FailSafe over EtherCAT User Manual

rt:labs

FailSafe over EtherCAT User Manual

rt-labs AB

https://rt-labs.com

Revision: 588

Contents

Chapter 1

About this manual

1 Scope

This user manual describes how to configure, create and use an FSoE master to slave connection or an FSoE slave to master connection in accordance with the ETG.5100 "Safety Over EtherCAT" specification using the provided FSoE stack implementation. The user manual does not include instructions or guidelines for how to implement functional safety software or hardware sufficient to qualify it as an FSoE device; instead follow the safety functional requirements described in ETG.5100 ch 6.1 "Safety functional requirements". ETG.9100 "FSoE Policy" defines further rules and requirements for using and implementing the Safety over EtherCAT technology. All requirements defined in the ETG.9100 that are applicable for a device shall be fully met. The user manual contains implementation checklists, in chapter Checklists, that the user shall go through.

2 Word usage: shall, should, may, can

The word **shall** is used to indicate mandatory requirement to be strictly followed.

The word **should** is used to indicate that among several possibilities one is recommended as particular suitable.

The word **may** is used to indicate a course of action permissible within the limits of a standard.

The word **can** is used for statements of possibility or capability.

3 Abbreviations

- FSoE - FailSafe over EtherCat or Safety over EtherCat
- PDU - Protocol Data Unit
- SCL - Safety Communication Layer. The FSoE stack is an SCL
- SPDU - Safety PDU

4 References

1. IEC 61784-3 Industrial communication networks - Profiles - Part 3: Functional Safety fieldbuses

2. ETG.5100 S(D) V1.2.0 - Safety Over EtherCAT specification

3. ETG.5120 S(R) V1.1.0 - FSoE Protocol Enhancements

4. ETG.5101 G(R) V1.3.0 - FSoE Implementation Guide

5. ETG.9100 S(R) V1.1.0 - FSoE Policy

6. Google Test framework - https://github.com/google/googletest/

5 Release Notes

5.1 Release 1.0.0

This is the first official release of the user manual. It corresponds to version 1.0.0 of the software stack.

Chapter 2

Introduction

1 Overview of FSoE

Safety over EtherCAT/FailSafe over EtherCAT (FSoE) describes a protocol for transferring safety data up to SIL3 between FSoE devices. Safety PDUs are transferred by a subordinate fieldbus that is not included in the safety considerations, since it can be regarded as a black channel. The Safety PDU exchanged between two communication partners is regarded by the subordinate fieldbus as process data that are exchanged cyclically.

FSoE uses a unique master/slave relationship between the FSoE Master and the FSoE Slave, called an FSoE Connection. In the FSoE Connection, Safety PDUs are exchanged between the FSoE Master and its FSoE Slave in a ping-pong scheme. The FSoE Master can handle more than one FSoE Connection to support several FSoE Slaves.

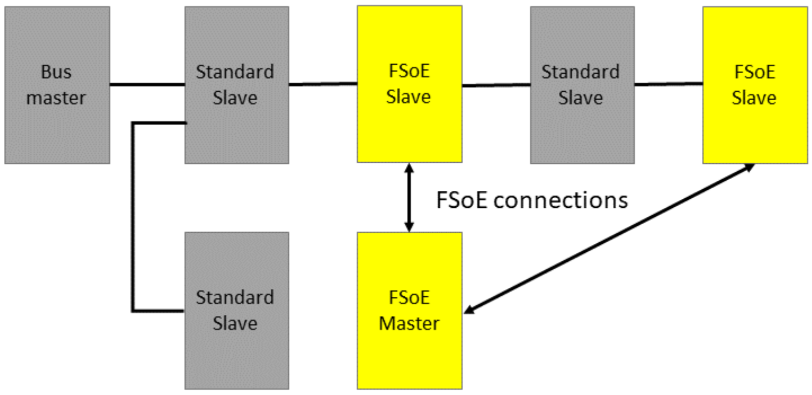


Figure 2.1: FSoE Connection

An FSoE Connection is controlled by two state machines, one on the slave side and one on the master side. Both the FSoE Master and the FSoE Slave starts up in Reset state. Master will then configure the slave through three intermediate states. Slave will verify that received configuration is valid. If configuration succeeds, Master will order Slave to enter Data state, which is the only state where interchange of process data takes place.

State transitions are initiated by the FSoE Master and acknowledged by the FSoE Slave, except if the FSoE Slave detects an error, in which case it will transition to the Reset state on its own.

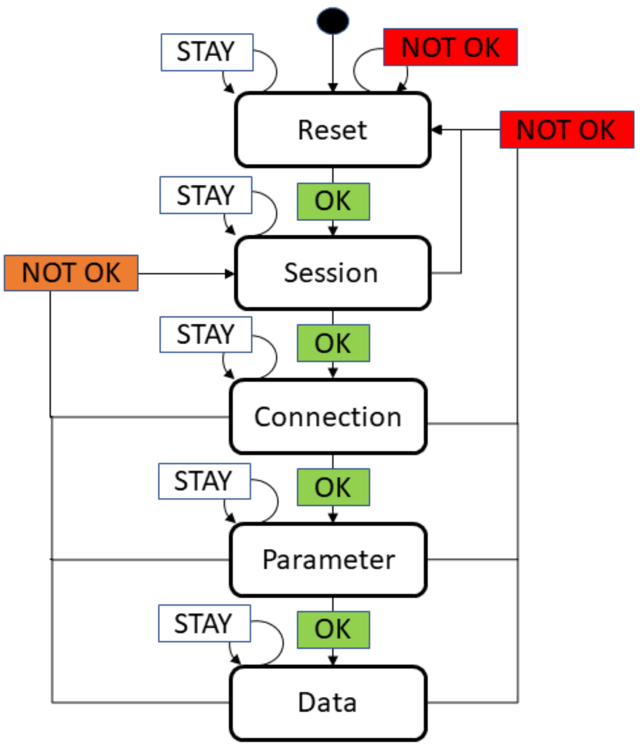


Figure 2.2: FSoE State Machine

The integrity of the safety data transfer is ensured as follows:

- Session-number for detecting buffering of a complete startup sequence.
- Sequence number for detecting interchange, repetition, insertion or loss of whole messages.
- Unique connection identification for detecting misrouted messages via a unique address relationship.
- Watchdog monitoring for detecting delays not allowed on the communication path.
- Cyclic redundancy checking for data integrity for detecting message corruption from source to sink.

As a fail-safe mechanism, the FSoE protocol ensures that data exchanged between the FSoE Master and Slave reverts to fail-safe data (defined as all zeros) if either end detects any communication error.

The FSoE Master and Slave both have a special flag variable specifying if sending valid process data is enabled, or if only fail-safe data may be sent. The flag variable on Slave is independent of the flag variable on the Master, with the exception that both will revert to false if either side detects an error. Thus, Slave may send valid process data to Master while Master only sends fail-safe data to Slave, or vice versa. At start-up, the flag variable is false on both Master and Slave. Neither the FSoE Slave nor the Master state machine will ever set the variable to true on their own. The FSoE protocol offers a way for the application on either side to set the variable to true.

2 Overview of the stack

The FSoE stack provides the safety user application with means to establish an FSoE Connection to a remote FSoE device, enabling cyclical exchange of safety process data.

The stack contains two major components: An implementation of the FSoE master state machine as specified in ETG.5100 and an implementation of the FSoE slave state machine also specified in ETG.↩ 5100. The stack is accessed from application through two main APIs; the header file fsoemaster.h for the master state machine and fsoeslave.h for the slave state machine. These APIs are described in chapters Master State Machine API and Slave State Machine API.

Some behaviour of the stack is application-specific. The application customises such behaviour to fit the application by implementing a set of callback functions. The callback functions are declared in the header file fsoeapp.h. The following callback functions are available:

- fsoeapp_send(): Send a complete FSoE PDU frame.
- fsoeapp_recv(): Try to receive a complete FSoE PDU frame.
- fsoeapp_generate_session_id(): Generate a random 16-bit number.
- fsoeapp_handle_user_error(): Handle user error.
- fsoeapp_verify_parameters(): Verify received parameters. For slave only.

An example of application-specific behaviour is the implementation of cross-checking, where the output of the stack is compared to the output from another FSoE stack acting on the same inputs. For this to work, the stacks need to generate the same Session IDs and receive the same FSoE PDU frames.

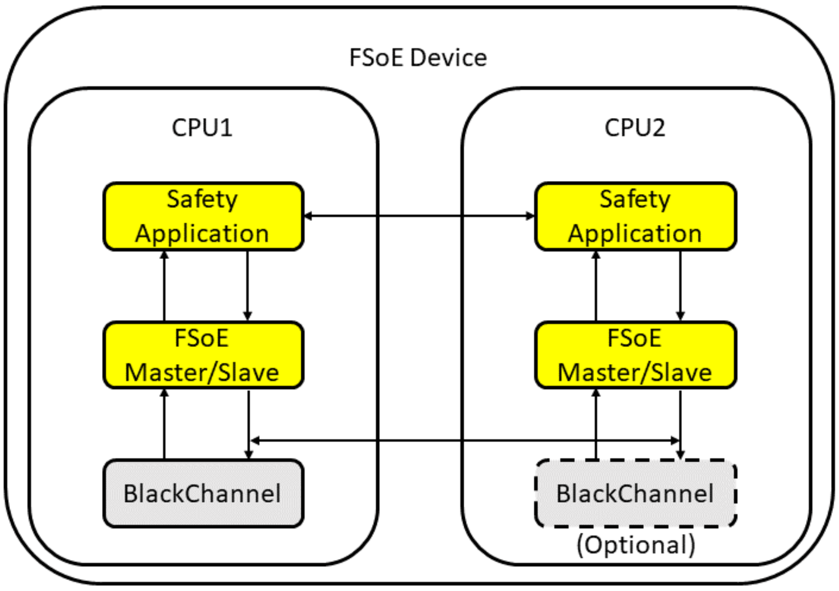


Figure 2.3: Example of cross-checking using two CPUs

The stack is tested for validity using a set of unittests with full code coverage. Both the stack and the unittests are delivered as source code for integration into target safety platform. The stack is written in C while the unittests are written in C++ using the Google Test framework. The source and header files are laid out in the following directory structure:

```
<fsoe stack source>
 |
 +-- cmake    - CMake project files for various targets.
 +-- include  - Header files for public stack API.
 +-- lnt      - PC-Lint configuration files.
 +-- src      - The stack source files.
 |   +-- port - Source files for existing ports.
 +-- test     - Unit tests and integration tests.
```

3 Porting the stack

The FSoE stack provides a platform abstraction layer. The abstraction layer supports both OS-hosted or bare-metal systems. It shall be implemented in a separate directory src/port/PLATFORM_NAME. Both the stack and the unittests should be ported to the target platform.

For more information, see Platform Abstraction Layer.

4 Building the stack

The FSoE stack are delivered with build instructions using CMake for various platforms. If your platform is not present it can be added. For further instructions on how to generate the CMake projects, read the file README.md.

It is also possible to build the source by just adding the source to an existing platform project. If CMake is not used, the header files options.h, fsoeoptions.h and fsoeexport.h must be created manually.

5 Configuring the stack

After the stack is built, its configuration may be changed using ccmake or cmake-gui. The following options may be changed:

- FSOE_PROCESS_DATA_MAX_SIZE: Max size of process data in bytes.
- FSOE_APPLICATION_PARAMETERS_MAX_SIZE: Max size of application parameters in bytes.
- FSOE_LOG_ENABLE: Enable logging.
- FSOE_LOG_LEVEL: Logging level.
- FSOE_LOG_SLAVE: Enable logging of slave state machine.
- FSOE_LOG_MASTER: Enable logging of master state machine.
- FSOE_LOG_CHANNEL: Enable logging of black channel.
- FSOE_LOG_FRAME: Enable logging of frame management.

Chapter 3

Master State Machine API

1 Overview

The FSoE Master state machine API provides the user with the following functionality:

- Support for sending process data outputs (called SafeOutputs in ETG.5100) to Slave and for receiving process data inputs (called SafeInputs in ETG.5100) from Slave.
- Support for user application to control if sending valid process data should be allowed, or if only fail-safe data should be sent (the default).
- Support for user application to reset the FSoE Connection whenever it wishes.

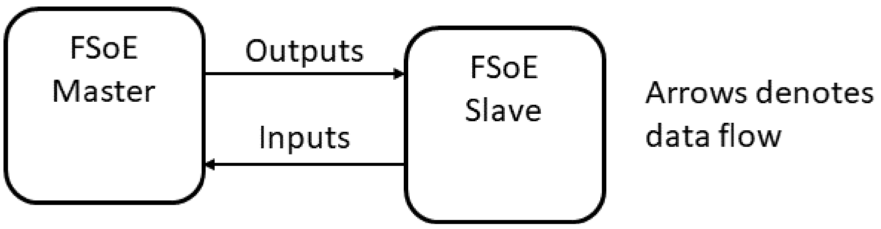


Figure 3.1: FSoE data flow

The user API in header file fsoemaster.h allows any number of master state machines to be configured, instantiated and used. Each state machine instance is independent of all other instances. Global variables are not used. Because of this, the stack has no way to verify that instances are not configured in a conflicting way, e.g. by using the same Connection IDs or Slave Addresses. The application shall ensure that instances are not configured in a way that causes conflicts. See Functions.

The callback API in header file fsoeapp.h declares four callback functions that shall be implemented by application. See Callback functions and types.

2 Macros

Public macro constants and functions ("defines") used by the master state machine are listed here. The following are available:

- FSOEMASTER_RESETREASON_xxx: Reason connection was reset.
- FSOEMASTER_FRAME_SIZE(): Calculate the size of an FSoE frame.
- FSOEMASTER_STATUS_xxx: API function status return codes

2.1 Reasons for reset of connection

These codes are sent between master and slave when either side requests connection to be reset. They are sent in Reset frames. Local reset (FSOEMASTER_RESETREASON_LOCAL_RESET) may be requested by any master or slave application. Local reset is also the reset reason sent by master to slave at startup. All other reset reasons are error conditions detected by an FSoE state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

2.1.1 Local reset

Master or slave application requested connection to be reset. Also sent by master state machine at startup.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_LOCAL_RESET          (0)
```

2.1.2 Invalid command

Master or slave state machine requested connection to be reset after receiving a frame whose type was not valid for current state.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_CMD          (1)
```

2.1.3 Unknown command

Master or slave state machine requested connection to be reset after receiving a frame of unknown type.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_UNKNOWN_CMD          (2)
```

2.1.4 Invalid Connection ID

Master or slave state machine requested connection to be reset after receiving a frame with invalid Connection ID.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_CONNID       (3)
```

2.1.5 Invalid CRC

Master or slave state machine requested connection to be reset after receiving a frame with invalid CRCs.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_CRC         (4)
```

2.1.6 Watchdog expired

Master or slave state machine requested connection to be reset after watchdog timer expired while waiting for a frame to be received.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_WD_EXPIRED          (5)
```

2.1.7 Invalid slave address

Slave state machine requested connection to be reset after receiving Connection frame with incorrect slave address from master. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_ADDRESS     (6)
```

2.1.8 Invalid configuration data

Master state machine requested connection to be reset after receiving Connection or Parameter frame from slave containing different data than what was sent to it. Never requested by slave state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_DATA        (7)
```

2.1.9 Invalid size of Communication parameters

Slave state machine requested connection to be reset after receiving Parameter frame with incorrect size of Communication Parameters from master. Never requested by master state machine. The only communication parameter is the watchdog timeout, whose size is always two bytes.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_COMPARALEN  (8)
```

2.1.10 Invalid Communication parameter data

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible watchdog timeout from master. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_COMPARA     (9)
```

2.1.11 Invalid size of Application parameters

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible size for Application Parameters. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_USERPARALEN (10)
```

2.1.12 Invalid Application parameter data (generic error code)

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible Application Parameters. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_USERPARA    (11)
```

2.1.13 Invalid Application parameter data (first device-specific error code)

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible Application Parameters. Never requested by master state machine. The device-specific error codes are in the range 0x80 ... 0xFF.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_USERPARA_MIN (0x80)
```

2.1.14 Invalid Application parameter data (last device-specific error code)

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible Application Parameters. Never requested by master state machine. The device-specific error codes are in the range 0x80 ... 0xFF.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOEMASTER_RESETREASON_INVALID_USERPARA_MAX (0xFF)
```

2.2 FSOEMASTER_FRAME_SIZE - Get frame size

Number of bytes in FSoE Safety PDU frame containing *data_size* data bytes

Parameters

in	*data_size*	Number of data bytes in frame Needs to be even or 1.

Returns

Size of frame in bytes.

```
#define FSOEMASTER_FRAME_SIZE(data_size) (\
   ((data_size) == 1) ? 6 : (2 * (data_size) + 3) )
```

2.3 API functions status return codes

User API functions return codes. Returned from each API function to indicate if user called the function correctly as described in the function's documentation.

2.3.1 Status OK

User called the API correctly

```
#define FSOEMASTER_STATUS_OK          (0)
```

2.3.2 Status ERROR

User violated the function's preconditions. fsoeapp_handle_user_error() callback will give detailed information about what caused the function to return ERROR.

```
#define FSOEMASTER_STATUS_ERROR       (-1)
```

3 Types

Public types for the FSoE master state machine are listed here. The following are available:

- fsoemaster_status_t: Status returned from API function.
- fsoemaster_state_t: Connection state.
- fsoemaster_resetevent_t: Reset event.
- fsoemaster_syncstatus_t: Status after call to fsoemaster_sync_with_slave().
- fsoemaster_cfg_t: Configuration of state machine.
- fsoemaster_t: A state machine instance.

3.1 API function status

Status returned from API function Returned from each API function to indicate if user called the function correctly as described in the function's documentation. See FSOEMASTER_STATUS_OK See FSOE↩ MASTER_STATUS_ERROR

```
typedef int32_t fsoemaster_status_t;
```

3.2 Connection state

```
typedef enum fsoemaster_state
{
   FSOEMASTER_STATE_RESET,        /*< Connection is reset */
   FSOEMASTER_STATE_SESSION,      /*< The session IDs are being transferred */
   FSOEMASTER_STATE_CONNECTION,   /*< The connection ID is being transferred */
   FSOEMASTER_STATE_PARAMETER,    /*< The parameters are being transferred */
   FSOEMASTER_STATE_DATA,         /*< Process or fail-safe data is being transferred */
} fsoemaster_state_t;
```

3.3 Connection reset event

A reset of connection between master and slave may be initiated by either side sending a Reset frame containing a code describing why the reset was initiated, such as an error detected by FSoE stack, system startup (only master to slave) or application request.

```
typedef enum fsoemaster_resetevent
{
   FSOEMASTER_RESETEVENT_NONE,       /*< No reset initiated. */
   FSOEMASTER_RESETEVENT_BY_MASTER, /*< Reset was initiated by master
                                     * application or state machine.
                                     * A Reset frame was sent to slave
                                     * containing the reset code.
                                     */
   FSOEMASTER_RESETEVENT_BY_SLAVE,  /*< Reset was initiated by slave
                                     * application or state machine.
                                     * A Reset frame was received from slave
                                     * containing the reset code.
                                     */
} fsoemaster_resetevent_t;
```

3.4 Communication status

Status after synchronisation with slave See fsoemaster_sync_with_slave()

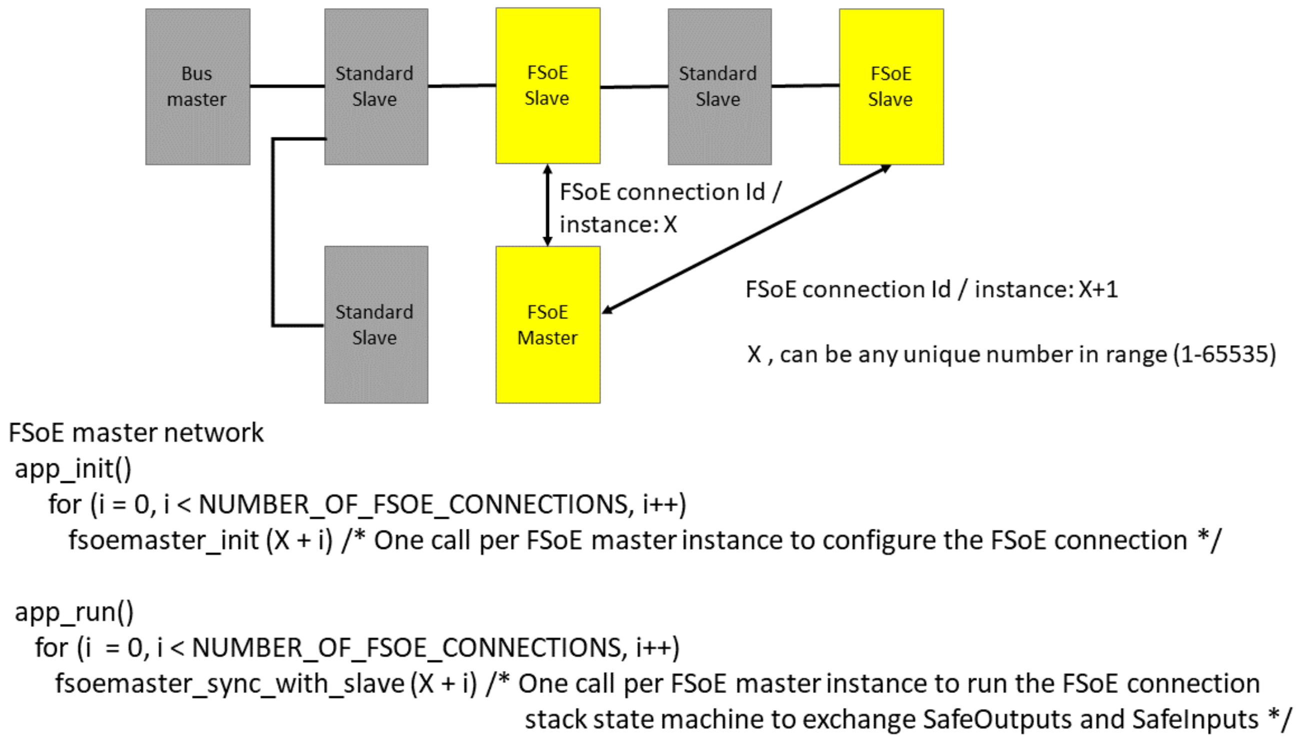
```
typedef struct fsoemaster_syncstatus
{
   bool is_process_data_received;   /*< Is process data received?
                                     * true:
                                     *   Valid process data was received in
                                     *   last FSoE cycle. The process data is
                                     *   stored in \a inputs buffer.
                                     *   Note that the process data could have
                                     *   been received in a previous call
                                     *   to fsoemaster_sync_with_slave(). It is
                                     *   still considered valid though as no
                                     *   communication error has occured, such
                                     *   as timeouts or CRC errors.
                                     * false:
                                     *   No valid process data was received in
                                     *   last FSoE cycle. The \a inputs buffer
                                     *   contains only zeroes.
                                     *   This will be returned if an error has
                                     *   been detected, if connection with
                                     *   slave is not established or if
                                     *   fail-safe data was received.
                                     */
   fsoemaster_resetevent_t reset_event; /*< Connection reset event.
                                     * If a reset event occured during this call
                                     * to fsoemaster_sync_with_slave(), this will
                                     * indicate if it was initiated by master or
                                     * slave. Otherwise it is set to
                                     * FSOEMASTER_RESETEVENT_NONE.
                                     * Note that the master state machine
                                     * will reset the connection at startup.
                                     */
   uint8_t reset_reason;            /*< Reason for connection reset.
                                     * In case a reset event occured, this
                                     * is the code sent/received in the
                                     * Reset frame. All codes except for
                                     * FSOEMASTER_RESETREASON_LOCAL_RESET
                                     * indicates that an error was detected.
                                     * See codes defined further up. Also see
                                     * fsoemaster_reset_reason_description().
                                     */
   fsoemaster_state_t current_state; /*< Current state of the state machine */
} fsoemaster_syncstatus_t;
```

3.5 Configuration

```
typedef struct fsoemaster_cfg
{
   /*
    * Slave Address

    * An address uniquely identifying the slave;
    * No other slave within the communication system may have the same
    * Slave Address. Valid values are 0 - 65535.

    * This value will be sent to slave when connection is established, which
    * will verify that the value matches its own Slave Address.
    * Slave will refuse the connection if wrong Slave Address is sent to it.

    * See ETG.5100 ch. 8.2.2.4 "Connection state".
    */
   uint16_t slave_address;

   /*
    * Connection ID

    * A non-zero address uniquely identifying the master;
    * No other master within the communication system may have the same
    * Connection ID.

    * This value will be sent to slave when connection is established.

    * See ETG.5100 ch. 8.2.2.4 "Connection state".
    */
   uint16_t connection_id;

   /*
    * Timeout value in milliseconds for the watchdog timer

    * This value will be sent to slave when connection is established.
    * Valid values are 1 - 65535.
    * Slave will refuse the connection if value is outside the slave's
    * supported range.

    * See ETG.5100 ch.8.2.2.5 "Parameter state".
    */
   uint16_t watchdog_timeout_ms;

   /*
    * Application parameters (optional)

    * The application parameters are device-specific and will be sent to
    * slave when connection is established.
    * May be set to NULL if no application parameters are needed.
    * Slave will refuse the connection if it determines that a parameter
    * has the wrong value.

    * See ETG.5100 ch. 8.2.2.5 "Parameter state".
    */
   const void * application_parameters;

   /*
    * Size in bytes of the application parameters
    * Valid values are 0 - FSOE_APPLICATION_PARAMETERS_MAX_SIZE.
    * This value will be sent to slave when connection is established.
    * May be zero if no application parameters are needed.
    * Slave will refuse the connection if it expected a different size.

    * See ETG.5100 ch. 8.2.2.5 "Parameter state".
    */
   size_t application_parameters_size;

   /*
    * Size in bytes of the outputs to be sent to slave

    * Only even values are allowed, except for 1, which is also allowed.
    * Maximum value is FSOE_PROCESS_DATA_MAX_SIZE.
    * Master and slave need to agree on the size of the outputs.
    * Communication between master and slave will otherwise not be possible.
    * The size of PDU frames received from slave will be
    * MAX (3 + 2 * outputs_size, 6).

    * See ETG.5100 ch. 4.1.2 (called "SafeOutputs").
    */
   size_t outputs_size;

   /*
    * Size in bytes of the inputs to be received from slave

    * Only even values are allowed, except for 1, which is also allowed.
    * Maximum value is FSOE_PROCESS_DATA_MAX_SIZE.

    * Master and slave need to agree on the size of the inputs.
    * Communication between master and slave will otherwise not be possible.
    * The size of PDU frames received from slave will be
    * MAX (3 + 2 * inputs_size, 6).

    * See ETG.5100 ch. 4.1.2 (called "SafeInputs").
    */
   size_t inputs_size;
} fsoemaster_cfg_t;
```

3.6 State machine instance

An FSoE master state machine instance handles the connection with a single slave. Multiple state machines instances are supported, where each one has their own Connection ID and associated slave.


Bus master
Standard Slave
FSoE Slave
Standard Slave
FSoE Slave
FSoE connection Id / instance: X
Standard Slave
FSoE Master
FSoE connection Id / instance: X+1
X , can be any unique number in range (1-65535)
FSoE master network

```
 app_init()
     for (i = 0, i < NUMBER_OF_FSOE_CONNECTIONS, i++)
       fsoemaster_init (X + i) /* One call per FSoE master instance to configure the FSoE connection */

 app_run()
    for (i  = 0, i < NUMBER_OF_FSOE_CONNECTIONS, i++)
      fsoemaster_sync_with_slave (X + i) /* One call per FSoE master instance to run the FSoE connection
                                            stack state machine to exchange SafeOutputs and SafeInputs */
```

Figure 3.2: FSoE instance and connection

User may allocate the instance statically or dynamically using malloc() or on the stack. To use an allocated instance, pass a pointer to it as the first argument to any API function.

This struct is made public as to allow for static allocation.

NOTE: User of the API is prohibited from accessing any of the fields as the layout of the structure is to be considered an implementation detail. Only the stack-internal file "fsoemaster.c" may access any field directly.

```
typedef struct fsoemaster
{
   ...
} fsoemaster_t;
```

4 Functions

These functions acts upon an instance:

- fsoemaster_get_state(): Get current state.
- fsoemaster_get_master_session_id(): Get generated Master Session ID.
- fsoemaster_get_slave_session_id(): Get received Slave Session ID.
- fsoemaster_get_time_until_timeout_ms(): Get time until timeout.
- fsoemaster_get_process_data_sending_enable_flag(): Get flag indicating if sending process data is enabled.
- fsoemaster_clear_process_data_sending_enable_flag(): Clear flag indicating if sending process data is enabled.
- fsoemaster_set_process_data_sending_enable_flag(): Set flag indicating if sending process data is enabled.
- fsoemaster_set_reset_request_flag(): Set reset request flag.
- fsoemaster_sync_with_slave(): Synchronise with slave.
- fsoemaster_init(): Initialise master state machine.

This function may be used if the SRA CRC feature is used:

- fsoemaster_update_sra_crc(): Update SRA CRC value.

These functions are mainly exposed for convenience when logging:

- fsoemaster_reset_reason_description(): Return description of reset reason.
- fsoemaster_state_description(): Return description of state.

For macros used by the functions, see Macros. For types used by the functions, see Types.

4.1 fsoemaster_reset_reason_description - Return description of reset reason

```
const char * fsoemaster_reset_reason_description (
   uint8_t reset_reason);
```

Return description of reset reason as a string literal

Parameters

in	*reset_reason*	Reset reason

Returns

String describing the reset reason, e.g. "local reset" or "INVALID_CRC", unless *reset_reason* is not a valid reset reason, in which case "invalid error code" is returned. In either case, the returned string is null- terminated and statically allocated as a string literal.

Example:

```
#include <fsoemaster.h>
void handle_connection_reset_by_slave (uint8_t reset_reason)
{
   printf ("Slave initiated connection reset due to %s (%u)\n",
         fsoemaster_reset_reason_description (reset_reason),
         reset_reason);
}
```

4.2 fsoemaster_state_description - Return description of instance state

```
const char * fsoemaster_state_description (
   fsoemaster_state_t state);
```

Return description of state machine state as a string literal

Parameters

in	*state*	State

Returns

String describing the state, unless *state* is not a valid state, in which case "invalid" is returned. In either case, the returned string is null- terminated and statically allocated as a string literal.

Example:

```
#include <fsoemaster.h>
fsoemaster_state_t state;

status = fsoemaster_get_state (master, &state);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Current state is %s\n",
      fsoemaster_state_description (state));
}
```

4.3 fsoemaster_update_sra_crc - Update SRA CRC value

```
fsoemaster_status_t fsoemaster_update_sra_crc (
   uint32_t * crc,
   const void * data,
   size_t size);
```

This function will calculate the SRA CRC for data in *data*. If this is the first time the function is called, then user should first set *crc* to zero before calling the function. If this is a subsequent call then the previously calculated CRC value will be is used as input to the CRC calculation. The CRC *crc* will be updated in-place.

SRA CRC is an optional feature whose use is not mandated nor specified by the FSoE ETG.5100 specification. If used, the SRA CRC should be sent to slave as Application parameter, where it should be placed first (encoded in little endian byte order).

See ETG.5120 "Safety over EtherCAT Protocol Enhancements", ch. 6.3 "SRA CRC Calculation".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *crc* and *data* are non-null.

Parameters

`in,out`	*crc*	SRA CRC value to update in-place. If this is the first call to fsoemaster_update_sra_crc() then its value should first be set to zero.
`in`	*data*	Buffer with data.
`in`	*size*	Size of buffer in bytes. If zero, *crc* will be left unmodified.

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status1;
fsoemaster_status_t status2;
uint32_t crc;

crc = 0;
status1 = fsoemaster_update_sra_crc (&crc, data1, sizeof (data1));
status2 = fsoemaster_update_sra_crc (&crc, data2, sizeof (data2));
if (status1 == FSOEMASTER_STATUS_OK &&
    status2 == FSOEMASTER_STATUS_OK)
{
   printf ("Calculated SRA CRC: 0x%x\n", crc);
}
else
{
   printf ("We called function incorrectly (with null-pointers)\n");
}
```

4.4 fsoemaster_get_state - Get current state

```
fsoemaster_status_t fsoemaster_get_state (
   const fsoemaster_t * master,
   fsoemaster_state_t * state);
```

Get current state of the FSoE master state machine.

See ETG.5100 ch. 8.4.1.1. table 29: "States of the FSoE master".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *master* and *state* are non-null.
fsoemaster_init() has been called for instance *master*.

Parameters

in	*master*	FSoE master state machine
out	*state*	Current state

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;
fsoemaster_state_t state;

status = fsoemaster_get_state (master, &state);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Current state is %s\n",
      fsoemaster_state_description (state));
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.5 fsoemaster_get_master_session_id - Get generated Master Session ID

```
fsoemaster_status_t fsoemaster_get_master_session_id (
   const fsoemaster_t * master,
   uint16_t * session_id);
```

The Master Session ID was generated by the master state machine when entering Session state.

Calling this function while master state machine is in Reset state is not allowed as no Master Session ID has yet been generated.

See ETG.5100 ch. 8.2.2.3: "Session state".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *master* and *session_id* are non-null.
fsoemaster_init() has been called for instance *master*.
Master state machine is at least in Session state.

Parameters

in	*master*	FSoE master state machine
out	*session↩ _id*	Current Master Session ID

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;
uint16_t session_id;

status = fsoemaster_get_master_session_id (master, &session_id);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Generated Master Session ID: %u\n", session_id);
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.6 fsoemaster_get_slave_session_id - Get received Slave Session ID

```
fsoemaster_status_t fsoemaster_get_slave_session_id (
   const fsoemaster_t * master,
   uint16_t * session_id);
```

The Slave Session ID was generated by slave and then received by the master state machine when entering Connection state.

Calling this function while master state machine is in Reset or Session state is not allowed as no Slave Session ID has yet been received.

See ETG.5100 ch. 8.2.2.3: "Session state".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *master* and *session_id* are non-null.
fsoemaster_init() has been called for instance *master*.
Master state machine is at least in Connection state.

Parameters

in	*master*	FSoE master state machine
out	*session_id*	Current Slave Session ID

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;
uint16_t session_id;

status = fsoemaster_get_slave_session_id (master, &session_id);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Received Slave Session ID: %u\n", session_id);
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.7 fsoemaster_get_time_until_timeout_ms - Get time until timeout

```
fsoemaster_status_t fsoemaster_get_time_until_timeout_ms (
   const fsoemaster_t * master,
   uint32_t * time_ms);
```

Get time remaining until watchdog timer timeouts, in milliseconds. This function is mainly used for unit-testing purposes.

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *master* and *time_ms* are non-null.
fsoemaster_init() has been called for instance *master*.

Parameters

in	*master*	FSoE master state machine
out	*time_ms*	Time remaining in milliseconds. If watchdog timer is not started, UINT32_MAX is returned.

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition..

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;
uint32_t time_ms;

status = fsoemaster_get_time_until_timeout_ms (master, &time_ms);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Time remaining until watchdog timeout: %u ms\n", time_ms);
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.8 fsoemaster_get_process_data_sending_enable_flag - Get flag indicating if sending process data is enabled

```
fsoemaster_status_t fsoemaster_get_process_data_sending_enable_flag (
   const fsoemaster_t * master,
   bool * is_enabled);
```

Get flag indicating if sending process data to slave is enabled. This will only check a flag indicating that everything is OK from the perspective of the application. Master state machine will not send normal process data if connection with slave is not fully established (Data state), even if application allows it.

See ETG.5100 ch. 8.4.1.2 "Set Data Command event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *master* and *is_enabled* are non-null.
fsoemaster_init() has been called for instance *master*.

Parameters

in	*master*	FSoE master state machine
out	*is_enabled*	Current process data send status: • true if master is allowed to send process data, • false if not.

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;
bool is_enabled;

status = fsoemaster_get_process_data_sending_enable_flag (master, &is_enabled);
if (status == FSOEMASTER_STATUS_OK)
{
   if (is_enabled)
   {
      printf ("Master is allowed to send process data to slave\n");
   }
   else
   {
      printf ("Master is not allowed to send process data to slave\n");
   }
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.9 fsoemaster_clear_process_data_sending_enable_flag - Clear flag indicating if sending process data is enabled

```
fsoemaster_status_t fsoemaster_clear_process_data_sending_enable_flag (
   fsoemaster_t * master);
```

This will clear a flag indicating that everything is OK from the perspective of the application. Master will only send fail-safe data (zeros) to slave. This is the default setting after power-on and after detection of any errors.

See ETG.5100 ch. 8.4.1.2 "Set Data Command event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointer *master* is non-null.
fsoemaster_init() has been called for instance *master*.

Parameters

`in,out`	*master*	FSoE master state machine

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;

status = fsoemaster_clear_process_data_sending_enable_flag (master);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Sending process data to slave is no longer allowed\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.10 fsoemaster_set_process_data_sending_enable_flag - Set flag indicating if sending process data is enabled

```
fsoemaster_status_t fsoemaster_set_process_data_sending_enable_flag (
   fsoemaster_t * master);
```

This will set a flag indicating that everything is OK from the perspective of the application. Setting the flag will cause master to send outputs containing valid process data once connection is established, assuming no errors are detected. If any errors are detected, this flag will revert to its disabled state and only fail-safe outputs will be sent.

See ETG.5100 ch. 8.4.1.2 "Set Data Command event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointer *master* is non-null.
fsoemaster_init() has been called for instance *master*.

Parameters

`in,out`	*master*	FSoE master state machine

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;

status = fsoemaster_set_process_data_sending_enable_flag (master);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Sending process data to slave is now allowed\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.11 fsoemaster_set_reset_request_flag - Set reset request flag

```
fsoemaster_status_t fsoemaster_set_reset_request_flag (
   fsoemaster_t * master);
```

This will set a flag, which in next call to fsoemaster_sync_with_slave() will cause the master state machine to send the Reset frame to slave and then enter the Reset state. Fail-safe mode will then be entered, where normal process data outputs will not be sent even after connection has been re-established. Application needs to explicitly re-enable process data outputs in order to leave fail-safe mode. See fsoemaster_set_process_data_sending_enable_flag().

See ETG.5100 ch. 8.4.1.2 "Reset Connection event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointer *master* is non-null.
fsoemaster_init() has been called for instance *master*.

Parameters

in,out	*master*	FSoE master state machine

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_status_t status;

status = fsoemaster_set_reset_request_flag (master);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Master state machine reset was requested\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.12 fsoemaster_sync_with_slave - Synchronise with slave

```
fsoemaster_status_t fsoemaster_sync_with_slave (
      fsoemaster_t * master,
      const void * outputs,
      void * inputs,
      fsoemaster_syncstatus_t * sync_status);
```

Needs to be called periodically in order to avoid watchdog timeout. It is recommended that delay between calls to the function is no more than half the watchdog timeout.

Depending on current state, the master state machine may try to send a single frame or read a single frame by calling fsoeapp_send() and/or fsoeapp_recv(), which are non-blocking functions.

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *master*, *outputs*, *inputs* and *sync_status* are non-null.
fsoemaster_init() has been called for instance *master*.

Parameters

`in,out`	*master*	FSoE master state machine
`in`	*outputs*	Buffer containing outputs to be sent to slave. Its size is given in configuration.
`out`	*inputs*	Buffer to store inputs received from slave. Its size is given in configuration. Whether inputs are valid or not is given by *sync_status*.
`out`	*sync_status*	Status of FSoE connection.

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
fsoemaster_syncstatus_t sync_status;
fsoemaster_status_t status;
uint8_t inputs[2];

outputs[0] = 0x12;
outputs[1] = 0x34;
status = fsoemaster_sync_with_slave (master, outputs, inputs, &sync_status);
if (status == FSOEMASTER_STATUS_OK)
{
   if (sync_status.reset_event != FSOEMASTER_RESETEVENT_NONE)
   {
      printf ("Connection was reset by %s. Cause: %s\n",
         sync_status.reset_event == FSOEMASTER_RESETEVENT_BY_MASTER ?
         "master" : "slave",
         fsoemaster_reset_reason_description (sync_status.reset_reason));
   }
   if (sync_status.is_process_data_received)
   {
      handle_received_data (inputs);
   }
   else
   {
      printf ("No valid process data was received\n");
   }
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.13 fsoemaster_init - Initialise master state machine

```
fsoemaster_status_t fsoemaster_init (
   fsoemaster_t * master,
   const fsoemaster_cfg_t * cfg,
   void * app_ref);
```

This will configure the instance according to supplied configuration.

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOEMASTER_STATUS_ERROR.

Precondition

The pointers *master* and *cfg* are non-null.
The fields in *cfg* are valid.

Parameters

out	*master*	FSoE master state machine
in	*cfg*	Configuration
in	*app_ref*	Application reference. This will be passed as first argument to callback functions implemented by application. The stack does not interpret this value in any way.

Returns

Status:

- FSOEMASTER_STATUS_OK if function was called correctly.
- FSOEMASTER_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoemaster.h>
const fsoemaster_cfg_t cfg =
{
   .slave_address                = 0x0304,
   .connection_id                = 8,
   .watchdog_timeout_ms          = 100,
   .application_parameters       = NULL,
   .application_parameters_size = 0,
   .outputs_size                 = 2,
   .inputs_size                  = 2,
};
fsoemaster_t master;
fsoemaster_status_t status;

status = fsoemaster_init (&master, &cfg, NULL);
if (status == FSOEMASTER_STATUS_OK)
{
   printf ("Master state machine was initialised\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

5 Callback functions and types

The following callback functions are declared in fsoeapp.h and shall be implemented in application:

- fsoeapp_send(): Send Safety PDU frame.
- fsoeapp_recv(): Receive Safety PDU frame.
- fsoeapp_generate_session_id(): Generate a Session ID.
- fsoeapp_handle_user_error(): Handle user error.

The functions fsoeapp_send() and fsoeapp_recv() are the means for the state machine to access the black channel. Application may implement these with support for cross-checking. The arrows in picture below denote direct function calls:

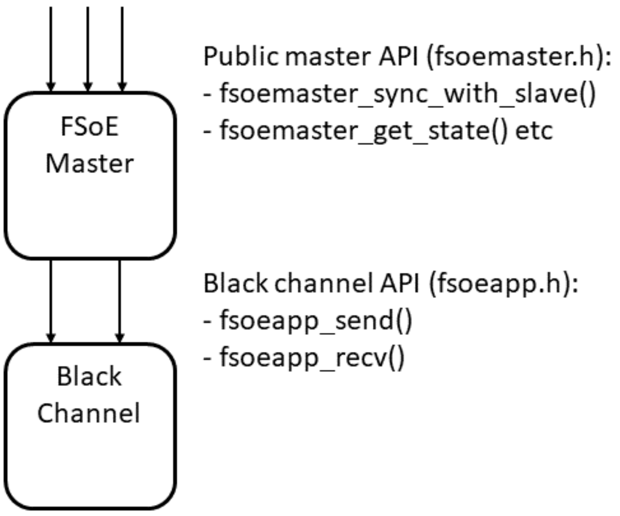

FSoE Master
Public master API (fsoemaster.h):
- fsoemaster_sync_with_slave()
- fsoemaster_get_state() etc
Black channel API (fsoeapp.h):
- fsoeapp_send()
- fsoeapp_recv()
Black Channel

Figure 3.3: FSoE stack layers

The master state machine does not verify that the Session ID returned from fsoeapp_generate_session↵ _id() is generated correctly. To pass the conformance test, the application should ensure that the Session ID is re-generated as a random number in accordance with the specification.

The function fsoeapp_handle_user_error() can be implemented as a system restart, or it can be implemented as an empty function which does nothing.

The header file also makes the following type available:

- fsoeapp_usererror_t: User error.

The header file also makes the following function available for convenience when logging:

- fsoeapp_user_error_description(): Return description of user error.

5.1 User error

Passed to fsoeapp_handle_user_error() when an API function detects that user violated a precondition.

```
typedef enum fsoeapp_usererror
{
   FSOEAPP_USERERROR_NULL_INSTANCE = 1,/*< User violated the API by passing
                                     * a null-pointer instance.
                                     */
   FSOEAPP_USERERROR_UNINITIALISED_INSTANCE, /*< User violated the API by
                                     * calling API function before calling
                                     * fsoemaster_init() or fsoeslave_init().
                                     */
   FSOEAPP_USERERROR_WRONG_INSTANCE_STATE, /*< User violated the API by
                                     * calling function when instance was in a
                                     * state prohibited by the function's
                                     * documentation.
                                     */
   FSOEAPP_USERERROR_NULL_ARGUMENT, /*< User violated the API by passing
                                     * a null-pointer argument (other than
                                     * the instance itself).
                                     */
   FSOEAPP_USERERROR_BAD_CONFIGURATION, /*< User violated the API by calling
                                     * fsoemaster_init() or fsoeslave_init()
                                     * with a configuration with bad field.
                                     */
} fsoeapp_usererror_t;
```

See also

fsoeapp_handle_user_error().
fsoeapp_user_error_description().

5.2 fsoeapp_user_error_description - Return description of user error

```
const char * fsoeapp_user_error_description (
   fsoeapp_usererror_t user_error);
```

Return description of user error as a string literal. This is just a helper function which may be used for logging the error code passed to fsoeapp_handle_user_error(). It is implemented by the FSoE stack itself.

Parameters

in	*user_error*	User error

Returns

String describing the user error, unless *user_error* is not a valid error, in which case "invalid error code" is returned. In either case, the returned string is null- terminated and statically allocated as a string literal

Example:

```
#include <fsoeapp.h>
void fsoeapp_handle_user_error (
   void * app_ref, fsoeapp_usererror_t user_error)
{
   printf ("We called an API function incorrectly: %s\n",
      fsoeapp_user_error_description (user_error));
}
```

See also

fsoeapp_handle_user_error().

5.3 fsoeapp_send - Send a Safety PDU frame

```
void fsoeapp_send (void * app_ref, const void * buffer, size_t size);
```

Send a complete FSoE PDU frame. An FSoE PDU frame starts with the Command byte and ends with the Connection ID. Its size is given by the formula MAX (3 + 2 $*$ data_size, 6), where data_size is the number of data bytes to send and is given by a field in fsoemaster_cfg_t or fsoeslave_cfg_t.

See ETG.5100 ch. 8.1.1 "Safety PDU structure".

This callback function is called by the FSoE stack when it wishes to send a frame. Application is required to be implement this by making an attempt to send the frame in supplied buffer. If application wishes to communicate an error condition to the FSoE stack then it may do so by calling fsoemaster_set_reset_↩ request_flag() or fsoeslave_set_reset_request_flag().

NOTE: If cross-checking is required it depends on implemented model if the data is cross-checked before sending or not, the example models from iec61784-3 suggest cross-checking to be performed before sending if only a single frame is sent, this is typically true for a FSoE device running on EtherCAT Slave Controller HW.

NOTE: If the FSoE stack runs on a different thread or execution context than the thread or execution context where the frame is actually sent, then the application has to ensure that no race condition can occur. Otherwise, a partial frame may be sent, causing CRC error and a reset of connection. Race conditions may be avoided by means of a mutex, a semaphore, by locking interrupts or some other mechanism.

Parameters

in,out	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_init() or fsoeslave_init(). Application is free to use this as it sees fit.
in	*buffer*	Buffer containing a PDU frame to be sent.
in	*size*	Size of PDU frame in bytes. Always the same, as given by formula above.

EtherCAT example iec61784-3, Annex A Model A

Example:

```
void fsoeapp_send (void * app_ref, const void * buffer, size_t size)
{
   /* Configure for cross-checking in HW or SW */
#if FSOE_REDUNDANT_SCL_IN_HW
   /* copy the HW checked and encoded SPDU to the black channel */
   memcpy (FSOE_Slave_Frame_Elements, buffer, size);
#else
   /* send encoded data to be cross-checked with encoded data from
    * second CPU. if the result match we copy it to the black channel.
    * if the result doesn't match, we do a reset.
    */
   if (cross_check_encoded_data (buffer, size) == ERROR)
   {
      fsoemaster_set_reset_request_flag (fsoe_master)
   }
   else
   {
      memcpy (FSOE_Slave_Frame_Elements, buffer, size);
   }
#endif
}
```

5.4 fsoeapp_recv - Receive a Safety PDU frame

```
size_t fsoeapp_recv (void * app_ref, void * buffer, size_t size);
```

Try to receive a complete FSoE PDU frame. An FSoE PDU frame starts with the Command byte and ends with the Connection ID. Its size is given by the formula MAX $(3 + 2 * \text{data\_size}, 6)$, where data_size is the number of data bytes to receive and is given by a field in fsoemaster_cfg_t or fsoeslave_cfg_t.

See ETG.5100 ch. 8.1.1 "Safety PDU structure".

This callback function is called by the FSoE stack when it wishes to receive a frame. Application is required to implement this by first checking if a frame was received. If no new frame was received then the function should either

- return without waiting for any incoming frame or
- copy previously received frame to buffer and return.

If a frame was received then its content should be copied to buffer. If application wishes to communicate an error condition to the FSoE stack then it may do so by calling fsoemaster_reset_connection() or fsoeslave_reset_connection().

NOTE: If cross-checking is required, no cross-checking is performed here but on decoded data returned by the stack.

NOTE: If the FSoE stack runs on a different thread or execution context than the thread or execution context where the frame is actually received, then the application has to ensure that no race condition can occur. Otherwise, fsoeapp_recv() may receive a partial frame, causing CRC error and a reset of connection. Race conditions may be avoided by means of a mutex, a semaphore, by locking interrupts or some other mechanism.

Parameters

`in,out`	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_init() or fsoeslave_init(). Application is free to use this as it sees fit.
`out`	*buffer*	Buffer where recieved PDU frame will be stored.
`in`	*size*	Size of PDU frame in bytes. Always the same, as given by formula above.

Returns

Number of bytes received. Should be equal to *size* if a frame was received. If no frame was received, it may be equal 0. Alternatively, the last received frame may be put in the buffer with *size* bytes returned.

Example:

```
/* FSoE stack receive data from black channel */
size_t fsoeapp_recv (void * app_ref, void * buffer, size_t size);
{
   /* Ok to discard data received in states below OP */
   if ((CC_ATOMIC_GET (ESCvar.App.state) & APPSTATE_OUTPUT) <= 1)
   {
      return 0;
   }

   /* copy black channel data received from FSoE Device partner,
    * size is fixed and set at configuration.
    * cross-checking is done on decoded FSoE data, not here */
   memcpy (buffer, FSOE_Master_Frame_Elements, size);
   return size;
}
```

5.5 fsoeapp_generate_session_id - Generate a Session ID

```
uint16_t fsoeapp_generate_session_id (void * app_ref);
```

Generate a Session ID. A Session ID is a random 16 bit number.

See ETG.5100 ch. 8.1.3.7 "Session ID".

This callback function is called by the FSoE stack after power-on and after each connection reset. Application is required to implement this by generating a random number which is sufficiently random that a (with high probability) different random number will be generated after each system restart. A normal pseudo-random algorithm with fixed seed value is not sufficient.

NOTE: If cross-checking is required, the two SCL must synchronise the random number used for session ID. All subsequent frames will "inherit" from this random number due the inclusion of received CRC_0 in sent frames.

Parameters

`in,out`	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_init() or fsoeslave_init(). Application is free to use this as it sees fit.

Returns

A generated Session ID

Example:

```
uint16_t fsoeapp_generate_session_id (void * app_ref)
{
   /* Configure for cross-checking in HW or SW */
#if FSOE_REDUNDANT_SCL_IN_HW
   return (uint16_t)platform_rand();
#else
   uint16_t my_rand = (uint16_t)platform_rand();
   uint16_t partner_rand = cross_check_fetch_partner();
   return my_rand + partner_rand;
#endif
}
```

5.6 fsoeapp_handle_user_error - Handle user error

```
void fsoeapp_handle_user_error (
   void * app_ref,
   fsoeapp_usererror_t user_error);
```

User called an API function in a way that violated a precondition. The API function detected this and before returning it called this function.

Application may implement this by restarting the system if running on an embedded target or quit the process if running on a PC. In these cases, the API function will not return and the user does not need to check the returned error code.

Application may also implement this by returning to the API function. The API function will then return the error to user, which should then handle the error. Note that any elaborate error handling by user is unlikely to succeed as it was the user who committed the error in the first place by violating the API function's preconditions.

Note

If using a debugger, this may be a good place to put a debug breakpoint.
This function will never be called if user calls all API functions correctly.

Parameters

`in,out`	*app_ref*	Application reference: • NULL if error was detected by a state machine function and *user_error* was either FSOEAPP_USERERROR_NULL_INSTANCE or FSOEAPP_USERERROR_UNINITIALISED_INSTANCE. • NULL if error was detected by the functions fsoeslave_update_sra_crc() or fsoemaster_update_sra_crc(). • Otherwise, this is the pointer with the same name passed to fsoeslave_init() or fsoemaster_init().
`in`	*user_error*	Type of error user made when calling the API function.

Example:

```
#include <fsoeapp.h>
void fsoeapp_handle_user_error (
   void * app_ref, fsoeapp_usererror_t user_error)
{
   printf ("User called API function incorrectly: %s, app_ref: %p\n"),
      fsoeapp_user_error_description (user_eror), app_ref);
   assert (0);
}
```

Chapter 4

Slave State Machine API

1 Overview

The FSoE Slave state machine API provides the user with the following functionality:

- Support for sending process data inputs (called SafeInputs in ETG.5100) to Master and for receiving process data outputs (called SafeOutputs in ETG.5100) from Master.
- Support for user application to control if sending valid process data should be allowed, or if only fail-safe data should be sent (the default).
- Support for user application to reset the FSoE Connection whenever it wishes.

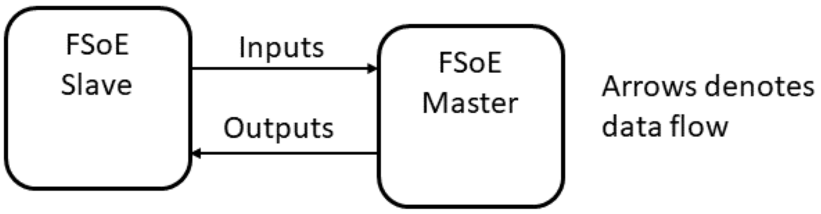


Figure 4.1: FSoE data flow

The user API in header file fsoeslave.h allows the slave state machine to be configured, instantiated and used. See Functions.

The callback API in header file fsoeapp.h declares five callback functions that shall be implemented by application. See Callback functions and types.

2 Macros

Public macro constants and functions ("defines") used by the slave state machine are listed here. The following are available:

- FSOESLAVE_RESETREASON_xxx: Reason connection was reset.

- FSOESLAVE_FRAME_SIZE(): Calculate the size of an FSoE frame.
- FSOESLAVE_STATUS_xxx: API function status return codes

2.1 Reasons for reset of connection

These codes are sent between master and slave when either side requests connection to be reset. They are sent in Reset frames. Local reset (FSOESLAVE_RESETREASON_LOCAL_RESET) may be requested by any master or slave application. Local reset is also the reset reason sent by master to slave at startup. All other reset reasons are error conditions detected by an FSoE state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

2.1.1 Local reset

Master or slave application requested connection to be reset. Also sent by master state machine at startup.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_LOCAL_RESET         (0)
```

2.1.2 Invalid command

Master or slave state machine requested connection to be reset after receiving a frame whose type was not valid for current state.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_CMD         (1)
```

2.1.3 Unknown command

Master or slave state machine requested connection to be reset after receiving a frame of unknown type.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_UNKNOWN_CMD         (2)
```

2.1.4 Invalid Connection ID

Master or slave state machine requested connection to be reset after receiving a frame with invalid Connection ID.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_CONNID      (3)
```

2.1.5 Invalid CRC

Master or slave state machine requested connection to be reset after receiving a frame with invalid CRCs.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_CRC         (4)
```

2.1.6 Watchdog expired

Master or slave state machine requested connection to be reset after watchdog timer expired while waiting for a frame to be received.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_WD_EXPIRED          (5)
```

2.1.7 Invalid slave address

Slave state machine requested connection to be reset after receiving Connection frame with incorrect slave address from master. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_ADDRESS     (6)
```

2.1.8 Invalid configuration data

Master state machine requested connection to be reset after receiving Connection or Parameter frame from slave containing different data than what was sent to it. Never requested by slave state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_DATA        (7)
```

2.1.9 Invalid size of Communication parameters

Slave state machine requested connection to be reset after receiving Parameter frame with incorrect size of Communication Parameters from master. Never requested by master state machine. The only communication parameter is the watchdog timeout, whose size is always two bytes.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_COMPARALEN  (8)
```

2.1.10 Invalid Communication parameter data

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible watchdog timeout from master. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_COMPARA     (9)
```

2.1.11 Invalid size of Application parameters

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible size for Application Parameters. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_USERPARALEN (10)
```

2.1.12 Invalid Application parameter data (generic error code)

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible Application Parameters. Never requested by master state machine.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_USERPARA    (11)
```

2.1.13 Invalid Application parameter data (first device-specific error code)

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible Application Parameters. Never requested by master state machine. The device-specific error codes are in the range 0x80 ... 0xFF.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_USERPARA_MIN (0x80)
```

2.1.14 Invalid Application parameter data (last device-specific error code)

Slave state machine requested connection to be reset after receiving Parameter frame with incompatible Application Parameters. Never requested by master state machine. The device-specific error codes are in the range 0x80 ... 0xFF.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

```
#define FSOESLAVE_RESETREASON_INVALID_USERPARA_MAX (0xFF)
```

2.2 FSOESLAVE_FRAME_SIZE - Get frame size

Number of bytes in FSoE Safety PDU frame containing *data_size* data bytes

Parameters

in	*data_size*	Number of data bytes in frame Needs to be even or 1.

Returns

Size of frame in bytes.

```
#define FSOESLAVE_FRAME_SIZE(data_size) (\
   ((data_size) == 1) ? 6 : (2 * (data_size) + 3) )
```

2.3 API functions status return codes

User API functions return codes. Returned from each API function to indicate if user called the function correctly as described in the function's documentation.

2.3.1 Status OK

User called the API correctly

```
#define FSOESLAVE_STATUS_OK         (0)
```

2.3.2 Status ERROR

User violated the function's preconditions. fsoeapp_handle_user_error() callback will give detailed information about what caused the function to return ERROR.

```
#define FSOESLAVE_STATUS_ERROR      (-1)
```

3 Types

Public types for the FSoE slave state machine are listed here. The following are available:

- fsoeslave_status_t: Status returned from API function.
- fsoeslave_state_t: Connection state.
- fsoeslave_resetevent_t: Reset event.
- fsoeslave_syncstatus_t: Status after call to fsoeslave_sync_with_master().
- fsoeslave_cfg_t: Configuration of state machine.
- fsoeslave_t: A state machine instance.

3.1 API function status

Status returned from API function Returned from each API function to indicate if user called the function correctly as described in the function's documentation. See FSOESLAVE_STATUS_OK See FSOESL↩ AVE_STATUS_ERROR

```
typedef int32_t fsoeslave_status_t;
```

3.2 Connection state

```
typedef enum fsoeslave_state
{
   FSOESLAVE_STATE_RESET,       /* Connection is reset */
   FSOESLAVE_STATE_SESSION,     /* The session IDs are being transferred */
   FSOESLAVE_STATE_CONNECTION,  /* The connection ID is being transferred */
   FSOESLAVE_STATE_PARAMETER,   /* The parameters are being transferred */
   FSOESLAVE_STATE_DATA,        /* Process or fail-safe data is being transferred */
} fsoeslave_state_t;
```

3.3 Connection reset event

A reset of connection between master and slave may be initiated by either side sending a Reset frame containing a code describing why the reset was initiated, such as an error detected by FSoE stack, system startup (only master to slave) or application request.

```
typedef enum fsoeslave_resetevent
{
   FSOESLAVE_RESETEVENT_NONE,        /*< No reset initiated. */
   FSOESLAVE_RESETEVENT_BY_MASTER,   /*< Reset was initiated by master
                                      * application or state machine.
                                      * A Reset frame was received from master
                                      * containing the reset code.
                                      */
   FSOESLAVE_RESETEVENT_BY_SLAVE,    /*< Reset was initiated by slave
                                      * application or state machine.
                                      * A Reset frame was sent to master
                                      * containing the reset code.
                                      */
} fsoeslave_resetevent_t;
```

3.4 Communication status

Status after synchronisation with master See fsoeslave_sync_with_master()

```
typedef struct fsoeslave_syncstatus
{
   bool is_process_data_received;    /*< Is process data received?
                                      * true:
                                      *   Valid process data was received in
                                      *   last FSoE cycle. The process data is
                                      *   stored in \a outputs buffer.
                                      *   Note that the process data could have
                                      *   been received in a previous call
                                      *   to fsoeslave_sync_with_master(). It is
                                      *   still considered valid though as no
                                      *   communication error has occured, such
                                      *   as timeouts or CRC errors.
                                      * false:
                                      *   No valid process data was received in
                                      *   last FSoE cycle. The \a outputs buffer
                                      *   contains only zeroes.
                                      *   This will be returned if an error has
                                      *   been detected, if connection with
                                      *   master is not established or if
                                      *   fail-safe data was received.
                                      */
   fsoeslave_resetevent_t reset_event; /*< Connection reset event.
                                      * If a reset event occured during this call
                                      * to fsoeslave_sync_with_master(), this will
                                      * indicate if it was initiated by slave or
                                      * master. Otherwise it is set to
                                      * FSOESLAVE_RESETEVENT_NONE.
                                      * Note that the slave state machine will wait
                                      * for master to reset the connection after
                                      * startup.
                                      */
   uint8_t reset_reason;             /*< Reason for connection reset.
                                      * In case a reset event occured, this
                                      * is the code sent/received in the
                                      * Reset frame. All codes except for
                                      * FSOESLAVE_RESETREASON_LOCAL_RESET
                                      * indicates that an error was detected.
                                      * See codes defined further up. Also see
                                      * fsoeslave_reset_reason_description().
                                      */
   fsoeslave_state_t current_state;  /*< Current state of the state machine */
} fsoeslave_syncstatus_t;
```

3.5 Configuration

```
typedef struct fsoeslave_cfg
{
   /*
    * Slave Address

    * An address uniquely identifying the slave;
    * No other slave within the communication system may have the same
    * Slave Address. Valid values are 0 - 65535.

    * This value will be received from master when connection is established,
    * and slave will verify that the value matches this value.
    * Slave will refuse the connection if wrong Slave Address is received.

    * See ETG.5100 ch. 8.2.2.4 "Connection state".
    */
   uint16_t slave_address;

   /*
    * Expected size in bytes of the application parameters

    * Valid values are 0 - FSOE_APPLICATION_PARAMETERS_MAX_SIZE.

    * Slave will check that the size of application parameters received
    * from master match this value. If it does not match, connection
    * will be rejected.
    */
   size_t application_parameters_size;

   /*
    * Size in bytes of the inputs to be sent to master

    * Only even values are allowed, except for 1, which is also allowed.
    * Maximum value is FSOE_PROCESS_DATA_MAX_SIZE.

    * Slave and master need to agree on the size of the inputs.
    * Communication between slave and master will otherwise not be possible.
    * The size of PDU frames sent to master will be
    * MAX (3 + 2 * inputs_size, 6).

    * See ETG.5100 ch. 4.1.2 (called "SafeOutputs").
    */
   size_t inputs_size;

   /*
    * Size in bytes of the outputs to be received from master

    * Only even values are allowed, except for 1, which is also allowed.
    * Maximum value is FSOE_PROCESS_DATA_MAX_SIZE.

    * Slave and master need to agree on the size of the outputs.
    * Communication between slave and master will otherwise not be possible.
    * The size of PDU frames received from master will be
    * MAX (3 + 2 * outputs_size, 6).

    * See ETG.5100 ch. 4.1.2 (called "SafeInputs").
    */
   size_t outputs_size;
} fsoeslave_cfg_t;
```

3.6 State machine instance

An FSoE slave state machine instance handles the connection with a single master.

User may allocate the instance statically or dynamically using malloc() or on the stack. To use an allocated instance, pass a pointer to it as the first argument to any API function.

This struct is made public as to allow for static allocation.

NOTE: User of the API is prohibited from accessing any of the fields as the layout of the structure is to be considered an implementation detail. Only the stack-internal file "fsoeslave.c" may access any field directly.

```
typedef struct fsoeslave
{
   ...
} fsoeslave_t;
```

4 Functions

These functions acts upon an instance:

- fsoeslave_get_state(): Get current state.
- fsoeslave_get_slave_session_id(): Get generated Slave Session ID.
- fsoeslave_get_master_session_id(): Get received Master Session ID.
- fsoeslave_get_process_data_sending_enable_flag(): Get flag indicating if sending process data is enabled.
- fsoeslave_clear_process_data_sending_enable_flag(): Clear flag indicating if sending process data is enabled.
- fsoeslave_set_process_data_sending_enable_flag(): Set flag indicating if sending process data is enabled.
- fsoeslave_set_reset_request_flag(): Set reset request flag.
- fsoeslave_sync_with_master(): Synchronise with master.
- fsoeslave_init(): Initialise slave state machine.

This function may be used if the SRA CRC feature is used:

- fsoeslave_update_sra_crc(): Update SRA CRC value.

These functions are mainly exposed for convenience when logging:

- fsoeslave_reset_reason_description(): Return description of reset reason.
- fsoeslave_state_description(): Return description of state.

For macros used by the functions, see Macros. For types used by the functions, see Types.

4.1 fsoeslave_reset_reason_description - Return description of reset reason

```
const char * fsoeslave_reset_reason_description (
   uint8_t reset_reason);
```

Return description of reset reason as a string literal

Parameters

in	*reset_reason*	Reset reason

Returns

String describing the reset reason, e.g. "local reset" or "INVALID_CRC", unless *reset_reason* is not a valid reset reason, in which case "invalid error code" is returned. In either case, the returned string is null- terminated and statically allocated as a string literal.

Example:

```
#include <fsoeslave.h>
void handle_connection_reset_by_master (uint8_t reset_reason)
{
   printf ("Master initiated connection reset due to %s (%u)\n",
         fsoeslave_reset_reason_description (reset_reason),
         reset_reason);
}
```

4.2 fsoeslave_state_description - Return description of instance state

```
const char * fsoeslave_state_description (
   fsoeslave_state_t state);
```

Return description of state machine state as a string literal

Parameters

in	*state*	State

Returns

String describing the state, unless *state* is not a valid state, in which case "invalid" is returned. In either case, the returned string is null- terminated and statically allocated as a string literal.

Example:

```
#include <fsoeslave.h>
fsoeslave_state_t state;

status = fsoeslave_state_description (slave, &state);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Current state is %s\n",
      fsoeslave_state_description (state));
}
```

4.3 fsoeslave_update_sra_crc - Update SRA CRC value

```
fsoeslave_status_t fsoeslave_update_sra_crc (
   uint32_t * crc,
   const void * data,
   size_t size);
```

This function will calculate the SRA CRC for data in *data*. If this is the first time the function is called, then user should first set *crc* to zero before calling the function. If this is a subsequent call then the previously calculated CRC value will be is used as input to the CRC calculation. The CRC *crc* will be updated in-place.

SRA CRC is an optional feature whose use is not mandated nor specified by the FSoE ETG.5100 specification. If used, the SRA CRC should be sent to slave as Application parameter, where it should be placed first (encoded in little endian byte order).

See ETG.5120 "Safety over EtherCAT Protocol Enhancements", ch. 6.3 "SRA CRC Calculation".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointers *crc* and *data* are non-null.

Parameters

`in,out`	*crc*	SRA CRC value to update in-place. If this is the first call to fsoeslave_update_sra_crc() then its value should first be set to zero.
`in`	*data*	Buffer with data.
`in`	*size*	Size of buffer in bytes. If zero, *crc* will be left unmodified.

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status1;
fsoeslave_status_t status2;
uint32_t crc;

crc = 0;
status1 = fsoeslave_update_sra_crc (&crc, data1, sizeof (data1));
status2 = fsoeslave_update_sra_crc (&crc, data2, sizeof (data2));
if (status1 == FSOESLAVE_STATUS_OK &&
    status2 == FSOESLAVE_STATUS_OK)
{
   printf ("Calculated SRA CRC: 0x%x\n", crc);
}
else
{
   printf ("We called function incorrectly (with null-pointers)\n");
}
```

4.4 fsoeslave_get_state - Get current state

```
fsoeslave_status_t fsoeslave_get_state (
   const fsoeslave_t * slave,
   fsoeslave_state_t * state);
```

Get current state of the FSoE slave state machine.

See ETG.5100 ch. 8.5.1.1 table 34: "States of the FSoE Slave".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointers *slave* and *state* are non-null.
fsoeslave_init() has been called for instance *slave*.

Parameters

in	*slave*	FSoE slave state machine
out	*state*	Current state

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status;
fsoeslave_state_t state;

status = fsoeslave_get_state (slave, &state);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Current state is %s\n",
      fsoeslave_state_description (state));
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.5 fsoeslave_get_slave_session_id - Get generated Slave Session ID

```
fsoeslave_status_t fsoeslave_get_slave_session_id (
   const fsoeslave_t * slave,
   uint16_t * session_id);
```

The Slave Session ID was generated by the slave state machine when entering Session state.

Calling this function while slave state machine is in Reset state is not allowed as no Slave Session ID has yet been generated.

See ETG.5100 ch. 8.2.2.3: "Session state".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointers *slave* and *session_id* are non-null.
fsoeslave_init() has been called for instance *slave*.
Slave state machine is at least in Session state.

Parameters

in	*slave*	FSoE slave state machine
out	*session_id*	Current Slave Session ID

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status;
uint16_t session_id;

status = fsoeslave_get_slave_session_id (slave, &session_id);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Generated Slave Session ID: %u\n", session_id);
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.6 fsoeslave_get_master_session_id - Get received Master Session ID

```
fsoeslave_status_t fsoeslave_get_master_session_id (
   const fsoeslave_t * slave,
   uint16_t * session_id);
```

The Master Session ID was generated by master and received by the slave state machine while in Session state.

Calling this function while slave state machine is in Reset or Session state is not allowed as no Slave Session ID has yet been received.

See ETG.5100 ch. 8.2.2.3: "Session state".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointers *slave* and *session_id* are non-null.
fsoeslave_init() has been called for instance *slave*.
Slave state machine is at least in Connection state.

Parameters

`in`	*slave*	FSoE slave state machine
`out`	*session↵ _id*	Current Master Session ID

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status;
uint16_t session_id;

status = fsoeslave_get_master_session_id (slave, &session_id);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Received Master Session ID: %u\n", session_id);
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.7 fsoeslave_get_process_data_sending_enable_flag - Get flag indicating if sending process data is enabled

```
fsoeslave_status_t fsoeslave_get_process_data_sending_enable_flag (
   const fsoeslave_t * slave,
   bool * is_enabled);
```

Get flag indicating if sending process data to master is enabled. This will only check a flag indicating that everything is OK from the perspective of the application. Slave will not send normal process data if connection with master is not fully established (Data state), even if application allows it.

See ETG.5100 ch. 8.5.1.2 "Set Data Command event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointers *slave* and *is_enabled* are non-null.
fsoeslave_init() has been called for instance *slave*.

Parameters

`in`	*slave*	FSoE slave state machine
`out`	*is_enabled*	Current process data send status: true if slave is allowed to send process data, false if not.

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status;
bool is_enabled;

status = fsoeslave_get_process_data_sending_enable_flag (slave, &is_enabled);
if (status == FSOESLAVE_STATUS_OK)
{
   if (is_enabled)
   {
      printf ("Slave is allowed to send process data to master\n");
   }
   else
   {
      printf ("Slave is not allowed to send process data to master\n");
   }
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.8 fsoeslave_clear_process_data_sending_enable_flag - Clear flag indicating if sending process data is enabled

```
fsoeslave_status_t fsoeslave_clear_process_data_sending_enable_flag (
   fsoeslave_t * slave);
```

This will clear a flag indicating that everything is OK from the perspective of the application. Slave will only send fail safe data (zeros) to master. This is the default setting after power-on and after detection of any errors.

See ETG.5100 ch. 8.5.1.2 "Set Data Command event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointer *slave* is non-null.
fsoeslave_init() has been called for instance *slave*.

Parameters

in,out	*slave*	FSoE slave state machine

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status;

status = fsoeslave_clear_process_data_sending_enable_flag (slave);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Sending process data to master is no longer allowed\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.9 fsoeslave_set_process_data_sending_enable_flag - Set flag indicating if sending process data is enabled

```
fsoeslave_status_t fsoeslave_set_process_data_sending_enable_flag (
   fsoeslave_t * slave);
```

This will set a flag indicating that everything is OK from the perspective of the application. Setting the flag will cause slave to send inputs containing valid process data once connection is established, assuming no errors are detected. If any errors are detected, this flag will revert to its disabled state and only fail-safe inputs will be sent.

See ETG.5100 ch. 8.5.1.2 "Set Data Command event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointer *slave* is non-null.
fsoeslave_init() has been called for instance *slave*.

Parameters

`in,out`	*slave*	FSoE slave state machine

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status;

status = fsoeslave_set_process_data_sending_enable_flag (slave);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Sending process data to master is now allowed\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.10 fsoeslave_set_reset_request_flag - Set reset request flag

```
fsoeslave_status_t fsoeslave_set_reset_request_flag (
   fsoeslave_t * slave);
```

This will set a flag, which in next call to fsoeslave_sync_with_master() will cause the slave state machine to send a Reset frame to master and then wait for master to re-establish connection. Fail-safe mode will then be entered, where normal process data inputs will not be sent even after connection has been re-established. Application needs to explicitly re-enable process data inputs in order to leave fail-safe mode. See fsoeslave_set_process_data_sending_enable_flag().

See ETG.5100 ch. 8.5.1.2 "Reset Connection event".

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointer *slave* is non-null.
fsoeslave_init() has been called for instance *slave*.

Parameters

`in,out`	*slave*	FSoE slave state machine

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_status_t status;

status = fsoeslave_set_reset_request_flag (slave);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Slave state machine reset was requested\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.11 fsoeslave_sync_with_master - Synchronise with master

```
fsoeslave_status_t fsoeslave_sync_with_master (
      fsoeslave_t * slave,
      const void * inputs,
      void * outputs,
      fsoeslave_syncstatus_t * sync_status);
```

Needs to be called periodically in order to avoid watchdog timeout. It is recommended that delay between calls to the function is no more than half the watchdog timeout.

Depending on current state, the slave state machine may try to send a single frame or read a single frame by calling fsoeapp_send() and/or fsoeapp_recv(), which are non-blocking functions.

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointers *slave*, *inputs*, *outputs* and *sync_status* are non-null.
fsoeslave_init() has been called for instance *slave*.

Parameters

`in,out`	*slave*	FSoE slave state machine
`in`	*inputs*	Buffer containing inputs to be sent to master. Its size is given in configuration.
`out`	*outputs*	Buffer to store outputs received from master. Its size is given in configuration. Whether outputs are valid or not is given by *sync_status*.
`out`	*sync_status*	Status of FSoE connection.

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
fsoeslave_syncstatus_t sync_status;
fsoeslave_status_t status;
uint8_t outputs[2];

inputs[0] = 0x56;
inputs[1] = 0x78;
status = fsoeslave_sync_with_master (slave, inputs, outputs, &sync_status);
if (status == FSOESLAVE_STATUS_OK)
{
   if (sync_status.reset_event != FSOESLAVE_RESETEVENT_NONE)
   {
      printf ("Connection was reset by %s. Cause: %s\n",
         sync_status.reset_event == FSOESLAVE_RESETEVENT_BY_MASTER ?
         "master" : "slave",
         fsoeslave_reset_reason_description (sync_status.reset_reason));
   }
   if (sync_status.is_process_data_received)
   {
      handle_received_data (outputs);
   }
   else
   {
      printf ("No valid process data was received\n);
   }
}
else
{
   printf ("We called function incorrectly\n");
}
```

4.12 fsoeslave_init - Initialise FSoE slave state machine

```
fsoeslave_status_t fsoeslave_init (
   fsoeslave_t * slave,
   const fsoeslave_cfg_t * cfg,
   void * app_ref);
```

This will configure the instance according to supplied configuration.

Before taking any action, function will first validate that its preconditions (see below) were respected. If this was not the case, the function fsoeapp_handle_user_error() will first be called after which function will exit with status FSOESLAVE_STATUS_ERROR.

Precondition

The pointers *slave* and *cfg* are non-null.
The fields in *cfg* are valid.

Parameters

out	*slave*	FSoE slave state machine
in	*cfg*	Configuration
in	*app_ref*	Application reference. This will be passed as first argument to callback functions implemented by application. The stack does not interpret this value in any way.

Returns

Status:

- FSOESLAVE_STATUS_OK if function was called correctly.
- FSOESLAVE_STATUS_ERROR if user violated a precondition.

Example:

```
#include <fsoeslave.h>
const fsoeslave_cfg_t cfg =
{
   .slave_address                = 0x0304,
   .application_parameters_size = 0,
   .inputs_size                  = 2,
   .outputs_size                 = 2,
};
fsoeslave_t slave;
fsoeslave_status_t status;
```

```
status = fsoeslave_init (&slave, &cfg, NULL);
if (status == FSOESLAVE_STATUS_OK)
{
   printf ("Slave state machine was initialised\n");
}
else
{
   printf ("We called function incorrectly\n");
}
```

5 Callback functions and types

The following callback functions are declared in fsoeapp.h and shall be implemented in application:

- fsoeapp_send(): Send a Safety PDU frame.
- fsoeapp_recv(): Receive a Safety PDU frame.
- fsoeapp_generate_session_id(): Generate a Session ID.
- fsoeapp_verify_parameters(): Verify received parameters.
- fsoeapp_handle_user_error(): Handle user error.

The functions fsoeapp_send() and fsoeapp_recv() are the means for the state machine to access the black channel. Application may implement these with support for cross-checking. The arrows in picture below denote direct function calls:

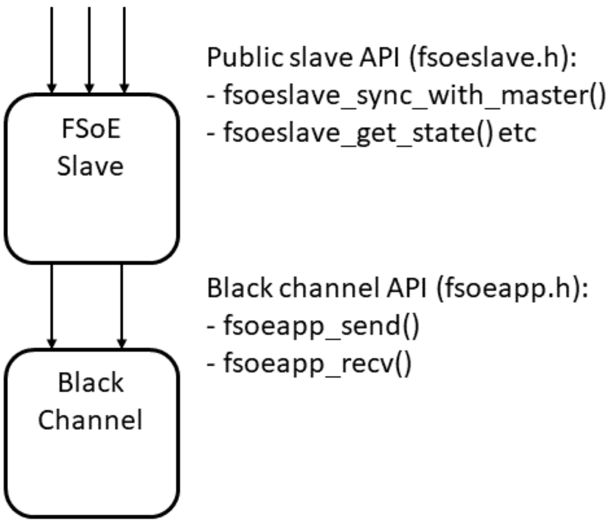

Public slave API (fsoeslave.h):
- fsoeslave_sync_with_master()
- fsoeslave_get_state() etc
FSoE Slave
Black channel API (fsoeapp.h):
- fsoeapp_send()
- fsoeapp_recv()
Black Channel

Figure 4.2: FSoE stack layers

The slave state machine does not verify that the Session ID returned from fsoeapp_generate_session_↩ id() is generated correctly. To pass the conformance test, the application should ensure that the Session ID is re-generated as a random number in accordance with the specification.

The function fsoeapp_verify_parameters() should verify that parameters received from Master are valid. The parameters include both the watchdog timeout and application-specific parameters.

The function fsoeapp_handle_user_error() can be implemented as a system restart, or it can be implemented as an empty function which does nothing.

The header file also makes the following macros and types available:

- FSOEAPP_STATUS_xxx: Status returned by fsoeapp_verify_parameters().
- fsoeapp_usererror_t: User error.

The header file also makes the following function available for convenience when logging:

- fsoeapp_user_error_description(): Return description of user error. Passed to fsoeapp_handle_↩ user_error() when an API function detects that user violated a precondition.

5.1 API functions status return codes

Note that values in the range 0x80 - 0xff are also allowed, indicating that some application-specific parameter is invalid.

See ETG.5100 ch. 8.3. table 28: "FSoE communication error codes".

5.1.1 Parameters are valid

```
#define FSOEAPP_STATUS_OK                 0
```

5.1.2 Invalid watchdog timeout

```
#define FSOEAPP_STATUS_BAD_TIMOUT         9
```

5.1.3 Invalid application-specific parameter

```
#define FSOEAPP_STATUS_BAD_APP_PARAMETER  11
```

5.2 User error

User error. Passed to fsoeapp_handle_user_error() when an API function detects that user violated a precondition. See fsoeapp_handle_user_error() and fsoeapp_user_error_description().

```
typedef enum fsoeapp_usererror
{
   FSOEAPP_USERERROR_NULL_INSTANCE = 1,/*< User violated the API by passing
                                        * a null-pointer instance.
                                        */
   FSOEAPP_USERERROR_UNINITIALISED_INSTANCE, /*< User violated the API by
                                        * calling API function before calling
                                        * fsoemaster_init() or fsoeslave_init().
                                        */
   FSOEAPP_USERERROR_WRONG_INSTANCE_STATE, /*< User violated the API by
                                        * calling function when instance was in a
                                        * state prohibited by the function's
                                        * documentation.
                                        */
   FSOEAPP_USERERROR_NULL_ARGUMENT, /*< User violated the API by passing
                                        * a null-pointer argument (other than
                                        * the instance itself).
                                        */
   FSOEAPP_USERERROR_BAD_CONFIGURATION, /*< User violated the API by calling
                                        * fsoemaster_init() or fsoeslave_init()
                                        * with a configuration with bad field.
                                        */
} fsoeapp_usererror_t;
```

See also

fsoeapp_handle_user_error().
fsoeapp_user_error_description().

5.3 fsoeapp_user_error_description - Return description of user error

```
const char * fsoeapp_user_error_description (
   fsoeapp_usererror_t user_error);
```

Return description of user error as a string literal This is just a helper function which may be used for logging the error code passed to fsoeapp_handle_user_error(). It is implemented by the FSoE stack itself.

Parameters

in	*user_error*	User error

Returns

String describing the user error, unless *user_error* is not a valid error, in which case "invalid error code" is returned. In either case, the returned string is null- terminated and statically allocated as a string literal

Example:

```
#include <fsoeapp.h>
void fsoeapp_handle_user_error (
   void * app_ref, fsoeapp_usererror_t user_error)
{
   printf ("We called an API function incorrectly: %s\n",
      fsoeapp_user_error_description (user_error));
}
```

See also

fsoeapp_handle_user_error().

5.4 fsoeapp_send - Send a Safety PDU frame

```
void fsoeapp_send (void * app_ref, const void * buffer, size_t size);
```

Send a complete FSoE PDU frame. An FSoE PDU frame starts with the Command byte and ends with the Connection ID. Its size is given by the formula MAX (3 + 2 $*$ data_size, 6), where data_size is the number of data bytes to send and is given by a field in fsoemaster_cfg_t or fsoeslave_cfg_t.

See ETG.5100 ch. 8.1.1 "Safety PDU structure".

This callback function is called by the FSoE stack when it wishes to send a frame. Application is required to be implement this by making an attempt to send the frame in supplied buffer. If application wishes to communicate an error condition to the FSoE stack then it may do so by calling fsoemaster_set_reset_↩ request_flag() or fsoeslave_set_reset_request_flag().

NOTE: If cross-checking is required it depends on implemented model if the data is cross-checked before sending or not, the example models from iec61784-3 suggest cross-checking to be performed before sending if only a single frame is sent, this is typically true for a FSoE device running on EtherCAT Slave Controller HW.

NOTE: If the FSoE stack runs on a different thread or execution context than the thread or execution context where the frame is actually sent, then the application has to ensure that no race condition can occur. Otherwise, a partial frame may be sent, causing CRC error and a reset of connection. Race conditions may be avoided by means of a mutex, a semaphore, by locking interrupts or some other mechanism.

Parameters

`in,out`	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_init() or fsoeslave_init(). Application is free to use this as it sees fit.
`in`	*buffer*	Buffer containing a PDU frame to be sent.
`in`	*size*	Size of PDU frame in bytes. Always the same, as given by formula above.

EtherCAT example iec61784-3, Annex A Model A

Example:

```
void fsoeapp_send (void * app_ref, const void * buffer, size_t size)
{
   /* Configure for cross-checking in HW or SW */
#if FSOE_REDUNDANT_SCL_IN_HW
   /* copy the HW checked and encoded SPDU to the black channel */
   memcpy(FSOE_Slave_Frame_Elements, buffer, size);
#else
   /* send encoded data to be cross-checked with encoded data from
    * second CPU. if the result match we copy it to the black channel.
    * if the result doesn't match, we do a reset.
    */
   if(cross_check_encoded_data(buffer, size) == ERROR)
   {
      fsoeslave_set_reset_request_flag (fsoe_slave)
   }
   else
   {
      memcpy(FSOE_Slave_Frame_Elements, buffer, size);
   }
#endif
}
```

5.5 fsoeapp_recv - Receive a Safety PDU frame

```
size_t fsoeapp_recv (void * app_ref, void * buffer, size_t size);
```

Try to receive a complete FSoE PDU frame. An FSoE PDU frame starts with the Command byte and ends with the Connection ID. Its size is given by the formula MAX (3 + 2 $*$ data_size, 6), where data_size is the number of data bytes to receive and is given by a field in fsoemaster_cfg_t or fsoeslave_cfg_t.

See ETG.5100 ch. 8.1.1 "Safety PDU structure".

This callback function is called by the FSoE stack when it wishes to receive a frame. Application is required to implement this by first checking if a frame was received. If no new frame was received then the function should either

- return without waiting for any incoming frame or
- copy previously received frame to buffer and return.

If a frame was received then its content should be copied to buffer. If application wishes to communicate an error condition to the FSoE stack then it may do so by calling fsoemaster_reset_connection() or fsoeslave_reset_connection().

NOTE: If cross-checking is required, no cross-checking is performed here but on decoded data returned by the stack.

NOTE: If the FSoE stack runs on a different thread or execution context than the thread or execution context where the frame is actually received, then the application has to ensure that no race condition can occur. Otherwise, fsoeapp_recv() may receive a partial frame, causing CRC error and a reset of connection. Race conditions may be avoided by means of a mutex, a semaphore, by locking interrupts or some other mechanism.

Parameters

`in,out`	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_init() or fsoeslave_init(). Application is free to use this as it sees fit.
`out`	*buffer*	Buffer where recieved PDU frame will be stored.
`in`	*size*	Size of PDU frame in bytes. Always the same, as given by formula above.

Returns

Number of bytes received. Should be equal to *size* if a frame was received. If no frame was received, it may be equal 0. Alternatively, the last received frame may be put in the buffer with *size* bytes returned.

Example:

```
/* FSoE stack receive data from black channel */
size_t fsoeapp_recv (void * app_ref, void * buffer, size_t size);
{
   /* Ok to discard data received in states below OP */
   if ((CC_ATOMIC_GET (ESCvar.App.state) & APPSTATE_OUTPUT) <= 1)
   {
      return 0;
   }

   /* copy black channel data received from FSoE Device partner,
    * size is fixed and set at configuration.
    * cross-checking is done on decoded FSoE data, not here */
   memcpy (buffer, FSOE_Master_Frame_Elements, size);
   return size;
}
```

5.6 fsoeapp_generate_session_id - Generate a Session ID

```
uint16_t fsoeapp_generate_session_id (void * app_ref);
```

Generate a Session ID. A Session ID is a random 16 bit number.

See ETG.5100 ch. 8.1.3.7 "Session ID".

This callback function is called by the FSoE stack after power-on and after each connection reset. Application is required to implement this by generating a random number which is sufficiently random that a (with high probability) different random number will be generated after each system restart. A normal pseudo-random algorithm with fixed seed value is not sufficient.

NOTE: If cross-checking is required, the two SCL must synchronise the random number used for session ID. All subsequent frames will "inherit" from this random number due the inclusion of received CRC_0 in sent frames.

Parameters

`in,out`	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_init() or fsoeslave_init(). Application is free to use this as it sees fit.

Returns

A generated Session ID

Example:

```
uint16_t fsoeapp_generate_session_id (void * app_ref)
{
   /* Configure for cross-checking in HW or SW */
#if FSOE_REDUNDANT_SCL_IN_HW
   return (uint16_t)platform_rand();
#else
   uint16_t my_rand = (uint16_t)platform_rand();
   uint16_t partner_rand = cross_check_fetch_partner();
   return my_rand + partner_rand;
#endif
}
```

5.7 fsoeapp_handle_user_error - Handle user error

```
void fsoeapp_handle_user_error (
   void * app_ref,
   fsoeapp_usererror_t user_error);
```

User called an API function in a way that violated a precondition. The API function detected this and before returning it called this function.

Application may implement this by restarting the system if running on an embedded target or quit the process if running on a PC. In these cases, the API function will not return and the user does not need to check the returned error code.

Application may also implement this by returning to the API function. The API function will then return the error to user, which should then handle the error. Note that any elaborate error handling by user is unlikely to succeed as it was the user who committed the error in the first place by violating the API function's preconditions.

Note

If using a debugger, this may be a good place to put a debug breakpoint.
This function will never be called if user calls all API functions correctly.

Parameters

in,out	*app_ref*	Application reference: • NULL if error was detected by a state machine function and *user_error* was either FSOEAPP_USERERROR_NULL_INSTANCE or FSOEAPP_USERERROR_UNINITIALISED_INSTANCE. • NULL if error was detected by the functions fsoeslave_update_sra_crc() or fsoemaster_update_sra_crc(). • Otherwise, this is the pointer with the same name passed to fsoeslave_init() or fsoemaster_init().
in	*user_error*	Type of error user made when calling the API function.

Example:

```
#include <fsoeapp.h>
void fsoeapp_handle_user_error (
   void * app_ref, fsoeapp_usererror_t user_error)
{
   printf ("User called API function incorrectly: %s, app_ref: %p\n"),
      fsoeapp_user_error_description (user_eror), app_ref);
   assert (0);
}
```

5.8 fsoeapp_verify_parameters - Verify received parameters

```
uint8_t fsoeapp_verify_parameters (
      void * app_ref,
      uint16_t timeout_ms,
      const void * app_parameters,
      size_t size);
```

The parameters include both communication parameters (the watchdog timeout) as well as application-specific parameters.

See ETG.5100 ch. 7.1 "FSoE Connection".

This callback function is called by FSoE slave when all parameters have been received from master. Application is required to be implement this by verifying that the parameters are valid, returning an error code if not. If error is returned, slave will reset the connection and send the specified error code to master. The master stack does not call this function.

Parameters

in,out	*app_ref*	Application reference. This pointer was passed to stack in fsoeslave_init(). Application is free to use this as it sees fit.
in	*timeout_ms*	Watchdog timeout in milliseconds.
in	*app_parameters*	Buffer with received application-specific parameters.
in	*app_parameters_size*	Size of application-specific parameters in bytes. Will always be equal to configured application parameter size. See fsoeslave_cfg_t.

Returns

Error code:

- FSOEAPP_STATUS_OK if all parameters are valid,
- FSOEAPP_STATUS_BAD_TIMOUT if the watchdog timeout is invalid,
- FSOEAPP_STATUS_BAD_APP_PARAMETER if application-specific parameters are invalid,
- 0x80-0xFF if application-specific parameters are invalid and cause is given by application-specific error code.

Example:

```
uint8_t fsoeapp_verify_parameters (
      void * app_ref,
      uint16_t timeout_ms,
      const void * app_parameters,
      size_t size)
{
   /* Verify watchdog */
   if (timeout_ms < FSOE_MIN_WATCHDOG || timeout_ms >= FSOE_MAX_WATCHDOG)
   {
      /* FSOEAPP_STATUS_BAD_TIMOUT */
      return FSOEAPP_STATUS_BAD_TIMOUT;
   }
   /* Verify application parameters */
   if(size > 0)
   {
      uint16_t * param1 = (uint16_t *)app_parameters;
      uint16_t * param2 = (uint16_t *)app_parameters + 1;
      parameters_t * temp_param =  (parameters_t *)app_ref;
      if(*param1 != temp_param->app_param1 ||
         *param2 != temp_param->app_param2)
      {
         /* FSOEAPP_STATUS_BAD_APP_PARAMETER */
         return FSOEAPP_STATUS_BAD_APP_PARAMETER;
      }
   }
   return FSOEAPP_STATUS_OK;
}
```

Chapter 5

Platform Abstraction Layer

1 Overview

The platform abstraction layer is composed of the macros and functions listed below. The functions are declared in the stack-internal header file fsoeport.h and should be implemented in a file fsoeport.c. The macros should be implemented in a file fsoeport_macros.h. Both files should be placed in a directory src/port/PLATFORM_NAME, where PLATFORM_NAME is the name of the platform added to the FSoE stack.

List of required macros:

- FSOEPORT_ASSERT(): Assert that expression is true.
- FSOEPORT_IS_BIG_ENDIAN: Flag indicating platform endianness.
- FSOEPORT_FORMAT(): Function attribute for printf-like functions.

List of required functions:

- fsoeport_current_time_us(): Get current microsecond time.
- fsoeport_memcpy(): Copy block of memory, like memcpy().
- fsoeport_memcmp(): Compare blocks of memory, like memcmp().
- fsoeport_memset(): Fill block of memory, like memset().

List of optional functions:

- fsoeport_log(): Write a log message with printf-like format.

2 Macros

2.1 FSOEPORT_ASSERT - Assert that expression is true

Assertions shall never fail. An assertion failure indicates that a fatal event has occurred, likely some kind of memory corruption. Program is no longer in a defined state and should be terminated as to avoid safety violations. Termination may be accomplished by means of a watchdog reset or some other mechanism.

Parameters

in	*exp*	Expression which should always be true

Example:

```
#include <assert.h>
#ifdef NDEBUG
#define FSOEPORT_ASSERT(exp)        fsoeport_assert (exp);
static inline void fsoeport_assert (int exp)
{
   if (!exp)
   {
      /* Wait for watchdog reset */
      for (;;);
   }
}
#else
#define FSOEPORT_ASSERT(exp)        assert (exp);
#endif /* NDEBUG */
```

2.2 FSOEPORT_IS_BIG_ENDIAN - Specify if this is a big endian platform

If set to 1 then this is a big endian platform. The most significant 8 bits of a word are stored at the lowest byte address. If set to 0 then this is a little endian platform. The least significant 8 bits of a word are stored at the lowest byte address.

Note

The defines used in sample code below might not be applicable for the user platform, The user shall ensure that correct endianness is configured.

Example:

```
#if BYTE_ORDER == LITTLE_ENDIAN
#define FSOEPORT_IS_BIG_ENDIAN      0
#else
#define FSOEPORT_IS_BIG_ENDIAN      1
#endif
```

2.3 FSOEPORT_FORMAT - Function attribute specifying printf-like format string

This function attribute is used to enable compiler to emit warnings if types of supplied arguments do not match supplied format string. If compiler does not have such function attribute then this can be defined as an empty macro.

Parameters

in	*str*	Position of the format string within the argument list. Argument positions start at 1.
in	*arg*	Position of first argument of the variable number of arguments. Argument positions start at 1.

Example:

```
#define FSOEPORT_FORMAT(str, arg)   __attribute__((format (printf, str, arg)))
```

3 Functions

3.1 fsoeport_current_time_us - Get current microsecond time

Time should be monotonically increasing (except for wrap-around). When time is close to 0xffffffff it should wrap around to a value close to zero.

Time will be used for determining whether the watchdog timer has expired or not. It may also be used for logging purposes.

Returns

Current time, in microseconds

Example:

```
uint32_t fsoeport_current_time_us (void)
{
   return (uint32_t)platform_us();
}
```

3.2 fsoeport_memcpy - Copy block of memory

Copies the values of *num* bytes from the location pointed to by *src* directly to the memory block pointed to by *dest*. The blocks may not overlap.

Parameters

`out`	*dest*	Pointer to the destination array where the content is to be copied.
`in`	*src*	Pointer to the source of data to be copied.
`in`	*num*	Number of bytes to copy.

Example:

```
#include <string.h>
void fsoeport_memcpy (void * dest, const void * src, size_t num)
{
   (void)memcpy (dest, src, num);
}
```

3.3 fsoeport_memcmp - Compare two blocks of memory

Compares the first *num* bytes of the block of memory pointed by *ptr1* to the first *num* bytes pointed by *ptr2*, returning zero if they all match or a value different from zero otherwise.

Parameters

`in`	*ptr1*	Pointer to block of memory.
`in`	*ptr2*	Pointer to block of memory.
`in`	*num*	Number of bytes to compare.

Returns

Result of comparison:

- Negative if the first byte that does not match in both memory blocks has a lower value in *ptr1* than in *ptr2* (if evaluated as unsigned char values).
- Zero if the contents of both memory blocks are equal.
- Positive if the first byte that does not match in both memory blocks has a greater value in *ptr1* than in *ptr2* (if evaluated as unsigned char values).

Example:

```
#include <string.h>
int32_t fsoeport_memcmp (const void * ptr1, const void * ptr2, size_t num)
{
   return memcmp (ptr1, ptr2, num);
}
```

3.4 fsoeport_memset - Fill block of memory

Sets the first *num* bytes of the block of memory pointed by *ptr* to the specified value.

Parameters

out	*ptr*	Pointer to the block of memory to fill.
in	*value*	Value to be set
in	*num*	Number of bytes to be set to the value.

Example:

```
#include <string.h>
void fsoeport_memset (void * ptr, uint8_t value, size_t num)
{
   (void)memset (ptr, value, num);
}
```

4 Optional functions

4.1 fsoeport_log - Write a log message with printf-like format

Logging is an optional feature useful for development purposes. If logging is not enabled, then this function does not need to be implemented.

Note

Enabling logging functions require support for printf macros defined by inttypes.h

Parameters

in	*type*	Type of log message
in	*fmt*	Printf-like format string
in	...	Additional parameters as specified by format string.

Example:

```
#include <stdio.h>
#include <stdarg.h>
void fsoeport_log (uint8_t type, const char * fmt, ...)
{
   va_list list;

   switch (FSOELOG_LEVEL_GET (type))
   {
   case FSOELOG_LEVEL_DEBUG:   printf ("[DEBUG] "); break;
   case FSOELOG_LEVEL_INFO:    printf ("[INFO ] "); break;
   case FSOELOG_LEVEL_WARNING: printf ("[WARN ] "); break;
   case FSOELOG_LEVEL_ERROR:   printf ("[ERROR] "); break;
   default: break;
   }

   va_start (list, fmt);
   vprintf (fmt, list);
   va_end (list);
   fflush (stdout);
}
```

Chapter 6

Checklists

1 Checklist for implementing FSoE master device

1.1 Implementation 'SHALL'

Ref	Description	Action
#1	The user shall consult Safety over EtherCAT Policy ETG.9100 for rules and requirements for using and implementing the Safety over EtherCAT Technology. ETG.9100 contains references to additional standards and requirements	
#2	The FSoE device shall be compliant to - IEC 61508 as basic safety standard or relevant sector/product standards with respect to safety requirements. - IEC 61784-3 general part. - ETG.5100 Safety over EtherCAT Specification.	
#3	Cross-checking shall be implemented if it is required to achieve claimed safety integrity level	
#4	The callback function fsoeapp_generate_session_id() shall share Session ID between multiple controllers if applicable	
#5	The callback functions fsoeapp_send() and fsoeapp_recv() shall expect frames on the format CMD, DATA0 ,CRC0,..., CRCn, CONN ID	
#6	The callback functions fsoeapp_send() and fsoeapp_recv() shall translate Safety P↩ DU format to black channel format, if needed	
#7	The callback function fsoeapp_recv() shall only return the configured size of Safe↩ Inputs or zero	
#8	The FSoE state machine API is not thread safe and shall be protected by the safety application from concurrent access	
#9	Null-pointers shall not be passed to API functions, with the exception of fsoemaster↩ _init()'s parameter *app_ref* and field *application_parameters* in fsoemaster_cfg_t.	
#10	The application shall not continue executing any FSoE API function if fsoemaster_↩ init() fails since the allocated instance is invalid	
#11	The application shall not access members of **fsoemaster_t** directly, only via the stack API. Internal members are only public to be enable the safety application to allocate memory statically	
#12	The Connection ID shall not be zero.	
#13	The configured Slave Address shall be the address used by the associated slave	

Ref	Description	Action
#14	The watchdog timeout shall be set to a value within the range supported by the associated slave	
#15	The size of SafeInputs shall be either 1 byte or an even number of bytes and not larger than FSOE_PROCESS_DATA_MAX_SIZE.	
#16	The size of SafeOutputs shall be either 1 byte or an even number of bytes and not larger than FSOE_PROCESS_DATA_MAX_SIZE.	
#17	The size of the application-specific parameters shall not be larger than FSOE_AP↩ PLICATION_PARAMETERS_MAX_SIZE.	
#18	The size of SafeInputs shall be the same as used by associated slave.	
#19	The size of SafeOutputs shall be the same as used by associated slave.	
#20	The size of the application-specific parameters shall be the same as used by associated slave.	
#21	The layout of the application-specific parameters shall be the same as used by associated slave.	
#22	If the SRA parameter CRC calculation feature is used, the SRA CRC shall: be placed first in *application_parameters*, be encoded in little endian byte order and its length should be added to *application_parameters_size*	
#23	fsoemaster_set_process_data_sending_enable_flag() shall not be called unless application has determined that no error condition is present and that application is ready to send process data	
#24	Application shall call the fsoemaster_sync_with_slave() periodically in order to avoid watchdog timeout. It is the safety application that shall ensure it is periodically executed and monitor that it is done, failure to do so shall be handled by the safety application	
#25	The size of the *outputs* buffer passed to fsoemaster_sync_with_slave() shall be as configured in fsoemaster_init()	
#26	The size of the *inputs* buffer passed to fsoemaster_sync_with_slave() shall be as configured in fsoemaster_init()	
#27	If fsoemaster_update_sra_crc() is called for the first time, the in-argument *crc* shall first be set to zero	
#28	fsoemaster_get_master_session_id() shall not be called if state machine is in Reset state	
#29	fsoemaster_get_slave_session_id() shall not be called if state machine is in Reset or Session state	

1.2 Implementation 'SHOULD'

Ref	Description	Action
#1	The callback function fsoeapp_recv() should not wait for frame to be received	
#2	The callback function fsoeapp_generate_session_id() should not generate the same Session ID after each start-up	
#3	The safety application or configuration tool should verify the uniqueness of Slave Addresses and Connection IDs	
#4	Application should call fsoemaster_set_reset_request_flag() if it detects an application-level error while communicating with slave	
#5	The FSoE Conformance Test should pass	
#6	If the stack may run on a 1 or 2 controller hardware solution, FSoE Application layer should be implemented to support either hardware setup	

Ref	Description	Action
#7	If the platform is missing **stdbool.h**, a local header file should be added: stdbool.h #define true 1 #define false 0 #define bool _Bool	
#8	If the platform is missing **stdint.h**, a local header file should be added	

1.3 Implementation 'MAY'

Ref	Description	Action
#2	If the platform is missing **inttypes.h**, a local header file may be added	

1.4 Implementation 'CAN'

Ref	Description	Action
#1	**FSOE_PROCESS_DATA_MAX_SIZE:** can be adjusted to decrease RAM usage for objects allocated by the safety application.	
#2	**FSOE_APPLICATION_PARAMETERS_MAX_SIZE:** can be adjusted to decrease RAM usage for objects allocated by the safety application.	
#3	**FSOE_LOG_ENABLE:** can be used to enable run-time logging provided by the stack	
#4	**FSOE_LOG_ENABLE:** the user implements the logging function	
#5	**FSOE_LOG_LEVEL:** sets the log level, given logging is enabled	
#6	**FSOE_LOG_MASTER:** enabled logging for specific object	
#7	**FSOE_LOG_SLAVE:** enabled logging for specific object	
#8	**FSOE_LOG_FRAME:** enabled logging for specific object	
#9	**FSOE_LOG_CHANNEL:** enabled logging for specific object	

1.5 Implementation 'Limitations'

Ref	Description	Action
#1	**MAX safetydata size:** Maximum data size is 126 bytes SafeOutputs and 126 bytes SafeInputs per FSoE connection. The limitation is implementation specific due to using the CRC algorithm provided by ETG.5100 "Safety Over EtherCAT" specification, it is possible to extend.	
#2	**Dynamic SafeData:** dynamic mapping of the SafeData is not supported	

2 Checklist for implementing FSoE slave device

2.1 Implementation 'SHALL'

Ref	Description	Action
#1	The user shall consult Safety over EtherCAT Policy ETG.9100 for rules and requirements for using and implementing the Safety over EtherCAT Technology. ETG.9100 contains references to additional standards and requirements	

Ref	Description	Action
#2	The FSoE device shall be compliant to - IEC 61508 as basic safety standard or relevant sector/product standards with respect to safety requirements. - IEC 61784-3 general part. - ETG.5100 Safety over EtherCAT Specification.	
#3	Cross-checking shall be implemented if it is required to achieve claimed safety integrity level	
#4	The callback function fsoeapp_generate_session_id() shall share Session ID between multiple controllers if applicable	
#5	The callback functions fsoeapp_send() and fsoeapp_recv() shall expect frames on the format CMD, DATA0 ,CRC0,..., CRCn, CONN ID	
#6	The callback functions fsoeapp_send() and fsoeapp_recv() shall translate Safety P↩ DU format to black channel format, if needed	
#7	The callback function fsoeapp_recv() shall only return the configured size of Safe↩ Outputs or zero	
#8	The callback function fsoeapp_verify_parameters() shall return FSOEAPP_STAT↩ US_BAD_TIMOUT if received watchdog timeout can not be supported	
#9	The callback function fsoeapp_verify_parameters() shall return FSOEAPP_STAT↩ US_BAD_APP_PARAMETER or a value in range 0x80 - 0xff if received watchdog timeout is OK but received application-specific parameters are not	
#10	If the SRA parameter CRC calculation feature is used, the callback function fsoeapp_verify_parameters() shall expect the SRA CRC to: be placed first in *application_parameters*, be encoded in little endian byte order	
#11	The FSoE state machine API is not thread safe and shall be protected by the safety application from concurrent access	
#12	Null-pointers shall not be passed to API functions, with the exception of fsoeslave↩ _init()'s parameter *app_ref*	
#13	The application shall not continue executing any FSoE API function if fsoeslave_init() fails since the allocated instance is invalid	
#14	The application shall not access members of **fsoeslave_t** directly, only via the stack API. Internal members are only public to be enable the safety application to allocate memory statically	
#15	The size of SafeInputs shall be either 1 byte or an even number of bytes and not larger than FSOE_PROCESS_DATA_MAX_SIZE.	
#16	The size of SafeOutputs shall be either 1 byte or an even number of bytes and not larger than FSOE_PROCESS_DATA_MAX_SIZE.	
#17	The size of the application-specific parameters shall not be larger than FSOE_AP↩ PLICATION_PARAMETERS_MAX_SIZE.	
#18	fsoeslave_set_process_data_sending_enable_flag() shall not be called unless application has determined that no error condition is present and that application is ready to send process data	
#19	Application shall call the fsoeslave_sync_with_master() periodically in order to avoid watchdog timeout. It is the safety application that shall ensure it is periodically executed and monitor that it is done, failure to do so shall be handled by the safety application	
#20	The size of the *inputs* buffer passed to fsoeslave_sync_with_master() shall be as configured in fsoeslave_init()	

Ref	Description	Action
#21	The size of the *outputs* buffer passed to fsoeslave_sync_with_master() shall be as configured in fsoeslave_init()	
#22	If fsoeslave_update_sra_crc() is called for the first time, the in-argument *crc* shall first be set to zero	
#23	fsoeslave_get_slave_session_id() shall not be called if state machine is in Reset state	
#24	fsoeslave_get_master_session_id() shall not be called if state machine is in Reset or Session state	

2.2 Implementation 'SHOULD'

Ref	Description	Action
#1	The callback function fsoeapp_recv() should not wait for frame to be received	
#2	The callback function fsoeapp_generate_session_id() should not generate the same Session ID after each start-up	
#3	Application should call fsoeslave_set_reset_request_flag() if it detects an application-level error while communicating with master	
#4	The FSoE Conformance Test should pass	
#5	If the stack may run on a 1 or 2 controller hardware solution, FSoE Application layer should be implemented to support either hardware setup	
#6	If the platform is missing **stdbool.h**, a local header file should be added: stdbool.h #define true 1 #define false 0 #define bool _Bool	
#7	If the platform is missing **stdint.h**, a local header file should be added	

2.3 Implementation 'MAY'

Ref	Description	Action
#1	If the platform is missing **inttypes.h**, a local header file may be added	

2.4 Implementation 'CAN'

Ref	Description	Action
#1	**FSOE_PROCESS_DATA_MAX_SIZE:** can be adjusted to decrease RAM usage for objects allocated by the safety application.	
#2	**FSOE_APPLICATION_PARAMETERS_MAX_SIZE:** can be adjusted to decrease RAM usage for objects allocated by the safety application.	
#3	**FSOE_LOG_ENABLE:** can be used to enable run-time logging provided by the stack	
#4	**FSOE_LOG_ENABLE:** the user implements the logging function	
#5	**FSOE_LOG_LEVEL:** sets the log level, given logging is enabled	
#6	**FSOE_LOG_MASTER:** enabled logging for specific object	
#7	**FSOE_LOG_SLAVE:** enabled logging for specific object	
#8	**FSOE_LOG_FRAME:** enabled logging for specific object	
#9	**FSOE_LOG_CHANNEL:** enabled logging for specific object	

2.5 Implementation 'Limitations'

Ref	Description	Action
#1	**MAX safetydata size:** Maximum data size is 126 bytes SafeOutputs and 126 bytes SafeInputs per FSoE connection. The limitation is implementation specific due to using the CRC algorithm provided by ETG.5100 "Safety Over EtherCAT" specification, it is possible to extend.	
#2	**Dynamic SafeData:** Dynamic mapping of the SafeData is not supported	

3 Checklist for implementing platform abstraction layer

3.1 Implementation 'SHALL'

Ref	Description	Action
#1	The macros FSOEPORT_ASSERT, FSOEPORT_IS_BIG_ENDIAN and FSOEPO↩ RT_FORMAT shall be implemented in a header file fsoeport_macros.h	
#2	**FSOEPORT_IS_BIG_ENDIAN** shall be defined to match the target platform running the FSoE stack	
#3	**fsoeport_current_time_us** shall be implemented using a local monotonically increasing timer (except for wrap-around)	

3.2 Implementation 'SHOULD'

Ref	Description	Action
#1	**fsoeport_memcpy** should use memcpy() or an equivalent function	
#2	**fsoeport_memcmp** should use memcmp() or an equivalent function	
#3	**fsoeport_memset** should use memset() or an equivalent function	
#4	**fsoeport_current_time_us** should at least support millisecond resolution	
#5	If CMake is not used, the header file options.h should be created manually and placed in src directory.	
#6	If CMake is not used, the header files fsoeoptions.h and fsoeexport.h should be created manually and placed in include directory.	
#7	**FSOEPORT_ASSERT**, if triggered, should perform proper (platform specific) error handling and halt or exit execution of FSoE stack	
#8	The unit-tests and integration tests should pass	

3.3 Implementation 'MAY'

Ref	Description	Action
#1	**fsoeport_current_time_us** may support microsecond resolution	
#2	**fsoeport_log** may be implemented if logging support is desired	
#3	The C99 header file stdint.h may be added if not included by default	
#4	The C99 header file stdbool.h may be added if not included by default	
#5	The C99 header file inttypes.h may be added if not included by default	

Ref	Description	Action
#6	**FSOEPORT_FORMAT** may be defined to enable compiler to emit warnings if types of supplied arguments to fsoeport_log() do not match supplied format string	

3.4 Implementation 'CAN'

Ref	Description	Action
#1	**FSOE_PROCESS_DATA_MAX_SIZE** can be adjusted to decrease RAM usage for objects allocated by the safety application.	
#2	**FSOE_APPLICATION_PARAMETERS_MAX_SIZE** can be adjusted to decrease RAM usage for objects allocated by the safety application.	
#3	**FSOE_LOG_ENABLE** can be used to enable run-time logging provided by the stack	
#4	**FSOE_LOG_ENABLE** the user implements the logging function	
#5	**FSOE_LOG_LEVEL** sets the log level, given logging is enabled	
#6	**FSOE_LOG_MASTER** enabled logging for specific object	
#7	**FSOE_LOG_SLAVE** enabled logging for specific object	
#8	**FSOE_LOG_FRAME** enabled logging for specific object	
#9	**FSOE_LOG_CHANNEL** enabled logging for specific object	

3.5 Implementation 'Limitations'

Ref	Description	Action
#1	**MAX safetydata size:** Maximum data size is 126 bytes SafeOutputs and 126 bytes SafeInputs per FSoE connection. The limitation is implementation specific due to using the CRC algorithm provided by ETG.5100 "Safety Over EtherCAT" specification, it is possible to extend.	

rt-labs AB
Värmlandsgatan 2
SE-413 28 Göteborg, Sweden

www.rt-labs.com
support@rt-labs.com