errno values. |
| errno values | |
| EINVAL | Page number is out of range or Buffer is NULL |
| EIO | Program failed on chip level, block should be considered bad. |

5.12.3.6.8. Factory bad block check

Reads the factory bad block marker from a block and reports status.

NOTE: This only gives information about factory marked bad blocks. Bad blocks that arise during use need to be handled by the application software.

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

| Return value | Description |
|--|----------------------------------|
| AAC_SYSFLASH_FACTORY_BAD_BLOCK_CLEARED | Block is OK. |
| AAC_SYSFLASH_FACTORY_BAD_BLOCK_MARKED | Block is marked bad. |
| errno values | |
| EINVAL | The block number is out of range |

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.

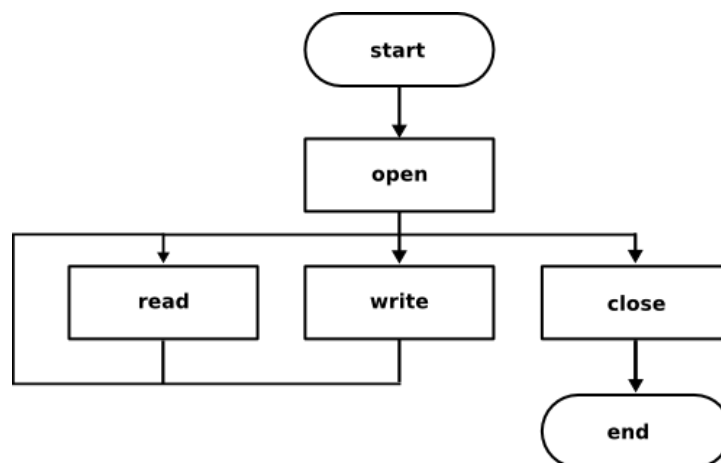


Figure 5-7 - RTEMS driver usage description

5.12.4.3. RTEMS application example

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

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

#define CONFIGURE_APPLICATION_NEEDS_SYSTEM_FLASH_DRIVER
.
.
#define CONFIGURE_INIT
rtems_task Init (rtems_task_argument argument);

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

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

Inclusion of `<fcntl.h>` and `<unistd.h>` are required for using the POSIX functions `open`, `close`, `lseek`, `read`, `write` and `ioctl` 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 smaller router is integrated onto their relative SoCs. The routers all use path addressing (see [RD2]) and given the topology illustrated in Figure 6-1, the routing addressing can be easily calculated.

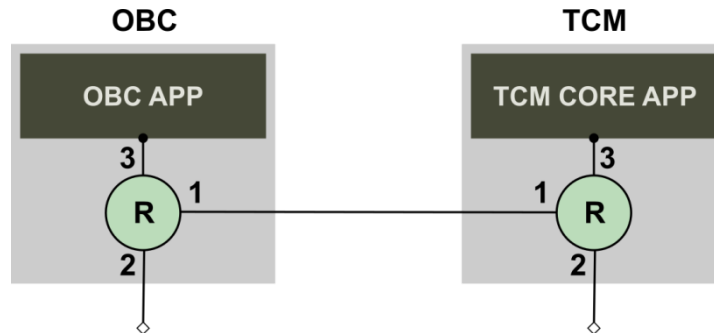


Figure 6-1 Integrated router location

In reference to the topology above, sending a package from the Sirius OBC to the Sirius TCM™ or vice versa, the routing address will be 1-3.

In addition to this, each end node, Sirius OBC or Sirius TCM, 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 of the recipient application/service on the Sirius TCM.

7. Sirius TCM

7.1. Description

The Sirius TCM handles receiving of Telecommands (TCs) and 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 enabled.
- Reed Solomon encoding of telemetry is enabled.
- Convolutional encoding of telemetry is disabled.
- The divisor of the TM clock is set to 25.
- 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 [RD9]

7.2. Block diagram

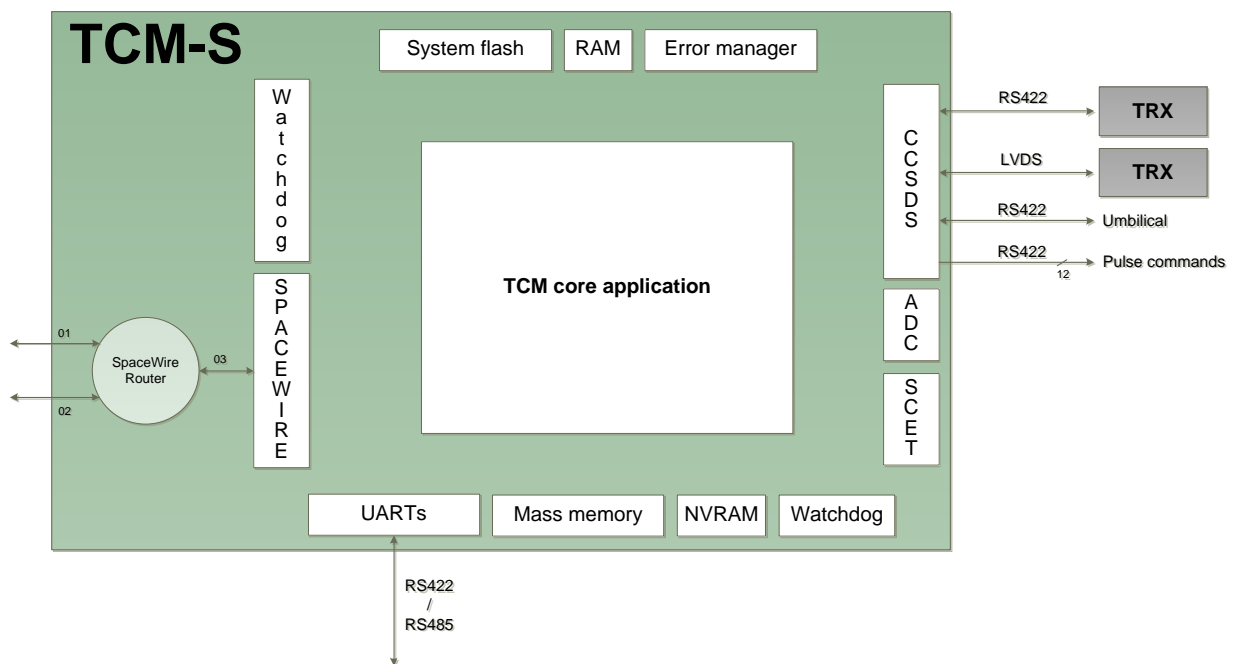


Figure 7-1 – Sirius TCM functionality layout with the external ports depicted

7.3. TCM-S application overview

The TCM-S application is partitioned into several software modules who each handles a specific functional part. An overview of the software architecture of the TCM-S is presented in Figure 7.2. A main design driver of the TCM-S 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 have DMA functionality, off-loading the CPU from mere datashuffling tasks while at the same time increasing performance by at least a 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.

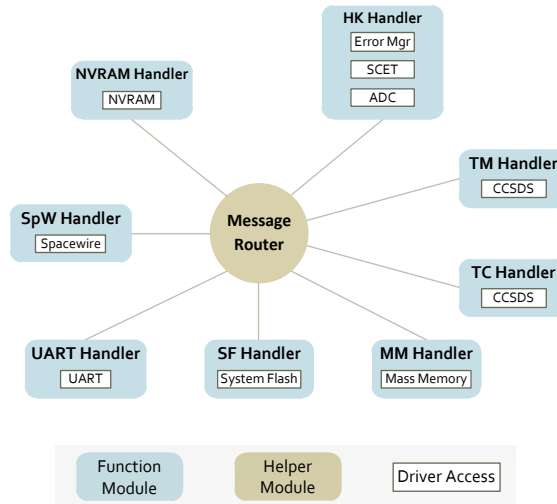


Figure 7.2 TCM-S software application overview

7.4. Configuration

The TCM-S can be configured for specific missions by parameters in NVRAM described in to . The parameters from NVRAM are read during initialization of the TCM-S application. If reading from NVRAM fails, a set of fallback parameters are used instead. The fallback parameters are described in chapter 7.4.1.

Partition configuration of mass memory is specified in .

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) |
| 10 | UINT8 | Specifies which virtual channel (0 or 1) to be used for downloading of the data in the partition. |
| 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 7-4. |

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: 10 = 375000 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 |
| Reserved | UINT8 | Reserved for padding and future use |

Paths on SpW-network are specified by table Table 7-4 below:

Table 7-4: NVRAM SpW path storage

| Data | Type | Description |
|------|----------------|--|
| Path | Array of UINT8 | A path on SpW network including the logic address of the receiving node. |

Telecommand can be routed to nodes on the SpW by APID as specified in Table 7-5 below:

Table 7-5: 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: 0 – Ext. APID, 1 – TCM-S APID Bit13:11 Not used Bit10:0 – APID |
| 2-3 | UINT16 | Upper APID in APID range Bit15: 0 – Single APID Routing, 1 – APID range Bit14: 0 – Ext. APID, 1 – TCM-S APID Bit13:11 – Not used Bit10:0 - APID |
| 4-5 | UINT16 | The index of the SpW-path of the APID. See Table 7-4. |
| 6-7 | UINT16 | Reserved for future use and padding. |

Configuration of the TM path is described in below:

Table 7-6: TM_CONFIG

| Data | Type | Description |
|----------------|--------|---|
| TM Clk divisor | UINT16 | The resulting TM bitrate is $0.5 * \text{System Frequency} / \text{TM Clk divisor}$ |
| TM Config | UINT16 | <p>Configuration of TM path.</p> <p>Bit6: 0 – Disable RS Encoder, 1 – Enable RS Encoder</p> <p>Bit5: 0 – Disable Conv. Encoder, 1 - Enable Conv. Encoder</p> <p>Bit4: 0 – Disable Randomizer, 1 – Enable Randomizer</p> <p>Bit3: 0 – Disable Idle Frames, 1 – Enable Idle Frames</p> <p>Bit2: 0 – Disable MCFC, 1 – Enable MCFC</p> <p>Bit1: 0 – Disable FECF, 1 – Enable FECF</p> <p>Bit0: 0 – Disable CLCW, 1 – Enable CLCW</p> |

Configuration of the TC path is described in below:

Table 7-7: TC_CONFIG

| Data | Type | Description |
|-----------|--------|--|
| TC Config | UINT32 | <p>Configuration of TC path.</p> <p>Bit1: 0 – Disable BCH Decoder, 1 – Enable BCH Decoder</p> <p>Bit0: 0 – Disable Derandomizer, 1 – Enable Derandomizer</p> |

7.4.1. 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-S 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 *****
```

7.5. Telemetry

Telemetry is simultaneously sent on all the transceiver interfaces, i.e. the RS422 (TRX1), the LVDS (TRX2) and umbilical (UMBI) interfaces. VC 0 and VC 1 are supported for TM Data and VC 7 is reserved for idle-frames. 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 not does 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 7.12):

- 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 (sent on VC7 when no data is sent on VC0 or VC1)
- Convolutional encoding
- Pseudo randomization

The TCM-S supports the format of TM Transfer Frames described in [RD8].

7.6. Telecommands

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

- All incoming signals on the inputs are synchronized to the system clock domain.
- 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.
- 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.
- The BCH interface does not “see” the data/clock until the start pattern is decoded correctly and the enable signal is set.

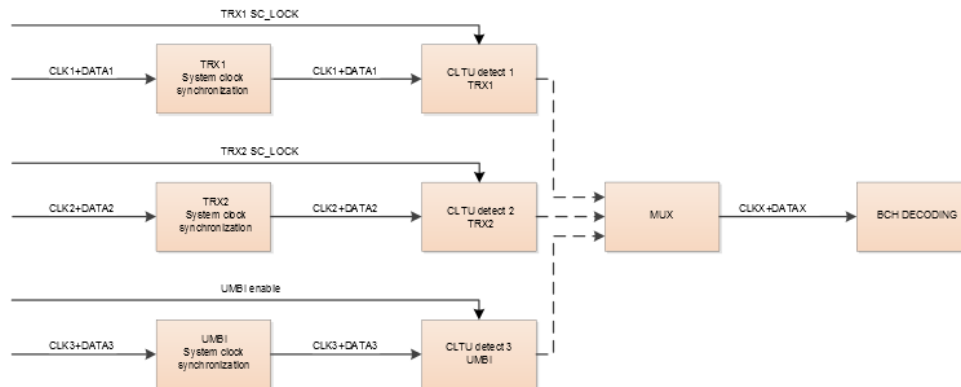


Figure 7.3 – Telecommand Input Multiplexer

Derandomization of TC can be enabled/disabled by RMAP command (see 7.12).
Telecommands sent to the TCM-S must include a segment header, see 4.1.3.2.2 in [RD9]

The TCM-S supports the format of TC Transfer Frames described in [RD9].

7.6.1. 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 e.g. be used to reset modules in a spacecraft and also 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 7.10.2

7.6.2. 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

The TCM-S has an internal SCET timer that can be synchronised to an external time source. In order 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-S will automatically sync SCET subseconds to the external PPS arrival time. A received SCETTime write command can then synchronise the seconds value, see 7.12.4.17.

If the PPS is not stable, the TCM-S 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 is 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-S.

7.7.1. 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 7.12.4.11 and 7.12.4.12 for further info.

7.8. Error Management and System Supervision

The Error Manager in the TCM-S 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 7.12.4.16

The status of the TM Downlink and TC Uplink are available through RMAP. See 7.12.4.14 and 7.12.4.1

A watchdog is enabled in the TCM-S that must be kicked by the TCM-S Application or a reset will occur.

7.9. Mass Memory Handling

The mass memory in the TCM-S 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

The mass memory is accessed through RMAP commands as described in chapters 7.12.4.19 to 7.12.4.26. The mass memory is nandflash-based and that also slightly colours its user interface, even though the detailed handling has been abstracted away. The primary storage units in a flash are the block of 2097152 bytes, i.e. 2 Mbytes, and the page of 16384 bytes, i.e. 16 kbytes, which will be used throughout the document where relevant. The total amount of mass memory available is 16 Gbytes.

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 taken into account 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).

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 7.12.4.24 for details.

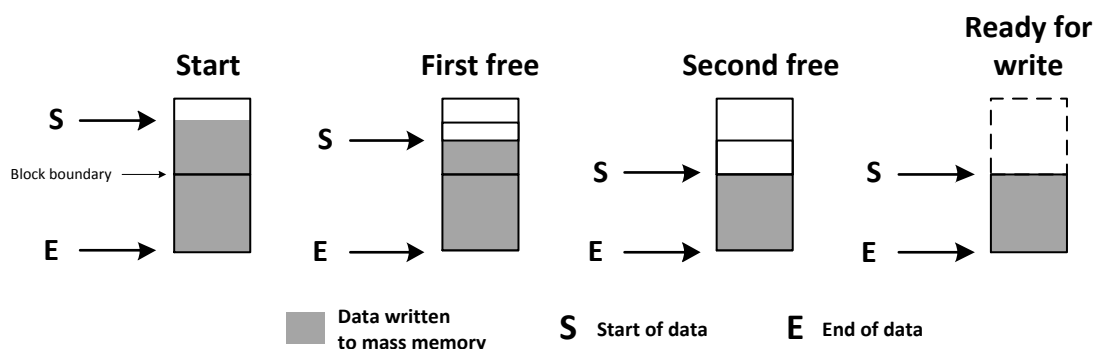


Figure 7.4 Illustration of free behaviour and block boundaries.

Note! Freeing data from a partition will physically erase it, i.e. once data has been freed it is not possible to recover.

7.9.1. Partition configuration

Partitions are configured via the NVRAM configuration tool, according to the format in 7.12.4.21, below follows some detailed information regarding certain configuration items.

7.9.1.1. 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). Obsolete 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 7.9.1.4.

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 7.9.1.4.

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. Read access and download of data is available from any arbitrary address within a partition, given that it has valid data (previously written). Obsolete data or data to overwrite need to be freed here as well, but can be freed on any valid address in the address space.

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

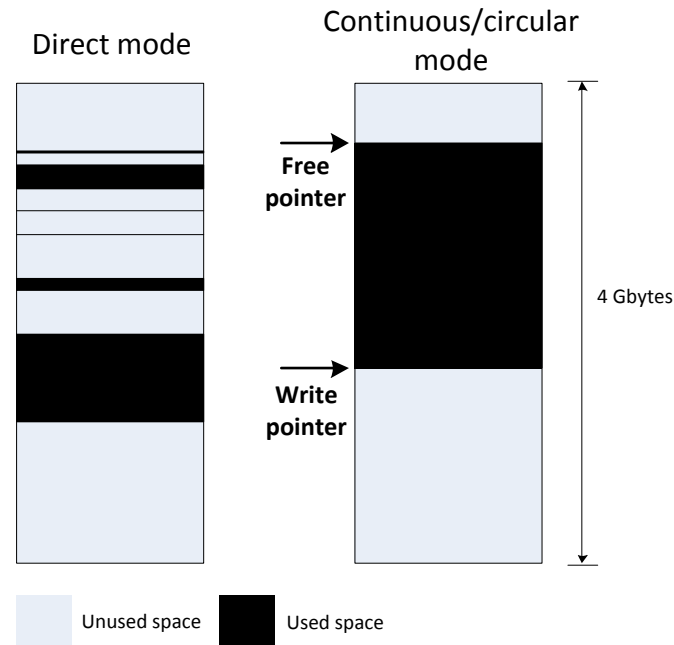


Figure 7.5 Illustration of partition modes and the free/write pointers

7.9.1.2. 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.

7.9.1.3. Partition type

Partitions can be of two types, PUS (see[RD4]) and raw.

Partitions of **type PUS** requires 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).

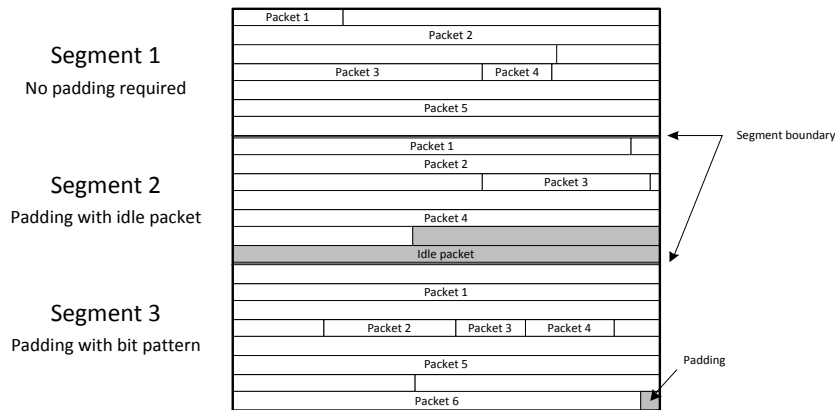


Figure 7.6 Illustration of packet placement inside segments with different padding (marked in grey)

Partitions of **type raw** are currently not supported for download, but imposes no other limitations.

7.9.1.4. 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.

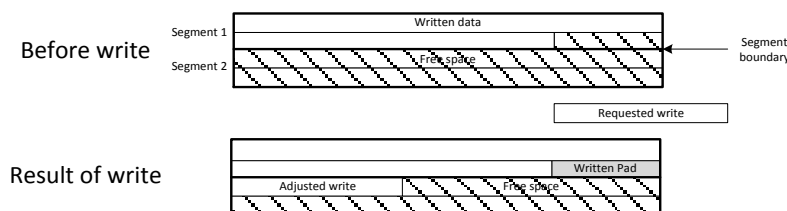


Figure 7-7 Illustration of auto-padding of a requested write.

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 take into account 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.

7.9.1.5. Partition virtual channel

This specifies which CCSDS virtual channel to be used for downloading of the data in the partition. Currently virtual channels 0 and 1 are supported.

7.9.2. Recovery

The mass memory handler utilises the NVRAM to store on-going operation data, and it is used in the initialisation step in order to recover consistency after an 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 in order to ensure that the partition data range is continuous (which is required in order 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.

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

- If not all data was freed from the block.
- 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. ECSS standard services

The TCM-S supports a subset of the services described in [RD4]

7.10.1. PUS-1 Telecommand verification service

The TCM-S 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 [RD4]. 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.

Table 7-8: Telecommand Acceptance Report – Failure (1,2)

| Packet ID | Packet Sequence Control | Code |
|-----------|-------------------------|----------------------------|
| UINT16 | UINT16 | UINT8. 0 – Illegal APID |

7.10.2. PUS-2 Device Command Distribution Service

The TCM-S supports the command pulse distribution unit (CPDU) pulse commands in hardware as defined in 7.2.2 in [RD4].

The CPDU listens on virtual channel 2, APID 2.

It has 12 controllable (0-11) output lines and can be toggled to supply different pulse lengths according to the following scheme:

Table 7-9 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 7-10 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.

7.11. Custom services

7.11.1. 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 9 for further info.

7.12. Spacewire RMAP

According to [RD3], 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-S according to Table 7-13 to separate between configuration commands and mass memory storage of data (partition handling).

The initiator logic address of output messages from the TCM-S, and the RMAP key that needs to be used for input messages and should be expected from output messages, are shown in Table 7-11.

Table 7-11: 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 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 7-12: RMAP predefined fields for routing

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

7.12.1. Input

The RMAP commands supported by the Sirius TCM are specified in the table below. See chapter 7.12.4 for details on each specific command.

Note! The Sirius TCM uses the RMAP Transaction ID to separate between outstanding replies to different units. When several nodes are addressing the Sirius 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 Sirius TCM uses the Transaction ID range 0x0000-0xFFF in all outgoing communication.

Table 7-13: RMAP commands to TCM

| Name | Ext. Addr | Address | Cmd | Description |
|----------|-----------|------------|-----|--------------------------------|
| TMStatus | 0xFF | 0x00000000 | R | Reads latest telemetry status. |

| | | | | |
|-------------------------|-----------|------------|-----|--|
| TMConfig | 0xFF | 0x00000200 | R | Reads telemetry configuration. |
| TMControl | 0xFF | 0x00000300 | W | Enable/Disable telemetry. |
| TMFEControl | 0xFF | 0x00000400 | W | Enable/Disable Frame Error Control Field for TM Transfer Frames. |
| TMMCFControl | 0xFF | 0x00000500 | W | Enable/Disable Master Channel Frame Counter Control for TM Transfer Frames. |
| TMIFControl | 0xFF | 0x00000600 | W | Enable/Disable Idle Frames. |
| TMPRControl | 0xFF | 0x00000700 | W | Enable/Disable Pseudo Randomization for telemetry. |
| TMCEControl | 0xFF | 0x00000800 | W | Enable/Disable Convolutional Encoding for telemetry. |
| TMBRControl | 0xFF | 0x00000900 | W | Sets telemetry clock frequency divisor. |
| TMOCFControl | 0xFF | 0x00000A00 | W | Enable/Disable inclusion of Operational Control field in TM Frames. |
| TMTSControl | 0xFF | 0x00000B00 | R/W | Configures Timestamp of telemetry. |
| TMTSStatus | 0xFF | 0x00000C00 | R | Latest timestamp of telemetry on virtual channel 0. |
| TMSend | 0xFF | 0x00001000 | W | Sends telemetry on virtual channel 0. |
| TCStatus | 0xFF | 0x01000000 | R | Reads latest telecommand status. |
| TCDRControl | 0xFF | 0x01000100 | W | Enables/Disables Derandomizer of telecommands. |
| HKData | 0xFF | 0x02000000 | R | Reads housekeeping data. |
| SCETTime | 0xFF | 0x02000100 | R/W | Reads/Sets SCET time. |
| UARTCommand | 0xFF | 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. |
| MMDData | 0x00-0x0F | 0xn timer | 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. |
| MMDDataRange | 0xFF | 0x0500010n | R | Address ranges of all stored data in partition n. |
| MMPartitionConfig | 0xFF | 0x0500030n | R | Configuration of partition n. |
| MMPartitionSpace | 0xFF | 0x0500040n | R | Space available in partition n. |
| MMDownloadPartitionData | 0xFF | 0x0500050n | W | Downloads partition n data via telemetry. |
| MMFree | 0xFF | 0x0500060n | W | Frees memory from partition n. |
| MMDownloadStatus | 0xFF | 0x0500070n | R | Amount of data downloaded in partition n. |

7.12.2. Output

The TCM-S publishes data to other nodes according to the address map below:

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

Table 7-14: 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 |

7.12.3. Status code in reply messages

In the status field of write/read, the values in Table 7-15 can be returned, this replaces the standard RMAP status codes described in Table 7-15[RD3], see individual commands for specific status code interpretations.

Table 7-15: Status code

| Code | Numeric value |
|----------|---------------|
| - | 0 |
| EIO | 5 |
| EEXIST | 17 |
| EINVAL | 22 |
| ENOSPC | 28 |
| EBADMSG | 77 |
| EALREADY | 120 |

7.12.4. RMAP input address details

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

7.12.4.1. TMStatus

Reads the latest telemetry status.

Table 7-16: TMStatus data

| Byte | Type | Description |
|------|-------|---|
| 0 | UINT8 | 0x00 – No Error 0x01 – FIFO error. |
| 1 | UINT8 | 0x00 – No transfer in progress. 0x01 – Transfer in progress. |

RMAP reply status:

Table 7-17: 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 |

7.12.4.2. TMConfig

Reads the telemetry configuration.

Table 7-18: TMConfig data

| Byte | Type | Description |
|------|--------|---|
| 0-1 | UINT16 | Telemetry clock bitrate divisor value, default 25. |
| 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 7-19: 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 |

7.12.4.3. TMControl

Controls generation of telemetry.

Table 7-20: TMControl data

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

RMAP reply status (if a reply is requested):

Table 7-21: 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 |

7.12.4.4. TMFEControl

Controls Frame Error Control Field inclusion for transfer frames.

Table 7-22: TMFEControl data

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

RMAP reply status (if a reply is requested):

Table 7-23: 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 |

7.12.4.5. TMMCFCCControl

Controls the Master Channel Frame Counter generation for transfer frames.

Table 7-24: TMMCFCCControl data

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

RMAP reply status (if a reply is requested):

Table 7-25: TMMCFCCControl 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 |

7.12.4.6. TMIFControl

Controls the Idle Frame generation for transfer frames.

Table 7-26: TMIFControl data

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

RMAP reply status (if a reply is requested):

Table 7-27: 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 |

7.12.4.7. TMPRControl

Controls the Pseudo Randomization for transfer frames.

Table 7-28: TMPRControl data

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

RMAP reply status (if a reply is requested):

Table 7-29: 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 |

7.12.4.8. TMOCFControl

Controls Operational Control Field inclusion in TM Transfer frames.

Table 7-30: TMOCFControl data

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

RMAP reply status (if a reply is requested):

Table 7-31: TMOFCControl 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 |

7.12.4.9. 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 7-32: TMCEControl data

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

RMAP reply status (if a reply is requested):

Table 7-33: 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 |

7.12.4.10. TMBRControl

Sets the telemetry clock frequency divisor.

The telemetry clock is fed to the radio. The frequency of the telemetry clock is the system clock (50 MHz) divided by the divisor. E.g. if the divisor value is set to 25, the telemetry clock frequency is 2 MHz

Note! If the convolutional encoding is **disabled**, as defined in subchapter 7.12.4.9, the telemetry clock is divided by two, i.e. 1 MHz from example above, to keep the net data rate the same.

Table 7-34: TMBRControl data

| Byte | Type | Description |
|------|--------|---|
| 0-1 | UINT16 | Bitrate divisor value (default 25). Minimum divisor is 4, maximum is 255. |

RMAP reply status (if a reply is requested):

Table 7-35: 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 |

7.12.4.11. TMTSControl

Configures the timestamping for transfer frames.

Table 7-36: TMTSControl data

| Byte | Type | Description |
|------|-------|--|
| 0 | UINT8 | 0x00 – No timestamping (default) 0x01 – Take a timestamp every time frame sent 0x02 – Take a timestamp every 2 nd time frame sent ... 0xFF – Take a timestamp every 255 th time frame sent |

RMAP reply status (if a reply is requested):

Table 7-37: TMTSControl 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. |

7.12.4.12. TMTSStatus

The latest timestamp of telemetry sent on virtual channel 0. Timestamping needs to be enabled before timestamps can be read. See 7.12.4.14.

Table 7-38: 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 7-39: TMTSStatus status codes

| Status code | Description |
|-------------|---|
| 0 | Success. |
| EINVAL | Timestamping is not enabled. See 7.12.4.11 |
| EIO | I/O error. The TM device cannot be accessed |

7.12.4.13. TMSend

Sends telemetry to the TM path on virtual channel 0. The data must contain **at least one** telemetry PUS Packet.

Table 7-40: TMSend data

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

RMAP reply status (if a reply is requested):

Table 7-41: 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 |

7.12.4.14. TCStatus

Reads current telecommand status.

Table 7-42: 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 7-43: 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 |

7.12.4.15. TCDRControl

Configures derandomization for telecommand frames.

Table 7-44: TCDRControl data

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

RMAP reply status (if a reply is requested):

Table 7-45: 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 |

7.12.4.16. HKData

Reads the housekeeping data.

Table 7-46: 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 [m°C] |
| 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 7-47: 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 |

7.12.4.17. 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 7-48: SCETTime data

| Byte | Type | Description |
|------|--------|----------------|
| 0 | UINT32 | SCETSeconds |
| 4 | UINT16 | SCETSubSeconds |

RMAP reply status (if a reply is requested):

Table 7-49: SCETTime status codes

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

7.12.4.18. UARTCommand

Send a command on the specified UART interface.

Table 7-50: UARTCommand data

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

RMAP reply status (if a reply is requested):

Table 7-51: UARTCommand status codes

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

7.12.4.19. MMData

Reads or writes data from/to a partition.

7.12.4.19.1. 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.

7.12.4.19.2. 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). 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.

The data field of the read/write RMAP message in Table 7-52 contains raw data written to or read from the partition.

Table 7-52: MMData data

| Byte | Type | Description |
|--------|----------------|-------------|
| 0 - nn | Array of UINT8 | Data |

RMAP reply status (if a reply is requested):

Table 7-53: 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! <i>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.</i> |
| 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. |
| EEXIST | Write operation to direct mode partition would overwrite existing data. |
| EALREADY | Write to circular partition that is being downloaded. |

7.12.4.20. MMDataRange

This command will return all data address ranges where data is written in this partition, see Table 7-54. 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 exclusive.

Table 7-54: 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 7-55: MMDataRange status codes

| Status code | Description |
|-------------|---------------------------|
| 0 | Success. |
| EINVAL | Invalid partition number. |

7.12.4.21. MMPartitionConfig

Reads the current partition configuration (see 1.1), the RMAP reply message data format is described in Table 7-56.

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

Table 7-56: 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 |
| 9 | UINT8 | Specifies type of data stored on the partition. 0 – PUS Packets 1 – Raw Data (not supported for download) |
| 10 | UINT8 | Specifies which virtual channel (0 or 1) to be used for downloading of the data in the partition. |
| 11 | UINT8 | Segment size for the partition. 1 - 16 kbyte 2 - 32 kbyte 3 - 48 kbyte 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 7-57: MMPartitionConfig data status codes

| Status code | Description |
|-------------|---------------------------|
| 0 | Success. |
| EINVAL | Invalid partition number. |

7.12.4.22. 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 7-58 MMPartitionSpace data

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

RMAP reply status:

Table 7-59: MMPartitionSpace status codes

| Status code | Description |
|-------------|---------------------------|
| 0 | Success. |
| EINVAL | Invalid partition number. |

7.12.4.23. MMDownloadPartitionData

Downloads data of the requested length from the partition using the virtual channel set in the partition configuration (see 1.1). 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 PUS packet length is encountered in a memory segment during download, the rest of the segment will be skipped and the download will continue with 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 1.1 on padding of data.

A download will not automatically free any data.

The RMAP write command data format is described in Table 7-60.

Table 7-60 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 has to be sent without fault for Success to be returned.

Table 7-61 MMDownloadPartitionData data status codes

| Status code | Description |
|-------------|--|
| 0 | Success. |
| ENOSPC | Not enough data on partition. Note! <i>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.</i> |
| 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. |

7.12.4.24. 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.

The RMAP write command data format is described in Table 7-62.

Table 7-62: 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 7-63: 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. |

7.12.4.25. MMDownloadStatus

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

Table 7-64: MMDownloadStatus data

| Byte | Type | Description |
|------|--------|----------------------------|
| 0-7 | UINT64 | Amount of data downloaded. |

RMAP reply status:

Table 7-65: MMFree status codes

| Status code | Description |
|-------------|---|
| 0 | Success. |
| EINVAL | Invalid partition number. |
| EIO | I/O error. Failed to access storage or NVRAM. |

7.12.4.26. MMStopDownloadData

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

RMAP reply status (if a reply is requested):

Table 7-66 MMStopDownload status codes

| Status code | Description |
|-------------|---------------------------|
| 0 | Success. |
| EINVAL | Invalid partition number. |

7.12.5. RMAP output address details

7.12.5.1. TCCommand

A fully formed PUS packet according to [RD4] containing a TC packet to be routed.

7.12.5.2. UARTData

Routed data from UART.

Table 7-67: UARTData data

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

7.13. Limitations

For performance reasons, the current TCM-S 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. System-on-Chip definitions

In this section the peripherals, memory sections and interrupts defined for the SoC for the Sirius OBC and Sirius TCM are described

8.1. Memory mapping

Table 8-1 - Sirius memory structure definition

| Memory Base Address | Function |
|-------------------------|--|
| 0xF0000000 | Boot ROM |
| 0xE0000000 | CCSDS (Sirius TCM only) |
| 0xCB000000 | Watchdog |
| 0xCA000000 | SpaceCraft Elapsed Time |
| 0xC1000000 | SoC info |
| 0xC0000000 | Error Manager |
| 0xBD000000 - 0xBF000000 | Reserved |
| 0xBC000000 | Reserved for SPI interface 1 |
| 0xBB000000 | Reserved for SPI interface 0 |
| 0xBA000000 | GPIO |
| 0xB6000000 | Reserved for ADC controller 1 |
| 0xB5000000 | ADC controller 0 |
| 0xB4000000 | Reserved |
| 0xB3000000 | Mass memory flash controller (Sirius TCM only) |
| 0xB2000000 | System flash controller |
| 0xB1000000 | Reserved |
| 0xB0000000 | NVRAM controller |
| 0xAC000000 | Reserved for PCIe |
| 0xAB000000 | Reserved for CAN |
| 0xAA000000 | Reserved for USB |
| 0xA9000000 - 0xA3000000 | Reserved |
| 0xA2000000 | Reserved for redundant SpaceWire |
| 0xA1000000 | SpaceWire |
| 0xA0000000 | Reserved for Ethernet MAC |
| 0x9C000000 - 0x9F000000 | Reserved |
| 0x9B000000 | Reserved for I2C interface 1 |
| 0x9A000000 | Reserved for I2C interface 0 |
| 0x99000000 | Reserved |
| 0x98000000 | UART 7 (Safe bus functionality, RS485) |
| 0x97000000 | UART 6 (PSU control functionality, RS485) |
| 0x96000000 | Reserved for High speed UART w. DMA |
| 0x95000000 | UART 4 (Routed to LVDS HK on Sirius TCM) |
| 0x94000000 | UART 3 (Routed to RS422 HK on Sirius TCM) |
| 0x93000000 | UART 2 |
| 0x92000000 | UART 1 |
| 0x91000000 | UART 0 |
| 0x90000000 | UART Debug (LVTTTL) |
| 0x80000000 - 0x8F000000 | Reserved for customer IP |
| 0x00000000 | SDRAM memory including EDAC (64 MB) |

8.2. Interrupt sources

The following interrupts are available to the processor:

Table 8-2 - Sirius interrupt assignment

| Interrupt no. | Function | Description |
|---------------|------------------|---|
| 0-1 | Reserved | Internal use |
| 2 | UART Debug | UART interrupt signal |
| 3 | UART 0 | UART interrupt signal |
| 4 | UART 1 | UART interrupt signal |
| 5 | UART 2 | UART interrupt signal |
| 6 | UART 3 | UART interrupt signal |
| 7 | UART 4 | UART interrupt signal |
| 8 | UART 5 | UART interrupt signal |
| 9 | UART 6 | UART interrupt signal |
| 10 | UART 7 | UART interrupt signal |
| 11 | ADC controller 0 | ADC controller 0 interrupt signal |
| 12 | - | Available (reserved for ADC controller 1) |
| 13 | - | Available (reserved for I2C interface 0) |
| 14 | - | Available (reserved for I2C interface 1) |
| 15 | - | Available |
| 16 | - | Available |
| 17 | SCET | SCET interrupt signal |
| 18 | Error manager | Error manager interrupt signal |
| 19 | - | Available (reserved for redundant SpaceWire) |
| 20 | System flash | System flash controller interrupt signal |
| 21 | Mass memory | Mass memory flash controller interrupt signal |
| 22 | Spacewire | SpaceWire interrupt signal |
| 23 | CCSDS | CCSDS interrupt signal |
| 24 | - | Available (reserved for Ethernet) |
| 25 | GPIO | GPIO interrupt signal |
| 26 | - | Available (reserved for SPI 0) |
| 27 | - | Available (reserved for SPI 1) |
| 28 | - | Available (reserved for custom adaptation) |
| 29 | - | Available (reserved for custom adaptation) |
| 30 | - | Available (reserved for custom adaptation) |

8.3. SCET timestamp trigger sources

Some of the peripherals in the SoC have the capability of sending a timestamp trigger signal on specific events. These signals are routed to the SCET which has a number of general purpose trigger registers (GP) where a snapshot of the SCET counter is stored for later retrieval by application software, see chapter 5.4. The tables below detail the mapping between the trigger signals and the general purpose trigger registers in the two products.

Table 8-3 General purpose trigger map

| GP number | Trigger source | Description |
|-----------|----------------|---|
| 0 | power_loss | Triggered when the voltage drops below a certain level, i.e. power is lost to the board |
| 1 | ccsds | Triggered when telemetry sending on virtual channel 0 starts (Sirius TCM only) |
| 2 | gpio | Triggered when one of the pins input changes states and edge detection and timestamping are enabled |
| 3 | adc | Triggered when an ADC conversion is started |

8.4. Boot images and boot procedure

8.4.1. Description

The bootrom is a small piece of software built into a read-only memory inside the System-on-Chip. Its main function is to load a software image from the system flash to RAM and start it by jumping to the reset vector (0x100). To make the system fault tolerant, there are two logical images of the main software, designated Updated and Safe. Each logical image is stored in three physical copies distributed over the system flash. By default the bootrom will first try to load the Updated image and if that fails fall back to the Safe image. The image to load can also be selected by setting the *Next FW* register in the Error Manager and doing a soft reset (see section 5.3 for more details). Boot order of the logical images and their physical copies is shown in Figure 8-1.

8.4.2. Block diagram

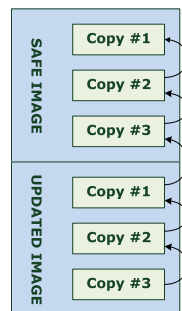


Figure 8-1 Software images in flash

8.4.3. Usage description

The locations in the system flash where the bootrom looks for software images are given in Table 8.4. The first two 32-bit words of the image are expected to be a header with image size and an XOR checksum, see Table 8.5. 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 is on top of the automatic EDAC on all flash data. The EDAC is handled by hardware and calculates one extra byte of redundancy data for each true data byte written to flash.

The bootrom loads a table of bad blocks from the NVRAM. 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.

Table 8.4 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 8.5 Software image header

| Field | Size | Description |
|------------|---------|--|
| Image size | 32 bits | The size in bytes of the software image, not including the header, stored as a 32-bit unsigned integer. A software image can be 264 Bytes – 63 MB. |
| Checksum | 32 bits | A cumulative XOR of all 32-bit words in the image including the size, so that a cumulative XOR of the whole image and header (including checksum) shall evaluate to 0. |

8.4.4. 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.

8.5. Reset behaviour

The SoC has a clock and reset block that synchronizes the external asynchronous reset to each clock domain. The internal soft reset, which can be commanded by software, follows the same design philosophy i.e. is also synchronized into the clock domain where it's used.

8.6. General synchronize method

All signals passing clock domain crossings are either handled via asynchronous two port FIFOs or synchronized into the other clock domain. Two flip-flops in series are used to reduce possible metastability effects. All external signals are synchronized into its clock domain following the above method.

8.7. Pulse command inputs

The pulse command inputs on 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-2 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.

8.8. SoC information map

The information included in the SoC info block for the Sirius products can be found in Table 8-6. This information must be fetched from the gdb prompt and can be used as a check of which SoC version that is flashed on the board. In a connected gdb prompt type:

`x/3xw 0xC1000000`

Table 8-6 Sirius SoC info

| Base address number | Function | Description |
|---------------------|-------------|---|
| 0x0 | TIME_STAMP | When building the SoC, a Unix timestamp is taken and put into the system. It is made as a 32 bit vector indicating seconds since 1970-01-01 (UTC). |
| 0x4 | PRODUCT_ID | <ul style="list-style-type: none"> 0x00 OBC S BB 0x01 OBC SR – With SPW router 3 ports |
| | | <ul style="list-style-type: none"> 0x08 OBC S FM 0x09 OBC SR FM – With SPW router 3 ports |
| | | <ul style="list-style-type: none"> 0x10 TCM S BB 0x11 TCM S R – With SPW router 3 ports |
| | | <ul style="list-style-type: none"> 0x18 TCM S FM 0x19 TCM SR FM – With SPW router 3 ports |
| | | <ul style="list-style-type: none"> 0x20-0xFF Reserved |
| 0x8 | SOC_VERSION | <p>Follows the methodology release 0.1.0 = Release-X.Y.Z, First eight bits are reserved: 0x00XXYYZZ X represents a major number, 8 bits Y represents a minor number, 8 bits Z represents a patch number, 8 bits Representated in 32 bits.</p> <p>Example: 0x00010203 = 1.2.3 Major version 1 Minor version 2 Patch number 3</p> |

9. Software upload

9.1. Description

During the lifetime of a satellite, the on-board software might need adjustments as bugs are detected or the mission parameters adjusted. This module tries to solve that by providing a means for updating the on-board software in orbit. The OBC-S and the TCM-S 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-S, regardless of the end target. The TCM-S simply acts as a router in this case, routing the PUS command to the intended source based on the PUS APID and the TCM-S 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-S itself) using the PUS extension of the CCSDS standard (see [RD4]). 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 2.3 and 2.4 will cover the two middle operations. The first (initial CCSDS handling) and the last (flash operations) are covered in [RD2]. The picture in Figure 9-1 shows the intended control flow when commanding the software update from ground.

9.2. Block diagram

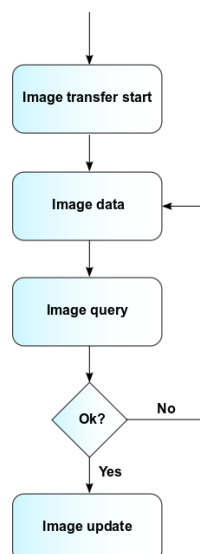


Figure 9-1 The intended software upload command flow

9.3. CCSDS API – custom PUS service 130

This service is provided to allow updates to the flight software on a node in a data handling system using AAC Microtec 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 (1,1/2) and telecommand execution complete as PUS services (1, 7/8) (see [RD4]) if requested in the telecommand PUS header. All reports are sent on the live telemetry virtual channel. 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 Table 9.1 without any parameters (see [RD4]).

Telecommand Execution Completed Report -Failure values are listed under each subtype heading. Errors noted as 'critical' will cause the whole software upload process to be aborted.

Table 9.1 Telecommand acceptance failure error types

| Error code | Data type | Error description |
|------------|-----------|--|
| 0 | UINT8 | Illegal APID (PAC error) |
| 1 | UINT8 | Incomplete or invalid length packet |
| 2 | UINT8 | Incorrect checksum |
| 3 | UINT8 | Illegal packet type |
| 4 | UINT8 | Illegal packet subtype |
| 5 | UINT8 | Illegal or inconsistent application data (unused) |

9.3.1. Subtype 1 – Image transfer start

A telecommand using this subtype has to 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 2.5 below.

Minimum image size is currently 272 bytes including header, and maximum image size is 16 Mbyte.

Table 9.2 Image transfer start command data structure

| Total number of bytes in image | Reserved (zero) | Reserved (zero) |
|--------------------------------|-----------------|-----------------|
| UINT32 | UINT32 | UINT32 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.3 in case of a failure.

Table 9.3 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 |

9.3.2. 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 9.4 below, with the data length given in the PUS header.

Table 9.4 Image data command structure

| Segment number | Segment length | Segment data | | | |
|----------------|----------------|--------------|-------|-------|-----|
| UINT16 | UINT16 | UINT8 | UINT8 | UINT8 | ... |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.5 in case of a failure.

Table 9.5 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 |

9.3.3. Subtype 3 – Verify uploaded image

This subtype calculates and compares the checksum of the uploaded software image with the checksum set in the command's payload data, see Table 9.6

Table 9.6 Verify uploaded image argument

| Checksum |
|----------|
| UINT32 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.7 in case of a failure.

Table 9.7 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 |

9.3.4. Subtype 4 – Write uploaded image

To actually do the updating of the flight image, this command is sent to the service provider which will then write the image to flash. To safe-guard against accidental update commanding, a correct CRC is required as input argument for this command, see Table 9.8

Table 9.8 Write image command argument

| Checksum |
|----------|
| UINT32 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.9 in case of a failure.

Table 9.9 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 |

9.3.5. 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 needs a verification. The telecommand takes the image copy number as argument (max value 6), see Table 9.10. Image copy number 1 - 3 is for the (non-updateable) safe image and 4 - 6 covers the updated image copies.

Table 9.10 Calculate CRC in flash command argument

| Image copy number |
|-------------------|
| UINT8 |

A telecommand execution complete report (if requested in the PUS header) will return the values listed in Table 9.11 in case of a failure.

Table 9.11 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 |

Furthermore, upon execution completed, a report will be generated using the same type and subtype as for the telecommand. This report will contain the calculated checksum, see Table 9.12

Table 9.12 Calculated flash CRC report

| Image copy number | Checksum |
|-------------------|----------|
|-------------------|----------|

| | |
|-------|--------|
| UINT8 | UINT32 |
|-------|--------|

9.4. 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.

9.4.1. 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 |

9.4.2. 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 |
|---------|-----------------------|

9.4.3. 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 not all segments have been added, it will instead return an error code and an array of the ten first missing segments (maximum).

| Argument name | Type | Direction | Description |
|---------------|------------|-----------|--|
| 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 amount of elements in the missing segment array. If only the checksum is of interest this may be a NULL pointer. |

| Return value | Description |
|--------------|--|
| 0 | Success |
| -ENODATA | Not enough data - some data segments missing |
| -ENOENT | No upload in progress |
| -EINVAL | NULL pointer in arguments |

9.4.4. 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 | Description |
|---------------|----------|-----------|--|
| checksum | uint32_t | in | Externally calculated checksum (checked against an internal calculation before update) |

| Return value | Description |
|--------------|--|
| 0 | Success |
| -EINVAL | Checksum argument doesn't match image checksum |
| -EIO | Error when accessing flash |

| | |
|---------|---|
| -ENOSPC | Out of non-bad blocks in one or more of the boot image areas in flash |
| -ENOENT | No upload in progress |

9.4.5. 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 | Description |
|---------------|------------|-----------|--|
| 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 |

9.5. Usage description

A user of the software upload module can either let the module handle all PUS commanding through the PUS API (see section 9.3) or handle all PUS packetizing and reporting internally and only hook into the functional interface described in section 9.4. A code example is provided in the directory `..\src\example`

9.6. Limitations

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

10. Updating the Sirius FPGA

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

10.1. Prerequisite hardware

- Microsemi FlashPro5 unit
- 104470 FPGA programming cable assembly

10.2. Prerequisite software

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

10.3. Generation of encryption key

When AAC Microtec 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.

10.4. 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 connector 4 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 10-1) and select the supplied .pro file.

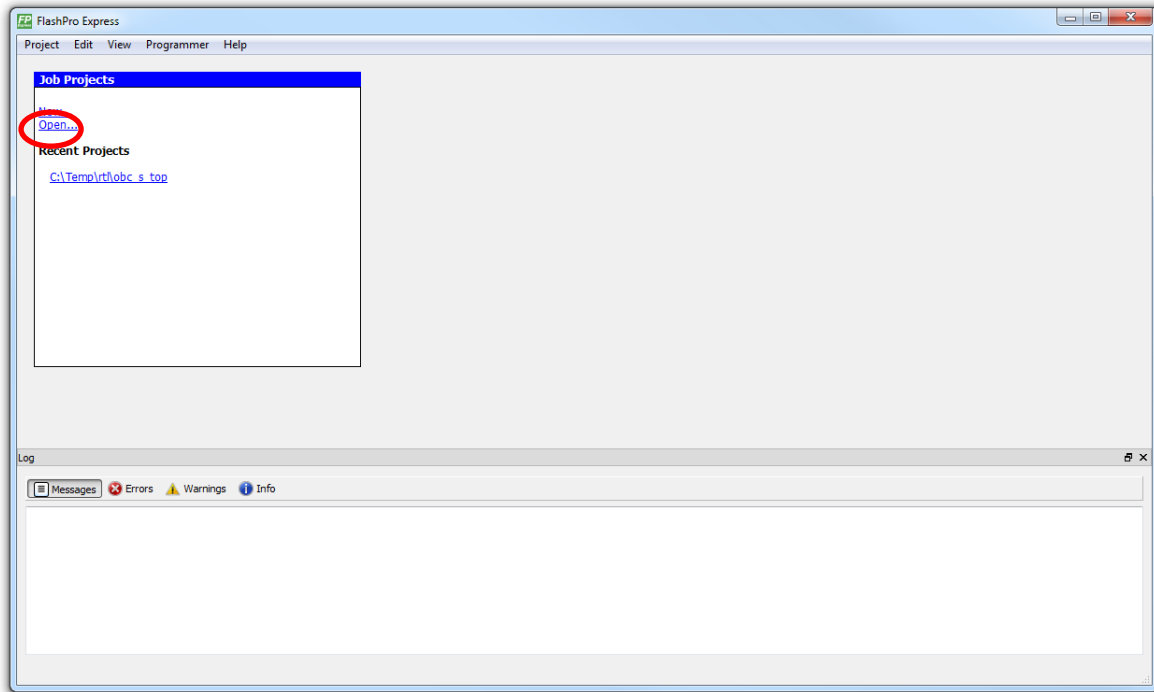


Figure 10-1 - Startup view of FlashPro Express

5. Once the file has loaded (warnings might appear), click RUN (see Figure 10-2). Please note that the connected FlashPro5 programmed ID should be shown.

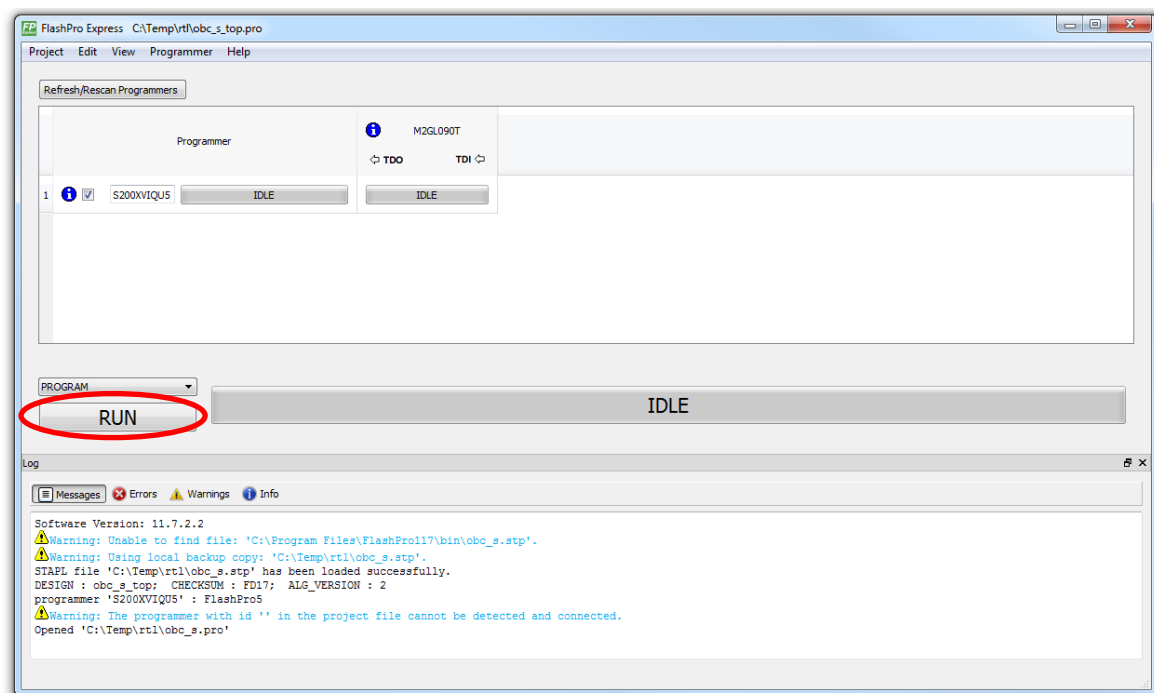


Figure 10-2 - View of FlashPro Express with project loaded.

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 10-3.

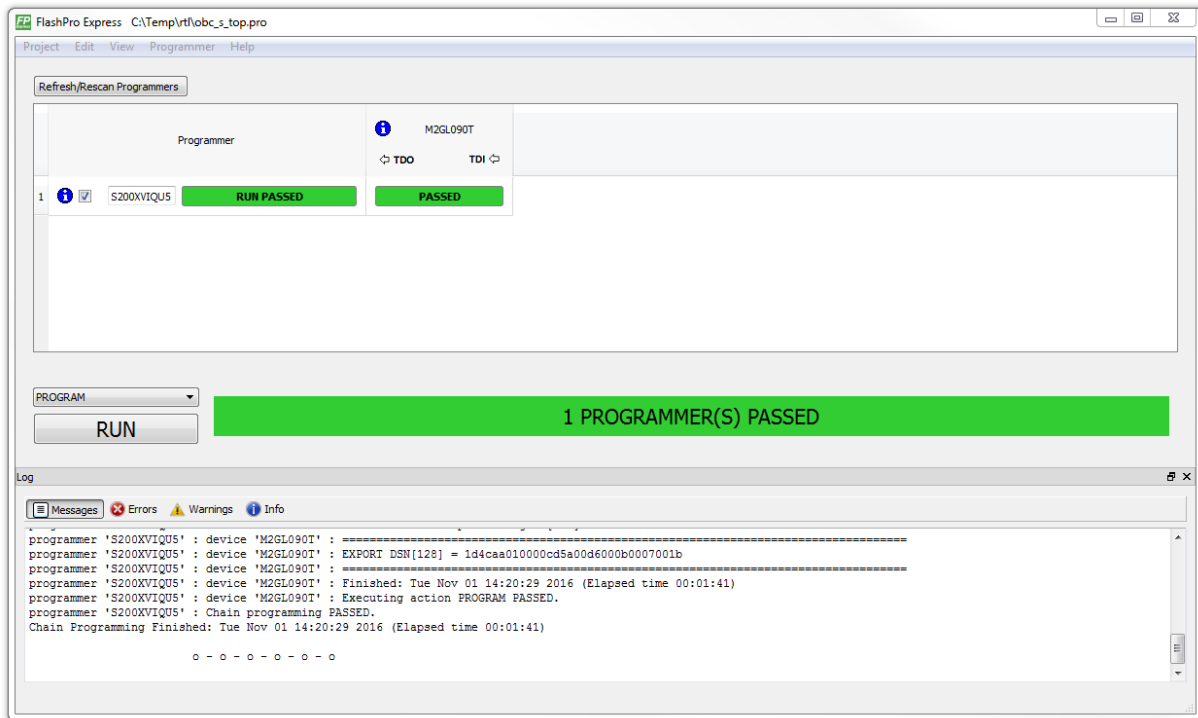


Figure 10-3 - View of FlashPro Express after program passed.

The Sirius FPGA image is now updated

11. Mechanical data

The total size of the Sirius board is 183x136 mm.

Mounting holes are $\varnothing 3.4$ mm with 4.5 mm pad size.

12. Glossary

| | |
|------------------|--|
| ADC | Analog Digital Converter |
| APID | Application Process ID |
| BSP | Board Support Package |
| CCSDS | The Consultative Committee for Space Data Systems |
| EDAC | Error Detection and Correction |
| EM | Engineering model |
| 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 |
| GCC | GNU Compiler Collection program (type of standard in Unix) |
| GPIO | General Purpose Input/Output |
| Gtkterm | Is a terminal emulator that drives serial ports |
| I ² C | Inter-Integrated Circuit, generally referred as “two-wire interface” is a multi-master serial single-ended computer bus invented by Philips. |
| JTAG | Joint Test Action Group, interface for debugging the PCBs |
| LVTTTL | Low-Voltage TTL |
| Minicom | Is a text based modem control and terminal emulation program |
| NA | Not Applicable |
| NVRAM | Non Volatile Random Access Memory |
| OBC | On Board Computer |
| OS | Operating System |
| PCB | Printed Circuit Board |
| PCBA | Printed Circuit Board Assembly |
| POSIX | Portable Operating System Interface |
| PUS | Packet Utilization Standard |
| RAM | Random Access Memory, however modern DRAM has not random access. It is often associated with volatile types of memory |
| ROM | Read Only Memory |
| RTEMS | Real-Time Executive for Multiprocessor Systems |
| SCET | SpaceCraft Elapsed Timer |
| SoC | System-on-Chip |
| SPI | Serial Peripheral Interface Bus is a synchronous serial data link which sometimes is called a 4-wire serial bus. |
| TC | Telecommand |
| TCL | Tool Command Language, a script language |
| TCM | Mass memory |
| TM | Telemetry |
| TTL | Transistor Transistor Logic, digital signal levels used by IC components |
| UART | Universal Asynchronous Receiver Transmitter that translates data between parallel and serial forms. |
| USB | Universal Serial Bus, bus connection for both power and data |



Contact us

ÅAC Microtec AB

Uppsala Science Park
Dag Hammarskjölds väg 48
SE-751 83 Uppsala
Sweden

Phone: +46 18-560130

info@aacmicrotec.com

AAC Microtec North America Inc.

5 Berry Patch Ln Columbia
Illinois 62236,
USA

Phone: +1 (602) 284-7997

info@aacmicrotecus.com

AAC Microtec UK Ltd

Atlas Building
Harwell Campus
Oxfordshire OX11-0QX
UK

Phone: +44 7500 934829

info@aacmicrotec.com

Clyde Space

Skypark 5
45 Finnieston Street
Glasgow G3 8JU
UK

Phone: +44 (0) 141 946 4440

info@aacmicrotec.com