MicroSFSC

Generated by Doxygen 1.9.2

Chapter 1

MicroSFSC

The Shop-Floor Service Connector (SFSC) is an easy-to-use, service-orientated communication framework for Industry 4.0, developed at the University of Stuttgart. MircoSFSC is a microcontroller ready implementation of the SFSC adapter role. This documentation provides a quick overview about the usage and the features of the MicroSFSC framework, where as an in deep analysis can be found in this white-paper(German).

This implementation is written in C and conforms the C99 freestanding standard. If your compiler only supports ANSI-C, some platform depended adjustments are required.

The RAM and ROM footprint is configuration depended; in default configuration 25kB ROM and about 7-11kB RAM are needed. A configuration for your specific use-case will most likely result in lower resource requirements.

This framework does not use dynamic memory allocation.

1.1 Repository structure

This repository is structured the following way:

- The **sfsc** folder contains the actual source code you should copy into your project.
- The **platforms** folder contains implementations for the platform dependencies for some platforms. Platform dependencies are explained in the next section.
- The **docs** folder contains a doxygen html documentation of the public header. It can be access via github pages. Also, the **docs** folder contains latex code and the complete documentation as PDF.
- The **examples/scenarios** folder contains the actual examples. Every example comes with its own preprocessor directive that must be defined to enable that example. Only one of the examples should be active at the same time. Where to define the directives is up to the used build system. The other subfolders of the **examples** folder contain the initialization logic and example build instructions to the get the examples running on the corresponding platform.

1.2 Porting to your platform

The **sfsc/platform** folder contains headers you have edit or implement for your platform if you want to use this framework. There are some existing platform ports in the **platforms** folder in the root of this repository. For example, there are implementations for POSIX (including Windows with MinGW) systems and the ESP32 microcontroller family. This framework does not contain an IP stack, so you need to provide one (most network ready platforms will already include one). Below are some more information on the headers you need to look at.

1.2.1 sfsc_types.h

`sfsc_types.h` contains declarations of different datatypes. If your platform provides `stdint.h` and `stdbool.h` (like all conforming freestanding C99 environments should), there is no need to edit this file. The other needed header files are from the ANSI-C (C90) standard.

1.2.2 sfsc_strings.h

This framework needs some functions from the `strings.h` header (namely `memcpy`, `memset` and `strlen`), which is not part of any freestanding C standard. Many platforms will provide it anyway, and if you platform does, make sure to define *HAS_STRING_H* in `sfsc_adapter_cofig.h` (more about configuration can be read below). If this define is missing, the framework will fall back to self-provided, but not as efficient implementations of these functions.

1.2.3 sfsc_platform.h

This header provides four functions you need to implement. They should behave as their function documentation demands, some important additional details are given here:

- `time_ms` is used for timing and must return some means of the current time, in millisecond resolution. It is not necessary (but allowed) to provide the absolute unix time, a relative value to indicate time since the system start (or even first call of the time_ms function) is sufficient.
- `random_bytes` needs to generate the requested amount of random bytes and write them to the specified buffer. These bytes are not used for cryptographically functions, so it's ok for them to be pseudo-random. On the other hand, it is very important that the generated byte sequences are different on each system start! If your platform does not provide such a mechanism you have to do this yourself: You can use a pseudo-random generator algorithms seeded with some random sensor noise (read more on the `von Neumann whitening algorithm`) or the absolute unix time (if you can access it). An other approach is to store the last generated bytes in persistent memory (EEPROM), and seed your pseudo-random algorithm with them on system start. If you use this framework on multiple microcontrollers simultaneously, the generated random sequences must not be the same, too!
- The `lock` and `unlock` functions are only needed for multi-threading, if you don't use multi-threading you can provide empty implementations of these functions. If you use multi-threading you have to ensure that only one thread is accessing a socket for writing at the same time. This should be no problem, since the most kernels and operating systems (which most likely provide the multi-threading in the first place) provide synchronization mechanisms. The framework will call you with an address to an uint_8. The address identifies the socket that should be locked/unlocked for access through other threads (you can treat the address like a numerical handle). The uint_8t is a single byte you can freely use as memory if needed (e.g. to store locking information).

1.2.4 sfsc_sockets.h

SFSC adapters communicate with SFSC cores using TCP/IP, and you need a to provide functions to establish this connection. The framework does not contain a own network-stack, since if your platform is IP-ready, it will most likely feature a platform optimized implementation. The API demanded by this framework is based on the POSIX Socket API (also known as BSD Socket), with the extension that most functions need to operate in a non-blocking way. Examples for providing this functions can be found in the **platforms** folder.

1.2.5 Protobuf

This framework uses NanoPB v0.4.2 for protobuf serialization/deserialization. It is configured to not use dynamic memory allocation, and is pointed towards `sfsc_strings.h` instead of the normal `strings.h`. NanoPB also depends on `stdbool.h`, `stdint.h`, `stddef.h`, and `limits.h` in its `pb.h`. If your compiler does not conform the freestanding C99 standard, you might need to edit the `pb.h` header. For more information on NanoPB see it's official page.

1.3 The Execution model

The framework needs some background tasks to be handled. But it does not include any concept of threading (since your platform might already have a concept, or is not powerful enough to have such a concept). Instead, it is your responsibility to call the background tasks in a cyclic manor. There are two main tasks you need to call: The system task and the user task. Both tasks are designed to be non-blocking, but for the user task, there are some restrictions.

1.3.1 System Task

The system task handles connection setup and heartbeating. It will also read the network using your `sfsc_socket.h` implementation. It is important that the system task is called with a high enough frequency, or it will fail to send heartbeats to the core in time. If this happens, the core will treat the corresponding adapter as disconnected. As a guideline, try to call the system task at least once every 5ms. The system tasks runtime is designed to be constant, which is achieved by only using non-blocking operations. As a consequence, the system task does not invoke your callbacks, as the framework can not know, what you are doing in your callbacks. So, instead of executing callbacks based on the data the system task receives from the network, it writes the data to an intermediate ring buffer, called the user ring.

1.3.2 User Task

Among other things, the user task will then go ahead and read data from the user ring, and based on them, invoke callbacks you defined during an API call. Since the user task directly executes your callbacks, what you do in them will influence the runtime of the user task. Reading one entry of data from the user ring and invoking a callback is called a micro step. How many micro steps should be taken in a single call to the user task function can be configured (using REPLAYS_PER_TASK). The data supplied as parameters to your callbacks are only valid during the current micro step, meaning that after your callback returns, the data will be removed from the user ring and you should no longer try to access them. If you need the data from the callback outside the callback, you have two options:

1. Simply copy the data somewhere else. This is the easiest and safest way, but requires additional memory.
2. If you can not afford to copy the data, you can enter the *user task pause state*. In the user task pause state, the user task won't advance to the next micro step after your callback returns. You will need to leave the pause state explicitly by calling a function. Keep in mind, that as long as you are in the pause state, no further callbacks will be invoked (exceptions are TODO). Note that even while you are currently in the pause state, you still must ensure that the system and user task functions are called (since as noted above, the user tasks needs to do other things then taking micro steps, too)!

1.3.3 Blocking the User Task

An other important question is: Am I allowed to perform a blocking or long taking operation in a callback? As stated above, system and user task are designed to be non-blocking. Consider the following code snippet:

```
// Note that this code is not runnable since it ignores multiple return values of functions that indicate
       the operability of the adapter
sfsc_adapter adapter;
start_session_bootstraped(&adapter,host,PORT);
while(1){ //global execution loop
    system_task(&adapter);
    user_task(&adapter);
}
```

So if your callback now blocks execution, the user task will also block, and therefore, the whole program is blocked, preventing the system task from running. An approach to move the blocking code out of the callback will lead to this:

```
sfsc_adapter adapter;
start_session_bootstraped(&adapter,host,PORT);
while(1){ //global execution loop
    system_task(&adapter);
    user_task(&adapter);
    blocking_user_code();
}
```

This is obviously not better than the frist approach, since it will also prevent `system_task` from being called. You can try to design your function in the system and user tasks manor and split your long taking operation in multiple substeps and use only non-blocking functions yourself. Then, each invocation of your function it will execute only a tiny step of your overall goal. This will result in the following, valid approach:

```
sfsc_adapter adapter;
start_session_bootstraped(&adapter,host,PORT);
int step_number=0;
while(1){ //global execution loop
    system_task(&adapter);
    user_task(&adapter);
    non_blocking_user_code(step_number);
    step_number++;
}
```

So to conclude, if your system task and user task run in the same thread, callbacks and other things you do in your global execution loop should not block.

1.3.4 Multi-threading

If you decide to use multi-threading you can set up the system and the user task as different threads. Then, blocking the user task won't interfere with the execution of the system task and is thus allowed. This framework does not include a scheduler, so you have to use your own threading solution. Keep in mind that if you use the a multi-threaded execution model, you might have to add some synchronization (see the *sfsc_platform.h* section above).

1.3.5 Message drop

Since the system task writes data to the user ring, and the user tasks takes data from the user ring, it can happen that the user ring gets filled faster than it is emptied (especially if the user task is paused for too long!). In this case, newly received data are dropped (according to the `ZMTP PUBSUB specification` on which SFSC is based).

1.3.6 Further reading

For an in deep discussion and reasoning why this execution model is used, see the white-paper, chapter 4.9.

1.4 Configuration

There are several configuration options to adjust the MircoSFSC framework. They can be found in `sfsc_adapter_config.h` and `zmtp_config.h` and will influence the RAM (and some even ROM) consumption of a `sfsc_adapter` struct. The table below list the configuration options, where they can be found, what they are for, and how they change memory consumption. Also, the default values (either a numerical value, or whether the option is defined or not) are listed. Note that all memory sizes are only guide values, the actual values will depend on your platform (e.g. on the pointer size or your memory alignment rules). A xN (where N is a number) in the memory column means that the configured value times N bytes of memory is needed.

Parameter Name	Header	Default	Memory Impact	Description
REPLAYS_PER_TASK	`sfsc_adapter_config.h`	0	-	The maximum number of micro steps that should be taken per `user_task` function invocation, 0 for as many as possible. Example: If there are 6 items in the user ring, and REPLAYS_PER_TASK is set to 4, only the first 4 of them will be processed in this `user_task` call, the next 2 have to wait until the next `user_task` call.
HEARTBEAT_SEND_RATE_MS	`sfsc_adapter_config.h`	400	-	The time to wait between sending outgoing heartbeats in milliseconds.
HEARTBEAT_DEADLINE_OUTGOING_MS	`sfsc_adapter_config.h`	HEARTBEAT_SEND_RATE_MS * 4	-	If it was not possible to send at least one heartbeat in this time (denoted in milliseconds) - most likely due to the `system_task` function not being called frequent enough - an error will be raised.
HEARTBEAT_DEADLINE_INCOMING_MS	`sfsc_adapter_config.h`	4000	-	The amount of time in milliseconds in which a heartbeat from the core needs to arrive. If there is no heartbeat in this amount of time, the SFSC session will be treated as terminated.

Parameter Name	Header	Default	Memory Impact	Description
HAS_STRING_H	sfsc_↩ adapter_↩ config.h	defined	a few bytes of ROM if NOT set	Should be defined if your platform has a memcpy, memset and strlen function. If not defined, inefficient fallback implementations of this functions will be used.
MAX_↩ PUBLISHERS	sfsc_↩ adapter_↩ config.h	6	x4 RAM	Amount of publisher services, a single adapter can operate at the same time. See the register↩ _publisher function documentation for more information.
MAX_↩ SUBSCRIBERS	sfsc_↩ adapter_↩ config.h	12	x4 RAM	Amount of subscriptions to publisher services, a single adapter can operate at the same time. See the register↩ _subscriber function documentation for more information.
MAX_PENDING↩ _ACKS	sfsc_↩ adapter_↩ config.h	6	x48 RAM	Amount of pending acknowledges to transmitted server-service-answers a single adapter can keep track of at the same time. See the answer_↩ request function documentation for more information.
MAX_SERVERS	sfsc_↩ adapter_↩ config.h	6	x4 RAM	Amount of server services, a single adapter can operate at the same time. See the register_↩ server function documentation for more information.

Parameter Name	Header	Default	Memory Impact	Description
MAX_↩ SIMULTANIOUS↩ _COMMANDS	sfsc_↩ adapter_↩ config.h	6	x24 RAM	Amount of commands (used for creating or deleting something from a SFSC cores service registry) a single adapter can issue at the same time. Needed in the register↩ _publisher, register↩ _server and unregister↩ _publisher, unregister_↩ server functions, see there for more information.
MAX_↩ SIMULTANIOUS↩ _REQUESTS	sfsc_↩ adapter_↩ config.h	6	x40 RAM	Amount pending (not-yet answered) requests a single adapter can make at the same time. See the request function for more information.
REGISTRY_↩ BUFFER_SIZE	sfsc_↩ adapter_↩ config.h	512	x1 RAM	When querying the service registry, received services are stored in a buffer of this size. Must not be greater then ZMTP_IN↩ _BUFFER_SIZE (see below; there is no point in storing a service you can not even receive).

Parameter Name	Header	Default	Memory Impact	Description
MAX_DELETED↩ _MEMORY	sfsc_↩ adapter_↩ config.h	32	x37 RAM	When querying the service registry, it is necessary to keep track of services in the registry that are marked as deleted (see chapter 4.2 in the white-paper). This value determines, how many delete events can be stored; if you know that there are many services in your service registry, you may want to increase this value. Note however, that this is not a 1:1 relation and you don't need to set this value to the total number of services (if you have 100 services, you don't need to set this value to 100, 32 might be fine). To find the value suited the most for your usecase, you need to experiment a little.
USER_RING_SIZE	sfsc_↩ adapter_↩ config.h	5120	x1 RAM	The size of the user ring. The default value is chosen this high to prevent message drop, for platforms with lower RAM capabilities, a more appropriate value might be 1024.
NO_CURVE	zmtp_↩ config.h	defined	5 KB(!) RAM and ROM if NOT set	Curve encryption is wip. It is recommended that you disable the functions related to CRUVE to speed up compilation and reduce the RAM and ROM footprint of the framework.

Parameter Name	Header	Default	Memory Impact	Description
ZMTP_IN_↩ BUFFER_SIZE	zmtp_↩ config.h	512	x4 RAM	The size of the ZMTP receive buffer; determines, how big a single ZMTP message can be. If you know, that the services you use need to receive bigger payloads (e.g. because you want to subscribe a publisher whose messages are 1KB in size), you need to adjust this value.
ZMTP_↩ METADATA_↩ BUFFER_SIZE	zmtp_↩ config.h	32	x8 RAM	ZMTP needs a place to store its meta data. You usually don't need to adjust this buffers size, unless you tweak the ZMTP implementation itself.

1.5 Using the API

All public API functions are defined in sfsc_adapter.h. You should include this header everywhere you want to use one of the frameworks function. To use most of the functions, you need a pointer to a sfsc_adapter. To declare a sfsc_adapter, you also need to include sfsc_adapter_struct.h. You should only include this header in the compilation unit you declare the adapter and not modify the struct fields of the adapter. The adapter has a stats field, which tells you some information about its current state. To access the stats field of an adapter, use the adapter_stats function. After declaring a adapter, you can use it to start a session. Use the start_session_bootstraped or start_session function with a pointer the the adapter. After the start_session method returns, your SFSC adapter is ready to perform the SFSC handshake, and you should start using the system_task function on the adapter. Once the state of the adapter (accessible through the stats field of the adapter struct) is operational, you can also start invoking the user_task function on the adapter, and use the other API functions with the adapter.

Most functions, including the system_task and user_task functions return error codes. An error code of SFSC_OK indicates, that a function call was successful. A full list of error codes can be found in sfsc_error_codes.h and zmtp_stats.h. All errors of zmtp_stats.h indicate that there was something wrong with data transport: either the network or the ZMTP protocol (on which SFSC is based) somehow failed. These errors are not recoverable, you should initiate a new SFSC session (you can reuse the adapter struct for that, as long as you set all the fields according to sfsc_adapter_DEFAULT_INIT, see section *Struct initalization* below). Errors of sfsc_error_codes.h are higher level and indicate problems with SFSC itself. They are well documented, and some are even recoverable, so look at their documentation.

1.5.1 Struct initalization

There are some struct types you'll encounter while using the framework. If you want to initialize a struct to its default values (just 0 in almost all cases), you can use the corresponding <struct_name>_DEFAULT_INIT macro. For some structs there is a constant default instance you can use to copy the default values over, and if there is, its name is <struct_name>_default.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 _relative_publisher_tags Struct Reference

Represents service tags specific to publisher services.

`#include <sfsc_adapter.h>`

Data Fields

- sfsc_size **topic_offset**
- sfsc_size **topic_len**
- sfsc_size **output_message_type_offset**
- sfsc_size **output_message_type_len**
- sfsc_bool **unregistered**

4.1.1 Detailed Description

Represents service tags specific to publisher services.

The format and idea of the fields is very similar to relative_sfsc_service_descriptor, see there for an explanation.

An exception to this is the unregistered field. See the sfsc_publisher struct documentation for an explanation.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.2 _relative_server_tags Struct Reference

Represents service tags specific to server services.

`#include <sfsc_adapter.h>`

Data Fields

- sfsc_size **topic_offset**
- sfsc_size **topic_len**
- sfsc_size **input_message_type_offset**
- sfsc_size **input_message_type_len**
- sfsc_size **output_message_type_offset**
- sfsc_size **output_message_type_len**
- sfsc_SfscServiceDescriptor_ServiceTags_ServerTags_AckSettings **ack_settings**

4.2.1 Detailed Description

Represents service tags specific to server services.

The format and idea of the fields is very similar to relative_sfsc_service_descriptor, see there for an explanation.

An exception to this is the ack_settings field. See the sfsc_server struct documentation for an explanation.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.3 _relative_sfsc_service_descriptor Struct Reference

Represents all infromation about a service.

`#include <sfsc_adapter.h>`

Data Fields

- sfsc_SfscId **core_id**
- sfsc_SfscId **adapter_id**
- sfsc_SfscId **service_id**
- sfsc_size **name_offset**
- sfsc_size **name_len**
- sfsc_size **custom_tags_offset**
- sfsc_size **custom_tags_len**
- sfsc_uint8 **service_type**
- union {
  relative_publisher_tags **publisher_tags**
  relative_server_tags **server_tags**
  } **service_tags**

4.3.1 Detailed Description

Represents all infromation about a service.

A relative_sfsc_service_descriptor is relative because it does not contain information about a service directly, but it merely serves as an index structure. The indexes (which end with _offset) are relative to a memory area. The memory areas address (called start in the following) is usually delivered with the descriptor. For example, to now access the name of the service, read name_len bytes from (start+name_offset). For reasoning, why this relative apporach is used, read the last paragraph.

An exception to the above are the core_id, adapter_id, and service_id fields, which actually contain the respective ids, and thereby denote the core and adapter this service belongs to, as well as this services own id.

The service_type field is either SERVICE_TYPE_SERVER or SERVICE_TYPE_PUBLISHER and indicates, how the service_tags union should be treated.

Why is this relative approach used? The binary size of a serivce in sfsc is not limited, so we can not know the size of the respective service fields in advance. On the other hand, this framework does not use dynamic memory allocation (malloc). A possible solution is to statically allocate memory for each field and add an length field, to indicate, how much memory is actually used. The difference between the allocated and actually used size is called waste. Instead of allocation a memory area for each field, we use one bigger memory area for all fields. The idea is that, because of the indivudual waste, this single field size can be smaller then sum of all indivudual field sizes, while still containing enough space to store all necessary information.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.4 _sfsc_adapter_stats Struct Reference

The sfsc_adapter_stats struct contains the infromation needed by the user.

```
#include <sfsc_adapter.h>
```

Data Fields

- const char * **address**
- sfsc_uint8 **adapter_id** [UUID_LEN]
- sfsc_uint8 **core_id** [UUID_LEN]
- sfsc_uint8 **state**
- sfsc_uint32 **discarded_message_count**
- sfsc_bool **query_in_progress**

4.4.1 Detailed Description

The sfsc_adapter_stats struct contains the infromation needed by the user.

The stats of an adapter should be accessd throught the adapter_stats function. The fields of this struct are all read-only and should not be modified by you.

The address field specifies the address of the core and is implicitly set by you during the start_session_bootstraped or start_session functions. The format of the address is up to you and can be anything, as long it is understood by your implementation of socket_connect (see sfsc_sockets.h).

The adapter_id and core_id fields indicate this adapters id and the id of the core it is connected to respectivly. They are both filled during the handshake, and ready to use once the adapters state is operational.

The state indicates to current connection state of an adapter. The various states are defined in sfsc_states.↩ h and are all prefixed with SFSC_STATE_. An adapter is considered operational if the value of this field is >= SFSC_STATE_OPERATIONAL.

Due to the execution and memory model of this framework it can occur that some messages receieved from the network are dropped (see the system_task function for details). The discarded_message_count keeps track of the number of lost messages.

The query_in_progress is set to 1 if you started a query process using query_services, and will be reset to 0 if the query process is terminated. See the query functions for more information.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.5 _sfsc_buffer Struct Reference

A simple structure to store binary data and their length.

`#include <sfsc_adapter.h>`

Data Fields

- const sfsc_uint8 * **content**
- sfsc_size **length**

4.5.1 Detailed Description

A simple structure to store binary data and their length.

This structure type is widely used in the framework for compact data storing. It is important to notice that some functions will take a sfsc_buffer struct as parameter directly, while others will work with pointers. In the most cases, this says something about the mutality of the content.

In general, for the use-time of a sfsc_buffer (usually defined in the respective functions documentation), the memory area the buffer points to must be valid and of the in the struct specified length.

When passing the struct to a function, it must be immutable, meaning that for the use-time of the sfsc_buffer, the content must not change.

When passing a pointer to a function, the content is allowed to be mutable, meaning that during the use-time of the sfsc_buffer, you are allowed to change the content pointer and thereby change the memory area are this sfsc_buffer is backed by. If you do so, don't forget to update the length field accordingly.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.6 _sfsc_channel_answer Struct Reference

A struct containing all information needed to answer a channel request.

`#include <sfsc_adapter.h>`

Data Fields

- sfsc_uint8 * **service_id**
- sfsc_uint8 * **adapter_id**
- sfsc_uint8 * **core_id**
- sfsc_buffer **publisher_output_topic**
- sfsc_buffer **name**
- sfsc_buffer **custom_tags**
- sfsc_buffer **output_message_type**
- sfsc_bool **unregistered**

4.6.1 Detailed Description

A struct containing all information needed to answer a channel request.

Instead of normal binary payload, a channel request is answered with the definition of a publisher service.

The core_id, adapter_id and service_id represent the core and adapter the publisher service is conencted to, as well as the publisher service itself. Their length must be UUID_LEN and their format must be standard-hexgroup-format (see the random_uuid function). The reason a sfsc_channel_answer uses sfsc_uint8* instead of sfsc_uint8[] is to remove copying and to save memory: The publisher you are describing is most likly hosted by the same adapter that will send this channel answer. Instead of allocating 3*UUID_LEN memory and copy the details from the adapter to the sfsc_channel_answer, you can just let the fields point to &adapter_states()->core_id and &adapter_states()->adapter_id respectivly.

All sfsc_buffers must be valid during the use-time of the sfsc_channel_answer struct and are optional. If publisher_output_topic is set to SERVICE_TOPIC_AUTOGEN the topic autogenerate rule (as described in sfsc_publisher) will be applied, and the service_id will be used as topic.

The unregistered should be set to 0 if the publisher you are describing in this sfsc_channel_answer is also registered in the service registry, or to 1 if it is not.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.7 _sfsc_publisher Struct Reference

State memory for a publisher service.

`#include <sfsc_adapter.h>`

Data Fields

- void(∗ **on_subscription_change**)(sfsc_adapter ∗adapter, sfsc_publisher ∗publisher, sfsc_uint8 last_↩ subscribers, sfsc_uint8 current_subscribers)
- sfsc_uint8 **last_subscribers**
- sfsc_uint8 **current_subscribers**
- sfsc_SfscId **service_id**
- sfsc_buffer **topic**
- sfsc_bool **unregistered**

4.7.1 Detailed Description

State memory for a publisher service.

To create a publisher, declare a sfsc_publisher struct, fill it, and use either the register_publisher or the register↩ _publisher_unregistered function. The sfsc_publisher struct must be valid until it is unregistered by an call to the unregister_publisher function.

Filling a publisher is optional and means to set values to the on_subscription_change, the service_id and topic fields.

If you want to choose a service id for the publisher yourself, you can fill the service_id before registration with an valid 128bit UUID in standard-hexgroup-format (see the random_uuid for more information). Usually, you want to set this field to sfsc_SfscId_init_default (what it already is for any instance initalized with sfsc_publisher_DEFAULT_↩ INIT) and let the framework automatically generate a service id. After registration, you should not change the value of this field.

If you want to choose a topic for the publisher yourself you can configure the topic buffer to point to a valid topic. The topic buffer must be valid and immutable (see the sfsc_buffer documentation for more insight on this) as long as the publisher is registered. If you want the framework to choose a topic for you, set this field to SERVICE_TOPIC_↩ AUTOGEN (what it already is for any instance initalized with sfsc_publisher_DEFAULT_INIT). To save memory, the framework will then simply use the service_id as topic.

If the subscription count of a publisher changes, the on_subscription_change callback is invoked. The parameters of the callback indicate the old and new subscription count. They are either 0 or at maximum 1, even if there might be more subscribers, since sfsc can only tell if there is at least 1 subscriber. During execution of the callback function, the last_subscribers and current_subscribers fields (not callback parameters!) are unfedined. After the callback function returns, they will match the values of the parameters. In any case, they are read-only!

The unregistered field is set by the framework and depends on wether you registered a publisher with the register↩ _publisher or the register_publisher_unregistered function. It indicates if the publisher is registered in the cores service registry. You must not change it!

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.8 _sfsc_publisher_or_server Struct Reference

Container to point either to a sfsc_server or sfsc_publisher.

`#include <sfsc_adapter.h>`

Data Fields

- sfsc_bool **is_server**
- union {
  sfsc_publisher * **publisher**
  sfsc_server * **server**
  } **service**

4.8.1 Detailed Description

Container to point either to a sfsc_server or sfsc_publisher.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.9 _sfsc_server Struct Reference

State memory for a server service.

`#include <sfsc_adapter.h>`

Data Fields

- void(* **on_request**)(sfsc_adapter *adapter, sfsc_server *server, sfsc_buffer payload, sfsc_int32 expected↩ _reply_id, sfsc_buffer reply_topic, sfsc_bool *b_auto_advance)
- sfsc_SfscServiceDescriptor_ServiceTags_ServerTags_AckSettings **ack_settings**
- sfsc_SfscId **service_id**
- sfsc_buffer **topic**
- sfsc_bool **is_channel**

4.9.1 Detailed Description

State memory for a server service.

To create a server, declare a sfsc_server struct, fill it, and use the register_service function. The sfsc_server struct must be valid until it is unregistered by an call to the unregister_server function.

In constrast to the creating process of a publisher, filling a server is NOT optional.

If you want to choose a service id for the server yourself, you can fill the service_id before registration with an valid 128bit UUID in standard-hexgroup-format (see the random_uuid for more information). Usually, you want to set this field to sfsc_SfscId_init_default (what it already is for any instance initalized with sfsc_server_DEFAULT_INIT) and let the framework automatically generate a service id. After registration, you should not change the value of this field.

If you want to choose a topic for the server yourself you can configure the topic buffer to point to a valid topic. The topic buffer must be valid and immutable (see the sfsc_buffer documentation for more insight on this) as long as the server is registered. If you want the framework to choose a topic for you, set this field to SERVICE_TOPIC↩ _AUTOGEN (what it already is for any instance initalized with sfsc_server_DEFAULT_INIT). To save memory, the framework will then simply use the service_id as topic.

The is_channel field indicates if this server service is a channel service. Channel services do not answer requests with normal binary payload, but answer them with publisher service definitions. Then, the adapter that made the request can subscribe to the publisher and recive values in a streamlike way. After registration, you should not change the value of this field.

The ack_settings field describe this servers acknwoledge strategy: usually, if a server answers a request, the requestor will send back an acknowledge message to the server, so that the server knows that the request was succesfully served. If this acknowledge message does not reach the server after ack_settings.timeout_ms milliseconds, the server will attempt to retransmit the answer, up to ack_settings.send_max_tries times. If ack_settings.send_↩ max_tries is set to 0, this server won't wait for acknowledges and use a fire-and-forget approach. This has some beneficial implications to the answer_request and answer_channel_request functions, documented there. In most cases, using a fire-and-forget approach is valid, since most transmission errors will be corrected on the tcp layer, and a performant sfsc core will rarely drop messages on the zmtp layer.

The on_request callback is invoked every time a request for this server service is receieved. It is allowed to change the on_request callback, even after registering th service. After receiving a request, you will usually take some actions based on the payload and eventually send an answer back to the requestor using the answer_request or answer_channel_request function.

The payload buffer is only valid during the current user task micro step (see the user_task documentation). b_↩ auto_advance is an out-parameter (meaning that you should set it), that lets you pause the user task on the current mirco step: if you set it to 0, you will enter the pause state and the payload pointer will be valid, even after the callback returns. On the other hand, the user task will not advance to the next micro step until you leave the pause state manually (see advance_user_ring).

The expected_reply_id and reply_topic are pull-throught parameters, meaning that you don't need to interact with them, but pass them to the answer_request or answer_channel_request once you want to answer the request. The reply_topic is like the payload also only valid during the current user task micro step.

If you attempt to answer an request right in the callback consider the following: For fire-and-forget servers, this is valid, since the reply_topic you receive by the on_request callback will be passed to the answer_request function, which will return before the on_request callback returns. For non-fire-and-forget servers, this is not valid, since the reply_topic must be valid till the corresonding on_ack call, and not till the answer_request function returns. To work arround this, you can either copy the content of the reply_topic somewhere global (recommended) or pause the user task, until the corresonding on_ack is invoked. Even if the user task is in pause state, the on_ack callback of an answer_request will be invoked, so there will be no deadlock.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

4.10 _sfsc_subscriber Struct Reference

Struct that contains the necessary state memory to subscribe to a publisher service.

`#include <sfsc_adapter.h>`

Data Fields

- sfsc_buffer **topic**
- void(∗ **on_data**)(sfsc_adapter ∗adapter, sfsc_subscriber ∗subscriber, sfsc_buffer payload, sfsc_bool ∗b_↩ auto_advance)

4.10.1 Detailed Description

Struct that contains the necessary state memory to subscribe to a publisher service.

To subscribe to a publisher service, declare a sfsc_subscriber somewhere, fill it and then register it with an adapter using the register_subscriber function. The subscriber struct must be valid as long as the subscriber is registered.

Filling a subscriber means to configure the topic field and the on_data callback.

Usually, you will recieve the topic of the publisher service you want to subscribe to from the query_services function. The topics region must be immutable while the subscriber is registered (for more information what immutable means, see the sfsc_buffer documentation).

The on_data field is allowed to change. It is invoked, when the corresponding publisher publishes a message. The payload buffer is only valid during the current user task micro step (see the user_task documentation). b_auto_↩ advance is an out-parameter (meaning that you should set it), that lets you pause the user task on the current mirco step: if you set it to 0, you will enter the pause state and the payload pointer will be valid, even after the callback returns. On the other hand, the user task will not advance to the next micro step until you leave the pause state manually (see advance_user_ring). Usually, you want to set b_auto_advance to 1.

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

- sfsc/sfsc_adapter/sfsc_adapter.h

Chapter 5

File Documentation

5.1 sfsc/sfsc_adapter/sfsc_adapter.h File Reference

Public header that contains all SFSC functions.

```
#include "../platform/sfsc_strings.h"
#include "../platform/sfsc_types.h"
#include "../proto_gen/generated.pb.h"
#include "sfsc_adapter_config.h"
#include "sfsc_error_codes.h"
#include "sfsc_states.h"
```

Data Structures

- struct _sfsc_adapter_stats

  The sfsc_adapter_stats struct contains the infromation needed by the user.
- struct _sfsc_buffer

  A simple structure to store binary data and their length.
- struct _sfsc_subscriber

  Struct that contains the necessary state memory to subscribe to a publisher service.
- struct _sfsc_publisher

  State memory for a publisher service.
- struct _sfsc_server

  State memory for a server service.
- struct _sfsc_publisher_or_server

  Container to point either to a sfsc_server or sfsc_publisher.
- struct _relative_publisher_tags

  Represents service tags specific to publisher services.
- struct _relative_server_tags

  Represents service tags specific to server services.
- struct _relative_sfsc_service_descriptor

  Represents all infromation about a service.
- struct _sfsc_channel_answer

  A struct containing all information needed to answer a channel request.

Macros

- #define **SFSC_OK** 0
- #define **SERVICE_TYPE_SERVER** 1
- #define **SERVICE_TYPE_PUBLISHER** 0
- #define **UUID_LEN** 36
- #define **sfsc_adapter_stats_DEFAULT_INIT** { NULL, {UUID_LEN}, {UUID_LEN}, SFSC_STATE_NONE, 0, 0 }
- #define **sfsc_buffer_DEFAULT_INIT** { NULL, 0 }
- #define **SERVICE_TOPIC_AUTOGEN** sfsc_buffer_DEFAULT_INIT
- #define **sfsc_subscriber_DEFAULT_INIT** { sfsc_buffer_DEFAULT_INIT, NULL }
- #define **sfsc_publisher_DEFAULT_INIT** { NULL, 0, 0, {0}, SERVICE_TOPIC_AUTOGEN, 0 }
- #define **sfsc_server_DEFAULT_INIT**
- #define **sfsc_publisher_or_server_INIT_DEFAULT** { 0, NULL }
- #define **relative_publisher_tags_DEFAULT_INIT** { 0, 0, 0, 0, 0 }
- #define **relative_server_tags_DEFAULT_INIT**
- #define **relative_sfsc_service_descriptor_DEFAULT_INIT**

Typedefs

- typedef struct _sfsc_adapter sfsc_adapter

  Contains all state memory for a sfsc_adapter instance.
- typedef struct _sfsc_adapter_stats sfsc_adapter_stats

  The sfsc_adapter_stats struct contains the infromation needed by the user.
- typedef struct _sfsc_buffer sfsc_buffer

  A simple structure to store binary data and their length.
- typedef struct _sfsc_subscriber **sfsc_subscriber**
- typedef struct _sfsc_publisher **sfsc_publisher**
- typedef struct _sfsc_server **sfsc_server**
- typedef struct _sfsc_publisher_or_server sfsc_publisher_or_server

  Container to point either to a sfsc_server or sfsc_publisher.
- typedef struct _relative_publisher_tags relative_publisher_tags

  Represents service tags specific to publisher services.
- typedef struct _relative_server_tags relative_server_tags

  Represents service tags specific to server services.
- typedef struct _relative_sfsc_service_descriptor relative_sfsc_service_descriptor

  Represents all infromation about a service.
- typedef struct _sfsc_channel_answer sfsc_channel_answer

  A struct containing all information needed to answer a channel request.
- typedef void() sfsc_command_callback(sfsc_adapter *adapter, sfsc_publisher_or_server service, sfsc_bool created)

  Called when a command to create or delete a service succeeds.
- typedef void() sfsc_query_callback(sfsc_adapter *adapter, relative_sfsc_service_descriptor descriptor, sfsc_uint8 *offset, sfsc_size length, sfsc_bool is_last)

  Called during a query process with a service descriptor, or to indicate that the query process is done.
- typedef void() sfsc_request_callback(sfsc_adapter *adapter, sfsc_buffer payload, sfsc_bool timeout, void *mapping_arg, sfsc_bool *b_auto_advance)

  Invoked when an answer to a request is receieved, or when the request times out.
- typedef void() sfsc_channel_request_callback(sfsc_adapter *adapter, sfsc_bool timeout, sfsc_int8 decode↩_error, relative_sfsc_service_descriptor *descriptor, sfsc_uint8 *descirptor_offset, sfsc_size descirptor_↩length, void *mapping_arg, sfsc_bool *b_auto_advance)

  Invoked when an answer to a channel request is receieved, or when the request times out.
- typedef void() sfsc_answer_ack_callback(sfsc_adapter *adapter, sfsc_server *server, sfsc_bool timeout, void *mapping_arg)

  Invoked if an ack message for an answer was receieved, or the servers timeout condition was meet.

Functions

- sfsc_adapter_stats * adapter_stats (sfsc_adapter *adapter)

  Recommended way to access the stats of an adapter.
- sfsc_int8 start_session_bootstraped (sfsc_adapter *adapter, const char *address, int original_control_pub↩ _port)

  Starts a sfsc adapter session with bootstraping.
- sfsc_int8 start_session (sfsc_adapter *adapter, const char *address, int original_control_pub_port, int original_control_sub_port, int original_data_pub_port, int original_data_sub_port)

  Stats a sfsc adapter session without bootstraping.
- sfsc_int8 register_subscriber (sfsc_adapter *adapter, sfsc_subscriber *subscriber)

  Subscribes to a sfsc publisher through the given adapter.
- sfsc_int8 unregister_subscriber (sfsc_adapter *adapter, sfsc_subscriber *subscriber)

  Unregisters a subscriber and unsubscribe messages on that topic.
- sfsc_int8 query_services (sfsc_adapter *adapter, sfsc_query_callback *callback)

  Starts a query process to obtain registered services from the core.
- void query_services_next (sfsc_adapter *adapter, sfsc_bool next)

  Tells the framework to continue or to end a currently ongoing query process.
- sfsc_int8 register_publisher (sfsc_adapter *adapter, sfsc_publisher *publisher, sfsc_buffer name, sfsc_buffer custom_tags, sfsc_buffer output_message_type, sfsc_command_callback *callback)

  Sets up a publisher service and registers it with the core.
- sfsc_int8 register_publisher_unregistered (sfsc_adapter *adapter, sfsc_publisher *publisher)

  Sets up a publisher you can publish with, but does not register it in the cores service registry.
- sfsc_int8 unregister_publisher (sfsc_adapter *adapter, sfsc_publisher *publisher, sfsc_command_callback *callback)

  Unregisters a publisher.
- sfsc_int8 publish (sfsc_adapter *adapter, sfsc_publisher *publisher, sfsc_buffer payload)

  Publishes data through a publisher.
- sfsc_int8 request (sfsc_adapter *adapter, sfsc_buffer topic, sfsc_buffer payload, sfsc_uint64 timeout_time, sfsc_request_callback *callback, void *mapping_arg)

  Makes a request call to a server service.
- sfsc_int8 channel_request (sfsc_adapter *adapter, sfsc_buffer topic, sfsc_buffer payload, sfsc_uint64 timeout_time, relative_sfsc_service_descriptor *descriptor, sfsc_uint8 *descriptor_space, sfsc_size descriptor_space_lenght, sfsc_channel_request_callback *callback, void *mapping_arg)

  Makes a request call to a channel server service.
- sfsc_int8 register_server (sfsc_adapter *adapter, sfsc_server *server, sfsc_buffer name, sfsc_buffer custom_tags, sfsc_buffer output_message_type, sfsc_buffer input_message_type, sfsc_command_callback *callback)

  Sets up a server service and registers it with the core.
- sfsc_int8 unregister_server (sfsc_adapter *adapter, sfsc_server *server, sfsc_command_callback *callback)

  Unregisters a server.
- sfsc_int8 answer_request (sfsc_adapter *adapter, sfsc_server *server, sfsc_int32 expected_reply_id, sfsc_buffer reply_topic, sfsc_buffer *payload, void *mapping_arg, sfsc_answer_ack_callback *on_ack)

  Answers a request.
- sfsc_int8 answer_channel_request (sfsc_adapter *adapter, sfsc_server *server, sfsc_int32 expected_reply↩ _id, sfsc_buffer reply_topic, sfsc_channel_answer *channel_answer, void *mapping_arg, sfsc_answer_ack_callback *callback)

  Used to answer a channel request.
- void random_uuid (sfsc_uint8 target[UUID_LEN])

  Generates and writes a random 128bit UUID in standard-hexgroup-format to the target buffer.
- sfsc_int8 system_task (sfsc_adapter *adapter)

  Executes a single system task step on the adapter.

- sfsc_int8 user_task (sfsc_adapter *adapter)

  Executes a single system task step on the adapter.
- void advance_user_ring (sfsc_adapter *adapter)

  Leaves the user task pause state.

Variables

- const sfsc_buffer **sfsc_buffer_default**
- const relative_publisher_tags **relative_publisher_tags_default**
- const relative_server_tags **relative_server_tags_default**

5.1.1 Detailed Description

Public header that contains all SFSC functions.

5.1.2 Macro Definition Documentation

5.1.2.1 relative_server_tags_DEFAULT_INIT

```
#define relative_server_tags_DEFAULT_INIT
```

Value:

```
    {                                                                                \
        0, 0, 0, 0, 0, 0,                                                            \
            sfsc_SfscServiceDescriptor_ServiceTags_ServerTags_AckSettings_init_default \
    }
```

5.1.2.2 relative_sfsc_service_descriptor_DEFAULT_INIT

```
#define relative_sfsc_service_descriptor_DEFAULT_INIT
```

Value:

```
    {                                                                  \
        sfsc_SfscId_init_default, sfsc_SfscId_init_default,            \
            sfsc_SfscId_init_default, 0, 0, 0, 0, SERVICE_TYPE_PUBLISHER, \
            relative_publisher_tags_DEFAULT_INIT                       \
    }
```

5.1.2.3 sfsc_server_DEFAULT_INIT

```
#define sfsc_server_DEFAULT_INIT
```

Value:

```
    {                                                                                \
        NULL,                                                                        \
            sfsc_SfscServiceDescriptor_ServiceTags_ServerTags_AckSettings_init_default, \
            sfsc_SfscId_init_default, SERVICE_TOPIC_AUTOGEN, 0                       \
    }
```

5.1.3 Typedef Documentation

5.1.3.1 relative_publisher_tags

```
typedef struct _relative_publisher_tags relative_publisher_tags
```

Represents service tags specific to publisher services.

The format and idea of the fields is very similar to relative_sfsc_service_descriptor, see there for an explanation.

An exception to this is the unregistered field. See the sfsc_publisher struct documentation for an explanation.

5.1.3.2 relative_server_tags

```
typedef struct _relative_server_tags relative_server_tags
```

Represents service tags specific to server services.

The format and idea of the fields is very similar to relative_sfsc_service_descriptor, see there for an explanation.

An exception to this is the ack_settings field. See the sfsc_server struct documentation for an explanation.

5.1.3.3 relative_sfsc_service_descriptor

```
typedef struct _relative_sfsc_service_descriptor relative_sfsc_service_descriptor
```

Represents all infromation about a service.

A relative_sfsc_service_descriptor is relative because it does not contain information about a service directly, but it merely serves as an index structure. The indexes (which end with _offset) are relative to a memory area. The memory areas address (called start in the following) is usually delivered with the descriptor. For example, to now access the name of the service, read name_len bytes from (start+name_offset). For reasoning, why this relative apporach is used, read the last paragraph.

An exception to the above are the core_id, adapter_id, and service_id fields, which actually contain the respective ids, and thereby denote the core and adapter this service belongs to, as well as this services own id.

The service_type field is either SERVICE_TYPE_SERVER or SERVICE_TYPE_PUBLISHER and indicates, how the service_tags union should be treated.

Why is this relative approach used? The binary size of a serivce in sfsc is not limited, so we can not know the size of the respective service fields in advance. On the other hand, this framework does not use dynamic memory allocation (malloc). A possible solution is to statically allocate memory for each field and add an length field, to indicate, how much memory is actually used. The difference between the allocated and actually used size is called waste. Instead of allocation a memory area for each field, we use one bigger memory area for all fields. The idea is that, because of the indivudual waste, this single field size can be smaller then sum of all indivudual field sizes, while still containing enough space to store all necessary information.

5.1.3.4 sfsc_adapter

```
typedef struct _sfsc_adapter sfsc_adapter
```

Contains all state memory for a sfsc_adapter instance.

In most cases, you do not need to interact with the fields of a sfsc_adapter struct directly. Therefor, the struct members are not exposed in this header. The for you relevant fields should be accessed throught the adapter_stats function.

Note however, that the compilation unit declaring the sfsc_adapter struct needs a full specification of it. Only this compilation unit should include sfsc_adapter_struct.h.

5.1.3.5 sfsc_adapter_stats

```
typedef struct _sfsc_adapter_stats sfsc_adapter_stats
```

The sfsc_adapter_stats struct contains the infromation needed by the user.

The stats of an adapter should be accessd throught the adapter_stats function. The fields of this struct are all read-only and should not be modified by you.

The address field specifies the address of the core and is implicitly set by you during the start_session_bootstraped or start_session functions. The format of the address is up to you and can be anything, as long it is understood by your implementation of socket_connect (see sfsc_sockets.h).

The adapter_id and core_id fields indicate this adapters id and the id of the core it is connected to respectivly. They are both filled during the handshake, and ready to use once the adapters state is operational.

The state indicates to current connection state of an adapter. The various states are defined in sfsc_states.↩ h and are all prefixed with SFSC_STATE_. An adapter is considered operational if the value of this field is >= SFSC_STATE_OPERATIONAL.

Due to the execution and memory model of this framework it can occur that some messages receieved from the network are dropped (see the system_task function for details). The discarded_message_count keeps track of the number of lost messages.

The query_in_progress is set to 1 if you started a query process using query_services, and will be reset to 0 if the query process is terminated. See the query functions for more information.

5.1.3.6 sfsc_answer_ack_callback

```
typedef void() sfsc_answer_ack_callback(sfsc_adapter *adapter, sfsc_server *server, sfsc_bool
timeout, void *mapping_arg)
```

Invoked if an ack message for an answer was receieved, or the servers timeout condition was meet.

If timeout is set to 1 all retransmission attempts failed (see the sfsc_server.ack_settings), if it is set to 0, the requestor acknowledged the answer.

The mapping_arg parameter is a mapping-argument (explained in the request functions documentation with an example).

5.1.3.7 sfsc_buffer

```
typedef struct _sfsc_buffer sfsc_buffer
```

A simple structure to store binary data and their length.

This structure type is widely used in the framework for compact data storing. It is important to notice that some functions will take a sfsc_buffer struct as parameter directly, while others will work with pointers. In the most cases, this says something about the mutality of the content.

In general, for the use-time of a sfsc_buffer (usually defined in the respective functions documentation), the memory area the buffer points to must be valid and of the in the struct specified length.

When passing the struct to a function, it must be immutable, meaning that for the use-time of the sfsc_buffer, the content must not change.

When passing a pointer to a function, the content is allowed to be mutable, meaning that during the use-time of the sfsc_buffer, you are allowed to change the content pointer and thereby change the memory area are this sfsc_buffer is backed by. If you do so, don't forget to update the length field accordingly.

5.1.3.8 sfsc_channel_answer

```
typedef struct _sfsc_channel_answer sfsc_channel_answer
```

A struct containing all information needed to answer a channel request.

Instead of normal binary payload, a channel request is answered with the definition of a publisher service.

The core_id, adapter_id and service_id represent the core and adapter the publisher service is conencted to, as well as the publisher service itself. Their length must be UUID_LEN and their format must be standard-hexgroup-format (see the random_uuid function). The reason a sfsc_channel_answer uses sfsc_uint8∗ instead of sfsc_uint8[] is to remove copying and to save memory: The publisher you are describing is most likly hosted by the same adapter that will send this channel answer. Instead of allocating 3∗UUID_LEN memory and copy the details from the adapter to the sfsc_channel_answer, you can just let the fields point to &adapter_states()->core_id and &adapter_states()->adapter_id respectivly.

All sfsc_buffers must be valid during the use-time of the sfsc_channel_answer struct and are optional. If publisher_output_topic is set to SERVICE_TOPIC_AUTOGEN the topic autogenerate rule (as described in sfsc_publisher) will be applied, and the service_id will be used as topic.

The unregistered should be set to 0 if the publisher you are describing in this sfsc_channel_answer is also registered in the service registry, or to 1 if it is not.

5.1.3.9 sfsc_channel_request_callback

```
typedef void() sfsc_channel_request_callback(sfsc_adapter *adapter, sfsc_bool timeout, sfsc_int8 decode_error, relative_sfsc_service_descriptor *descriptor, sfsc_uint8 *descirptor_offset, sfsc_size descirptor_length, void *mapping_arg, sfsc_bool *b_auto_advance)
```

Invoked when an answer to a channel request is receieved, or when the request times out.

The timeout parameter is set to 1 if the callback invokation is caused by a timeout, to 0 if an answer is deleivered.

The mapping_arg parameter is a mapping-argument (explained in the request functions documentation with an example).

If decode_error is not SFSC_OK, an error occured during decoding the received service definition. This will most likley happen due to a too small memory area for the descriptor. In this case decode_error is set to E_BUFFER_INSUFFICIENT, and descirptor_length indicates, how much memory would have been needed.

In contrast to the payload in a normal sfsc_request_callback, you have allocated the descriptor struct and the descriptor memory area yourself, so they will be valid until you do something with them, and won't become invalid once the callback returns. Thus, the b_auto_advance does not influence the validity, and is just there as a convenience tool to enter the user task pause state (see the user_task documentation).

5.1.3.10 sfsc_command_callback

```
typedef void() sfsc_command_callback(sfsc_adapter *adapter, sfsc_publisher_or_server service,
sfsc_bool created)
```

Called when a command to create or delete a service succeeds.

A create command means that the service was registered in the cores event-log, a delete command means taht the service was unregistred from it.

Parameters

adapter	the executing adapter
service	a struct describing the service
created	1 if the command was a create command, 0 if it was a delete command

5.1.3.11 sfsc_query_callback

```
typedef void() sfsc_query_callback(sfsc_adapter *adapter, relative_sfsc_service_descriptor
descriptor, sfsc_uint8 *offset, sfsc_size length, sfsc_bool is_last)
```

Called during a query process with a service descriptor, or to indicate that the query process is done.

The descriptor is valid as long as you don't continue the query process (using query_services_next). If you need to access the service data of a service delivered by the callback after continuation, you need to write the descriptor to a global place, and also copy length bytes from offset to a global place. Alternatively, if you call query_services_next to end the query process (by setting next to 0), the descriptor is valid until the next query process.

In the last call to this method, is_last is set to 1. The last call might not contain a service. An offset value of NULL indicates that this invokation of the callback does not contain a service.

Even after the is_last call you have to invoke query_services_next with next set to 0, to explicitly end the query process.

5.1.3.12 sfsc_request_callback

```
typedef void() sfsc_request_callback(sfsc_adapter *adapter, sfsc_buffer payload, sfsc_bool
timeout, void *mapping_arg, sfsc_bool *b_auto_advance)
```

Invoked when an answer to a request is receieved, or when the request times out.

The timeout parameter is set to 1 if the callback invokation is caused by a timeout, to 0 if an answer is deleivered.

mapping_arg is a mapping-argument (explained in the request functions documentation with an example).

The payload buffer is only valid during the current user task micro step (see the user_task documentation). b_↩ auto_advance is an out-parameter (meaning that you should set it), that lets you pause the user task on the current mirco step: if you set it to 0, you will enter the pause state and the payload pointer will be valid, even after the callback returns. On the other hand, the user task will not advance to the next micro step until you leave the pause state manually (see advance_user_ring). Usually, you want to set b_auto_advance to 1. If the request timed out, you should not modify b_auto_advance, as it will point to NULL.

5.1.4 Function Documentation

5.1.4.1 adapter_stats()

```
sfsc_adapter_stats* adapter_stats (
            sfsc_adapter * adapter )
```

Recommended way to access the stats of an adapter.

Parameters

adapter	The target adapter

Returns

sfsc_adapter_stats* Pointer to that adapers stats

5.1.4.2 advance_user_ring()

```
void advance_user_ring (
            sfsc_adapter * adapter )
```

Leaves the user task pause state.

Some callback functions allow you to pause the user task and freeze it on the current micro step. To continue execution you have to call this function. You should only call this funtion if you entered the pause state, and never call it from the callback you entered the pause state, as this will skip messages and leave your adapter in a undefined state.

Parameters

adapter	The adapter whichs user task is currently in the pause state and should continue execution

5.1.4.3 answer_channel_request()

```
sfsc_int8 answer_channel_request (
            sfsc_adapter * adapter,
            sfsc_server * server,
            sfsc_int32 expected_reply_id,
            sfsc_buffer reply_topic,
            sfsc_channel_answer * channel_answer,
            void * mapping_arg,
            sfsc_answer_ack_callback * callback )
```

Used to answer a channel request.

This function must only be called by channel server services. It behaves just like answer_request, so see there for documentation.

The only difference is that instead of passing a pointer to a binary payload, you have to pass a pointer to a sfsc_↩ channel_answer. The channel_answers use-time equals the use-time of the payload in the answer_request function. It is also mutable, meaning that you are allowed to edit it.

5.1.4.4 answer_request()

```
sfsc_int8 answer_request (
            sfsc_adapter * adapter,
            sfsc_server * server,
            sfsc_int32 expected_reply_id,
            sfsc_buffer reply_topic,
            sfsc_buffer * payload,
            void * mapping_arg,
            sfsc_answer_ack_callback * on_ack )
```

Answers a request.

After sending an answer, the original requestor will (hopefully) receive it and send an acknowledge for the answer. If the acknowledge is received in time, the on_ack callback will be invoked. If not, an attempt is automatically made to send it again. The maximal send again attempt count and the wait time are configured in the server_struct (see there for more information). If a server does not require its answers to be acknowledged and which are thus only send once, it is called a fire-and-forget server.

The expected_reply_id and reply_topic are given to you by the on_request function of the corresponding sfsc_↩ server. The use-time of the reply_topic is the time until the on_ack callback is invoked. During this time, it must be valid and immutable. For fire-and-forget servers, the use-time of the reply_topic is only this functions runtime, meaning that its only neccessary to be vaid and immutable until this function returns.

The payload parameter is a pointer to the actual payload you want to transmitt in the answer. The use-time for this is either until the on_ack callback is invoked, or for fire-and-forget servers, this functions runtime. Since this is a pointer to a sfsc_buffer and not a sfsc_buffer, both, the pointer to the sfsc_buffer and the content pointer inside that buffer must be valid during the use-time. The payload buffer can be mutable: you are allowed to change the content pointer or the content it points to. This is usefull in some situation, e.g.: Imaging answering a request with a sensor measurement, stored behind payload->content. You do not receieve an acknowledge in time, so the answer is send again. But during this peroid, the the measurement changed. Since you were allowed to change payload->content, you updated it, and the retransmission of the answer will contain the new measurement.

Answering a request to an non-fire-and-forget server will require a free acknowledge memory slot. If an attempt is made to answer a request on such a server and there are already MAX_PENDING_ACKS answer processes ongoing, a E_NO_FREE_SLOT will be returned. The acknowledge memory slot is freed once the on_ack callback is invoked.

The mapping_arg parameter is a mapping-argument (explained in the request functions documentation with an example).

Parameters

adapter	An operational adapter
server	The server that is answering a request
expected_reply↩ _id	An answer identification number obtained by the original request
reply_topic	The answers reply topic, obtained by the original request
payload	The actual answer payload
mapping_arg	Optional; Serves as mapping-argument
on_ack	A callback to invoke once an acknowledge is received, can be NULL

Returns

sfsc_int8 SFSC_OK or one of the error codes below

Return values

E_NO_FREE_SLOT	If there is no free acknowledge memory slot while trying to answer a non-fire-and-forget request
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.5 channel_request()

```
sfsc_int8 channel_request (
            sfsc_adapter * adapter,
            sfsc_buffer topic,
            sfsc_buffer payload,
            sfsc_uint64 timeout_time,
            relative_sfsc_service_descriptor * descriptor,
            sfsc_uint8 * descriptor_space,
            sfsc_size descriptor_space_lenght,
            sfsc_channel_request_callback * callback,
            void * mapping_arg )
```

Makes a request call to a channel server service.

This function behaves like the request function, so see there for documentation. The only difference is, what the answer is, and how it is delivered.

Instead of normal binary payload, a channel_request will result in a publisher service description as answer, in the same format as the query_service function would return it. The difference to the query_service function is, that the framework can not know, how many simultaneous channel requests you want to make. Thus it can not statically allocate memory for the service descriptors. As consequence, you have to allocate this memory yourself and pass it to this method. The memory must be large enough to hold the received service. If you do not know anything about the size of the service, it might be a good idea to use REGISTRY_BUFFER_SIZE as orientation. What happens if the descriptor_space_length is insufficient is described in the sfsc_channel_request_callback documentation.

Parameters

descriptor	Pointer to a relative_sfsc_service_descriptor which will be filled
descriptor_space	Pointer to the start of the usable memory area
descriptor_space_lenght	Length of the usable memory area

5.1.4.6 publish()

```
sfsc_int8 publish (
            sfsc_adapter * adapter,
```

```
            sfsc_publisher * publisher,
            sfsc_buffer payload )
```

Publishes data through a publisher.

The publisher must be registered with this adapter, either by the register_publisher or the register_publisher_↩ unregistered function.

The use-time of the payload is this functions runtime and must be immutable for the duration.

Parameters

adapter	An operational adapter this publisher is registered to
publisher	The publisher to publish the data
payload	The data to publish

Returns

sfsc_int8 SFSC_OK

5.1.4.7 query_services()

```
sfsc_int8 query_services (
            sfsc_adapter * adapter,
            sfsc_query_callback * callback )
```

Starts a query process to obtain registered services from the core.

Only one query process might be running simultaniously. During a query process, all valid services will be delivered through the callback. The query process queries the cores event log in reversed order, so that more recently registered services will be found first.

After a service is found and delivered to the calling application throught the callback, the query process is paused, until it is explicitly continued by a call to the query_services_next function. See the documentation of the sfsc_↩ query_callback for more information on the delivered service data.

Parameters

adapter	An operational adapter throught which the core is queried
callback	The callback all receieved services are delivered to

Returns

sfsc_int8 SFSC_OK if starting the query process was successfull, or one of the error codes below

Return values

E_QUERY_IN_PROGRESS	If there is already an other query process
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.8 query_services_next()

```
void query_services_next (
            sfsc_adapter * adapter,
            sfsc_bool next )
```

Tells the framework to continue or to end a currently ongoing query process.

Once a service is found during the query process, the associated callback is invoked and the query process enters a pause state. It is then either continued by a call to this function with next set to 1 or ended with next set to 0.

It is not neccesary, but allowed, that this function is invoked from a query callback. It is also allowed for it to be invoked from any other part of the programm, as long as it is invoked eventually.

Calling this function while there is no query process ongoing (checkable by the adapter_stats function) will result in undefined behaviour.

Parameters

adapter	The adapter which is currently in a query process
next	1 to continue the query process, 0 to end it

5.1.4.9 random_uuid()

```
void random_uuid (
            sfsc_uint8 target[UUID_LEN] )
```

Generates and writes a random 128bit UUID in standard-hexgroup-format to the target buffer.

Since the generated UUID is random, it can be classified as type 4 UUID.

This method generates 2 random sfsc_uint64 and formats them in standard-hexgroup-format. Standard-hexgroup-format consists of 5 datagroups, each group represended by ASCII letters a-f and numbers 0-9. The groups are separated by the - ASCII symbol.

Examples: 23236572-6963-6973-7473-757065722323, 550e8400-e29b-11d4-a716-446655440000

Parameters

target	A at least UUID_LEN bytes long buffer to write the formated UUID to

5.1.4.10 register_publisher()

```
sfsc_int8 register_publisher (
            sfsc_adapter * adapter,
```

```
sfsc_publisher * publisher,
sfsc_buffer name,
sfsc_buffer custom_tags,
sfsc_buffer output_message_type,
sfsc_command_callback * callback )
```

Sets up a publisher service and registers it with the core.

If you want to set up the services id and topic manually, see the sfsc_publisher struct documentation for instructions.

After successfull registration, this function will set the unregistered field of the publisher to 0.

All sfsc_buffer parameters for this method must be immutable and mut be valid until this function returns (their use-time equals this functions run time). For more information about mutability, see the sfsc_buffer struct documentation. To omit an optional sfsc_buffer parameter use sfsc_buffer_default as value.

Registering a publisher saves a pointer to that publisher in the adapter state, so you must not copy the publisher struct around as long as it is registered with the adapter! Also, registering will fail if there is no free publisher memory slot to store the pointer to that publisher in the adapter. In this case E_NO_FREE_SLOT will be returned. The maximal number of at the same time registered publisher per adapter can be configured using MAX_PUBLISHERS.

Also, since the publisher will be registered in the service registry, this call requires a free command memory slot and will also return E_NO_FREE_SLOT if their are currently already MAX_SIMULTANIOUS_COMMANDS ongoing. The command memory slot will be occupied until the callback is invoked.

Parameters

adapter	An operational adapter
publisher	The publisher to register with this adapter, see the sfsc_publisher struct for more information
name	The name of this publisher
custom_tags	Optional; custom tags for the publisher
output_message_type	Optional; can be used to annotate the format of the published messages
callback	Optional; a callback that is invoked after service registration with the core was successful

Returns

sfsc_int8 SFSC_OK or one of the error codes below

Return values

SFSC_OK	This indicates that registering the publisher into the adapter was successfull, and that the appropriate command to register the publisher into the cores service registry was issued.
E_NO_FREE_SLOT	There are already MAX_PUBLISHERS registered with this adapter, so there is no memory slot left for this one OR if there are currently already MAX_SIMULTANIOUS_COMMANDS ongoing
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.11 register_publisher_unregistered()

```
sfsc_int8 register_publisher_unregistered (
            sfsc_adapter * adapter,
            sfsc_publisher * publisher )
```

Sets up a publisher you can publish with, but does not register it in the cores service registry.

This function behaves mostly like register_publisher, but since this publisher is not registered into the service registry, it does not have a name, custom_tags or an ouput_message_type.

It will still have an service id and a topic, which are either setup manually or automaticly generated, according to the rules specified in the sfsc_publisher documentation. Also, for registering the publisher with the adapter, the adapter needs to have a free publisher memory slot (again, see register_publisher).

This function will set the unregistred field of the publisher to 1.

A subscriber can subscribe to an unregistred publisher, if it knows the unregistered publishers topic. In most cases, this will happen in the context of a channel request.

Even if the publisher is not registered into the service registry, it sill needs to be properly unregistered form the adapter after usage by the unregister_publisher function!

Parameters

adapter	An operational adapter
publisher	The publisher to register with this adapter, see the sfsc_publisher struct for more information

Returns

sfsc_int8 SFSC_OK or one of the error codes below

Return values

SFSC_OK	This indicates that registering the publisher into the adapter was successfull, and that the appropriate command to register the publisher into the cores service registry was issued.
E_NO_FREE_SLOT	There are already MAX_PUBLISHERS registered with this adapter, so there is no memory slot left for this one
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.12 register_server()

```
sfsc_int8 register_server (
            sfsc_adapter * adapter,
            sfsc_server * server,
            sfsc_buffer name,
            sfsc_buffer custom_tags,
            sfsc_buffer output_message_type,
```

```
        sfsc_buffer input_message_type,
        sfsc_command_callback * callback )
```

Sets up a server service and registers it with the core.

If you want to set up the services id and topic manually, see the sfsc_server struct documentation for instructions.

Also, the function that is invoked when recieving requests, the servers acknowledge strategy and if this is a channel service must be configured in the sfsc_server before calling this function, so see their for instructions.

All sfsc_buffer parameters for this method must be immutable and must be valid until this function returns (their use-time equals this functions run time). For more information about mutability, see the sfsc_buffer struct documentation. To omit an optional sfsc_buffer parameter use sfsc_buffer_default as value.

Registering a server saves a pointer to that server in the adapter state, so you must not copy the server struct around as long as it is registered with the adapter! Also, registering will fail if there is no free server memory slot to store the pointer to that server in the adapter. In this case E_NO_FREE_SLOT will be returned. The maximal number of at the same time registered server per adapter can be configured using MAX_SERVERS.

Also, since the server will be registered in the service registry, this call requires a free command memory slot and will also return E_NO_FREE_SLOT if their are currently already MAX_SIMULTANIOUS_COMMANDS ongoing. The command memory slot will be occupied until the callback is invoked.

Parameters

adapter	An operational adapter
server	The server to register with this adapter, see the sfsc_server struct for more information
name	The name of this server
custom_tags	Optional; custom tags for the server
output_message_type	Optional; can be used to annotate the format of the messages the server sends as answers
input_message_type	Optional; can be used to annotate the format of the messages the server accepts on requests
callback	Optional; a callback that is invoked after service registration with the core was successful

Returns

sfsc_int8 SFSC_OK or one of the error codes below

Return values

SFSC_OK	This indicates that registering the server into the adapter was successfull, and that the appropriate command to register the into into the cores service registry was issued.
E_NO_FREE_SLOT	There are already MAX_SERVERS registered with this adapter, so there is no memory slot left for this one OR if there are currently already MAX_SIMULTANIOUS_COMMANDS ongoing
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.13 register_subscriber()

```
sfsc_int8 register_subscriber (
            sfsc_adapter * adapter,
            sfsc_subscriber * subscriber )
```

Subscribes to a sfsc publisher through the given adapter.

The publishers topic which should be subscribed to and the callback function which should be invoked can be set in the corresponding sfsc_subscriber struct. For mutability rules of topic and callback function (if they are allowed to change), see the sfsc_subscriber structs documentation.

Registering a subscriber saves a pointer to that subscriber in the adapter state, so you must not copy the subscriber struct around as long as it is registered with the adapter! Also, registering will fail if there is no free subscriber memory slot to store the pointer to that subscriber in the adapter. In this case E_NO_FREE_SLOT will be returned. The maximal number of at the same time registered subscribers per adapter can be configured using MAX_↩ SUBSCRIBERS.

Calling this function with an non-operational adapter will result in unpredictable behaviour! Registering the same subscriber multiple times will also result in unpredictable behaviour!

Parameters

adapter	An operational adapter which will be used for the subscription
subscriber	Pointer to the subscriber struct holding the topic and callback information

Returns

sfsc_int8 SFSC_OK on success or one of the error codes below

Return values

E_NO_FREE_SLOT	There are already MAX_SUBSCRIBERS registered with this adapter, so there is no memory slot left for this one
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.14 request()

```
sfsc_int8 request (
            sfsc_adapter * adapter,
            sfsc_buffer topic,
            sfsc_buffer payload,
            sfsc_uint64 timeout_time,
            sfsc_request_callback * callback,
            void * mapping_arg )
```

Makes a request call to a server service.

The topic of the server service is usually obtained by a registry query. The use-time of the topic and the payload equal the functions runtime (until this function returns) and must both be immutable for this time (see the sfsc_buffer documentation for more information about mutability).

If an answer to the request is received, the callback is invoked. For information on how the data will be delivered, see the callbacks documentation.

You can specify a timeout_time in ms. If this time passes without receiving an answer to this request, the callback is invoked with timeout set to 1.

Each request you want to make needs a request memory slot in the given adapter for the time of the request (until the callback is invoked). If there is no free request memory slot, E_NO_FREE_SLOT will be returned. You can configure the amount of available request memory slots using MAX_SIMULTANIOUS_REQUESTS.

As the use-time of the request topic and payload is only the functions runtime, they are both not necessarily valid when an answer is recieved, and thus can not be passed to the callback. But most certainly you want to know to which request the recieved answer belongs. One way is to declare a separate callback function for each of your requests. An other way to let you know which request function call the callbacks invokation belongs to, is to use the optional mapping_arg parameter as a so called mapping-argument. If and what you store behind the mapping_arg pointer is opaque to the framework. The callback function will be invoked with this mapping_arg. You can then, based on the mapping_arg, reason, what request call the answer belongs to. For example, if you know that even if the use-time of the request topic buffer is over it will continue to be valid, you can pass this as mapping_arg. Or you define that you will use the mapping_arg pointer as numerical value and treat it as an normal integer number instead of a pointer. Then you can assign unique request ids to your topics.

Parameters

adapter	The operational adapter to make the request with
topic	The server services topic to make the request to
payload	The payload of the request
timeout_time	The timeout time in ms, 0 for no timeout restrictions
callback	The callback that is invoked when receiving an answer or on timeout
mapping_arg	Optional; Serves as mapping-argument (see the text above for explanation)

Returns

sfsc_int8 SFSC_OK or one of the error codes below

Return values

SFSC_OK	This indicates that registering the publisher into the adapter was successfull, and that the appropriate command to register the publisher into the cores service registry was issued.
E_NO_FREE_SLOT	There currently already MAX_SIMULTANIOUS_REQUESTS MAX_PUBLISHERS ongoing
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.15 start_session()

```
sfsc_int8 start_session (
            sfsc_adapter * adapter,
            const char * address,
            int original_control_pub_port,
            int original_control_sub_port,
```

```
            int original_data_pub_port,
            int original_data_sub_port )
```

Stats a sfsc adapter session without bootstraping.

See start_session_bootstraped for details.

5.1.4.16 start_session_bootstraped()

```
sfsc_int8 start_session_bootstraped (
            sfsc_adapter * adapter,
            const char * address,
            int original_control_pub_port )
```

Starts a sfsc adapter session with bootstraping.

The adapter is not operational after this function returns. It first needs to do a handshake and connect to the other sfsc sockets of the core. You should start system-tasking (see the system_task function) with this adapter and check the state field of the stats object (accessible with the adapter_stats function) after each step. It will eventually become SFSC_STATE_OPERATIONAL, making the adapter operational.

Parameters

adapter	Pointer to the adapter struct, all state information will be saved there
address	The address of the core, it will be passed to your socket implementation
original_control_pub_port	The port of the cores control pub socket

Returns

sfsc_int8 SFSC_OK or an error code

5.1.4.17 system_task()

```
sfsc_int8 system_task (
            sfsc_adapter * adapter )
```

Executes a single system task step on the adapter.

This function is non-blocking, and must be called cyclicly, with a high enough frequency on an adapter, on which the start_session or start_session_bootstraped function was called. To read more about the execution model, see the readme.

This function returns an error code, which is SFSC_OK on success. An other return code indicates an error, a list or error codes can be found in sfsc_errors.h and zmtp_states.h. Some errors are recoverable, again, see the readme.

Parameters

adapter	An adapter

Returns

sfsc_int8 SFSC_OK or an error code

5.1.4.18 unregister_publisher()

```
sfsc_int8 unregister_publisher (
            sfsc_adapter * adapter,
            sfsc_publisher * publisher,
            sfsc_command_callback * callback )
```

Unregisters a publisher.

Unregistering frees the publishers publisher memory slot in the adapter.

If this publisher is registered with the service registry, it will be removed from it. Removing requires a free command memory slot and will return E_NO_FREE_SLOT if there are currently already MAX_SIMULTANIOUS_COMMANDS ongoing. The command memory slot will be occupied until the callback is invoked.

Even if the publisher is not registered with the service registry (because it was created with register_publisher_↩ unregistered), this function must still be called!

The callback is invoked after the publisher was removed from the service registry, with created set to 0. Since with unregistered publishers this is never the case, the callback is ignored and should be set to NULL.

Parameters

adapter	An operational adapter
publisher	The publisher to shut down
callback	For registered publishers only; Invoked after the publisher was removed from the service registry

Returns

sfsc_int8 SFSC_OK or one of the error codes below

Return values

SFSC_OK	This indicates that unregistering the publisher from the adapter was successfull, and if this publisher is registered in the service registry, that the appropriate command to unregister the publisher from there was issued
E_NO_FREE_SLOT	There are currently already MAX_SIMULTANIOUS_COMMANDS ongoing
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.19 unregister_server()

```
sfsc_int8 unregister_server (
            sfsc_adapter * adapter,
```

```
            sfsc_server * server,
            sfsc_command_callback * callback )
```

Unregisters a server.

Unregistering frees the servers server memory slot in the adapter.

The server will be removed from the service registry. Removing requires a free command memory slot and will return E_NO_FREE_SLOT if there are currently already MAX_SIMULTANIOUS_COMMANDS ongoing. The command memory slot will be occupied until the callback is invoked. The callback is invoked after the server was removed from the service registry, with created set to 0.

Parameters

adapter	An operational adapter
server	The server to shut down
callback	Invoked after the server was removed from the service registry

Returns

sfsc_int8 SFSC_OK or one of the error codes below

Return values

SFSC_OK	Indicates that unregistering the server from the adapter was successfull and that the appropriate command to unregister the server from the service registry was issued
E_NO_FREE_SLOT	There are currently already MAX_SIMULTANIOUS_COMMANDS ongoing
Various	Network Errors There are serveral other, network-related errors that can occure

5.1.4.20 unregister_subscriber()

```
sfsc_int8 unregister_subscriber (
            sfsc_adapter * adapter,
            sfsc_subscriber * subscriber )
```

Unregisters a subscriber and unsubscribe messages on that topic.

If the given subscriber is registered with this adapter to receieve published messages, it will be unregistered and a subscriber memory slot in the adapter will be freed. After this function returns, the callback of the subscriber will not be invoked again.

Unregistering a not registered subscriber will result in a success.

Parameters

adapter	The operational adapter, from which to unregister the subscriber
subscriber	The subscriber to remove

Returns

sfsc_int8 SFSC_OK on success or one of the Various Network Errors

5.1.4.21 user_task()

```
sfsc_int8 user_task (
            sfsc_adapter * adapter )
```

Executes a single system task step on the adapter.

The adapter must be in an operational state, or this call will lead to undefined behaviour.

If this function is called, it will execute callbacks you registered. Whether using blocking or long taking code in your callbacks is allowed or not, depends on your execution modell (see the corresponding section in the readme).

It is important to continue calling this function, even if you entered the user task pause state (mentioned in the documentation of most callback functions).

This function returns an error code, which is SFSC_OK on success. An other return code indicates an error, a list or error codes can be found in sfsc_errors.h and zmtp_states.h. Some errors are recoverable, again, see the readme.

Parameters

adapter	An operational adapter

Returns

sfsc_int8 SFSC_OK or an error code

5.2 sfsc/sfsc_adapter/sfsc_error_codes.h File Reference

SFSC related error codes.

Macros

- #define E_PROTO_DECODE -19

  Failed to decode an incoming sfsc message.
- #define E_PROTO_ENCODE -20

  Failed to encode an outgoing sfsc message.
- #define E_TOO_SLOW -21

  This incdicates that you are calling the system_task function with a too low frequency.
- #define E_HEARTBEAT_MISSING -22

  The core failed to send a heartbeat in time.
- #define W_MESSAGE_DROP -23

  Indicates that the system task was not able to write a message to the user ring, so the message is dropped instead.
- #define E_QUERY_IN_PROGRESS -24

  Indicates that there is already a query process ongoing.
- #define E_NO_FREE_SLOT -25

  Indicates that an adapter has no free memory slot of a certain type, but a function would need one.

5.2.1 Detailed Description

SFSC related error codes.

5.2.2 Macro Definition Documentation

5.2.2.1 E_HEARTBEAT_MISSING

```
#define E_HEARTBEAT_MISSING -22
```

The core failed to send a heartbeat in time.

If this error happens, the core is most likely not longer available, either because of a core or a network failure. You should treat this session as expired and start a new one.

This error is not recoverable, meaning that you start a new sfsc session once you encounter it!

5.2.2.2 E_NO_FREE_SLOT

```
#define E_NO_FREE_SLOT -25
```

Indicates that an adapter has no free memory slot of a certain type, but a function would need one.

This error is recoverable, meaning that you can continue to use the sfsc adapter.

5.2.2.3 E_PROTO_DECODE

```
#define E_PROTO_DECODE -19
```

Failed to decode an incoming sfsc message.

This usually means that your communication peer sends malformed messages, or at least that you are receiveing malformed messages. There is not much you can do to prevent this error.

This error is not recoverable, meaning that you start a new sfsc session once you encounter it!

5.2.2.4 E_PROTO_ENCODE

```
#define E_PROTO_ENCODE -20
```

Failed to encode an outgoing sfsc message.

This usually happens if you call a sfsc function with malformed content. If you are working with sfsc_buffers, check that they are valid for as long as the function you are using requires them.

This error is not recoverable, meaning that you start a new sfsc session once you encounter it!

5.2.2.5 E_QUERY_IN_PROGRESS

```
#define E_QUERY_IN_PROGRESS -24
```

Indicates that there is already a query process ongoing.

This error will usually occur if there is already a query process ongoing, and you try to start another one. Keep in mind that you have to end a query process manually and that it counts as running until you end id. See the qery_services function for more details.

This error is recoverable, meaning that you can continue to use the sfsc adapter.

5.2.2.6 E_TOO_SLOW

```
#define E_TOO_SLOW -21
```

This incdicates that you are calling the system_task function with a too low frequency.

This error happens if the system task can not send outgoing heartbeats in time. The solution is not to rise the HEARTBEAT_SEND_RATE_MS, but to call the system task more often.

This error is not recoverable, meaning that you start a new sfsc session once you encounter it!

5.2.2.7 W_MESSAGE_DROP

```
#define W_MESSAGE_DROP -23
```

Indicates that the system task was not able to write a message to the user ring, so the message is dropped instead.

To understand the relation between system task and user ring, see the readme, section execution model.

This is more a warning than a error, and the framework should continue working.

If you encounter message drops very frequent, try to rise USER_RING_SIZE (see sfsc_adapter_config.h).