SFSF

0.1

Generated by Doxygen 1.8.14

Contents

1 SmallSat Flight Software Framework

1.1 Introduction

SFSF of (SF)^2 states for SmallSat Flight Software Framework, which is a small software framework aimed to help CubeSat student projects with the development of the flight software. It consists of a bunch of services and functions, which can be used for rapidly developing the flight software. The framework is open source and was developed during 2018 by the GW-CubeSat team, thanks to the cooperation between SETEC Lab and MpNL.

1.1.1 Framework Architecture

The framework itself is composed of two software libraries. One is CubeSat Space Protocol (CSP), which is an open-source library developed by a group of students from Aalborg University, this library offers functions for communication, like sending and receiving messages between satellite and ground stations. The other library is the SFSF Services, which are a bunch of useful functions, which can be used for rapidly developing the flight software for CubeSats. This second library is dependent from CSP, in other words, the SFSF Services require functions from CSP to work, however, CSP can work without the SFSF Services.

A better way to explain this relationship is by calling the libraries as *layers*. So we can stack layers, by placing the less dependent library at the bottom, above this we can place the library that depends on the first library. Following this logic, the framework can be illustrated in the following figure. So if we remove the layer on top, the underlying layer can still prevail. However, if we remove the layer at the bottom, the layer on top cannot prevail, because it is dependent.

To build a CubeSat flight software, it is necessary to include other two software layers, an Operating System (OS) and an Application. CSP is actually no completely independent, in fact, this library depends on an OS, which provides functionalities for creating, synchronizing and communicating tasks. Therefore we can say that the OS layer is located below CSP. Examples of OS are Linux, Windows, FreeRTOS, RETEMS. The framework was designed to work with different OSs, so that the same flight software can be executed in a Desktop computer with Linux, but also in an embedded system with FreeRTOS. This feature is called portability and will be discussed further in section Porting the Framework. This is the reason why the OS is initially not part of the framework, and it has to be chosen and included by the team developing the flight software.

Finally, on top of the three previously mentioned layers, is located the Application layer. This makes use of the functions from the three underlying layers, to conduct the mission according to what is required. The Application is the "smart" section of the flight software because this knows how to respond to commands and perform the mission activities. The Application is unique for each CubeSat mission because each mission has different goals, therefore this section is the responsibility of the team developing the CubeSat mission.

1.1.2 Framework Organization

The framework organization is very simple, each layer is contained in its own directory. The directory *app* contains the code of the Application layer. The directory *libcsp* contains the code of CSP, the network layer. The directory *libsfsf* contains the code of the SFSF services. The directory *docu* contains the required files for generating the documentation with Doxygen. Finally, the directory *examples* contain an application example.

Directory	Content
app	Application Files
libcsp	CubeSat Space Protocol
libsfsf	SFSF Services
docu	Documentation
examples	Example Applicaton

1.1.3 Documentation

Each API file from the SFSF Services (files in the *include* directory), contains its own documentation, as comments in the source code.

A more detailed HTML documentation can be generated using Doxygen. For this download the Doxygen tool. Open the file *Doxyfile* located at the *docu* directory with the tool, then on the tab *Run*, hit the *Run Doxygen* button. Or run Doxygen with the terminal as:

```
doxygen Doxyfile
```

The documentation will be generated into the *docu* directory in a new directory called *html*. Open the file *index.html* with a Web Browser to visualize it.

A PDF file with the same documentation is also included in the *docu* directory, but the HTML version is recommended.

1.1.4 Example

The folder *examples* contains an Application example for Linux, also contains a very basic ground station for being able to receive the telemetry data and sending commands from and to the application. See the *README* file in *examples/linux* for more info.

1.1.5 CSP CubeSat Space Protocol

To implement a communication network across the satellite and ground segment, it is essential to establish a reliable protocol for sending and receiving data. The CubeSat Space Protocol (CSP) was established specifically to meet the needs of CubeSat missions. CSP offers a wide range of functionalities, for sending and receiving messages between satellite and ground stations.

CSP features are very extensive and are not included within the scope of this docum, to learn more about CSP refers to the official distribution on GitHub: https://github.com/libcsp/libcsp.

The official documentation: https://github.com/libcsp/libcsp/tree/master/doc.

The API: https://github.com/libcsp/libcsp/tree/master/include/csp. Is recommended to read the file “csp.h”, it contains some of the most important and common functions for communication.

1.1.5.1 CSP Configuration

CSP requires the file "csp_autoconffig.h" to work, this files contains all the configurations for CSP. Generally the file is not included within the distribution but can be generated with the waf building tool, which is included within the distribution.

1.1.5.2 Examples

Examples of how to use CSP: https://github.com/libcsp/libcsp/tree/master/examples. Is recommended to read and run the “kiss.c” example, to get an idea of CSP.

1.1.6 SFSF Services

The SFSF Services offers a bunch of generic functions and services which can be used for rapidly developing the flight software for CubeSats. The code is organized as follow, the API is located in the directory "include", the implementation in the directory "src". And as with CSP, the SFSF Services require a configuration file to work, this file is sfsf_config.h.

The library consists of seven modules or "services" listed as follow:

Command Service	sfsf_cmd.h
Debug Service	sfsf_debug.h
Housekeeping Service	sfsf_hk.h
Log Service	sfsf_log.h
Parameter Service	sfsf_param.h
Storage Service	sfsf_storage.h
Time Service	sfsf_time.h

Header files from API (include directory) define and describes the functions, structures, and types that can be used. The directory also contains an interface for the hardware called simple_port.h, which defines the functions that need to be implemented by the user, in order to enable all features. The file sfsf.h contains some definitions which are necessary for all services, therefore this file should be included before any other header file from SFSF Services, but user don't need to modify it.

1.1.6.1 SFSF Services Configuration

As mentioned before, the SFSF services require a configuration file to work. Located at the root of the library directory, sfsf_config.h is the only file that the user needs to modify. The file contains configurations that modifies the behavior of each service. Each configuration has it owns description.

1.1.6.2 The Main and execution flow

When using the framework, **the main function becomes part of the framework and not from the user, therefore the user shall not implement** or modify it. The main function is located in the sfsf_main.h file, and this establish the execution flow during the initialization of the software. The functions called by the main are listed as follow, some of them shall be implemented by the user:

1. init_hardware(): first init_hardware() is called, in this function user shall write all the code for initializing the hardware. The body of this function is located in init_functions.c.
2. init_csp(): second init_csp() is called, in this function user shall write all the code and configurations for initializing CSP. See CSP examples. The body of this function is located in init_functions.c.
3. set_up_services(): This is the third function called by the main. In this function, the user shall set the dependencies for the SFSF Services, i.e. the Command Table, Parameter Table. Here is also possible to change the Telemetry collector or Log timestamp generator function if wanted. The body of this function is located in init_functions.c.
4. init_services(): fourth init_services() is called, this is the only function user shall not implement. This function is part form framework and initializes al SFSF Services based on the configurations from sfsf_config.h.
5. CSP_DEFINE_TASK(app_task): the fifth app_task is started, in this function user shall write the endless loop that represents the actual application routine. This routine usually resets the watchdog timers, check for incoming commands and executes the routines for the specific mode of operation for the mission. This task should be defined with the macro CSP_DEFINE_TASK(). The body is located in app_task.c. See `Application`.
6. late_init_routine(): finally late_init_routine() is called. Some platforms (OS and/or Hardware) require to run some routines to make effective previous function calls. For example FreeRTOS require to call vTaskStart↩ Scheduler() to start the tasks. The body of this function is located in init_functions.c.

An example of each one is provided in the example application directory. The execution flow at initialization is illustrated in the following image.

1.1.7 Application

The Application is the where the magic happens. This is the segment of the flight software which choose what to do and when, its unique for each mission because each mission has its own goals. Remeber that before the Application starts, the user should set up the required dependencies for the SFSF Services. For example: set the Parameter and Command Tables, set up the Telemetry collector function, set up the Log Timestamp generator function.

An Application is composed by:

- The Application loop (see app_task.c)
- The Parameter Table (see param_table.h)
- The Command Table (see cmd_table.h)
- The Command Routines (see cmd_routines.c)
- Whatever the user wants to add

There is an example for each one in the examples directory.

The loop is the actual application, it should never end, and some basic routines it may execute, are for example: perform the mission maneuvers, listen for new incoming CSP packages, handle packets whit commands (see command_handler()), reset Watchdog timers (see sfsf_time.h), update the parameter table. This loop can be illustrated as follow:

1.1.8 Porting the Framework

One of the main goals of this framework is to make it easy to execute in different hardware, in other words, easy to port, because all CubeSats missions use different equipment.

1.1.8.1 CSP Port

Clearly a radio system is necessary for establishing the communication network, however, the code for handling the incoming and outgoing messages between the OBC and the radio differs depending on the hardware. Therefore CSP offers interfaces for the hardware, which defines all the functions that the library requires to work. The implementation of the actual functions for the specific hardware should be written by the CubeSat development team.

The directory `drivers`, consist of the interface for the hardware. It is recommended to read the file "usart.h" on this directory to get an idea of which features are expected for the developers to write, for supporting the specific radio system.

CSP also offers an interface for the OS, which defines all the OS functions that the library requires to work, for example creating and synchronizing tasks, and creating queues. The implementation of the actual functions is part of the OS. What "connects" the function's definition from CSP interface with the actual function from the OS, is a so-called *port*. The library already offers ports for running CSP on FreeRTOS, Linux and Windows, however writing a new port for another OS is not difficult. The directory `arch` consist of the interface for the OS. It is recommended to read all files on this directory to get an idea of which features are expected for the OS.

1.1.8.2 SFSF port

Porting the SFSF services is easier, because the library doesn't require a port for the OS as it uses the CSP OS interface. On the other hand, regarding the hardware, the SFSF services require functions for printing in the debug output, writing files and performing system reboots and shutdowns.

Therefore the file sfsf_port.h is provided, this file specifies all the functions and definitions that needs to be implemented in a port, however, the flight software can be compiled and executed without implementing all functions, but the features won't be available. An SFSF port its conformed by a header file with the name "port.h" and a source file (file with ".c" extension), for keeping order each port should be located in its own directory in the "ports/" directory. See the directory "ports", it contains the corresponding ports for the Atmel AVR32UC3C microcontroller and for Linux.

1.1.9 License

SFSF services source code is completely free and open software. Written by students for students. Released under the MIT License. See the LICENSE file for more details.

CubeSat Space Protocol source code is available under an LGPL 2.1 license. See COPYING for the license text.

This work was possible thanks to The Space System Laboratory (SETEC Lab) at Costa Rica Institute of Technology, The GW-CubeSat Lab and The Micropropulsion and Nanotechnology Laboratory (MpNL) at The George Washington University.

By students for students with .

2 Data Structure Documentation

2.1 cmd_handle_t Struct Reference

Struct with the handle of a Command Routine.

Data Fields

- const uint8_t cmd_code
- const uint8_t cmd_args_num
- const cmd_exit_status_t(∗ cmd_routine_p)(csp_conn_t ∗, cmd_packet_t ∗)

2.1.1 Detailed Description

Struct with the handle of a Command Routine.

Contains the command code corresponding to the routine, the number of arguments expected and the pointer to the routine with the command work. The command table is composed by a collection of this struct.

See also

cmd_table_t

2.1.2 Field Documentation

2.1.2.1 cmd_args_num

```
const uint8_t cmd_args_num
```

Expected amount of arguments to receive, if any amount expected use ARGS_NUM_ANNY.

2.1.2.2 cmd_code

```
const uint8_t cmd_code
```

Code of Command.

2.1.2.3 cmd_routine_p

```
const cmd_exit_status_t(* cmd_routine_p) (csp_conn_t *, cmd_packet_t *)
```

Pointer to command routine, use DEFINE_CMD_ROUTINE to define the function.

The documentation for this struct was generated from the following file:

- D:/sfsf/include/sfsf_cmd.h

2.2 cmd_packet_t Struct Reference

Struct with info of a incoming command.

Data Fields

- uint8_t cmd_code
- trigger_type_t trigger_type
- char * cmd_next_arg_p
- char cmd_arg_list [CONF_CSP_BUFF_SIZE-2]

2.2.1 Detailed Description

Struct with info of a incoming command.

Contains the command code, the trigger type for event detection, a pointer to the t argument and a buffer with the arguments separated by comma.

2.2.2 Field Documentation

2.2.2.1 cmd_arg_list

```
char cmd_arg_list[CONF_CSP_BUFF_SIZE-2]
```

Buffer to store arguments for command, use get_next_arg() and rewind_arg_list().

2.2.2.2 cmd_code

```
uint8_t cmd_code
```

Code of Command, should be defined by the user.

2.2.2.3 cmd_next_arg_p

```
char* cmd_next_arg_p
```

Pointer to retrieve arguments one by one, use get_next_arg() and rewind_arg_list().

2.2.2.4 trigger_type

```
trigger_type_t trigger_type
```

Trigger for event detection, see trigger_type_t

The documentation for this struct was generated from the following file:

- D:/sfsf/include/sfsf_cmd.h

2.3 param_t Struct Reference

Parameter struct.

Data Fields

- const char name [CONF_PARAM_NAME_SIZE]
- const param_type_t type
- const uint8_t size
- uint8_t opts
- void * value

2.3.1 Detailed Description

Parameter struct.

Holds the data for a parameter in the table. The Parameter Table is composed by a collection of this struct.

See also

param_table_t

Do not handle this struct directly, use instead param_handle_t type and the API defined functions.

See also

param_handle_t

2.3.2 Field Documentation

2.3.2.1 name

```
const char name[CONF_PARAM_NAME_SIZE]
```

Name of the param.

2.3.2.2 opts

```
uint8_t opts
```

Special options, from param_opts_t.

2.3.2.3 size

```
const uint8_t size
```

Size in bytes, from param_size_t, or custom if STRING_PARAM.

2.3.2.4 type

```
const param_type_t type
```

Param Type, from param_type_t.

2.3.2.5 value

```
void* value
```

Pointer to param space where value is stored.

The documentation for this struct was generated from the following file:

- D:/sfsf/include/sfsf_param.h

3 File Documentation

3.1 D:/sfsf/app/app_task.c File Reference

Application Task.

Functions

- CSP_DEFINE_TASK (app_task)

  Application Task, execute the main application loop.

3.1.1 Detailed Description

Application Task.

Application Task

Application task Template.

3.1.2 Function Documentation

3.1.2.1 CSP_DEFINE_TASK()

```
CSP_DEFINE_TASK (
            app_task  )
```

Application Task, execute the main application loop.

In this function user should write the endless loop that represents the actual application routine. This routine usually resets the watchdog timers, check for incoming commands and executes the routines for the specific mode of operation for the mission.

3.2 D:/sfsf/app/cmd_routines.c File Reference

Command Routines.

3.2.1 Detailed Description

Command Routines.

Command Routines

Command Routines Template.

3.3 D:/sfsf/app/cmd_table.h File Reference

Command Table.

3.3.1 Detailed Description

Command Table.

Command Table

Command Table Template.

3.4 D:/sfsf/app/init_functions.c File Reference

Initialization Functions for hardware and software.

Functions

- void init_hardware (int argc, char **argv)
  Init hardware calls and configurations.
- void set_up_services (void)
  Set Up the SFSF services.
- void late_init_routine (void)
  Last init calls.

3.4.1 Detailed Description

Initialization Functions for hardware and software.

Initialization Functions

Initialization Functions for hardware and software.

3.4.2 Function Documentation

3.4.2.1 init_hardware()

```
void init_hardware (
            int argc,
            char ** argv )
```

Init hardware calls and configurations.

This is the first function to be executed when the software starts. In this function user should write all the code for initializing
the hardware.

3.4.2.2 late_init_routine()

```
void late_init_routine (
            void  )
```

Last init calls.

Some platforms (OS and/or Hardware) require to run some routines to make effective previous function calls. This function is optional.

3.4.2.3 set_up_services()

```
void set_up_services (
            void  )
```

Set Up the SFSF services.

This is the third function called by the main. In this function the user should set the SFSF Services dependencies. For example the Command Table, Parameter Table, or change the Telemetry collector or Log timestamp generator function.

3.5 D:/sfsf/app/param_table.h File Reference

Parameter Table.

Variables

- param_table_t mission_param_table
  Parameter Table.

3.5.1 Detailed Description

Parameter Table.

Parameter Table

This is a template for the Param Table.

3.6 D:/sfsf/include/sfsf.h File Reference

SFSF Framework Private Configurations.

3.6.1 Detailed Description

SFSF Framework Private Configurations.

Private Configurations

This file contains configurations and definitions the the user don't need to modify. However this file should be included in any other where a SFSF Services will be used, hence all services needs this configurations.

3.7 D:/sfsf/include/sfsf_cmd.h File Reference

API for Command Service.

Data Structures

- struct cmd_packet_t

  Struct with info of a incoming command.
- struct cmd_handle_t

  Struct with the handle of a Command Routine.

Macros

- #define ARGS_NUM_ANNY 0xff

  Define a table entry, of command table, that expect any amount of arguments.
- #define DEFINE_CMD_ROUTINE(cmd_routine_name) cmd_exit_status_t cmd_routine_name(csp_conn_↩ t ∗conn, cmd_packet_t ∗ cmd_packet)

  Define functions as Command Routines.

Typedefs

- typedef cmd_exit_status_t(∗ cmd_routine_t) (csp_conn_t ∗conn, cmd_packet_t ∗cmd_packet)

  Type of a command routine.
- typedef const cmd_handle_t cmd_table_t[ ]

  Type to define the command table.

Enumerations

- enum cmd_exit_status_t

  Types of exist status of processing a command.
- enum trigger_type_t {
  ON_REAL_TIME = 1, ON_EQUALS = 2, ON_NOT_EQUALS = 3, ON_LESS = 4,
  ON_LESS_OR_EQUAL = 5, ON_GREATER = 6, ON_GREATER_OR_EQUAL = 7, ON_IN_BETWEEN = 8,
  TRIGGER_TYPE_COUNT }

  Contains the Type of Triggers for Event detection.

Functions

- int init_cmd_queue (void)

  Start Command Queue task, for executing commands based on events.
- int set_cmd_table (cmd_table_t ∗cmd_table, uint16_t cmd_table_size)

  Register the Command Table.
- int decode_cmd_message (csp_packet_t ∗in_csp_packet, cmd_packet_t ∗out_cmd_packet)

  Decodes a CSP packet with a command.
- cmd_exit_status_t command_handler (csp_conn_t ∗conn, csp_packet_t ∗packet)

  Decode and Executes a command packet in a csp_packet_t.
- int count_csv (char ∗str_buff)

  Count amount of comma separated values.
- int get_next_arg (cmd_packet_t ∗cmd_packet, char ∗out_buff)

  Retrieves the next argument in Argument List.
- void rewind_arg_list (cmd_packet_t ∗cmd_packet)

  Rewind the argument list.
- cmd_handle_t ∗ get_cmd_table_entry (uint8_t cmd_code)

  Get a command table entry by the given code.
- csp_thread_handle_t get_cmd_task_handle (void)

  Get Service Task Handle.
- int send_message (csp_conn_t ∗connection, csp_packet_t ∗csp_packet, const char ∗message_buff)

  Send message // TODO decouple.

3.7.1 Detailed Description

API for Command Service.

Command service

Features Summary

- Handles incoming commands.
- Decodes commands contained in CSP packets.
- Definitions to create the command table.
- Executes command routines defined in the command table.
- Enqueue commands for event detection. (Still not implemented)

Module Description

The Command Service is in charge of providing the ground segment with the ability of controlling the spacecraft at any time. This module is able to accept and commands coming from the ground in real-time.

Commands arrive encoded inside packages from the Network layer, therefore this module is also capable of decoding an interpreting the content of a command package.

Normally, ground segment expects a result for each command sent to the spacecraft, therefore this service is capable of capturing the result from any command routine and sending it back to the ground.

Depending on the mission requirements, storing commands is also necessary, this provides the ability to execute commands, even during the section of the orbit without a communication link with the ground. For this, the command packages should also specify when to execute the command routine, this can be specified with an eventual condition, for example, when the time reaches a certain value, or when the GPS coordinates are between a certain range, or even when the battery charge drops under a certain threshold. For simplicity, this framework treats time-based conditions as other event condition, taking the time stamp as the trigger parameter. See trigger_type_t.

The specific action performed by a command will be called command routine. The amount of command routines needed and the actions performed by each command routine are specify of the mission and shall be implemented by the user. This is done by creating a command table with the cmd_handle_t type, and command routines with the DEFINE_CMD_ROUTINE definition.

Command Format

The following table represents the format of the encoded command within an CSP packet.

Cmd Code	Trigger Type	Argument List (Comma Separated)
0	1	3 to CONF_CSP_BUFF_SIZE

1. Cmd code: is the specific code that identifies a command, the code and the corresponding command routine should be specified in the command table (see cmd_table_t type).
2. Trigger Type: specifies when to execute the command, see trigger_type_t. If Trigger Type is ON_REAL_TIME, the command will be executed immediately. Otherwise will be enqueued until the trigger condition occurs. If

Trigger Type is different to ON_REAL_TIME, an on-board parameter name and a value should be specified to trigger the event. In this case, the first Argument from the Argument List is expected to be the on-board parameter name, and the following Argument the value to compare the on-board parameter with. If Trigger Type is ON_IN_BETWEEN, then an additional Argument should be specified. The remaining Arguments on Arguments List will be passed to the specific command execution.

3. Argument List: command routines may require input arguments, for example changing the satellite's pointing direction requires as arguments the new direction. The Argument List contains the arguments for the execution of the command routine. Arguments are contained in a string, each value speared by comma. See get_next_arg() and rewind_arg_list() to retrieve arguments contained in a cmd_packet.

Command Table

The Service needs a way to know which action to execute, when arrives a command. For this the user needs to create a Command Table with the typedef cmd_table_t. The table contains the corresponding code for a command, a trigger_type_t flag that indicate when to execute the command, and a pointer to the corresponding routine which should attend the command, this routine has to be defined with DEFINE_CMD_ROUTINE().

Example:

First define the routines that should attend the commands with DEFINE_CMD_ROUTINE().

```
DEFINE_CMD_ROUTINE(dummy)
{
    // Here the code you want the command to perform
    return CMD_OK;
}
```

Then create the table with cmd_table_t and add the corresponding command code and the pointer to the routine.

```
cmd_table_t mission_cmd_table = {
    //Code of command   Amount of args          Pointer to routine
    {.cmd_code = 1,     .cmd_args_num = 0,      .cmd_routine_p = &dummy},
};
```

Then register the table during initialization, in init_services() function at init_functions.c.

```
set_cmd_table(&mission_cmd_table, sizeof( mission_cmd_table)/sizeof(*mission_cmd_table));
```

Handling Commands

Once with the table and the routines ready. Incoming commands can be executed usin the function command_handler(). This is the key function from the Command Service. When a packet arrives with a command call this function to handle it. This decodes the command from the CSP packet, look-up the Command Table to find the corresponding routine to execute the activity. If command is ON_REALTIME executes the command immediately , otherwise enqueue the command for eventual execution.

The Application task should be in charged of checking for new packages with commands, and when one arrives, to call command_handler().

Command Routines

As stated before command routines are defined by user with the macro DEFINE_CMD_ROUTINE(), and are executed when a command with the correspondign code arrives. Inside this functions the user is allowed to write the desired code to attend the command. For example if the command is take a picture, the routine code should communicate with a camera to take a picture.

Note that the command routines receive two arguments: the CSP connection, and the CSP packet. With the connection the routine is able to send messages back to ground, using the CSP API. And with the CSP packet, the routine is able to retrieve the incoming arguments for performing the required action. For example, if the command is to set a parameter, the packet will contain two arguments: the name of the parameter and the value to set to the parameter. This arguments can be retrieved with get_next_arg().

3.7.2 Macro Definition Documentation

3.7.2.1 DEFINE_CMD_ROUTINE

```
#define DEFINE_CMD_ROUTINE(
              cmd_routine_name ) cmd_exit_status_t cmd_routine_name(csp_conn_t *conn, cmd_packet_t
* cmd_packet)
```

Define functions as Command Routines.

Define Command Routines with this macro, command routines should receive a CSP connection. and a cmd_packet_t, and return a cmd_exit_status_t. Functions defined with this macro are able to send response messages with the CSP connection argument conn, and to retrieve the arguments in the CSP packet cmd_packet with the function get_next_arg().

3.7.3 Typedef Documentation

3.7.3.1 cmd_routine_t

```
cmd_routine_t
```

Type of a command routine.

Use DEFINE_CMD_ROUTINE() to create command routines.

3.7.3.2 cmd_table_t

```
cmd_table_t
```

Type to define the command table.

Note

The table should be registered at init with the function set_cmd_table()

Example:

```
cmd_table_t mission_cmd_table = {
   //Code of command   Amount of args                    Pointer to routine
   {.cmd_code = 1,     .cmd_args_num = 0,                 .cmd_routine_p = &dummy},
   {.cmd_code = 2,     .cmd_args_num = ARGS_NUM_ANNY,     .cmd_routine_p = &cmd_get_param},
   {.cmd_code = 3,     .cmd_args_num = ARGS_NUM_ANNY,     .cmd_routine_p = &cmd_set_param},
   {.cmd_code = 4,     .cmd_args_num = 0,                 .cmd_routine_p = &cmd_reboot_obc}
};
```

3.7.4 Enumeration Type Documentation

3.7.4.1 trigger_type_t

```
enum trigger_type_t
```

Contains the Type of Triggers for Event detection.

Enumerator

ON_REAL_TIME	Immediately
ON_EQUALS	==
ON_NOT_EQUALS	!=
ON_LESS	<
ON_LESS_OR_EQUAL	<=
ON_GREATER	>
ON_GREATER_OR_EQUAL	>=
ON_IN_BETWEEN	< <
TRIGGER_TYPE_COUNT	Keep this always at last position in enum.

3.7.5 Function Documentation

3.7.5.1 command_handler()

```
cmd_exit_status_t command_handler (
            csp_conn_t * conn,
            csp_packet_t * packet )
```

Decode and Executes a command packet in a csp_packet_t.

This is the key function from the Command Service. When a packet arrives with a command call this function to handle it. This decodes the command from the CSP packet, look-up the Command Table to find the corresponding routine to execute the activity. If command is ON_REALTIME executes the command immediately , otherwise enqueue the command for eventual execution.

Warning

This function is not reentrant, not thread-safe, i.e. process one command at the time

Parameters

conn	Pointer to the new connection
packet	Pointer to the packet, obtained by using csp_read()

Returns

cmd_exit_status_t

3.7.5.2 count_csv()

```
int count_csv (
            char * str_buff )
```

Count amount of comma separated values.

Count amount of comma separated values (tokens) in a string, usefully to count the amount of Arguments within a command routine.

Parameters

str_buff	Pointer to the string

Returns

amount of comma separated values, 0 if none

3.7.5.3 decode_cmd_message()

```
int decode_cmd_message (
            csp_packet_t * in_csp_packet,
            cmd_packet_t * out_cmd_packet )
```

Decodes a CSP packet with a command.

Decodes a string from a CSP packet, and creates a cmd-pack struct with the command info This means, extract the command code, the event trigger type and the argument list, also check if this info (command code and amount of arguments) match with info in command table

Parameters

in_csp_packet	CSP Packet with command info
out_cmd_packet	Pointer to a cmd_packet_t to store command info

Returns

-1 if error , 0 if OK

3.7.5.4 get_cmd_table_entry()

```
cmd_handle_t* get_cmd_table_entry (
            uint8_t cmd_code )
```

Get a command table entry by the given code.

Parameters

cmd_code	Command code

Returns

NULL if nor found, cmd_handle_t pointer if OK

3.7.5.5 get_cmd_task_handle()

```
csp_thread_handle_t get_cmd_task_handle (
            void  )
```

Get Service Task Handle.

Returns

csp_thread_handle_t

3.7.5.6 get_next_arg()

```
int get_next_arg (
            cmd_packet_t * cmd_packet,
            char * out_buff )
```

Retrieves the next argument in Argument List.

Store at out_buff the next Argument from cmd_arg_list buff. If all args already retrieved, use rewind_arg_list() to start from the first again.

Parameters

cmd_packet	pointer to cmd_packet with the Argument List
out_buff	Destination buffer to store next arg, should be big enough

Returns

size of retrieved param, 0 if nothing retrieved or end of args

3.7.5.7 init_cmd_queue()

```
int init_cmd_queue (
            void  )
```

Start Command Queue task, for executing commands based on events.

Returns

-1 if error , 0 if OK

3.7.5.8 rewind_arg_list()

```
void rewind_arg_list (
            cmd_packet_t * cmd_packet )
```

Rewind the argument list.

Reset Argument List pointer, in order to be able to retrieve first Argument again with get_next_arg()

Parameters

cmd_packet	Pointer to cmd_packet with Argument List pointer to reset

Returns

void

3.7.5.9 send_message()

```
int send_message (
            csp_conn_t * connection,
            csp_packet_t * csp_packet,
            const char * message_buff )
```

Send message // TODO decouple.

Parameters

connection	CSP Connection
csp_packet	CSP Packet empty
message_buff	Message string to send

Returns

-1 if error , 0 if OK

3.7.5.10 set_cmd_table()

```
int set_cmd_table (
            cmd_table_t * cmd_table,
            uint16_t cmd_table_size )
```

Register the Command Table.

Call this function during initialization, in init_services() function at init_functions.c.

Parameters

cmd_table	Pointer to Command Table
cmd_table_size	Size of command Table

Returns

-1 if error , 0 if OK

3.8 D:/sfsf/include/sfsf_debug.h File Reference

API for Debug Service.

Functions

- void print_debug (const char ∗str)
  Print a string of characters on the debug output.
- void print_debug_char (char c)
  Print a character on the debug output.
- void print_debug_hex (char c)
  Print a char as hex representation on the debug output.
- void print_debug_uint (unsigned int n)
  Prints an unsigned integer on the debug output.

3.8.1 Detailed Description

API for Debug Service.

Debug Service

Features Summary

- Interface for printing information on the debug output.

Module Description

The debug Service is a small interface. It provides functions for printing information on the debug output. It is important to note that the actual implementation for this functions may differ between platforms, therefore is responsibility of the user to port this functions. All the functions should be ported to make functional the debug configurations options from sfsf_config.h See sfsf_port.h.

3.8.2 Function Documentation

3.8.2.1 print_debug()

```
void print_debug (
            const char * str )
```

Print a string of characters on the debug output.

Note

This function should be ported, see sfsf_port.h

Parameters

str	The string to print

3.8.2.2 print_debug_char()

```
void print_debug_char (
            char c )
```

Print a character on the debug output.

Note

This function should be ported, see sfsf_port.h

Parameters

c	The character to print.

3.8.2.3 print_debug_hex()

```
void print_debug_hex (
            char c )
```

Print a char as hex representation on the debug output.

Note

This function should be ported, see sfsf_port.h

Parameters

c	The hex character to print.

3.8.2.4 print_debug_uint()

```
void print_debug_uint (
            unsigned int n )
```

Prints an unsigned integer on the debug output.

Note

This function should be ported, see sfsf_port.h

Parameters

n	The integer to print.

3.9 D:/sfsf/include/sfsf_hk.h File Reference

API for Housekeeping Service.

Typedefs

- typedef void(∗ telemetry_collector_t) (char ∗dest_buf, size_t buf_len)

  Typedef of a Telemetry Data Collector function.

Functions

- int init_hk_service (void)

  Init HK task, collects, stores and broadcasts telemetry data periodically.
- int send_beacon (csp_packet_t ∗beacon_packet)

  Broadcast a CSP packet as a Beacon.
- void set_telemetry_collector (telemetry_collector_t telemetry_collector_p)

  Set the Telemetry Collector function.
- void stop_hk_broadcast (void)

  Stop Beacons broadcasting.
- void resume_hk_broadcast (void)

  Resume Beacons broadcasting.
- void resume_hk_storage (void)

  Resume Beacons storage.
- void stop_hk_storage (void)

  Stop Beacons storage.
- uint32_t get_beacon_count (void)

  Returns the count of Beacons sent.
- csp_thread_handle_t get_hk_task_handle (void)

  Get HK Handle.

Variables

Parameterizable Variables

Use the parateerize() Macro to parameterize this variables into the Parameters Table, this will simplify the control of HK Service, by providing a way to change the behavior . See sfsf_param.h.

Example:

```
param_t param_table[]= {
  ...
  parameterize(  "beacon_count",        UINT32_PARAM,   UINT32_SIZE,
      TELEMETRY|PERSISTENT|READ_ONLY, beacon_counter ),
  parameterize(  "beacon_period",   UINT32_PARAM,   UINT32_SIZE,
      PERSISTENT,                     beacon_period ),
  ...
}
```

- uint32_t beacon_counter
- uint32_t beacon_period
- uint8_t beacon_packet_prio
- uint8_t beacon_dport
- uint8_t beacon_sport
- uint8_t beacon_broadcast_padlock
- uint8_t beacon_storage_padlock

3.9.1 Detailed Description

API for Housekeeping Service.

Housekeeping Service

Features Summary

- Transmits telemetry data periodically.
- Stores telemetry data periodically.

Module Description

The Housekeeping (HK) Service is in charge of providing the ground segment with telemetry data about the state and health of the spacecraft.This service is able to automatically collect, store and transmit telemetry data. Storing telemetry data may be of interest for the mission, if it is required to know the state of the spacecraft even during the section of the orbit without a communication link with ground.

3.9.2 Typedef Documentation

3.9.2.1 telemetry_collector_t

```
telemetry_collector_t
```

Typedef of a Telemetry Data Collector function.

If set, this function will be called automatically by the service to collect telemetry data. It should be set during initialization with set_telemetry_collector().

3.9.3 Function Documentation

3.9.3.1 get_beacon_count()

```
uint32_t get_beacon_count (
            void  )
```

Returns the count of Beacons sent.

Returns

Count of Beacons sent

3.9.3.2 get_hk_task_handle()

```
csp_thread_handle_t get_hk_task_handle (
            void  )
```

Get HK Handle.

Returns

csp_thread_handle_t

3.9.3.3 init_hk_service()

```
int init_hk_service (
            void  )
```

Init HK task, collects, stores and broadcasts telemetry data periodically.

Note

When the Flight Software starts Beacons wont be stored neither broadcasted, this for avoid radio transmissions during deployment of the satellite. App should start Beacons transmission and storage with resume_hk_broadcast() and resume_hk_storage().
For the HK service to collect Telemetry Data automatically a collector function should by set during initialization with set_telemetry_collector().

Returns

-1 if error , 0 if OK

3.9.3.4 send_beacon()

```
int send_beacon (
            csp_packet_t * beacon_packet )
```

Broadcast a CSP packet as a Beacon.

Parameters

beacon_packet	CSP packet with telemetry data to broadcast

Returns

-1 if error , 0 if OK exit status

3.9.3.5 set_telemetry_collector()

```
void set_telemetry_collector (
            telemetry_collector_t telemetry_collector_p )
```

Set the Telemetry Collector function.

Note

collect_telemetry_params() fomr Param Service is situable for this.

See also

sfsf_param.h

Parameters

telemetry_collector_p	Function that collects telemetry data into dest_buf

3.9.4 Variable Documentation

3.9.4.1 beacon_broadcast_padlock

```
uint8_t beacon_broadcast_padlock
```

For pausing a resuming Beacon Transmission. Paused if 0, Resumed if 1.

3.9.4.2 beacon_counter

```
uint32_t beacon_counter
```

Counts the amount of beacons sent.

3.9.4.3 beacon_dport

```
uint8_t beacon_dport
```

CSP Destination Port of Bacon packets.

3.9.4.4 beacon_packet_prio

```
uint8_t beacon_packet_prio
```

CSP Priority of Bacon packets.

3.9.4.5 beacon_period

```
uint32_t beacon_period
```

The period between each beacon transmission.

3.9.4.6 beacon_sport

```
uint8_t beacon_sport
```

CSP Source Port of Bacon packets.

3.9.4.7 beacon_storage_padlock

```
uint8_t beacon_storage_padlock
```

For pausing a resuming Beacon Storage. Paused if 0, Resumed if 1.

3.10 D:/sfsf/include/sfsf_log.h File Reference

API for Log Service.

Typedefs

- typedef size_t(∗ timestamp_generator_t) (char ∗dest_buffer, size_t buff_size)
  Typedef of a Timestamp generator function.

Functions

- int init_log_service (void)
  Init tasks which stores Log messages.
- void set_log_timestamp_generator (timestamp_generator_t timestamp_generator)
  Set the Timestamp generator function.
- int log_print (const char ∗str)
  Pint a string on the Log File.
- int log_print_int (const char ∗name, int value)
  Pint a int value with format "key:value" on the Log File.
- int log_print_float (const char ∗name, float value)
  Print a float value with format "key:value" on the Log File
- csp_thread_handle_t get_log_task_handle (void)
  Get Task Handle.

3.10.1 Detailed Description

API for Log Service.

Log Service

Features Summary

- Store data about the behavior of the spacecraft
- Store data with timestamp.

Module Description

Provides an easy way store data about the behavior of the spacecraft into a file. It provides functions for storing a string message and variable values. Some data that may be valuable to store in the Log file is for example occurrence of events or errors, the value of a variable at a given time, incoming commands and the result. The Log Service can print every message with the timestamp, but a function which provides the timestamp as string should be set with the function set_log_timestamp_generator(), the function get_timestamp_str() from the Time Service can be assigned. If desired to print the Log messages also to the debug output, enable the CONF_LOG_DEBUG.

3.10.2 Typedef Documentation

3.10.2.1 timestamp_generator_t

```
timestamp_generator_t
```

Typedef of a Timestamp generator function.

The function should receive the destination buffer where the timestamp will be stored, and the size of the buffer. It should return the size of the string if success, zero if fails. Set with set_log_timestamp_generator() during initialization.

Note

The function get_timestamp_str() from Time Service meets this requirements.

3.10.3 Function Documentation

3.10.3.1 get_log_task_handle()

```
csp_thread_handle_t get_log_task_handle (
            void  )
```

Get Task Handle.

Returns

csp_thread_handle_t

3.10.3.2 init_log_service()

```
int init_log_service (
            void  )
```

Init tasks which stores Log messages.

Init Log Service, which provides persistence for the messages.

Note

Storage Service functions should be ported, see sfsf_port.h.

Returns

-1 if error , 0 if OK

3.10.3.3 log_print()

```
int log_print (
            const char * str )
```

Pint a string on the Log File.

Parameters

str	String to be printed on Log file

Returns

-1 if error , 0 if OK

3.10.3.4 log_print_float()

```
int log_print_float (
            const char * name,
            float value )
```

Print a float value with format "key:value" on the Log File

Parameters

name	Name of the value, "key"
value	Corresponding value

Returns

-1 if error , 0 if OK

3.10.3.5 log_print_int()

```
int log_print_int (
            const char * name,
            int value )
```

Pint a int value with format "key:value" on the Log File.

Parameters

name	Name of the value, "key"
value	Corresponding value

Returns

-1 if error , 0 if OK

3.10.3.6 set_log_timestamp_generator()

```
void set_log_timestamp_generator (
            timestamp_generator_t timestamp_generator )
```

Set the Timestamp generator function.

If set, log messages will be printed with the Timestamp.

Note

The function get_timestamp_str() from Time Service is situable.

Parameters

timestamp_generator	Function that generates the timestamp into dest_buf

3.11 D:/sfsf/include/sfsf_param.h File Reference

API for Parameter Service.

Data Structures

- struct param_t

  Parameter struct.

Macros

- #define parameterize(variable_name) (void∗)&variable_name

  Macro to parameterize a variable into the Parameters Table.
- #define set_param(handle, type, value) { type aux_var = value; set_param_val(handle, (void∗)&aux_var);}

  Macro for easing setting the value of a Parameters.
- #define get_param(handle, dest_var) { get_param_val(handle, (void∗)&dest_var);}

  Macro for easing getting the value of a Parameters.

Typedefs

- typedef param_t param_table_t[ ]

  Parameters Table Type.
- typedef param_t ∗ param_handle_t

  Parameter Handle Type.
- typedef int16_t param_index_t

  Index to a parameter Type.

Enumerations

- enum param_type_t

  Parameter Types.
- enum param_size_t

  Parameter Sizes.
- enum param_opts_t { TELEMETRY = 0b00000001, PERSISTENT = 0b00000010, READ_ONLY = 0b00000100 }

  Parameter Options.

Functions

- int load_param_table (char ∗file_name)

  Load parameters in a file to the Parameter Table.
- int init_param_persistence (void)

  Init the task that stores parameters in persistent memory.
- int set_param_table (param_table_t ∗param_table, uint16_t param_table_size)

  Set the parameter table.
- param_handle_t get_param_handle_by_name (const char ∗name)

  Get the handle of a Parameter by the name.
- param_index_t get_param_index (const char ∗name)

  Get the index in table of a Parameter by the name.
- param_handle_t get_param_handle_by_index (param_index_t index)

  Get the handle of a Parameter by the index.
- int set_param_val (param_handle_t param_h, void ∗in_p)

  Set the value of a Parameter.
- int get_param_val (param_handle_t param_h, void ∗out_p)

  Get the value of a Parameter.
- int param_to_str (param_handle_t param_handle, char ∗out_buff, int buff_size)

  Store the value of a param as string in a buffer.
- int str_to_param (param_handle_t param_handle, char ∗in_buff)

Set the value of a string to a param.

- uint16_t get_table_size (void)

  Get amount of params in table.

- void collect_telemetry_params (char *dest_buff, size_t buff_size)

  Collect all params with TEMELETRY.

- int collect_telemtry_header (char *dest_buff, int buff_size)

  Collect the references between TAG and params with TELEMETRY.

- void print_pram_table (void)

  Print all params names and values in debug output.

3.11.1 Detailed Description

API for Parameter Service.

Parameter Service

To avoid confusion:

- Variable: variables from programming language, a storage location with an associated name.
- Argument: one of the pieces of data provided as input to a function.
- Parameter: a characteristic that models or describes a system.

Features Summary

- Set and get parameters.
- Supports any type of parameter.
- Definitions to create the parameter table.
- Automatically collects parameters with Telemetry option.
- Automatically stores parameters in persistent memory. (Still not implemented)
- Parameters protection with read only option.

Module Description

The Parameter Service acts as a database for the spacecraft's parameters, provides an easy way to set and get the value of any parameter. A parameter can be described as a variable that helps model or describe a system, it can be for example the pointing direction of the ADCS, or the output voltage of the EPS, or even if the communication system is on or off.

Implementing the mission specific application using the Parameters Service instead of built-in variables from the platform, facilitates the control and monitoring of the spacecraft, because any parameter value can be modified or retrieved at any time. Parameters can be accessed by a name or by an index, this gives the ability to modify or retrieve the value of a parameter from the ground.

The Parameter Table

Parameters are stored in a table, the "Parameters Table". Each entry of the table is of type param_t, which contains the name, type, size, options and a pointer to the value of the parameter. In other words the Parameters Table is an array of param_t.

There can only be one table and should be created by the user, with the type param_table_t. It should be registered at init with the function set_param_table(). The table isstatic and it is created at compilation, this means new parameters can not be added at run time. Therefore, users should include in the table, all required parameters during coding. There are two ways to add parameters to the table:

- When the parameter does not exist, a new memory space can be created to store its value. This is done by adding a param_t struct to the table, as following:

```
{.param_name = "example1", .type = UINT8_PARAM, .size = UINT8_SIZE, .opts =
      TELEMETRY|PERSISTENT },
```

- When the parameter is an existing variable, the variable can be "parameterized" and added to the table. This is done with the macro parameterize(), by adding a line as the following to the table:

```
{.param_name = "example1", .type = UINT8_PARAM, .size = UINT8_SIZE, .opts =
      TELEMETRY|PERSISTENT, parameterize(variable_name) },
```

As shown in the two last examples, parameters require:

- A unique name, the max name size is determined by the configuration CONF_PARAM_NAME_SIZE.
- Type: all supported types are enlisted in enum param_type_t.
- Sze: size can be assigned with the macro sizeof(), or with the values of enum param_size_t. For a parameter of type STRING_PARAM the size can be any, it should be decided by the user.
- Options: there are some special options that can be assigned to parameters, these are enlisted in enum param_opts_t. Options are not mandatory, parameters can have no options. All parameters with option T↩ ELEMTRY can be collected into a string with the function collect_telemetry_params(). All parameters with option PERSITENT will be stored in persistent memory. This is useful for restoring the configuration after a reboot from OBC. Note that init_param_persistence() should be called at init and Storage Service functions should be ported to make effective the PERSOSTENT option. The option READ_ONLY is only applicable for parameters created with the parameterize() macro, this inhibits the parameter value to be modified throughout the Parameter Service API. This option is useful when there are parameters that should not be modified from the ground, but retrieved, like a counter, or the output of a sensor.

An example of a Parameters Table:

```
// Variables to be parameterized
uint32_t    variable_name;
float       variable2_name;
// Parameters Table
param_table_t mission_param_table = {
    //  NAME                    TYPE                Size                Options
       Variable
    {.name="example1",  .type=UINT8_PARAM,  .size=UINT8_SIZE,   .opts=TELEMETRY},
    {.name="example2",  .type=STRING_PARAM, .size=20,           .opts=TELEMETRY|
      PERSISTENT|READ_ONLY},
    {.name="example3",  .type=UINT32_PARAM, .size=UINT32_SIZE,}, // Params may have no options
    {.name="example4",  .type=UINT32_PARAM, .size=UINT32_SIZE,  .opts=TELEMETRY|
      PERSISTENT,             .value=parameterize(variable_name)},
    {.name="example5",  .type=FLOAT_PARAM,  .size=FLOAT_SIZE,                                .
      value=parameterize(variable2_name)}
};
```

Then register the table during initialization, in init_services() function at init_functions.c.

```
set_param_table(&mission_param_table,  sizeof(
      mission_param_table)/sizeof(*mission_param_table));
```

Handling Parameters

Whit a Parameter Table populated and registered, we can retrieve and modify the value of parameters. You may not access the param_t structures from table directly. Use the a Parameter Handle instead, the typedef param↩_handle_t, which is a pointer to a table entry. So once you have a handle pointing to a param, the access to the value is in immediate, there is no need to look-up through the table for the parameter. There are two ways to obtain a handle, by the name of the parameter, or by the index on the table.

The easiest, by the name, with the function get_param_handle_by_name(). This functions scans the table for a parameter with the name. If the parameter is accessed very frequently, is not efficient.

Example:

```
param_handle_t example1_h = get_param_handle_by_name("example1");
```

The other way is by the index in table, using the function get_param_handle_by_index(). This functions is way more efficient that the previous one, because there is not need to scan the table, the access is directly, but you need to keep track of the indexes in table.

Example:

```
#define EXAMPLE1_INDEX  1   // Recommended to keep indexes as macros
param_handle_t example1_h = get_param_handle_by_index(EXAMPLE1_INDEX);
```

Once with the handle of a param, in other words a pointer to it, the acces to its value can be done with two functions: get_param_val() and set_param_val().

To retreive the value of a parameter use get_param_val().

Example:

```
uint8_t aux_var;    // Variable where param value will be stored
param_handle_t example1_h = get_param_handle("example1");
get_param_val(example1_h, (void*)&aux_var );
```

To modify the value of a parameter use set_param_val().

Example:

```
uint8_t example1 = 4;   // Value to set to param
param_handle_t example1_h = get_param_handle("example1");
set_param_val(example1_h, (void*)&example1 );
```

Also you can use the macros get_param() and set_param(), which require less code.

Example:

```
// Set the value, note for set_param() you can place the value as argument
set_param(example1_h, int8_t, -4);

// Get the value
uint8_t example1 = 4;
get_param( example1_h, example1 );
```

The true advantage from the Param Service is the ability to access values by a string, the name. This way the ground segment can modify or retrieve variables in the spacecraft by knowing the name. Incoming and outgoing messages from ground may contain the value of parameters as strings, therefore you can also use the functions str_to_param() and param_to_str().

To set the value of a parameter with a string with the value use str_to_param().

Example:

```
str_to_param(example_handle, "example string");
```

To retrieve the value of a parameter as a string use param_to_str().

Example:

```
char buffer[20];
param_to_str(example_handle, buffer, 20);
```

3.11.2 Macro Definition Documentation

3.11.2.1 get_param

```
#define get_param(
              handle,
              dest_var ) { get_param_val(handle, (void*)&dest_var);}
```

Macro for easing getting the value of a Parameters.

```
Example:
param_handle_t example_handle_uint16 = get_param_handle("example_uint16");  // get handle
uint16_t other_var;
get_param( example_handle_uint16, other_var );              // get value with macro
```

Parameters

handle	Handle of the parameter
dest_var	Destination variable where value will be stored

3.11.2.2 parameterize

```
#define parameterize(
              variable_name ) (void*)&variable_name
```

Macro to parameterize a variable into the Parameters Table.

Use this macro to create parameters into the Parameters Table using a variable. Variable should be visible in the scope where the Table is declared, this can be done with the "extern" keyword.

Example:

```
uint32_t example_var;   // Variable to parameterize
// The Parameter Table
param_table_t param_table= {
    {.name="example",   .type=UINT32_PARAM, .size=UINT32_SIZE,  .opts=
      TELEMETRY .value=parameterize(example_var),
}
```

Parameters

variable_name	Name of the variable to parameterize

3.11.2.3 set_param

```
#define set_param(
              handle,
```

```
            type,
            value ) { type aux_var = value; set_param_val(handle, (void*)&aux_var);}
```

Macro for easing setting the value of a Parameters.

```
//Example:
param_handle_t example_handle_uint16 = get_param_handle("example_uint16");  // get handle
set_param(example_handle_uint16, uint16, 100);                              // Set value with
      macro
```

Parameters

handle	Handle of the parameter
type	C build in type of the value, NOT a param_type_t, (i.e. uin8_t, int32_t, float, etc.)
value	Value or variable with value to be assigned

3.11.3 Typedef Documentation

3.11.3.1 param_handle_t

```
param_handle_t
```

Parameter Handle Type.

Pointer to a parameter struct, i.e: entry in parameters table. Use this type and the API functions for handling parameters.

See also

set_param_val(), get_param_val(), set_param(), get_param()

3.11.3.2 param_index_t

```
param_index_t
```

Index to a parameter Type.

Type to retrieve a parameter handler by the index in table.

3.11.3.3 param_table_t

```
param_table_t
```

Parameters Table Type.

Type to define the parameters table.

Note

The table should be register during initialization with set_param_table()

Example:

```
// Variables to be parameterized
uint32_t    variable_name;
float       variable2_name;
// Parameters Table
param_table_t mission_param_table = {
 // NAME                    TYPE                Size                Options
       Variable
 {.name="example1", .type=UINT8_PARAM,  .size=UINT8_SIZE,    .opts=TELEMETRY},
 {.name="example2", .type=STRING_PARAM, .size=20,            .opts=TELEMETRY|
      PERSISTENT|READ_ONLY},
 {.name="example3", .type=UINT32_PARAM, .size=UINT32_SIZE,}, // Params may have no options
 {.name="example4", .type=UINT32_PARAM, .size=UINT32_SIZE,  .opts=TELEMETRY|
      PERSISTENT,             .value=parameterize(variable_name)},
 {.name="example5", .type=FLOAT_PARAM,  .size=FLOAT_SIZE,                                    .value=
      parameterize(variable2_name)}
};
```

3.11.4 Enumeration Type Documentation

3.11.4.1 param_opts_t

```
enum param_opts_t
```

Parameter Options.

Note

READ_ONLY is only applicable for parameterized variables.

Macros for assigning the options in Parameters Table

Enumerator

TELEMETRY	Automatic collect this param for telemetry.
PERSISTENT	Persist the value on non volatile memory.
READ_ONLY	Prohibited to write with param_service functions, only applicable for parameterized variables.

3.11.4.2 param_size_t

```
enum param_size_t
```

Parameter Sizes.

Macros for assigning the size in Parameters Table, note that STRING_PARAM can have any size, is decision from user to assign the size.

3.11.4.3 param_type_t

```
enum param_type_t
```

Parameter Types.

Supported types for parameters

3.11.5 Function Documentation

3.11.5.1 collect_telemetry_params()

```
void collect_telemetry_params (
            char * dest_buff,
            size_t buff_size )
```

Collect all params with TEMELETRY.

Collects all params with TEMELETRY option and store the value with a tag into dest_buff, if buff is not big enough, not all params will be collected. Format: "TAG:value,TAG:value,TAG:value" Example: "A:123,B:-3,C:0.001234" Where "TAG" is a incremental alphabetical character (A,B,C,...,AA,AB,...) and "value" is the value of the corresponding param. For getting the reference between TAG and param see collect_telemtry_header()

See also

collect_telemtry_header

Parameters

dest_buff	Buffer where telemetry data will be stored
buff_size	Size of dest_buff

3.11.5.2 collect_telemtry_header()

```
int collect_telemtry_header (
            char * dest_buff,
            int buff_size )
```

Collect the references between TAG and params with TELEMETRY.

Collect the references between TAG and params with TELEMETRY option into dest_buff, if buff is not big enough, not all params will be collected. Format: "TAG:param_name,TAG:param_name,TAG:param_name" Example: "↩ A:reset_cause,B:temperature,C:gps_lat" Where "TAG" is a incremental alphabetical character (A,B,C,...,AA,AB,...) and "param_name" is the name of the param assigned in the params table.

Parameters

dest_buff	Buffer where telemetry data will be stored
buff_size	Size of dest_buff

Returns

-1 if error, size of string if OK

3.11.5.3 get_param_handle_by_index()

```
param_handle_t get_param_handle_by_index (
            param_index_t index )
```

Get the handle of a Parameter by the index.

Parameters

index	Index in table of the parameter

Returns

handle to param if OK, NULL if error (not found)

3.11.5.4 get_param_handle_by_name()

```
param_handle_t get_param_handle_by_name (
            const char * name )
```

Get the handle of a Parameter by the name.

Parameters

name	Name of the parameter

Returns

handle to param if OK, NULL if error or not found

3.11.5.5 get_param_index()

```
param_index_t get_param_index (
            const char * name )
```

Get the index in table of a Parameter by the name.

Parameters

name	Name of the parameter

Returns

index to param if OK, -1 if error (not found)

3.11.5.6 get_param_val()

```
int get_param_val (
            param_handle_t param_h,
            void * out_p )
```

Get the value of a Parameter.

Parameters

param_h	Handle of the parameter
out_p	Void pointer to buffer to store param value, should be enough to store value

Returns

0 if OK, -1 if error or Param no exists, or buffer too small

3.11.5.7 get_table_size()

```
uint16_t get_table_size (
            void  )
```

Get amount of params in table.

Returns

Number of params in table

3.11.5.8 init_param_persistence()

```
int init_param_persistence (
            void  )
```

Init the task that stores parameters in persistent memory.

Note

Storage Service functions should be ported. See simle_port.h.

See also

sfsf_port.h

Returns

-1 if error, 0 if OK

3.11.5.9 load_param_table()

```
int load_param_table (
            char * file_name )
```

Load parameters in a file to the Parameter Table.

When enabling the Parameter persistence Service with init_param_persistence(), prameters are stored in a file. After a rebbot form OBC, to restore the Parameter Table use this function. The Param Table should be created befor, using set_param_table().

Note

Param Table should be created befor!

See also

set_param_table()

Note

Storage Service functions should be ported. See simle_port.h.

See also

sfsf_port.h

Parameters

file_name	Name of file where parameters are stored in persistent memory

Returns

-1 if error, 0 if OK

3.11.5.10 param_to_str()

```
int param_to_str (
            param_handle_t param_handle,
            char * out_buff,
            int buff_size )
```

Store the value of a param as string in a buffer.

Parameters

param_handle	Handle of the param
out_buff	Destination buffer
buff_size	Size of the destination buffer

Returns

0 if OK, -1 if error

3.11.5.11 print_pram_table()

```
void print_pram_table (
            void  )
```

Print all params names and values in debug output.

Note

Debug Service function should be ported.

See also

sfsf_port.h

3.11.5.12 set_param_table()

```
int set_param_table (
            param_table_t * param_table,
            uint16_t param_table_size )
```

Set the parameter table.

Call this function during initialization, in init_services() function at init_functions.c.

Parameters

param_table	Pointer to the param table
param_table_size	Num of entries of param_table

Returns

-1 if error, 0 if OK

3.11.5.13 set_param_val()

```
int set_param_val (
            param_handle_t param_h,
            void * in_p )
```

Set the value of a Parameter.

Warning

Not thread-safe, user shall use mutex if sharing a param between many tasks, as with variables

Parameters

param_h	Handle of the param
in_p	Void pointer to memory where value is stored

Returns

0 if OK, -1 if error (Param no exists, or buffer too small)

3.11.5.14 str_to_param()

```
int str_to_param (
            param_handle_t param_handle,
            char * in_buff )
```

Set the value of a string to a param.

Convert the value of a string to the type of the param, and store it in the param value space

Parameters

param_handle	Handle of the param
in_buff	Buffer with the value as string, to be assigned to the param

Returns

0 if OK, -1 if error

3.12 D:/sfsf/include/sfsf_port.h File Reference

Function declarations to ported.

Functions

System Functions

Include the header of your port

Your port should be conformed by a header file (port.h) and a source file (port.c).

See also

src/ports Port this functions for rebooting and shooting down the OBC.

- void cpu_reset (void)
  Reboot the system. Use csp_sys_reboot()!
- void cpu_shutdown (void)
  Shutdown the system. Use csp_sys_shutdown()!

Debug Functions

Port this functions to print debug info

- void print_debug_port (const char *str)
  Print a string of characters on the debug output.
- void print_debug_char_port (char c)
  Print a character on the debug output.
- void print_debug_hex_port (char c)
  Print a char as hex representation on the debug output.
- void print_debug_uint_port (unsigned int n)
  Prints an unsigned integer on the debug output.

Storage Functions

Port this functions for File System Calls

- #define FILE_T void
  Type of File Descriptor for file system.
- #define FILE_MODE_T int
  Type of File Open Modes.
- int file_open_port (FILE_T *fp, const char *path, FILE_MODE_T mode)
  Open or create a file.
- int file_close_port (FILE_T *fp)
  Close a file descriptor.
- char * file_read_port (FILE_T *fp, char *buff, int len)
  Read a string (until new-line or end-of-file) from a file descriptor.
- int file_write_port (FILE_T *fp, const char *str)
  Write a string to a file descriptor.
- int file_remove_port (const char *path)
  Remove a file.

3.12.1 Detailed Description

Function declarations to ported.

SFSF Port

Some features form Services require functions for interacting with hardware, for example rebooting the OBC, print messages to debug output, or writing a file in a external memory. Each satellite is build with different hardware, therefore the implementation of this functions may vary. It is expected that the developers of each mission implement this functions for the specific hardware.

This file contains the definition of all the functions that need to be implemented. This can be done in a source file (file with ".c" extension), see the ports for AVR32_UC3C and Linux in folder services/src/ports .

See also

services/src/ports

Note

Function prototype with "weak attribute" means the function may be implemented by user, but is not mandatory to do so.

3.12.2 Macro Definition Documentation

3.12.2.1 FILE_MODE_T

```
#define FILE_MODE_T int
```

Type of File Open Modes.

Define FILE_MODE_T as the File Modes Type, in your port header file.

3.12.2.2 FILE_T

```
#define FILE_T void
```

Type of File Descriptor for file system.

Define FILE_T as the File Descriptor Type, in your port header file.

3.12.3 Function Documentation

3.12.3.1 cpu_reset()

```
void cpu_reset (
            void  )
```

Reboot the system. Use csp_sys_reboot()!

Implement this function to enable csp_sys_reboot().

Note

Dont use this function! use csp_sys_reboot() instead.

3.12.3.2 cpu_shutdown()

```
void cpu_shutdown (
            void  )
```

Shutdown the system. Use csp_sys_shutdown()!

Implement this function to enable csp_sys_shutdown().

Note

Dont use this function! use csp_sys_shutdown() instead.

3.12.3.3 file_close_port()

```
int file_close_port (
            FILE_T * fp )
```

Close a file descriptor.

Parameters

fp	File descriptor of open file

Returns

-1 if fails, 0 if OK

3.12.3.4 file_open_port()

```
int file_open_port (
            FILE_T * fp,
```

```
            const char * path,
            FILE_MODE_T mode )
```

Open or create a file.

Parameters

fp	Pointer to a File descriptor to store file info
path	Pathname of file to open or create
mode	Mode to open or create file

Returns

-1 if fails, 0 if OK

3.12.3.5 file_read_port()

```
char* file_read_port (
            FILE_T * fp,
            char * buff,
            int len )
```

Read a string (until new-line or end-of-file) from a file descriptor.

Parameters

fp	File descriptor of open file
buff	Destination buffer to store read bytes
len	Limit bytes to read

Returns

-1 if fails, the number of bytes read if OK

3.12.3.6 file_remove_port()

```
int file_remove_port (
            const char * path )
```

Remove a file.

Parameters

path	Pathname of file to remove

Returns

-1 if fails, 0 if OK

3.12.3.7 file_write_port()

```
int file_write_port (
            FILE_T * fp,
            const char * str )
```

Write a string to a file descriptor.

Parameters

fp	File descriptor of open file
str	Source buffer whit sting to write into file

Returns

-1 if fails, the number of bytes written if OK

3.12.3.8 print_debug_char_port()

```
void print_debug_char_port (
            char c )
```

Print a character on the debug output.

Parameters

c	The character to print.

3.12.3.9 print_debug_hex_port()

```
void print_debug_hex_port (
            char c )
```

Print a char as hex representation on the debug output.

Parameters

c	The hex character to print.

3.12.3.10 print_debug_port()

```
void print_debug_port (
            const char * str )
```

Print a string of characters on the debug output.

Parameters

str	The string to print

3.12.3.11 print_debug_uint_port()

```
void print_debug_uint_port (
            unsigned int n )
```

Prints an unsigned integer on the debug output.

Parameters

n	The integer to print.

3.13 D:/sfsf/include/sfsf_storage.h File Reference

API for Storage Service.

Functions

- int file_open (FILE_T *fp, const char *path, FILE_MODE_T mode)
  Open or create a file.
- int file_close (FILE_T *fp)
  Close a file descriptor.
- char * file_read (FILE_T *fp, char *buff, int len)
  Read a string (until new-line or end-of-file) from a file descriptor.
- int file_write (FILE_T *fp, const char *str)
  Write a string to a file descriptor.
- int file_remove (const char *path)
  Remove a file.

3.13.1 Detailed Description

API for Storage Service.

Storage Service

Features Summary

- Open and close files
- Write and read bytes into files
- Retrieve file stats

Module Description

The storage service provides the essential functions for handling files. Other services, like the Param Service, Log Service, Housekeeping Service, may require this functions for providing persistence features, if so specific in the configuration file. Note that all the functions for managing files shall be ported, see sfsf_port.h.

3.13.2 Function Documentation

3.13.2.1 file_close()

```
int file_close (
            FILE_T * fp )
```

Close a file descriptor.

Parameters

fp	File descriptor of open file

Returns

-1 if fails, 0 if OK

3.13.2.2 file_open()

```
int file_open (
            FILE_T * fp,
            const char * path,
            FILE_MODE_T mode )
```

Open or create a file.

Parameters

fp	Pointer to a File descriptor to store file info
path	Pathname of file to open or create
mode	Mode to open or create file

Returns

-1 if fails, 0 if OK

3.13.2.3 file_read()

```
char* file_read (
            FILE_T * fp,
            char * buff,
            int len )
```

Read a string (until new-line or end-of-file) from a file descriptor.

Parameters

fp	File descriptor of open file
buff	Destination buffer to store read bytes
len	Limit bytes to read

Returns

-1 if fails, the number of bytes read if OK

3.13.2.4 file_remove()

```
int file_remove (
            const char * path )
```

Remove a file.

Parameters

path	Pathname of file to remove

Returns

-1 if fails, 0 if OK

3.13.2.5 file_write()

```
int file_write (
            FILE_T * fp,
            const char * str )
```

Write a string to a file descriptor.

Parameters

fp	File descriptor of open file
str	Source buffer whit sting to write into file

Returns

-1 if fails, the number of bytes written if OK

3.14 D:/sfsf/include/sfsf_time.h File Reference

API for Time Service.

Functions

- int init_sw_wdt (void)
  Init Software Watchdog Timer.
- void reset_sw_wdt (void)
  Reset Software Watchdog Timer.
- void inti_time (void)
  Init Time Service.
- uint32_t time_since_boot_ms (void)
  Return milliseconds science boot
- uint32_t time_since_boot_s (void)
  Return seconds science boot
- int sync_timestamp (uint32_t new_timestamp_s)
  Sync Timestamp with ground.
- uint32_t get_timestamp_s (void)
  Return the local timestamp in seconds.
- size_t get_timestamp_str (char *dest_buffer, size_t buff_size)
  Stores the local timestamp as string into dest_buffer.

Variables

Parameterizable Variables

Use the parateerize() Macro to parameterize this variables into the Parameters Table, this will simplify the control of Time Service, by providing a way to change the behavior .

See also

sfsf_param.h.

- uint32_t sw_wdt_timout_ms

3.14.1 Detailed Description

API for Time Service.

Time Service

Features Summary

- Retrieve the time since boot
- Synchronize time with the ground
- Retrieve the time from the ground
- Enable and reset a software Watchdog Timer

Module Description

The Time Service is a small module which provides functions for collecting the current timestamp and the time since boot of the system. Also provides functions for enabling and managing a software Watchdog timer.

3.14.2 Function Documentation

3.14.2.1 get_timestamp_s()

```
uint32_t get_timestamp_s (
            void  )
```

Return the local timestamp in seconds.

Returns

Timestamp in milliseconds

3.14.2.2 get_timestamp_str()

```
size_t get_timestamp_str (
            char * dest_buffer,
            size_t buff_size )
```

Stores the local timestamp as string into dest_buffer.

Parameters

dest_buffer	Destination Buffer where timestamp will be stored
buff_size	Size of dest_buffer

Returns

length of the resulting C string

3.14.2.3 init_sw_wdt()

```
int init_sw_wdt (
            void  )
```

Init Software Watchdog Timer.

Returns

EXIT_FAILURE if error , EXIT_SUCCESS if OK

3.14.2.4 inti_time()

```
void inti_time (
            void  )
```

Init Time Service.

Init Time Service for gathering time since boot, run at init.

3.14.2.5 sync_timestamp()

```
int sync_timestamp (
            uint32_t new_timestamp_s )
```

Sync Timestamp with ground.

Parameters

new_timestamp_s	New timestamp to be stored in s

Returns

EXIT_FAILURE if error , EXIT_SUCCESS if OK

3.14.2.6 time_since_boot_ms()

```
uint32_t time_since_boot_ms (
            void  )
```

Return milliseconds science boot

Returns

Milliseconds science boot

3.14.2.7 time_since_boot_s()

```
uint32_t time_since_boot_s (
            void  )
```

Return seconds science boot

Returns

Seconds science boot

3.14.3 Variable Documentation

3.14.3.1 sw_wdt_timout_ms

```
uint32_t sw_wdt_timout_ms
```

Timeout for software watchdog timer.

3.15 D:/sfsf/sfsf_config.h File Reference

SFSF Services Configurations.

Macros

Enable and Disable Macros

Some configurations may be enabled or disabled with this macros.

- #define ENABLE 1
- #define DISABLE 0

OS CONFIGURATIOS

Some OS, like FreeRTOS, require to define the stack size for creating Tasks If your OS don't requires to define the stack size, comment the following lines

- #define **CONF_MINIMAL_STACK_SIZE** 256

CSP Configurations

It is necessary to know the size assigned to the CSP buffers. Set the same value used for csp_buffer_init().

- #define **CONF_CSP_BUFF_SIZE** 300

Debug Configurations

Uncomment to print debug info of each service.

Note

Debug functions should be implemented by user.

See also

sfsf_port.h

- #define CONF_CMD_DEBUG ENABLE
- #define CONF_HK_DEBUG ENABLE
- #define CONF_LOG_DEBUG ENABLE
- #define CONF_PARAM_DEBUG ENABLE
- #define CONF_TIME_DEBUG ENABLE

Param Service Configurations

Configurations for sfsf_param.h

- #define CONF_PARAM_PERSIST_ENABLE ENABLE
- #define CONF_PARAM_FILE_NAME "params.txt"
- #define CONF_PARAM_NAME_SIZE 16
- #define CONF_PARAM_MAX_PARAM_SIZE 64

Command Service Configurations

- #define **CONF_CMD_QUEUE_ENABLE** ENABLE
- #define CONF_CMD_QUEUE_SIZE 10
- #define CONF_CMD_ARGS_DELIMITERS ",;"

Housekeeping Service Configurations

- #define CONF_HK_ENABLE ENABLE
- #define CONF_HK_BEACONS_FILE "beacons.txt"
- #define CONF_HK_SPORT 10
- #define CONF_HK_DPORT 10
- #define CONF_HK_BEACON_PACKET_PRIORITY 2
- #define CONF_HK_BEACON_PERIOD_MS 20000

Log Service Configurations

- #define CONF_LOG_PERSIST_ENABLE ENABLE
- #define CONF_LOG_FILE_NAME "log.txt"
- #define CONF_LOG_QUEUE_SIZE 10
- #define CONF_LOG_MESSAGE_SIZE 128

Time Service Configurations

- #define CONF_TIME_SW_WDT_ENABLE ENABLE
- #define CONF_TIME_SW_WDT_TIMEOUT_MS 10000
- #define CONF_TIME_TIMESTAMP_FORMT "%Y-%m-%d %T"

3.15.1 Detailed Description

SFSF Services Configurations.

SFSF Services Configurations

The SFSF services require a configuration file to work. This is the only file from library that the user needs to modify. The file contains configurations that modifies the behavior of each service. Each configuration has its owns description.

3.15.2 Macro Definition Documentation

3.15.2.1 CONF_CMD_ARGS_DELIMITERS

```
#define CONF_CMD_ARGS_DELIMITERS ",;"
```

Separators chars of Command Arguments within Command packet

3.15.2.2 CONF_CMD_DEBUG

```
#define CONF_CMD_DEBUG ENABLE
```

Print cmd queue debug info to debug output.

3.15.2.3 CONF_CMD_QUEUE_SIZE

```
#define CONF_CMD_QUEUE_SIZE 10
```

Max amount of commands in queue waiting to be executed.

3.15.2.4 CONF_HK_BEACON_PACKET_PRIORITY

```
#define CONF_HK_BEACON_PACKET_PRIORITY 2
```

CSP Packet Priority for beacons.

3.15.2.5 CONF_HK_BEACON_PERIOD_MS

```
#define CONF_HK_BEACON_PERIOD_MS 20000
```

Period between beacons in ms.

3.15.2.6 CONF_HK_BEACONS_FILE

```
#define CONF_HK_BEACONS_FILE "beacons.txt"
```

Name of the file where beacons will be stored if CONF_HK_STORE_BEACONS is ENABLE.

3.15.2.7 CONF_HK_DEBUG

```
#define CONF_HK_DEBUG ENABLE
```

Print beacons to debug output.

3.15.2.8 CONF_HK_DPORT

```
#define CONF_HK_DPORT 10
```

CSP destination port for Beacons.

3.15.2.9 CONF_HK_ENABLE

```
#define CONF_HK_ENABLE ENABLE
```

Enable or disable the Beacon transmission and storage.

3.15.2.10 CONF_HK_SPORT

```
#define CONF_HK_SPORT 10
```

CSP source port for Beacons.

3.15.2.11 CONF_LOG_DEBUG

```
#define CONF_LOG_DEBUG ENABLE
```

Print log messages to debug output.

3.15.2.12 CONF_LOG_FILE_NAME

```
#define CONF_LOG_FILE_NAME "log.txt"
```

Name of Log file.

3.15.2.13 CONF_LOG_MESSAGE_SIZE

```
#define CONF_LOG_MESSAGE_SIZE 128
```

Max size of a Log message.

3.15.2.14 CONF_LOG_PERSIST_ENABLE

```
#define CONF_LOG_PERSIST_ENABLE ENABLE
```

Enable or disable the Log task, which stores log messages in file.

3.15.2.15 CONF_LOG_QUEUE_SIZE

```
#define CONF_LOG_QUEUE_SIZE 10
```

Max Log messages waiting to be stored.

3.15.2.16 CONF_PARAM_DEBUG

```
#define CONF_PARAM_DEBUG ENABLE
```

Print param debug info to debug output.

3.15.2.17 CONF_PARAM_FILE_NAME

```
#define CONF_PARAM_FILE_NAME "params.txt"
```

File name where params will be stored in persistent memory.

3.15.2.18 CONF_PARAM_MAX_PARAM_SIZE

```
#define CONF_PARAM_MAX_PARAM_SIZE 64
```

Max size of Parameter of type STRING_PARAM.

3.15.2.19 CONF_PARAM_NAME_SIZE

```
#define CONF_PARAM_NAME_SIZE 16
```

Max size of params names in bytes.

3.15.2.20 CONF_PARAM_PERSIST_ENABLE

```
#define CONF_PARAM_PERSIST_ENABLE ENABLE
```

Enable or disable the Param task, which stores Params in file.

3.15.2.21 CONF_TIME_DEBUG

```
#define CONF_TIME_DEBUG ENABLE
```

Print software watchdog timer debug info to debug output.

3.15.2.22 CONF_TIME_SW_WDT_ENABLE

```
#define CONF_TIME_SW_WDT_ENABLE ENABLE
```

Enable or disable the Software Watchdog Timer.

3.15.2.23 CONF_TIME_SW_WDT_TIMEOUT_MS

```
#define CONF_TIME_SW_WDT_TIMEOUT_MS 10000
```

Timeout of the software Watchdog timer.

3.15.2.24 CONF_TIME_TIMESTAMP_FORMT

```
#define CONF_TIME_TIMESTAMP_FORMT "%Y-%m-%d %T"
```

Timestamp string format, see strftime() Posix function for more info.

3.15.2.25 DISABLE

```
#define DISABLE 0
```

Disable a configuration.

3.15.2.26 ENABLE

```
#define ENABLE 1
```

Enable a configuration.

Index