FailSafe over EtherCAT User Manual

rt:labs

FailSafe over EtherCAT User Manual

rt-labs AB

http://www.rt-labs.com

Revision: 549:550M

DISCLAIMER

RT-LABS AB MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, rt-labs AB, reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of rt-labs AB, to notify any person of such changes.

September 15, 2019

Contents

Chapter 1

FailSafe over EtherCAT Introduction

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, it is called FSoE Connection. In the FSoE Connection, each device only returns its own new message once a new message has been received from the partner device. The complete transfer path between FSoE Master and FSoE Slave is monitored by a separate watchdog timer on both devices, in each FSoE Cycle. The FSoE Master can handle more than one FSoE Connection to support several FSoE Slaves.

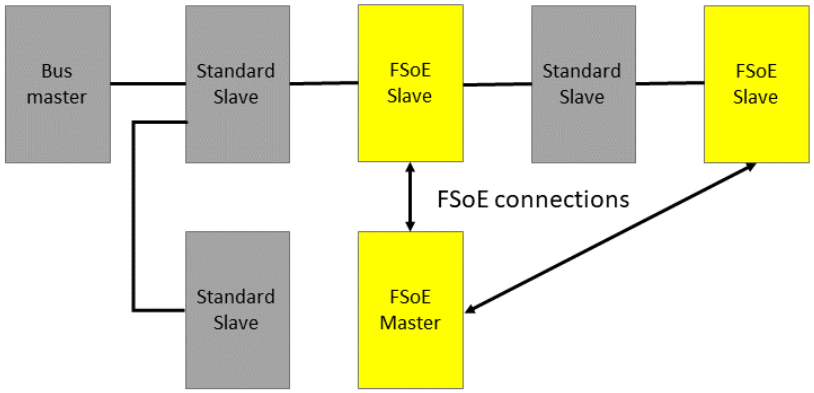


Figure 1.1: FSoE Connection

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 safely detecting misrouted messages via a unique address relationship.
- watchdog monitoring for safely detecting delays not allowed on the communication path.

- cyclic redundancy checking for data integrity for detecting message corruption from source to sink.

State transitions are initiated by the FSoE Master and acknowledged by the FSoE Slave. The FSoE state machine also involves exchange and checking of information for the communication relation.

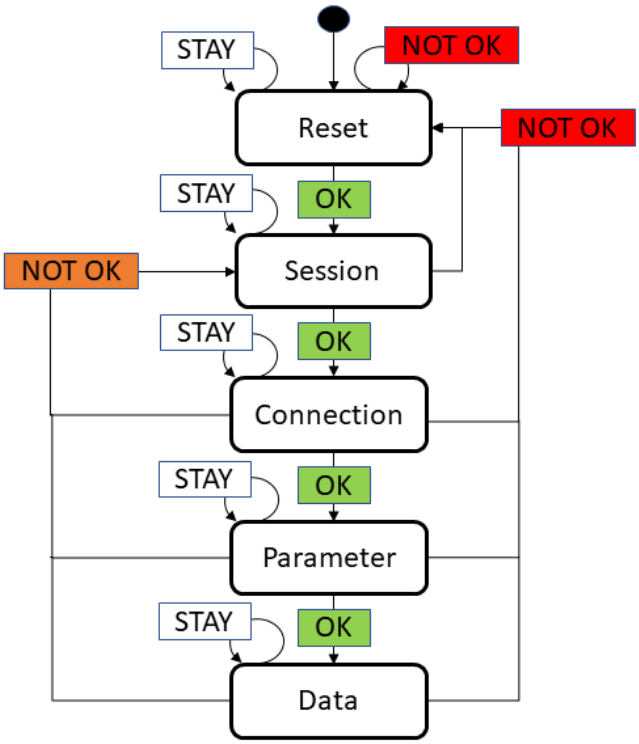


Figure 1.2: FSoE State Machine

1 FSoE stacks Requirements and Dependencies

The FSoE Master and Slave stacks are written i C and provide the safety user application with means to establish an FSoE Connection to an FSoE Device, enabling exchange of safety process data.

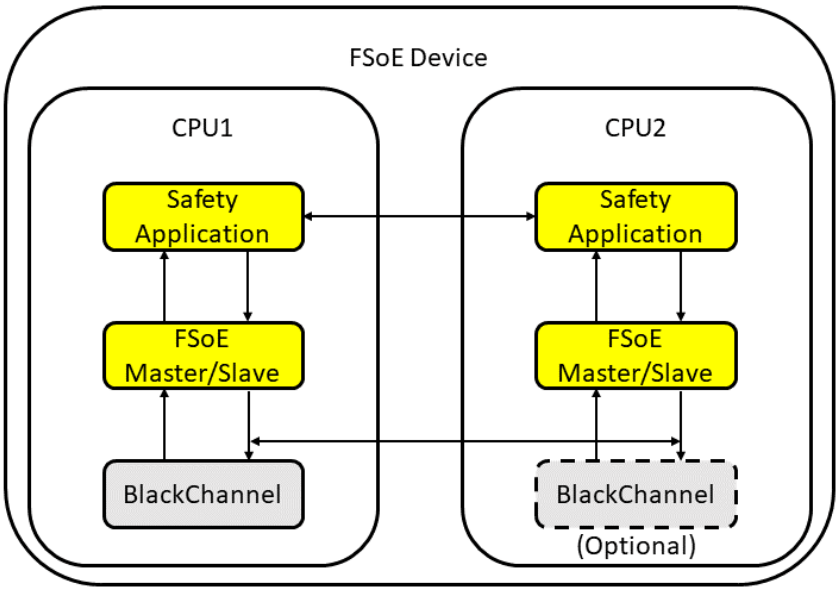


Figure 1.3: FSoE on typical architecture

1.1 Source deliverables

The FSoE stacks are delivered as source code for integration into target safety platform. It is mandatory to port the unittests and run on target platform.

```
<fsoe stack source>
 |
 | - cmake - CMake project files for various targets.
 | - include - headers for FSoE stack public API, no other headers should be included.
 | - lint - PC lint configuration files.
 | - src - the fsoe stack source files including existing osal ports.
 | - test - FSoE unit tests and integration tests implemented using Google Test framework, should be ported
       and run on platform target.
```

1.2 Build instructions

The FSoE stacks 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 fsoe stack source README.md.

It is also possible to build the source by just adding the source to an existing platform project.

1.3 Operating Systems or bare-metal

The FSoE stacks provide a platform abstraction layer enabling the use of any OS or bare-metal application to compile and run the stack. The abstraction layer must be provided for the stack to function properly. Depending on safety architecture used, some of the functions may require extra functionality exchanging information with a second FSoE stack instance running on the same or different hardware. List of functions

- os_log write a log message with printf-like format (optional).
- os_malloc, allocate memory, like malloc.

- os_free, deallocate memory, like free().
- os_get_current_time_us, get current microsecond time.
- fsoeapp_generate_session_id, get a random 16-bit number.
- fsoeapp_send, send a complete FSoE PDU frame.
- fsoeapp_recv, try to receive a complete FSoE PDU frame.

FSoE Slave stack only

- fsoeapp_verify_parameters, verify received parameters.

1.4 Scope

This FSoE stack user manual describe how to configure, create and use an FSoE master to slave connection or FSoE slave to master connection in accordance to ETG.5100 "Safety Over EtherCAT" specification using the belonging FSoE master/slave stack implementation. The user manual doesn't include instructions or guidelines howto implement functional safety software or hardware sufficient to qualify it as an FSoE device, instead follow the safety functional requirements described, See ETG ETG.5100 ch 6.1 "Safety functional requirements".

1.5 Limitations

The FSoE stacks only support a maximum of 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

The FSoE master stack provide support to create and use an FSoE master to slave connection, it is not an complete FSoE master implementation, e.g. the FSoE master to slave connections are not aware of each other to be able to verify uniqueness of connection ids, slave addresses and similar data in an FSoE network according to ETG.5100. That functionallity should be provided by the FSoE master implenetation using the FSoE master stack.

2 Abbreviations

- SCL - safety communication layer eg, The FSoE stack is a SCL.
- PDU - protocol data unit
- SPDU - safety PDU

3 References

1. IEC 61784-3 Industrial communication networks - Profiles - Part 3: Functional Safety fieldbuses
2. ETG.5100 - Safety Over EtherCAT specification

Chapter 2

FSoE Master

1 Overview

1.1 FSoE Master Overview

An FSoE master handles the connection with a single FSoE slave.

After power-on, master will try to establish a connection with its slave. Once established, it will periodically send outputs to the slave. The slave will respond by sending back its inputs.

Inputs and outputs may contain valid process data or they may contain fail-safe data (all zeroes). By default, they contain fail-safe data. They will only contain valid process data if sender (master for outputs, slave for inputs) determines that everything is OK. The sender may send valid process data while receiving fail-safe data or vice verca. Inputs and outputs have fixed size, but they need not be the same size.

A user of the master API will have to explicitly enable it in order for valid process data to be sent. Communication errors will cause the connection to be reset. Master will then disable the process data outputs and try to re-establish connection with its slave. If successful, it restarts sending outputs as fail-safe data. A user of the master API may then re-enable process data outputs.

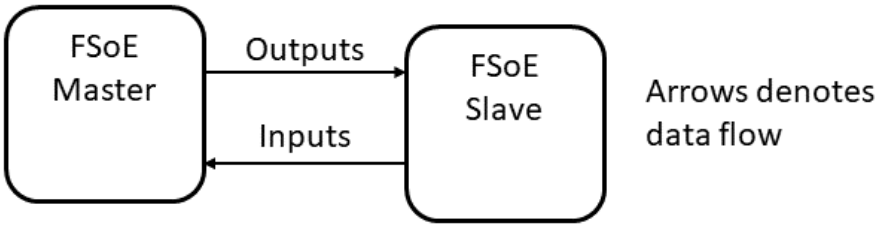


Figure 2.1: FSoE data flow

1.2 Black Channel Communication

At a lower level, the master communicates with the slave through a "black channel". Master does not know how the black channel is implemented, it just knows how to access it - by calling fsoeapp_send() and fsoeapp_recv(). The application implementer needs to implement these two functions.

The arrows in picture below denotes direct function calls:

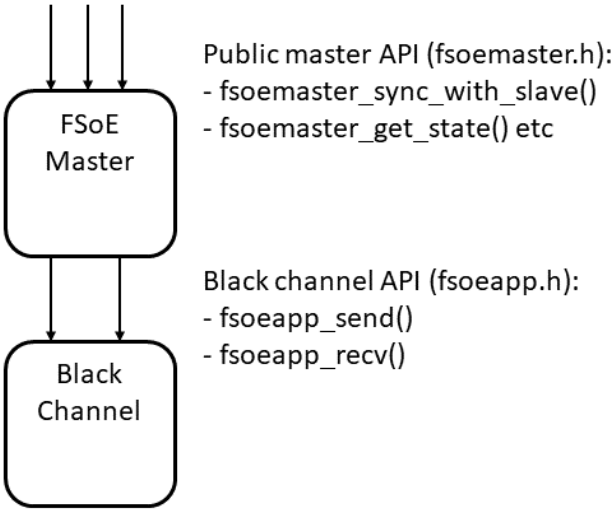


Figure 2.2: FSoE stack layers

1.3 FSoE connection state

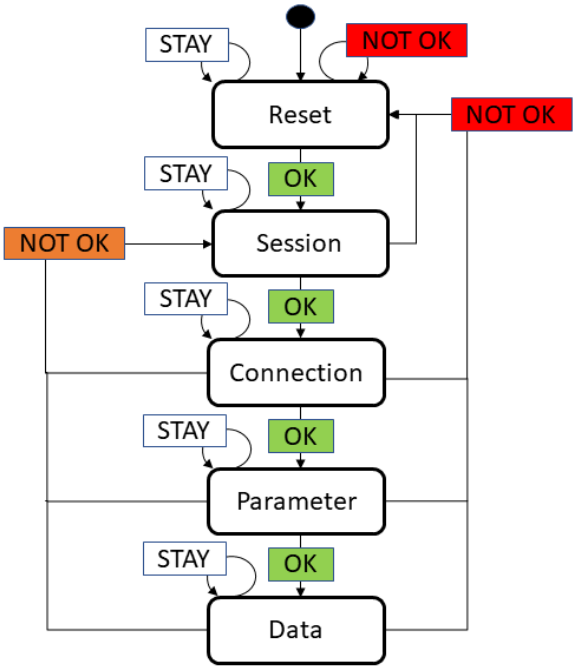


Figure 2.3: FSoE Connection

After power-on, master is in Reset state. In Reset state, master is not connected with any slave. Interchange of process data with a slave only takes place when master is in Data state. Before Data state is entered, master first has to configure the slave by sending it configuration data. This takes place in the intermediate states Session, Connection and Parameter.

Master transitions to new states once configuration data sent to slave has been ACKed by slave. It also sets the slave's state by means of sending a corresponding frame. For example, master sending a Connection frame will cause slave to enter Connection state, assuming that state transition is allowed.

The connection state describes what kind of frame master is waiting for to receive from slave:

Reset: The master will wait for a Reset frame from slave.

Session: The master will wait for a Session frame from slave.

Connection: The master will wait for a Connection frame from slave.

Parameter: The master will wait for a Parameter frame from slave.

Data: The master will wait for either a ProcessData frame or a FailSafeData frame from slave.

2 FSoE Master Stack

2.1 FSoE Master Stack Types

Public types used by the FSoE stack

2.1.1 Communication status

Status after synchronisation with slave See fsoemaster_sync_with_slave()

```
typedef struct fsoemaster_status
{
   bool is_process_data_received;/*  true:
                                  *      Valid process data was received in
                                  *      last FSoE cycle. The process data is
                                  *      stored in \a inputs buffer.
                                  *   false:
                                  *      No process data was received in
                                  *      last FSoE cycle. The \a inputs buffer
                                  *      contains only zeroes.
                                  *      This will be returned if master has
                                  *      detected an error, if connection with
                                  *      slave is not established or if fail-safe
                                  *      data was received.
                                  */
} fsoemaster_status_t;
```

2.1.2 FSoE connection state

```
typedef enum fsoemaster_state
{
   FSOEMASTER_STATE_RESET,       /*< Connection is reset */
   FSOEMASTER_STATE_SESSION,     /*< The session ID is 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;
```

2.1.3 Configuration of FSoE master

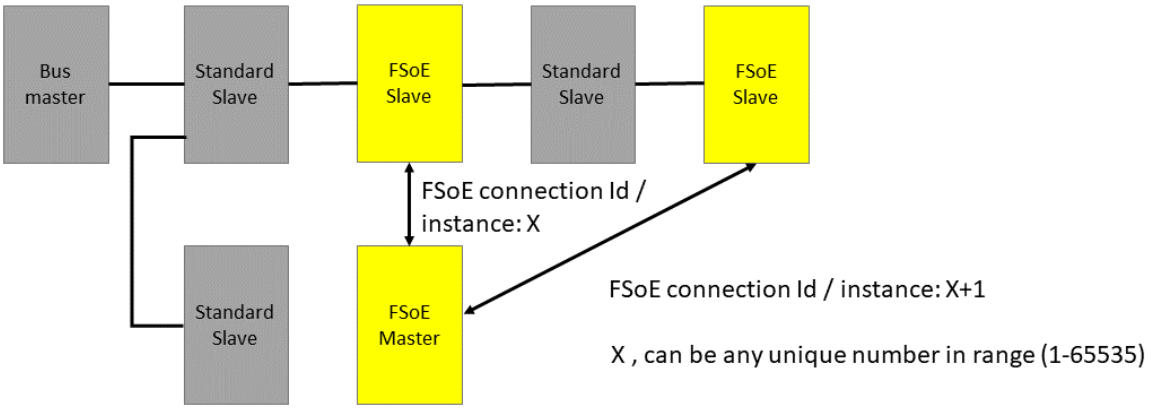
```
typedef struct fsoemaster_cfg
{
   /*
    * Slave Address

    * 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

     * 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 126.

     * 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 126.

    * 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;
```

2.1.4 FSoE master instance

An FSoE master instance handles the connection with a single slave. Multiple FSoE master instances are supported, where each one has their own Connection ID. From an FSoE network perspective we have an FSoE master, the FSoE master create and use a single FSoE connection per FSoE slave. From the FSoE master stack perspective every FSoE connection is created and cyclically run using a FSoE master instance per FSoE connection.




```
FSoE network
 app_init()
    for (i = 0, i < NUMBER_OF_FSOE_CONNECTIONS, i++)
      fsoemaster_create (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 2.4: FSoE instance and connection

```
typedef struct fsoemaster fsoemaster_t;
```

2.2 FSoE Master Stack API

2.2.1 fsoemaster_create - Create FSoE master instance

This will allocate memory used by the instance, including itself. The instance will then be configured according to supplied configuration. If memory allocation fails or fields in the configuration are invalid, an assertion error will be triggered and program execution will halt.

Parameters

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

FSoE master instance. Can't be NULL

```
fsoemaster_t * fsoemaster_create (
   const fsoemaster_cfg_t * cfg,
   void * app_ref);

void main
{
   /*
    * Buffer to store inputs received from slave
    * Note that examples in ETG.5100 uses 4 byte inputs
    */
   uint8_t inputs[4];
   /*
    * Outputs to be sent to slave
    * Note that examples in ETG.5100 uses 4 byte outputs
    */
   const uint8_t outputs[4] = { 4, 5, 6, 7 };
   /*
    * Application parameters to be sent to slave
    * Note that examples in ETG.5100 uses 2 byte application parameters
    */
   const uint8_t application_parameters[2] = { 8, 9 };

   /* Configuration of master instance */
   const fsoemaster_cfg_t cfg =
   {
      0x0304,                          // slave_address
      0x0102,                          // connection_id
      0x0506,                          // watchdog_timeout_ms
      application_parameters,          // application_parameters
      sizeof (application_parameters), // application_parameters_size
      sizeof (outputs),                // outputs_size
      sizeof (inputs),             // inputs_size (excluding overflow detection byte)
   };

   master_ref_t my_master;

   /* Create a master instance, pass local master ref as parameter and save the
    * ref to the fsoe master
    */
   my_master.fsoe_master = fsoemaster_create (&cfg, &my_master);

   if(my_master.fsoe_master == NULL)
   {
      return ERROR;
   }
```

2.2.2 fsoemaster_destroy - Destroy FSoE master instance

All allocated memory will be freed. The instance may no longer be used after calling this function.

Parameters

in,out	*master*	FSoE master instance

```
void fsoemaster_destroy (fsoemaster_t * master);

...
   /* Create a master instance */
   my_master.fsoe_master = fsoemaster_create (&cfgm, &my_master);
...
   for(;;)
   {
      if(stop)
         break;
   }
   fsoemaster_destroy(my_master.fsoe_master);
```

2.2.3 fsoemaster_sync_with_slave - Run state machine

Run FSoE master state machine. 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, master 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.

Parameters

in,out	*master*	FSoE master instance
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 *status*.
out	*status*	Status of FSoE connection..

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

void app_run(void)
   ...
   /* Fetch data to be sent */
   safeOutput1.SafeOutput1 = gpio_get(GPIO_BUTTON2);
   /* Run stack to decode data */
   fsoemaster_sync_with_slave(fsoe_master,
         &safeOutput1,
         &safeInput1,
         &safetdata_status);
   if(safetdata_status.is_process_data_received)
      /* Use data received */
      gpio_set(GPIO_LED2, safeInput1.SafeInput1);
```

2.2.4 fsoemaster_enable_sending_process_data - Enable FSoE sending process data

Enable master to send valid process data to slave. 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).

Parameters

in,out	*master*	FSoE master instance

```
void fsoemaster_enable_sending_process_data (fsoemaster_t * master);

void some_function(void)
...
   /* We're ready to send process data */
   if(local_application_ready && fsoe_state_ok)
   {
      fsoemaster_enable_sending_process_data(fsoe_master);
   }
```

2.2.5 fsoemaster_disable_sending_process_data - Disallow FSoE sending process data

Disallow master from sending normal process data to slave. This will clear a flag indicating that everything is OK from the perspective of the application. Master will only send fail safe data (zeroes) 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).

Parameters

in,out	*master*	FSoE master instance

```
void fsoemaster_disable_sending_process_data (fsoemaster_t * master);

void some_function(void)
...
   /* Application want to output failsafe data? */
   if(local_application_goto_failsafe)
   {
      fsoemaster_disable_sending_process_data(fsoe_master);
   }
```

2.2.6 fsoemaster_is_sending_process_data_enabled - Check if sending process enabled

Check if FSoE master is allowed to send normal process data to slave. This will only check a flag indicating that everything is OK from the perspective of the application. Master 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).

Parameters

in,out	*master*	FSoE master instance

```
bool fsoemaster_is_sending_process_data_enabled (fsoemaster_t * master);

void safety_app_run(void)
{
...
```

```
/* The application always allows processdata when we're in state DATA */
if(fsoemaster_get_state(fsoe_master) == FSOESLAVE_STATE_DATA &&
   fsoemaster_is_sending_process_data_enabled(fsoe_master) == FALSE)
{
   fsoemaster_enable_sending_process_data(fsoe_master);
}
```

2.2.7 fsoemaster_get_state - Get current 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".

Parameters

in	*master*	FSoE master instance

Returns

Current state of the FSoE master

```
fsoemaster_state_t fsoemaster_get_state (fsoemaster_t * master);

void safety_app_run(void)
{
...
   /* The application always allows processdata when we're in state DATA */
   if(fsoemaster_get_state(fsoe_master) == FSOESLAVE_STATE_DATA &&
      fsoemaster_is_process_data_outputs_enabled(fsoe_master) == FALSE)
   {
      fsoemaster_enable_process_data_outputs(fsoe_master);
   }
```

2.2.8 fsoemaster_reset_connection - Reset connection

Reset connection with slave. Master will send the Reset command to slave and then enter the Reset state. See ETG.5100 ch. 8.4.1.2 (Reset Connection).

Parameters

in,out	*master*	FSoE master instance

```
void fsoemaster_reset_connection (fsoemaster_t * master);

void safety_app_run(void)
{
...
   /* The application detects an error and reset the connection */
   if(local_application_error)
   {
      fsoemaster_reset_connection(fsoe_master);
   }
```

2.2.9 fsoemaster_time_until_timeout_ms - Get watchdog timer

Get time remining until watchdog timer timeouts, in milliseconds. For unittest purpose

Parameters

in	*master*	FSoE master instance

Returns

Time remaining in milliseconds. If watchdog timer is not started, UINT32_MAX is returned.

```
void fsoemaster_time_until_timeout_ms (fsoemaster_t * master);

void safety_app_run(void)
{
...
   /* For unittest purpose */
   uint32_t time_to_verify = fsoemaster_time_until_timeout_ms(fsoe_master);
   if(UINT32_MAX == time_to_verify)
      not_yet_started = TRUE;
```

3 FSoE Master Stack Porting API (Black Channel)

The FSoE master stack is designed to run on any bare-metal or OS given that the platform can support the following functionality. The FSoE master stack also support implementing all example Models in iec61784-3, Annex A(informative) - Example functional safety communication models. However, this porting guide will/can not give details on functions named cross_check_xxx since selected model would require implementation specific knowledge, instead the actions needed will be described in writing.

3.1 Stack Mandatory functions

3.1.1 os_malloc - OSAL Allocate memory

Allocate memory, the FSoE stack only allocate memory at startup when calling fsoemaster_create, if fsoemaster_create return without error no more memory is allocated by the FSoE stack.

Parameters

in	*size*	Size of memory block to allocate, in bytes.

Returns

Pointer to allocated memory, or NULL if memory could not be allocated.

Generic example

```
void * os_malloc (size_t size)
{
   return malloc (size);
}
```

3.1.2 os_free - OSAL Deallocate memory

Free memory, the FSoE stack only allocate memory at startup, not during runtime. Hence, if startup of the stack is done successfully it only use "statically" allocated memory. Free make it possible to shutdown

the FSoE stack and free that memory via fsoemaster_destroy which use os_free to free the allocated memory.

Generic example

```
void os_free (void * p)
{
   free (p);
}
```

3.1.3 os_get_current_time_us - OSAL 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 if the watchdog timer has expired or not. It may also be used for logging purposes.

Returns

Current time, in microseconds

Generic example

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

3.1.4 fsoeapp_generate_session_id - OSAL Get session id

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 synchronize the random number used for session ID to be able to run, since all subsequent frames will "inherit" from this random number due the inclusion of received CRC_0 in sent frames. See ETG.5100 ch. 8.1.3.7. To achieve this, one should make the two running FSoE instances exchange random values when necessary, the session id is generated at RESET_OK, and to be able to make a transition to SESSION a new session id must be generated and shared, either when needed or if it already have been done.

Parameters

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

Returns

A generated Session ID

Generic 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
}
```

3.1.5 fsoeapp_send - OSAL Send a Safety PDU frame

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_reset_connection() or fsoeslave_reset_connection().

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.

Parameters

in,out	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_create or fsoeslave_create(). 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

```
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_reset_connection(fsoe_master);
   }
   else
   {
      memcpy(FSOE_Slave_Frame_Elements, buffer, size);
   }
#endif
}
```

3.1.6 fsoeapp_recv - OSAL Receive Safety PDU

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 a) return without waiting for any incoming frame or b) 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.

Parameters

`in,out`	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_create or fsoeslave_create(). 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 byte 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 and returning *size*.

EtherCAT 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;
}
```

3.2 Optional functions

3.2.1 os_log - OSAL Write a log message

Write a log message with printf-like format. Optional function for development purpose.

Parameters

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

Generic example

```
void os_log (uint8_t type, const char * fmt, ...)
{
   va_list list;
   char message[100];
   const char * level;
   int written;

   switch (LOG_LEVEL_GET (type))
   {
   case LOG_LEVEL_DEBUG:   level = "DEBUG"; break;
   case LOG_LEVEL_INFO:    level = "INFO "; break;
   case LOG_LEVEL_WARNING: level = "WARN "; break;
   case LOG_LEVEL_ERROR:   level = "ERROR"; break;
   default:                level = "INVAL"; break;
   }
   written = snprintf (
         message,
         sizeof (message),
         "%10"PRIu32" [%s] ",
         tick_get (),
         level);

   if (written > 0 && (size_t)written < sizeof (message))
   {
      va_start (list, fmt);
      vsnprintf (&message[written], sizeof (message) - written, fmt, list);
      va_end (list);

      printf ("%s", message);
   }
}
```

4 Release Notes

4.1 Release 1.0.0

This is the first official release of the software.

Chapter 3

FSoE Slave

1 Overview

1.1 FSoE Slave Overview

A FSoE slave handles the connection with a single FSoE master.

After power-on, slave will listen for connection requests from a master. Once established, slave will wait for outputs from master. When received, it will respond by sending back its inputs to master. Inputs and outputs may contain valid process data or they may contain fail-safe data (all zeroes). By default, they contain fail-safe data. They will only contain valid process data if sender (slave for inputs, master for outputs) determines that everything is OK. The sender may send valid process data while receiving fail-safe data or vice verca. Inputs and outputs have fixed size, but they need not be the same size.

A user of the slave API will have to explicitly enable it in order for valid process data to be sent. Communication errors will cause the connection to be reset. Slave will then disable the process data inputs and start listening for new connection requests from a master. If successful, it restarts sending inputs as fail-safe data. A user of the slave API may then re-enable process data inputs.

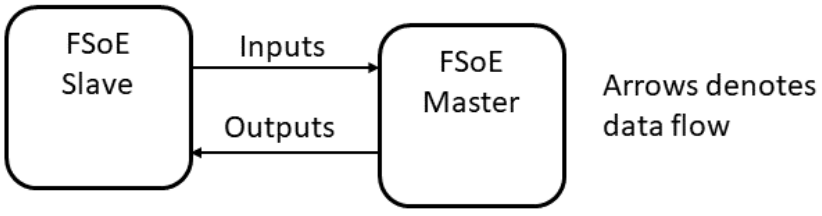


Figure 3.1: FSoE data flow

1.2 Black Channel Communication

At a lower level, the slave communicates with the master through a "black channel". Slave does not know how the black channel is implemented, it just knows how to access it - by calling fsoeapp_send() and fsoeapp_recv(). The application implementer needs to implement these two functions.

The arrows in picture below denotes direct function calls:

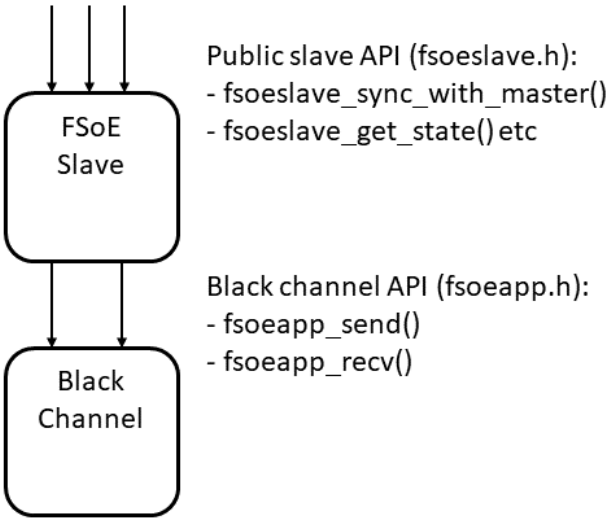


Figure 3.2: FSoE stack layers

1.3 FSoE connection state

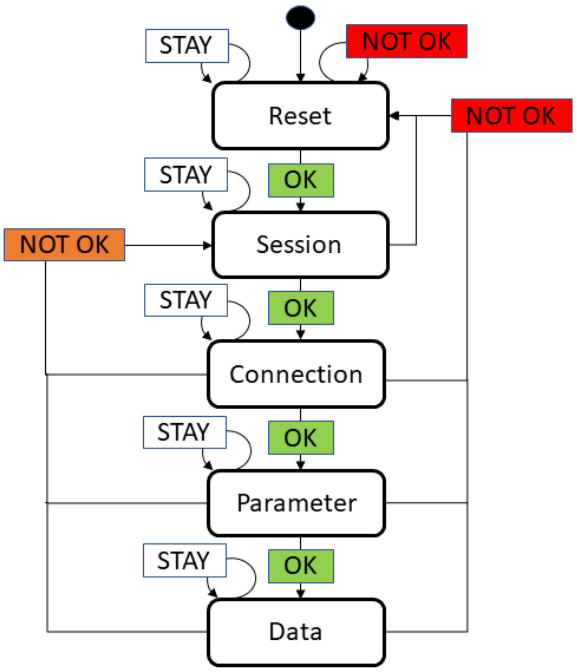


Figure 3.3: FSoE Connection

After power-on, slave is in Reset state. In Reset state, slave is not associated with any master. Interchange of process data with a master only takes place when slave is in Data state. Before Data state is entered, a master first has to configure the slave by sending it configuration data. This takes place in the intermediate states Session, Connection and Parameter.

With the exception of transitions to the Reset state, slave does not change state on its own. Instead, it is the master which orders the slave to enter a new state by means of sending a corresponding frame. For

example, master sending a Connection frame will cause slave to enter Connection state, assuming that state transition is allowed. Slave will enter Reset state on its own if it detects an error.

Reset: Slave is not associated with any master and will listen for incoming requests from masters.

Session: Slave has been associated with a master but has not yet been configured. Slave will remain associated with this master until connection is reset.

Connection: Master is identifying itself using its unique address, the Connection ID. Master will also send the Slave Address, which slave will check.

Parameter: Slave is being configured for data transfer by master. In particular, the watchdog timeout is received. Additional application- specific parameters may also be received.

Data: Connection with master is fully established. The watchdog timer is enabled. Process data or fail-safe data are sent and received periodically at a pace decided by master. Slave will continue to check for errors and reset connection if any error is detected.

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

2 FSoE Slave Stack

2.1 FSoE Slave Stack Types

Public types used by the FSoE stack

2.1.1 Communication status

Status after synchronisation with master See fsoeslave_sync_with_master()

```
typedef struct fsoeslave_status
{
   bool is_process_data_received;/*  true:
                                  *      Valid process data was received in
                                  *      last FSoE cycle. The process data is
                                  *      stored in \a outputs buffer.
                                  *   false:
                                  *      No process data was received in
                                  *      last FSoE cycle. The \a outputs buffer
                                  *      contains only zeroes.
                                  *      This will be returned if slave has
                                  *      detected an error, if connection with
                                  *      master is not established or if
                                  *      fail-safe data was received.
                                  */
} fsoeslave_status_t;
```

2.1.2 FSoE connection state

```
typedef enum fsoeslave_state
{
   FSOESLAVE_STATE_RESET,       /* Connection is reset */
   FSOESLAVE_STATE_SESSION,     /* The session ID is 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;
```

2.1.3 Configuration of FSoE slave

```
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 by 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 - 65535.

    * 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 126.

    * 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 126.

    * 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;
```

2.1.4 FSoE slave instance

An FSoE slave instance handles the connection with a single master. Multiple slaves are supported, where each one has their own Slave Address.

```
typedef struct fsoeslave fsoeslave_t;
```

2.2 FSoE Slave Stack API

2.2.1 fsoeslave_create - Create FSoE slave instance

This will allocate memory used by the instance, including itself. The instance will then be configured according to supplied configuration. If memory allocation fails or fields in the configuration are invalid, an assertion error will be triggered and program execution will halt.

Parameters

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

FSoE slave instance. Can't be NULL

```
fsoeslave_t * fsoeslave_create (const fsoeslave_cfg_t * cfg);

void main
   /* Configuration of slave instance */
   const fsoeslave_cfg_t cfg =
   {
      Mb.Safe_Address.FSoE_Address,     /* slave_address */
      4,                                /* application_parameters_size */
      sizeof (safety_in_t),             /* inputs_size */
      sizeof (safety_out_t),            /* outputs_size */
   };

   slave_ref_t my_slave;

   /* Create a slave instance, pass local slave ref as parameter and save the
    * ref to the fsoe slave
    */
   my_slave.fsoe_slave = fsoeslave_create (&cfg, &my_slave);

   if(my_slave.fsoe_slave == NULL)
   {
      return ERROR;
   }
```

2.2.2 fsoeslave_destroy - Destroy FSoE slave instance

All allocated memory will be freed. The instance may no longer be used after calling this function.

Parameters

in,out	*slave*	FSoE slave instance

```
void fsoeslave_destroy (fsoeslave_t * slave);

...
   /* Create a slave instance */
   my_slave.fsoe_slave = fsoeslave_create (&cfg, &my_slave);
...
   for(;;)
   {
      if(stop)
         break;
```

```
    }
    fsoeslave_destroy(my_slave.fsoe_slave);
```

2.2.3 fsoeslave_sync_with_master - Run state machine

Run FSoE slave state machine. 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, slave 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.

Parameters

`in,out`	*slave*	FSoE slave instance
`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 returned *status*.
`out`	*status*	Status of FSoE connection.

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

void app_run(void)
   ...
   /* Fetch data to be sent */
   safeInput1.SafeInput1 = gpio_get(GPIO_BUTTON2);
   /* Run stack to decode data */
   fsoeslave_sync_with_master(fsoe_slave,
         &safeInput1,
         &safeOutput1,
         &safetdata_status);
   if(safetdata_status.is_process_data_received)
      /* Use data received */
      gpio_set(GPIO_LED2, safeOutput1.SafeOutput1);
```

2.2.4 fsoeslave_enable_sending_process_data - Enable FSoE sending process data

Enable FSoE slave to send valid process data to master. 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).

Parameters

`in,out`	*slave*	FSoE slave instance

```
void fsoeslave_enable_sending_process_data (fsoeslave_t * slave);

void some_function(void)
...
   /* We're ready to send process data */
   if(local_application_ready && fsoe_state_ok)
   {
```

```
      fsoeslave_enable_sending_process_data(fsoe_slave);
   }
```

2.2.5 fsoeslave_disable_sending_process_data - Disallow FSoE sending process data

Disallow FSoE slave from sending normal process data to master This will clear a flag indicating that everything is OK from the perspective of the application. Slave will only send fail safe data (zeroes) 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).

Parameters

in,out	*slave*	FSoE slave instance

```
void fsoeslave_disable_sending_process_data (fsoeslave_t * slave);

void some_function(void)
...
   /* Application want to output failsafe data? */
   if(local_application_goto_failsafe)
   {
      fsoeslave_disable_sending_process_data(fsoe_slave);
   }
```

2.2.6 fsoeslave_is_sending_process_data_enabled - Check if sending process data is enabled

Check if FSoE slave is allowed to send normal process data to master. 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).

Parameters

in,out	*slave*	FSoE slave instance

```
bool fsoeslave_is_sending_process_data_enabled (fsoeslave_t * slave);

void safety_app_run(void)
{
...
   /* The application always allows processdata when we're in state DATA */
   if(fsoeslave_get_state(fsoe_slave) == FSOESLAVE_STATE_DATA &&
      fsoeslave_is_sending_process_data_enabled(fsoe_slave) == FALSE)
   {
      fsoeslave_enable_sending_process_data(fsoe_slave);
   }
```

2.2.7 fsoeslave_get_state - Get current 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".

Parameters

in	*slave*	FSoE slave instance

Returns

Current state of the FSoE slave

```
fsoeslave_state_t fsoeslave_get_state (fsoeslave_t * slave);

void safety_app_run(void)
{
...
   /* The application always allows processdata when we're in state DATA */
   if(fsoeslave_get_state(fsoe_slave) == FSOESLAVE_STATE_DATA &&
      fsoeslave_is_process_data_inputs_enabled(fsoe_slave) == FALSE)
   {
      fsoeslave_enable_process_data_inputs(fsoe_slave);
   }
```

2.2.8 fsoeslave_reset_connection - Reset connection

Reset connection with master. Slave will send a Reset frame to master and then wait for master to re-establish connection. Fail-safe mode will be entered, where normal process data inputs will not be sent even after connection has been reestablished. Application needs to explicitly re-enable process data inputs in order to leave fail-safe mode. See ETG.5100 ch. 8.5.1.2 (Reset Connection).

Parameters

in,out	*slave*	FSoE slave instance

```
void fsoeslave_reset_connection (fsoeslave_t * slave);

void safety_app_run(void)
{
...
   /* The application detects an error and resets the connection */
   if(local_application_error)
   {
      fsoeslave_reset_connection(fsoe_slave);
   }
```

3 FSoE Slave Stack Porting API (Black Channel)

The FSoE slave stack is designed to run on any bare-metal or OS given that the platform can support the following functionality. The FSoE slave stack also supports implementing all example Models in iec61784-3, Annex A(informative) - Example functional safety communication models. However, this porting guide will/can not give details on functions named cross_check_xxx since selected model would require implementation specific knowledge, instead the actions needed will be described in writing.

3.1 Stack Mandatory functions

3.1.1 os_malloc - OSAL Allocate memory

Allocate memory. The FSoE stack only allocates memory at startup when calling fsoeslave_create, if fsoeslave_create return without error no more memory is allocated by the FSoE stack.

Parameters

in	*size*	Size of memory block to allocate, in bytes.

Returns

Pointer to allocated memory, or NULL if memory could not be allocated.

Generic example

```
void * os_malloc (size_t size)
{
   return malloc (size);
}
```

3.1.2 os_free - OSAL Deallocate memory

Free memory, the FSoE stack only allocate memory at startup, not during runtime. Hence, if startup of the stack is done successfully it only use "statically" allocated memory. Free make it possible to shutdown the FSoE stack and free that memory via fsoeslave_destroy which use os_free to free the allocated memory.

Generic example

```
void os_free (void * p)
{
   free (p);
}
```

3.1.3 os_get_current_time_us - OSAL 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 if the watchdog timer has expired or not. It may also be used for logging purposes.

Returns

Current time, in microseconds

Generic example

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

3.1.4 fsoeapp_generate_session_id - OSAL Get session id

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 synchronize the random number used for session ID to be able to run, since all subsequent frames will "inherit" from this random number due the inclusion of received CRC_0 in sent frames. See ETG.5100 ch. 8.1.3.7. To achieve this, one should make the two running FSoE instances exchange random values when necessary, the session id is generated at

RESET_OK, and to be able to make a transition to SESSION a new session id must be generated and shared, either when needed or if it already have been done.

Parameters

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

Returns

A generated Session ID

Generic 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
}
```

3.1.5 fsoeapp_send - OSAL Send a Safety PDU frame

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_reset_connection() or fsoeslave_reset_connection().

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.

Parameters

in,out	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_create or fsoeslave_create(). 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

```
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_reset_connection(fsoe_slave);
    }
    else
    {
       memcpy(FSOE_Slave_Frame_Elements, buffer, size);
    }
#endif
}
```

3.1.6 fsoeapp_recv - OSAL Receive Safety PDU

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 a) return without waiting for any incoming frame or b) 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.

Parameters

in,out	*app_ref*	Application reference. This pointer was passed to stack in fsoemaster_create or fsoeslave_create(). 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 byte 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 and returning *size*.

EtherCAT 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,
    * length is fixed and set at configuration.
    * cross-checking is done on decoded FSoE data, not here */
   memcpy(buffer, FSOE_Slave_Frame_Elements, size);
   return size;
}
```

3.1.7 fsoeapp_verify_parameters - OSAL Verify received parameters

Verify received parameters. 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_create(). 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 applicaton- specific error code.

EtherCAT 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;
}
```

3.2 Optional functions

3.2.1 os_log - OSAL Write a log message with printf-like format

Optional function for development purpose

Parameters

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

Generic example

```
void os_log (uint8_t type, const char * fmt, ...)
{
   va_list list;
   char message[100];
   const char * level;
   int written;

   switch (LOG_LEVEL_GET (type))
   {
   case LOG_LEVEL_DEBUG:   level = "DEBUG"; break;
   case LOG_LEVEL_INFO:    level = "INFO "; break;
   case LOG_LEVEL_WARNING: level = "WARN "; break;
   case LOG_LEVEL_ERROR:   level = "ERROR"; break;
   default:                level = "INVAL"; break;
   }
   written = snprintf (
         message,
         sizeof (message),
         "%10"PRIu32" [%s] ",
         tick_get (),
         level);

   if (written > 0 && (size_t)written < sizeof (message))
   {
      va_start (list, fmt);
      vsnprintf (&message[written], sizeof (message) - written, fmt, list);
      va_end (list);

      printf ("%s", message);
   }
}
```

4 Release Notes

4.1 Release 1.0.0

This is the first official release of the software.

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

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

rt:labs