Implementation of eProtocol

This document describes the implementation of eProtocol under embOBJ environment, for use on the EMS boards and on the PC104 host.

Approval History

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Version | Author | | Date | Approved | | Date |
| 1.0 | Accame | iCub Unit |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 1.0 | 24 Jul 13 | M. Accame | First edit of the document |
| 1.1 | 19 Dec 13 | M. Accame | Added EOproxy and EOagent plus behaviour of proxied variables. The document is aligned w/ revision 27817 of SVN repository in eBcode/embobj/plus/comm-v2. |
|  |  |  |  |

Table of Contents

[1 The implementation of eProtocol 1](#_Toc375243158)

[2 The communication between the host and the board 2](#_Toc375243159)

[2.1 The object EOtheBOARDtransceiver 3](#_Toc375243160)

[2.1.1 The object EOnvSet 4](#_Toc375243161)

[2.1.2 The object EOtransceiver inside EOtheBOARDtransceiver 7](#_Toc375243162)

[2.2 The object EOhostTransceiver 11](#_Toc375243163)

[2.2.1 The object EOnvSet 11](#_Toc375243164)

[2.2.2 The object EOtransceiver inside EOhostTransceiver 12](#_Toc375243165)

[2.2.3 Example of use of the EOconfirmationManager 14](#_Toc375243166)

[3 Most common operations 15](#_Toc375243167)

[3.1 Operations on the host 15](#_Toc375243168)

[3.1.1 Initialisation 15](#_Toc375243169)

[3.1.2 Configuration of regular ROPs 16](#_Toc375243170)

[3.1.3 Regular use 16](#_Toc375243171)

[3.1.4 Protection from concurrent use 16](#_Toc375243172)

[3.1.5 Send a ROPframe 17](#_Toc375243173)

[3.1.6 Add an occasional ROP to the transceiver 18](#_Toc375243174)

[3.1.7 Manage the reception of a ROP 19](#_Toc375243175)

[3.1.8 How to propagate data received through sig<> ROPs to higher layers 21](#_Toc375243176)

[3.1.9 How to manage a blocking ask<> and say<> 22](#_Toc375243177)

[3.1.10 Have access of a given variable from its ID 23](#_Toc375243178)

[3.1.11 How to build IDs 24](#_Toc375243179)

[3.2 Operations on the board 25](#_Toc375243180)

[3.2.1 Initialisation 25](#_Toc375243181)

[3.2.2 Configuration of regular ROPs 26](#_Toc375243182)

[3.2.3 Have access to a given variable from its ID 26](#_Toc375243183)

[3.2.4 Manage the reception of a ROP set<> 27](#_Toc375243184)

[3.2.5 Build a ROP sig<> 28](#_Toc375243185)

[3.2.6 Manage the reception of a ROP ask<> 29](#_Toc375243186)

[4 Configuration of a board 30](#_Toc375243187)

[4.1 The eOprot\_Bxx.h file 30](#_Toc375243188)

[4.2 The eOprot\_Bxx.c file 32](#_Toc375243189)

[5 Change of an endpoint 33](#_Toc375243190)

[5.1 Adding a tag on an existing entity 33](#_Toc375243191)

[5.2 Adding a new entity 33](#_Toc375243192)

[6 Adding an entire endpoint 35](#_Toc375243193)

The implementation of eProtocol

This document describes the implementation of eProtocol under embOBJ environment for use both on the EMS boards and PC104 host.

In short, the implemented mechanism allows exchanging ROPframes containing ROPs related to a pre-loaded set of network variables. The ROPs are automatically managed by the above mechanism which calls user-defined functions when the RAM of a given network variable is changed.

At first it is described which are the objects which are used to exchange ROPframes and how the ROPframes are decoded and actions related to the contained ROPs are executed. Also, it is shown how the objects can be configured to “know about” a given set of network variables.

It is also shown which methods are available to the user to perform simple operations such as: build a ROP, send / receive a ROPframe, personalise actions on reception of ROPs, have access to the RAM of a given network variable.

Later it is described how it is possible to operate changes in the set of network variables, so that a user can add or remove network variables.

The communication between the host and the board

The ROPframe exchange between the host (PC104) and the board (EMS) is done using two specialised objects, the EOhostTransceiver and the EOtheBOARDtransceiver, which reside respectively on the PC104 and the EMS. The board object is a singleton as there must be only one instance of it, whereas the host object can be instantiated many times.

On the PC104 there are many instances of EOhostTransceiver, each of them paired with the EOtheBOARDtransceiver of a given EMS.

ROPFRAME EXCHANGE

PC104

eOnvset\_DEVcfg\_t nvsB1

EOhostTransceiver

eOnvset\_DEVcfg\_t nvsB2

EOhostTransceiver

BOARD 1

eOnvset\_DEVcfg\_t nvsB1

EOtheBOARDtransceiver

BOARD 2

eOnvset\_DEVcfg\_t nvsB2

EOtheBOARDtransceiver

EOropframe

EOropframe

EOropframe

EOropframe

**Figure 1**: The EOropframe content is exchanged through the network between pairs of transceivers of kind EOhostTransceiver on the PC104 and EOtheBOARDtransceiver on the EMS board. Both transceivers are initialised with the same configuration of network variables.

Both objects contain one EOtransceiver and one EOnvSet. The EOtransceiver is responsible of managing ROPs and ROPframes and it uses the EOnvSet to perform operations on the network variables. The EOnvSet contains knowledge of all the network variables of a given board and also contains the RAM where the ROPs operate. The difference between the EOtheBOARDtransceiver and the EOhostTransceiver is that the former directly owns the variables inside the EOnvSet, whereas the latter uses the EOnvSet to form ROPs which sets values on the board and to cache inside its RAM the values that the board sends.

As an example, when the host wants to set the value of the PID of a joint on a board, the EOhostTransceiver queries its EOnvSet about presence of that variable on the remote board, and then the EOtransceiver builds the ROP and puts it inside a ROPframe, so that the sending socket can retrieve it for transmission.

When the board receives the UDP packet it gives that to the EOtheBOARDtransceiver, where the ROPs are extracted by the EOtransceiver. The ROP is processed with the aid of the EOnvSet. Only if the ID inside the ROP is recognised, then the ROP is processed. In the case of a set<> the data field of the ROP is copied inside the RAM owned by the EOnvSet and a callback is executed.

On the other hand if the host wants to verify the values of the PID of a joint on the associated board, the EOhostTransceiver builds an ask<> ROP and sends it to the board. The board processes the ROP and forms a reply with a say<> ROP which contains data copied from the RAM owned by its EOnvSet. Finally the EOhostTransceiver receives the say<> ROP and copies its data inside the RAM owned by the EOnvSet and executes a callback.

There is also the case in which the value of the PID is logically associated to the board but it is effectively owned by another board in a CAN subnet. In such a case, the variable with value is said to be proxied and the actions called by the ROPs must be propagated to the owner of the variable.

The set<> received by a board shall behave in the same manner as the non-proxied case, but the user must use the callback to send the received value to the owner of the variable over the CAN bus.

The ask<> received by a board shall not generate an automatic ask<> but will instead execute a callback in which the user must send over the CAN bus proper messages to the owner of the variable to request the value back. Upon reception of the value, the user must manually feed the value to the transceiver which shall send the say<> back to the host.

The above mechanism is executed as long as both EOhostTransceiver and the EOtheBOARDtransceiver have instances of EOnvSet configured with the same eOnvset\_DEVcfg\_t data.

## The object EOtheBOARDtransceiver

The object EOtheBOARDtransceiver is a singleton which manages ROP-based communication with a set of variables which are locally owned. As such, it manages incoming ROPs of type ask<> and set<> and sends out ROPs of type say<> and sig<>.

Whenever it receives a set<> ROP it: (a) copies the received data inside the variable identified by the ID, and (b) calls a user-defined update function which alerts the application about the change of value.

When it receives ask<> ROP it: (a) prepares a say<> ROP with the content of the queried variable.

Finally, it can send spontaneous sig<> ROPs to inform the host about the value of a variable.

An interesting feature of the EOtheBOARDtransceiver is that it can be configured (via a ROP obviously!) to emit sig<> ROPs on a regular basis.

STRUCTURE OF EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOtransceiver

**Figure 2**: The EOtheBOARDtransceiver is a singleton on the EMS board. It contains one EOtransceiver and one EOnvSet object. The EOtransceiver is composed of one EOtransmitter and one EOreceiver which processes EOropframe objects. The EOnvSet contains knowledge of all the network variables of the specific board. It is also responsible of allocating the RAM for the variables.

### The object EOnvSet

The EOnvSet is responsible of loading the configuration of a given board which personalises the behaviour of EOtheBOARDtransceiver. The configuration is contained in a variable of type eOnvset\_DEVcfg\_t.

EOnvSet in EOtheBOARDtransceiver

BOARD 1

eOnvset\_DEVcfg\_t nvsB1

EOtheBOARDtransceiver

EOnvSet

HEAP

RAM of netvars

Contains information about every endpoint present on that board, on how many entities are contained (joints, motors, etc), about the size of RAM required, etc.

ROM

Endpoints, num of entities, user-def functions, etc.

Protocol library

EOtransceiver

**Figure 3**: The EOnvSet object is initialised using a constant structure which contains information on the network variables of the board. For instance, it contains information on the managed endpoints and on how many entities are in any endpoint, so that the EOnvSet can allocate the RAM used for the network variables and initialises other data structures.

EOnvSet in EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

eOnvset\_DEVcfg\_t nvsB1

1. Loads every endpoint in EOnvSet with the number of entities specified for that board.
2. Allocates the RAM for the variables in the endpoints.
3. Initializes the protocol library passing to it the relevant RAM and the number of entities of each endpoint of the board.
4. Calls the user-defined init() function of every network variable

EOtransceiver

**Figure 4**: In its initialisation, the EOnvSet performs a series of operations such as: load the endpoints specified in the eOnvset\_DEV\_cfg\_t board configuration, allocate the required RAM, initialise the protocol library by loading into it the number of entities and the RAM for each endpoint, calling the user-defined init() functions for every network variable.

Protocol library used by EOnvSet inside EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

HEAP

Protocol library

ROM

EP

* loads RAM,
* loads number of entities,
* already contains for each variable the data structure used by EOnv

EOtransceiver

**Figure 5**: The protocol library is configured by the EOnvSet for each board and each endpoint the board supports. In particular, for each endpoint of a given board the protocol library is assigned the RAM containing all the variables and pointers to ROM data containing the number of entities. The protocol library already contains internal reference to all the data structure required for each variable. For instance if a board supports the motion-control endpoint with 4 joints, 4 motors and 1 controller, then the initialisation consists of passing a const array with [4, 4, 1], and a pointer to N bytes of RAM.

Main services of protocol library

BOARD 1

EOtheBOARDtransceiver

EOnvSet

eOprotID32\_t ID = eoprot\_ID\_get(EP, ENT, IND, TAG);

eObool\_t res = eoprot\_id\_isvalid(eoprot\_board\_localboard, ID);

void \*var = eoprot\_variable\_ramof\_get(eoprot\_board\_localboard, ID);

HEAP

RAM of netvars

Protocol library

ROM

Endpoints, num of entities, user-def functions, etc.

EOtransceiver

**Figure 6**: After the protocol library is configured by the EOnvSet, it is able to offer services such as: verifying if an ID of a variable is supported by the board, retrieving the RAM and ROM information relative to an ID, convert an ID into (EP, ENT, IND, TAG) and the other way round.

EOnvSet and EOnv in EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

HEAP

RAM of netvars

EOnv

EOnv nv;

eo\_nvset\_NV\_Get(ID, &nv);

- pointer to RAM of variable

- size of variable

- mutex of the variable

- init() function

- update() function

Protocol library

ROM

Endpoints, num of entities, user-def functions, etc.

EOtransceiver

**Figure 7**: Inside the EOtheBOARDtransceiver, the EOnvSet object is used to retrieve a given data structure associated to a network variable: the EOnv. The EOnv contains pointer to the RAM of the network variable but also its size and the mutex to be used when manipulating its data and some associate user-defined functions: init() and update(). The EOnv can be used to set / get values into / from the network variable. The EOtransceiver and associated objects use the EOnv to process the ROP.

EOnvSet in EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

void \*var = eo\_nvset\_RAMofVariable\_Get(ID); // RAM of PID of a given joint

void \*ent = eo\_nvset\_RAMofEntity\_Get(EP, entity, index); // RAM of a given joint

HEAP

RAM of netvars

Protocol library

ROM

Endpoints, num of entities, user-def functions, etc.

EOtransceiver

**Figure 8**: The EOnvSet object also offers methods to retrieve the RAM and the size of a given network variable or of a given entity. These methods offer a more immediate access to data of variables than using a EOnv.

### The object EOtransceiver inside EOtheBOARDtransceiver

The EOtransceiver is responsible of decoding received EOropframe objects, performing actions related to the EOrop object inside them, and encoding EOropframe with replies or spontaneous signalling. It also manages EOrop related to proxied variables. For such it forwards set or get requests on fresh data from the relevant node.

As such it contains: an EOreceiver, an EOagent, an EOtransmitter, an EOproxy, a EOconfirmationManager. It also uses the services of the singletons EOtheParser and EOtheFormer.

EOtransceiver in EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOtransceiver

EOtransceiver

EOtransmitter

EOreceiver

EOconfManager

EOagent

EOproxy

Transmits a ROPframe

Receives a ROPframe

Performs actions on ROP

Manages proxied variables

Manages confirmation requests

**Figure 9**: The EOtransceiver contains the EOreceiver, the EOtransmitter, the EOagent, the EOconfirmationManager, and the EOproxy.

EOtransceiver in EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOtheParser

EOtheFormer

EOropframe

EOropframe

EOtransceiver

**Figure 10**: The EOtransceiver also uses singleton EOtheParser to extract EOrop objects from the received EOropframe and uses singleton EOtheFormer to prepare an EOrop to transmit.

Reception inside EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOrop

1. Finds the EOnv associated to received ID,
2. If EOnv is fully local, then processes it by using the RAM of the netvars.
3. If EOnv is proxied, then processes it by forwarding actions to the terminal node which owns the fresh data.

For every EOrop inside the received EOropframe:

EOropframe

HEAP

RAM of netvars

EOtransceiver

EOtheParser

EOagent

EOreceiver

**Figure 11**: The EOreceiver extracts all EOrop from the input EOropframe with the help of the EOtheParser, and passes them to the EOagent which performs an action. If the associated EOnv is fully local, then the EOagent processes it using only the local RAM of the netvars. If instead the EOnv is proxied, then the EOagent uses data contained in the terminal node.

Reception inside EOtheBOARDtransceiver: processing the EOrop of a fully local EOnv

BOARD 1

EOtheBOARDtransceiver

EOnvSet

1. If ROP is set<>: writes data inside EOnv, calls associated update() function. If requested, prepares a confirmation and passes it to EOtransmitter.
2. If ROP is ask<>: reads data from EOnv, prepares a reply EOrop, and passes it to EOtransmitter.

If EOnv is fully local:

EOtransceiver

EOtransmitter

EOreceiver

EOagent

**Figure 12**: If the EOnv inside the EOrop is fully local, the EOagent performs an action using only local data. In case of a set<>, it writes the data in the RAM of the EOnv and calls a user-defined function update(). In case of ask<>, it reads the RAM of the EOnv and prepares a say<> which is put inside the EOtransmitter.

Reception inside EOtheBOARDtransceiver: processing the EOrop of a proxied EOnv

BOARD 1

EOtheBOARDtransceiver

EOnvSet

1. If ROP is set<>: writes data inside EOnv, calls associated update() function, where the user MUST forward the order to the relevant node. If requested it prepares a confirmation EOrop and passes it to EOtransmitter.
2. If ROP is ask<>: puts details of the ROP inside the EOproxy and calls update() function where the user MUST forward the request to the relevant node. The say<> is built by the EOproxy upon asynchronous reception of the reply. See (\*).

If EOnv is proxied:

EOtransceiver

EOtransmitter

EOreceiver

EOagent

EOproxy

(\*) When a reply from the relevant node arrives, then the user MUST fill the data inside EOproxy. The EOproxy retrieves details of the recently received ask<>, forms a say<> with the fresh data and loads it inside the EOtransmitter.

**Figure 13**: If the EOnv inside the EOrop is proxied, the EOagent propagates the action to the relevant node using the user-defined update() function and the EOproxy object. In case of set<>, it writes the data of EOrop in its RAM and calls function update() in which the user must take care of propagating the order to the relevant node (e.g., by sending a CAN message of type set). In case of ask<>, it does not reply directly but: (a) stores details of the ask<> inside the EOproxy, and (b) calls the update() function where the user must query the relevant node with the actual value. Only when the user receives a reply from the relevant node, the actual value is passed to the EOproxy which updates the RAM of the EOnv and prepares a say<> EOrop which is put inside the EOtransmitter.

Transmission inside EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOropframe

EOtransmitter

occasionals

regulars

replies

EOtransceiver

EOtransmitter

EOproxy

**Figure 14**: The EOtransmitter contains three internal EOropframe: one with regular EOrop to be transmitted every time, one with the EOrop with replies coming from the EOreceiver or the EOproxy, and one with the EOrop which the board wants to sends occasionally. The output EOropframe is the concatenation of the three. When the output EOropframe is retrieved, the EOproxy is ticked so that timed-out operations on proxied variables are removed from its internal queue.

## The object EOhostTransceiver

The object EOhostTransceiver differs from the EOtheBOARDtransceiver in a few aspects.

First, it is not a singleton and thus can be instantiated many times: one for each board. Second, it is configured to hold knowledge of network variables which are remotely owned, thus it transmits ROPs of type ask<> and set<> and manages incoming ROPs of type say<> and sig<>. Third, it does not have a EOproxy because it does not hold local variables thus not even proxied. Fourth, it can be configured to have a EOconfirmationManager which executes actions on out-coming ROPs with confirmation request flagged on and on reception of ACK or NAK for those ROPs.

### The object EOnvSet

The EOnvSet is initialised by loading a given eOnvset\_DEVcfg\_t data structure for each board. For any of its endpoints is responsible to allocate the RAM and to initialise the protocol library with this RAM and with the number of entities. As a result of initialisation of every EOhostTransceiver, the protocol library shall be initialised with data for every board.

Internals of EOhostTransceiver

BOARD 1

eOnvset\_DEVcfg\_t nvsB1

EOhostTransceiver

EOnvSet

HEAP

RAM of netvars of BRD1

ROM

Protocol library

BOARD 2

eOnvset\_DEVcfg\_t nvsB2

EOhostTransceiver

EOnvSet

EOtransceiver

BRD2

BRD1

EOtransceiver

RAM and ROM for every endpoint of that board

RAM of netvars of BRD2

EP-MC config BRD1

EP-AS config BRD1

EP-MC config BRD2

Callbacks EP-MC for all boards

Callbacks EP-AS for all boards

**Figure 15**: The EOnvSet object of each instance of EOhostTransceiver is initialised with a proper eOnvset\_DEVcfg\_t data structure, use proper RAM and ROM to initialise the protocol library for its own board. In the above picture the BRD1 has both motion control and analog sensors endpoints, whereas the BRD2 has only the motion control.

### The object EOtransceiver inside EOhostTransceiver

The EOtransceiver is the same as in the EOtheBOARDtransceiver, but is configured in a different way due to the different requirements.

The EOtransmitter sends occasional ROPs of kind ask<> and set<>, possibly with confirmation request. The EOreceiver processes received ROPs of kind say<>, sig<>, or of ACK/NAK type. The EOconfirmationManager is uses by the EOtransmitter to call a user-defined function when a ROP requests confirmation info and by the EOreceiver to call a user-defined function when the ACK/NAK is received.

As such the EOhostTransceiver uses an EOtransmitter, an EOreceiver, an EOagent, an EOconfirmationManager, the singletons EOtheFormer and EOtheParser. It does not use and EOproxy,

Insertion of an occasional ROP inside EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOtransmitter

occasionals

regulars

replies

EOtransceiver

EOtransmitter

EOconfManager

EOtheFormer

EOagent

EOropdescriptor

**Figure 16**: The EOtransceiver can accept an occasional ROP for later transmission using function eo\_transceiver\_OccasionalROP\_Load(). The EOagent prepares the EOrop, asks the EOtheFormer to prepare the stream form of the ROP, and puts it inside the EOropframe called occasionals. If the ROP descriptor specifies a conformation request, the EOtransceiver also calls the EOconfirmationManager to register the EOrop.

Transmission inside EOtheBOARDtransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOropframe

EOtransmitter

occasionals

regulars

replies

EOtransceiver

EOtransmitter

EOconfManager

**Figure 17**: The EOtransmitter contains three internal EOropframe but typically uses only one of them: the one with the EOrop which the board wants to sends occasionally. The other two, one with regular EOrop and one with replies from the EOreceiver are typically empty. The output EOropframe is the concatenation of the three. The EOtransmitter also calls function eo\_confman\_Confirmation\_Requested() of the EOconfirmationManager for every ROP that has a confirmation request flagged on.

Reception inside EOhostTransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

EOrop

1. Find the EOnv associated to received ID,
2. If ROP is sig<> or say<>: write data inside EOnv and calls associated update() function.
3. If ROP is ack or nak of a set<> or ask<>: calls the proper function of the object EOconfManager.

For every EOrop inside the received EOropframe:

EOropframe

HEAP

RAM of netvars

EOtransceiver

EOagent

EOreceiver

EOconfManager

EOtheParser

**Figure 18**: The EOreceiver extracts EOrop from the input EOropframe with the help of the EOtheParser and then passes it to the EOagent. If it is a say<> or sig<> it writes the data of EOrop in its RAM and calls a user-defined function update(). If the ROP holds ACK/NAK information, it also calls eo\_confman\_Confirmation\_Received() of the EOconfirmationManager.

Use of EOconfirmationManager inside EOhostTransceiver

BOARD 1

EOtheBOARDtransceiver

EOnvSet

* If ROP is ack or nak of a set<> or ask<>: calls eo\_confman\_Confirmation\_Received().

When receiving a EOropframe the EOconfirmationManager:

EOropframe

EOtransceiver

EOconfManager

EOropframe

* If ROP is a ask<> or set<> with a confirmation request: calls eo\_confman\_Confirmation\_Requested().

When sending a EOropframe the EOconfirmationManager:

**Figure 19**: The function eo\_confman\_Confirmation\_Requested() is called by the EOtransmitter when the user transmits an occasional ROP which requires confirmation. The function eo\_confman\_Confirmation\_Received() is called when the EOreceiver processes an incoming ROP which is a ACK/NAK.

### Example of use of the EOconfirmationManager

The functions eo\_confman\_Confirmation\_Requested() and eo\_confman\_Confirmation\_Received() are called by the by the EOtransmitter when the user loads an occasional ROP which requires confirmation and when the EOreceiver processes an incoming ROP which is a ACK/NAK.

The default implementation of the EOconfirmationManager contains empty functions which can be specialised at creation of the object by passing user-defined callback functions. If the user writes them in a proper way he/she can effectively manage the verification of delivery of a ROP. An example is the following.

The eo\_confman\_Confirmation\_Requested() calls a user-defined function which puts the transmitted ROP inside a list and starts a countdown timer. The countdown timer is used to do an action if after a timeout no ACK/NAK is received. The function eo\_confman\_Confirmation\_Received() is called when the EOreceiver processes the ACK/NAK of a previously transmitted ROP. This method calls call a user-defined function which searches for the sent ROP inside the list and stops the countdown timer. If it the timer expires before having received a ACK/NAK, then the user may issue a warning of ROP not delivered or organise a new transmission.

Most common operations

Here are described the most common operations on the PC104 and on the EMS.

## Operations on the host

The host uses multiple instances of the EOhostTransceiver, each dedicated to a particular remote board (the EMS). The EOhostTransceiver is used in a multi-threading environment and in concurrency. There are two threads which perform reception and transmission, other threads which require transmission of a ROP and sometimes wait for a reply, others which read the RAM of variables. The transmission thread wakes up with a period of 1 msec and sends a packet out only if there are ROPs to be sent. The reception thread is activated on reception of a packet which under regular behaviour is sent every 1 ms by the EMS board. The other threads activate asynchronously.

### Initialisation

For each remote board it is created an object EOhostTransceiver with a given configuration. The configuration contains: (a) information about the protocol endpoints managed by the remote board, (b) IP address and port of the remote board, (c) maximum sizes of the packet, of the single ROP, of the internal ROPframe objects, (d) configuration of the EOconfirmationManager, (e) protection vs. concurrent access of the EOtransceiver, (f) protection vs. concurrent access of the network variables managed by EOnvSet through EOnv objects.

Initialisation

Thread INIT

// we must create a host transceiver for a given board

// and having access to its internal transceiver so that use can sue its methods

#define BRD1 0

const eOhosttransceiver\_cfg\_t\* brd1cfg = getit(BRD1); // the configuration

EOhostTransceiver\* hosttrans1 = eo\_hosttransceiver\_New(brd1cfg);

EOtransceiver\* txrx1 = eo\_hosttransceiver\_GetTransceiver(hosttrans1);

EOnvSet\* nvset1 eo\_hosttransceiver\_GetNVset(hosttrans1);

eOnvBRD\_t brd1 = eo\_hosttransceiver\_GetBoardNumber(hosttrans1);

eOipv4addr\_t ip1 = eo\_hosttransceiver\_GetRemoteIP (hosttrans1);

EOhostTransceiver

EOnvSet

EOtransceiver

**Figure 20**: Initialisation of a single EOhostTransceiver

On the PC104 it is possible to instantiate one EOhostTransceiver for each board. The initialisation is done in the same way for all boards, but each instance of EOhostTransceiver shall use the configuration eOhosttransceiver\_cfg\_t relevant to each board.

Multiple instances of EOhostTransceiver shall have their own RAM where to place the values of the variables of the remote board. However they all share the same init() and update() functions. In other words, if two boards send a sig<> ROP, then the same update() function is called. To know which board has send the ROP it is necessary to evaluate one of the arguments.

### Configuration of regular ROPs

The EOhostTransceiver is NOT configured to transmit regular ROPs. It could be done in theory, for instance to regularly transmit a ROP ask<>.

### Regular use

There are four main activities which the PC104 can perform: (a) processing a received EOropframe, (b) adding occasional ROPs for later transmission, (c) retrieving an object EOropframe for its transmission, and (d) reading RAM of the network variables which contain cached copies of the values in the board.

TX, RX, ROP send, data read

Thread TX

EOpacket\* pkt;

uint16\_t numofrops2tx;

eo\_transceiver\_Transmit(txrx1, &pkt, &numofrops2tx);

< use pkt to send the UDP packet to the socket >

EOhostTransceiver

EOnvSet

EOtransceiver

Thread RX

EOpacket rxpkt;

< read from the socket and fill rxpkt>

uint16\_t numofrxrops;

eo\_transceiver\_Receive(txrx1, &pkt, &numofrxrops, NULL);

// the above function also calls the update() functions

Thread ROP send

Thread ROP send

eOropdescriptor\_t ropdes;

< fill the ropdes >

eo\_transceiver\_OccasionalROP\_Load (txrx1, &ropdes);

Thread data read

Thread data read

eOmc\_joint\_status\_t jstatus; uint16\_t size; EOnv nv;

eOnvID32\_t id32j1status = ...;

eo\_nvset\_NV\_Get(nvset1, ip1, id32j1status, &nv);

eo\_nv\_Get(&nv, eo\_nv\_strg\_volatile, &jstatus, &size);

// OK: now you can use your local copy of jstatus

**Figure 21**: Use of EOhostTransceiver by multiple threads.

These activities can be executed by at least four different threads (one for transmission, one for reception, one or more for occasional insertion of ROPs and one or more for data read), thus some sort of protection vs. concurrent use of shared resources is required.

### Protection from concurrent use

The exact mechanism of use of shared resources in EOhostTransceiver is explained in the following figure.

Use of shared resources

eo\_transceiver\_Transmit()

EOhostTransceiver

EOnvSet

EOtransceiver

eo\_transceiver\_OccasionalROP\_Load()

eo\_nv\_Get()

eo\_transceiver\_Receive()

WRITES

EOropframe replies

EOnv

EOnv

WRITES

of rx ROPs and calls update()

READS

EOropframe regulars

EOnv

EOnv

READS

to update regulars ROPs

READS

EOropframe occasionals

and CLEARS

READS

EOropframe replies

and CLEARS

EOnv

EOnv

READS

of chosen variables

WRITES

EOropframe occasionals

**Figure 22**: Use of shared resources in EOhostTransceiver.

It is clear that the internal EOropframe objects can be written and read concurrently with very dangerous consequences. And also the EOnv objects retrieved through EOnvSet may be read while being written producing incoherent results.

To solve such problems the EOhostTransceiver has a dual protection mode, one for the EOtransceiver and one for the EOnvSet, which can be activated at creation time by means of the configuration of EOhostTransceiver (of type eOhosttransceiver\_cfg\_t).

If any of these protection modes are activated, it is required to fill in the configuration also the function pointer to the constructor of recursive mutex object. In this way the EOhostTransceiver can allocate and use the mutexes required to give the specified protection.

In particular it uses three mutexes to protect the EOtransceiver: one of each for the regulars, occasionals and replies EOropframe objects. It also uses one or more mutexes to protect the manipulation of the EOnv objects retrieved through EOnvSet. It may use only one mutex for the entire board, or one for each endpoint or even one for each EOnv.

An alternative mode to give protection is to turn off the internal protection and use a single external mutex to protect every method of EOtransceiver and also every method of the EOnv objects retrieved obtained through the EOnvSet of the EOhostTransceiver.

### Send a ROPframe

The TX thread at a regular pace retrieves a packet to be transmitted. The EOtransceiver fills its internal EOropframe inside an object EOpacket which has a payload, its size, the destination IP address and port. The EOpacket is then given to the socket for transmission.

How to retrieve a EOpacket from EOhostTransceiver

// here is a handle to a packet

EOpacket\* packet = NULL;

uint16\_t numberofrops = 0;

// then we retrieve the pointer of the fully formed packet internal to the transceiver

// numberofrops contains the number of ROPs inside the EOropframe contained inside the payload

eOresult\_t res = **eo\_transceiver\_Transmit**(txrx1, &packet, &numberofrops);

uint8\_t \*payload\_data;

uint16\_t payload\_size;

eOipv4addr\_t ipaddr;

eOipv4port ipport;

res = eo\_packet\_Payload\_Get(packet, &payload\_data, &payload\_size);

res = eo\_packet\_Addressing\_Get(packet, &ipaddr, &ipport);

// now we can send payload\_size bytes contained inside payload\_data to a given ip address and port.

// by the way … ip and port returned by eo\_transceiver\_Transmit() are those specified in the brd1cfg

// variable used to create the transceiver.

**Figure 23**: The occasional ROP set<> is added using a method of EOtransceiver.

### Add an occasional ROP to the transceiver

For the TX thread to transmit a EOropframe with some ROPs inside it is required that another thread adds a ROP. It is possible to add set<> or ask<> ROPs.

#### Add an occasional ROP set<> to the transceiver

It is possible to add a set<> ROP to the EOtransceiver as in the following example. It is shown how to set the PID of position for the joint 1 (the second one) with a give value. The identification of the variable comes through its ID which can be obtained from its endpoint, entity, index and tag.

How to add a set<> ROP to EOhostTransceiver

// here we have the variable that we want to set

eOmc\_joint\_config\_t jointcfg = { }; // fill with desired values

void \*var\_data = &jointcfg.pidposition;

uint16\_t var\_size = sizeof(jointcfg.pidposition);

eOnvID32\_t var\_id = eoprot\_ID\_get( eoprot\_endpoint\_management, // the endpoint: mc

eoprot\_entity\_mc\_joint, // the entity: joint

1, // the index: the second one

eoprot\_tag\_mc\_joint\_config\_pidposition // the tag: the pid position

);

// then we need to build a rop descriptor

eOropdescriptor\_t ropdescr =

{

.control =

{

.confinfo = eo\_ropconf\_none,

.plustime = 0,

.plussign = 0,

.rqsttime = 0,

.rqstconf = 0,

.version = EOK\_ROP\_VERSION\_0

},

.ropcode = **eo\_ropcode\_set**,

.size = var\_size, // of the target variable

.id32 = **var\_id**, // of the target variable

.data = **var\_data**, // which we want to set

.signature = 0,

.time = 0

};

// now we can add the rop as an occasional transmission

eOresult\_t res = **eo\_transceiver\_OccasionalROP\_Load**(txrx1, &ropdescr);

**Figure 24**: The occasional ROP set<> is added using a method of EOtransceiver.

#### Add an occasional ROP ask<> to the transceiver

It is possible to ask for a given value by forming ROP ask<>. In the following we show the request of the entire configuration of joint 1.

How to add a ask<> ROP to EOhostTransceiver

// then we define the variable that we want to ask for

eOmc\_joint\_config\_t jointcfg = { }; // fill with desired values

uint16\_t var\_size = sizeof(eOmc\_joint\_config\_t);

eOnvID32\_t var\_id = eoprot\_ID\_get( eoprot\_endpoint\_management, // the endpoint: mc

eoprot\_entity\_mc\_joint, // the entity: joint

1, // the index: the second one

eoprot\_tag\_mc\_joint\_config // the tag: the config

);

// then we need to build a rop descriptor

eOropdescriptor\_t ropdescr =

{

.control =

{

.confinfo = eo\_ropconf\_none,

.plustime = 0,

.plussign = 0,

.rqsttime = 0,

.rqstconf = 0,

.version = EOK\_ROP\_VERSION\_0

},

.ropcode = eo\_ropcode\_ask,

.size = 0, // not required

.id32 = var\_id, // of the target variable

.data = NULL, // not required

.signature = 0,

.time = 0

};

// now we can add the rop as an occasional transmission

eOresult\_t res = eo\_transceiver\_OccasionalROP\_Load(txrx1, &ropdescr);

**Figure 25**: The occasional ROP ask<> is added using a method of EOtransceiver.

### Manage the reception of a ROP

When the RX thread retrieves a UDP packet, it calls the method eo\_transceiver\_Receive(). In case of ROPs of kind sig<> or say<> it copies the data inside the ROP into the RAM of the network variable and calls the relevant update() function. If protection of EOnvSet is configured, these operations are protected by a mutex. See following figure for exact timing sequence.

When update() is called

EOhostTransceiver

EOnvSet

EOtransceiver

eo\_transceiver\_Receive()

EOnv

WRITES

1. mutex\_take(mutex);
2. memcpy(nv->ram, data, size);
3. nv->update();
4. mutex\_release(mutex);

**Figure 26**: The memory copy and the call of update() are protected by the configured mutex.

#### How to override and use the update() function.

The default update function is empty, however it can be overridden. To override the function it is necessary to define a new function with the same name as declared in EoProtocolXX.h, where XX is the short for the endpoint. The following example shows the update() function for the status of the joint in motion control endpoint. It is shown how to retrieve important information from the two parameters.

The call of update()at reception of a ROP inside EOhostTransceiver

// the function is empty by default and for motion control is weakly defined inside EoProtocolMC\_fun.c

\_\_weak extern void eoprot\_fun\_UPDT\_mc\_joint\_status(const EOnv\* nv, const eOropdescriptor\_t\* rd) {}

// the function can be redefined as in the following

#include “EoProtocolMC.h”

extern void eoprot\_fun\_UPDT\_mc\_joint\_status(const EOnv\* nv, const eOropdescriptor\_t\* rd)

{

// which board with which IP? Which is the ID of the variable? And its prog number?

eOnvBRD\_t brd = eo\_nv\_GetBRD(nv); // or brd = nv->brd

eOipv4addr\_t ip = eo\_nv\_GetIP(nv); // or ip = nv->ip

eOnvID32\_t id = eo\_nv\_GetID32(nv); // or id = nv->id, or id = rd->id32;

eOprotProgNumber\_t prog = eoprot\_id2prognum(brd, id);

// which joint index?

eOprotIndex\_t jointindex = eoprot\_ID2index(id);

// which operation?

eOropcode\_t ropcode = rd->ropcode;

// which is the value of the variable joint status? And its size?

eOmc\_joint\_status\_t status; eOmc\_joint\_status\_t\* stat = NULL; uint16\_t size = 0;

// mode 1: through the rop descriptor. IT IS THE QUICKEST and CLEANEST

stat = (eOmc\_joint\_status\_t\*) rd->data;

size = rd->size;

memcpy(&status, stat, size);

// mode 2: through methods of the EOnv. IT ALSO CALLS THE MUTEX (IF CONFIGURED)

eo\_nv\_Get(nv, eo\_nv\_strg\_volatile, &status, &size); // copied into status.

// mode 3: through direct access to protected fields of the EOnv.

stat = (eOmc\_joint\_status\_t\*) nv->ram;

size = nv->rom.capacity;

memcpy(&status, stat, size);

// now I can alert higher levels.

alert\_motioncontrol\_received\_rop(brd, ip, id, prog, jointindex, ropcode, &status, size);

}

**Figure 27**: The ROP is received and the relevant update() function is called.

There is only one update function for a given ID (endpoint, entity, index, tag), thus the same function is called by all the instances of EOhostTransceiver for any board. It is important to get the correct board number to address the proper higher layer data.

The RX thread is the only one that modifies the RAM of the variable as all others only read it. Thus, inside the overridden update() any read of the RAM of the variable does not need to be protected vs concurrent access and direct use of the pointer is safe.

### How to propagate data received through sig<> ROPs to higher layers

In the PC104 the receiving thread uses the EOhostTransceiver to automatically manage the received sig<> ROPs, either occasionally sent by the remote board or regularly sent after the PC104 has configured the board to do so. What is typically sent is some status or some warning message.

There are two modes to let higher layers know about the received values. The first mode is to rely on the automatic mechanism of ROP reception to refresh the values of variables and let higher layers to read the values when they need directly from the network variables or when the update() alerts them. The second is to use the update() function to move data to higher layers and let higher layers to read the values from there without reading the network variables.

#### The values stop at the network variable layer

The update() does nothing. The reading thread which is interested in the value of a variable can simply retrieve its value through the eo\_nv\_Get() method. The EOnv gives protection vs. concurrent access by means of the configured mutex.

The update() does nothing on reception of a sig<>

// this function is called by the asking thread to get the value specified by the id.

extern void get\_remote\_value\_by\_ram(eOnvID32\_t id, void\* value, uint16\_t\* size)

{

// access to variables is protected vs. concurrency by the mutex of EOnvSet.

EOnv nv;

eo\_nvset\_NV\_Get(nvset1, ip1, id, &nv); // ok, we must know that we are in board 1

eo\_nv\_Get(&nv, eo\_nv\_strg\_volatile, value, size);

}

// this function is called by the receiving thread

extern void eoprot\_fun\_UPDT\_mc\_motor\_status(const EOnv\* nv, const eOropdescriptor\_t\* rd)

{

if(eo\_ropcode\_sig == rd->ropcode)

{

return; // do nothing

}

// other things in case of different rop codes

< other code in here>

}

**Figure 28**: A reading task just reads the RAM of the variable using a protected mode. The update() function does nothing on reception of a sig<>.

#### The values are copied into higher layers

An alternative mode is to use the update() function called on reception of the sig<> ROP to copy the RAM into a higher layer buffer. It is this buffer which is retrieved by the reading thread. However, it is necessary to protect concurrent reading and writing of this buffer with a mutex.

The update() on reception of a sig<> copies into higher layers

// this function is called by the asking thread to get the value specified by the id.

MyClass::get\_remote\_value(void\* data, int\* size, int ID)

{

MyClass::mutex->lock();

<copy MyClass::buffer identified by ID into data and assign \*size>

MyClass::mutex->unlock();

}

// this function is called by the receiving thread

extern void eoprot\_fun\_UPDT\_mc\_motor\_status(const EOnv\* nv, const eOropdescriptor\_t\* rd)

{

if(eo\_ropcode\_sig == rd->ropcode)

{

MyClass::put\_remote\_value(rd->data, rd->size, ID\_mc\_motor\_config);

return;

}

// other things in case of different rop codes

<other code in here>

}

// this function is called by the receiving thread inside the update(0 function.

MyClass::set\_remote\_value(void\* data, int size, int ID)

{

MyClass::mutex->lock();

<copy size bytes of data into MyClass::buffer identified by ID>

MyClass::mutex->unlock();

}

**Figure 29**: A reading task just calls the method of a class. The update() function calls a method of the class to put the RAM of the signalled variable inside the class.

### How to manage a blocking ask<> and say<>

When a thread of the PC104 wants to know the remote values of a variable it sends a ROP ask<> and waits for the arrival of a ROP say<>. The PC104 may implement a blocking mechanism which stops the asking thread until the receiving task alerts it about the ROP reception. For that, it can be useful using the signature, a 32-bit number that is embedded in the ROP ask<> and which the protocol embeds also in the ROP say<>.

See the example below, where the signature of the received ROP s directly used to manipulate the semaphore. More accurate implementations could use the signature as a key to look for in a map which contains the semaphore. In such a way one could better manage portability to a 64-bit system and also ease life in case of expired timeouts.

The update() is used to unblock a semaphore

// this function is called by the asking thread to get the value specified by the id.

// it blocks the thread until a reply is reached. The BOARD is BRD1

extern void get\_remote\_value\_by\_rop(eOnvID32\_t id, void\* value, uint16\_t\* size)

{

Semaphore\* sem = new Semaphore(0); // counter is already zero, so that a decrement blocks

eOropdescriptor\_t ropdescr = {0};

memcpy(&ropdescr.control, &eok\_ropctrl\_basic, sizeof(eOropctrl\_t));

ropdescr.plussign = 1; // PLUS SIGNATURE !!!!!!!!!

ropdescr.ropcode = eo\_ropcode\_ask;

ropdescr.signature = (uint32\_t) &sem; // valid ONLY IF pointers are 32 bits

// now we add the rop as an occasional transmission

eOresult\_t res = eo\_transceiver\_RegularROP\_Load(txrx1, &ropdescr);

// and we wait until some other receiving thread increments the semaphore.

sem->decrement(); // it blocks in here

// we are in here if we have received the say<> with our unique signature.

// thus we get the value and the size.

get\_remote\_value\_RAM\_mapped\_with\_EOnv(id, value, size);

delete sem; // remember to delete it ...

}

// this function is called by the receiving thread

extern void eoprot\_fun\_UPDT\_mc\_motor\_config(const EOnv\* nv, const eOropdescriptor\_t\* rd)

{

if((1 == rd->control.plussign) && (eo\_ropcode\_say == rd->ropcode))

{

Semaphore\* sem = (Semaphore\*) rd->signature;

sem->increment(); // now the asking thread can execute (when scheduled)

return;

}

<other code in here for other opcodes>

}

**Figure 30**: A task sends a ask<ID> with a given signature and waits until some other task which receives the relevant say<ID, val> increments a semaphore inside the update function.

### Have access of a given variable from its ID

The safest mode to access a variable is through the relevant EOnv object retrieved by means of the EOnvSet of the EOhostTransceiver. In here the concurrent access is managed by the internal mutex. As an alternative one can directly use the functions of the protocol library. However, protection is not given.

Access of a variable from its ID using the EOnv

// The BOARD is BRD1

extern void get\_remote\_value\_RAM\_mapped\_with\_EOnv(eOnvID32\_t id, void\* value, uint16\_t\* size)

{

// mode 0: by EOnv. Access to variables is protected vs concurrency by the mutex of EOnvSet.

EOnv nv;

eo\_nvset\_NV\_Get(nvset1, ip1, id, &nv);

eo\_nv\_Get(&nv, eo\_nv\_strg\_volatile, value, size);

}

**Figure 31**: In the example the function access the RAM of a variable a call to the eo\_nv\_Get() method which uses the mutex (if configured).

Access of a variable from its ID using the protocol library

// The BOARD is BRD1

extern void get\_remote\_value\_RAM\_mapped\_with\_library(eOnvID32\_t id, void\* value, uint16\_t\* size)

{

// mode 1: by library. Access to variables is not protected vs concurrency.

\*size = eoprot\_variable\_sizeof\_get(brd1, id);

void\* temp = eoprot\_variable\_ramof\_get(brd1, id);

memcpy(value, temp, \*size);

}

**Figure 32**: In the example the function uses the RAM of a variable directly with the protocol library. Protection vs. concurrent access is not present.

### How to build IDs

The IDs are built using the method eoprot\_ID\_get() of the protocol library, where its arguments are taken in the following way. The endpoint and entity are taken from file EoProtocol.h, the index is just a number, and the tags are taken from file EoProtocolXX.h, where XX is AS for analog sensors, MC for motion control, MN for management, SK for skin etc.

How to build IDs

#include “EoProtocol.h”

#include “EoProtocolMC.h”

#include “EoMotionControl.h”

eOprotID32\_t id;

uint8\_t MaxIndex;

// joint …

MaxIndex = eoprot\_entity\_numberof\_get( brd1, // from 0 to eoprot\_boards\_maxnumberof-1

eoprot\_endpoint\_motioncontrol, // see enum eOprot\_endpoint\_t

eoprot\_entity\_mc\_joint // see enum eOprot\_entity\_t

);

id = eoprot\_ID\_get( eoprot\_endpoint\_motioncontrol, // see enum eOprot\_endpoint\_t

eoprot\_entity\_mc\_joint, // see enum eOprot\_entity\_t

3, // from 0 to MaxIndex-1

eoprot\_tag\_mc\_joint\_status // see enum eOprot\_tag\_mc\_joint\_t

);

eOmc\_joint\_status\_t joint\_status;

**Figure 33**: An ID in motion control.

How to build IDs

#include “EoProtocol.h”

#include “EoProtocolAS.h”

#include “EoAnalogSensors.h”

eOprotID32\_t id;

uint8\_t MaxIndex;

// strain …

MaxIndex = eoprot\_entity\_numberof\_get( brd1, // from 0 to eoprot\_boards\_maxnumberof-1

eoprot\_endpoint\_analogsensors, // see enum eOprot\_endpoint\_t

eoprot\_entity\_as\_strain // see enum eOprot\_entity\_t

);

// the strain is present only if MaxIndex is > 0.

id = eoprot\_ID\_get( eoprot\_endpoint\_analogsensors, // see enum eOprot\_endpoint\_t

eoprot\_entity\_as\_strain, // see enum eOprot\_entity\_t

0, // from 0 to MaxIndex-1

eoprot\_tag\_as\_strain\_config // see enum eOprot\_tag\_mc\_joint\_t

);

eOas\_strain\_config\_t strain\_cfg;

**Figure 34**: An ID in analog sensors.

## Operations on the board

On the board, the reception, transmission, insertion of a ROP, use of RAM are done in sequence by the same task, thus there is no need to protect vs. concurrent access.

### Initialisation

It is created only one object EOtheBOARDtransceiver with a given configuration for the board. The configuration contains: (a) information about the protocol endpoints managed by the remote board, (b) IP address and port of the remote board, (c) maximum sizes of the packet, of the single ROP, of the internal ROPframe objects, (d) no use of the EOconfirmationManager, (e) no protection vs. concurrent access for the EOtransceiver, (f) no protection vs. concurrent access of the network variables managed by EOnvSet through EOnv objects.

Initialisation

Thread INIT

// we must create a board transceiver for a given board

// and having access to its internal transceiver so that use can sue its methods

#define BRD1 0

const eOboardtransceiver\_cfg\_t\* brd1cfg = getit(BRD1); // the configuration

EOtheBOARDtransceiver\* brdtrans1 = eo\_boardtransceiver\_Initialise(brd1cfg);

EOtransceiver\* txrx1 = eo\_boardtransceiver\_GetTransceiver(brdtrans1);

EOnvSet\* nvset1 eo\_boardtransceiver\_GetNVset(brdtrans1);

eOnvBRD\_t brd1 = eo\_boardtransceiver\_GetBoardNumber(brdtrans1);

eOipv4addr\_t ip1 = eok\_ipv4addr\_localhost;

EOtheBOARDTransceiver

EOnvSet

EOtransceiver

EOtransmitter

EOreceiver

EOconfManager

**Figure 35**: Initialisation.

### Configuration of regular ROPs

It is done upon reception of a ROP set<> send by the PC104.

The relevant update function is eoprot\_fun\_UPDT\_mn\_comm\_cmmnds\_ropsigcfg(). It calls methods of the EOtransceiver such as eo\_transceiver\_RegularROPs\_Clear(),eo\_transceiver\_RegularROP\_Load() or eo\_transceiver\_RegularROP\_Unload().

### Have access to a given variable from its ID

It can be done in two ways, as in the PC104; by use of the relevant EOnv or directly using its RAM. It is possible to retrieve the RAM by means of the protocol library in several modes: by variable and by entity.

Access of a variable from its ID using the protocol library

// The BOARD is eoprot\_board\_localboard (there is only one board)

extern void get\_remote\_value\_RAM\_mapped\_with\_library(eOnvID32\_t id, void\* value, uint16\_t\* size)

{

// mode 1: by library. Access to variables is not protected vs concurrency.

\*size = eoprot\_variable\_sizeof\_get(eoprot\_board\_localboard, id);

void\* temp = eoprot\_variable\_ramof\_get(eoprot\_board\_localboard, id);

memcpy(value, temp, \*size);

}

**Figure 36**: In the example the function uses the RAM of a variable directly with the protocol library. Protection vs. concurrent access is not present.

Access of a complete entity using the protocol library

// The BOARD is eoprot\_board\_localboard, the entity is the first joint

eOjoint\_t joint; uint16\_t size;

get\_remote\_value\_RAM\_mapped\_with\_library( eoprot\_endpoint\_motioncontrol,

eoprot\_entity\_mc\_joint,

0,

&joint, &size);

extern void get\_remote\_entity\_RAM\_mapped\_with\_library( eOprotEndpoint\_t ep,

eOprotEntity\_t entity,

eOprotIndex\_t index,

void\* value, uint16\_t\* size)

{

\*size = eoprot\_entity\_sizeof\_get(eoprot\_board\_localboard, ep, entity);

void\* temp = eoprot\_entity\_ramof\_get(eoprot\_board\_localboard, ep, entity, index);

memcpy(value, temp, \*size);

}

**Figure 37**: In the example the function uses the RAM of the entity directly with the protocol library. Protection vs. concurrent access is not present.

### Manage the reception of a ROP set<>

The EOreceiver parses the received EOropframe and for each set<> ROP it writes the data inside the ROP into the RAM of the network variable and calls the relevant update() function.

The update function is typically used to propagate to other SW objects the value the ROP has written into the RAM. It is possible to have access to the RAM of a variable from inside its update function by simply using the ROP descriptor or the EOnv handle.

If the variable is a command, then the relevant action must be executed, else if it is a configuration which must remain into the board it is just set a flag to tell somebody to use the new values.

Finally if it is a configuration which must be propagated via CAN to other boards the CAN packets are prepared and put inside a queue. It is the case of a proxied variable.

The following example shows the update() function used to configure the transmission of regular ROPs.

The update() function of tag eoprot\_tag\_mn\_comm\_cmmnds\_ropsigcfg of entity eoprot\_entity\_mn\_comm in endpoint eoprot\_endpoint\_management

// this function is called by the receiving thread

extern void eoprot\_fun\_UPDT\_mn\_comm\_cmmnds\_ropsigcfg(const EOnv\* nv, const eOropdescriptor\_t\* rd)

{

eOmn\_ropsigcfg\_command\_t\* ropsigcfgcmd = (eOmn\_ropsigcfg\_command\_t\*)rd->data;

EOarray \*array = (EOarray\*)&ropsigcfgcmd->array;

eOmn\_ropsigcfg\_commandtype\_t cmmnd = (eOmn\_ropsigcfg\_commandtype\_t)ropsigcfgcmd->cmmnd;

uint8\_t size = eo\_array\_Size(array);

EOtransceiver\* theemstxrx = eo\_boardtransceiver\_GetTransceiver(eo\_boardtransceiver\_GetHandle());

switch(cmmnd)

{

uint8\_t i;

case ropsigcfg\_cmd\_clear:

{ // just clear all the regular ROPs

eo\_transceiver\_RegularROPs\_Clear(theemstxrx);

} break;

case ropsigcfg\_cmd\_append:

{ // dont clear and load all the sigcfg in the array

for(i=0; i<size; i++)

{

sigcfg = (eOropSIGcfg\_t\*)eo\_array\_At(array, i);

memcpy(&ropdesc.control, &eok\_ropctrl\_basic, sizeof(eOropctrl\_t));

ropdesc.control.plustime = (eobool\_true == ropsigcfgcmd->plustime) ? (1) : (0);

ropdesc.control.plussign = (eobool\_true == ropsigcfgcmd->plussign) ? (1) : (0);

ropdesc.ropcode = eo\_ropcode\_sig;

ropdesc.id32 = sigcfg->id32;

ropdesc.signature = ropsigcfgcmd->signature;

eo\_transceiver\_RegularROP\_Load(theemstxrx, &ropdesc);

}

} break;

case ropsigcfg\_cmd\_assign:

{ // clear and load all the sigcfg in the array

eo\_transceiver\_RegularROPs\_Clear(theemstxrx);

< code is omitted >

}

case ropsigcfg\_cmd\_remove:

{ // remove all the sigcfg in the array

for(i=0; i<size; i++)

{

sigcfg = (eOropSIGcfg\_t\*)eo\_array\_At(array, i);

memcpy(&ropdesc.control, &eok\_ropctrl\_basic, sizeof(eOropctrl\_t));

ropdesc.control.plustime = (eobool\_true == ropsigcfgcmd->plustime) ? (1) : (0);

ropdesc.control.plussign = (eobool\_true == ropsigcfgcmd->plussign) ? (1) : (0);

ropdesc.ropcode = eo\_ropcode\_sig;

ropdesc.id32 = sigcfg->id32;

ropdesc.signature = ropsigcfgcmd->signature;

eo\_transceiver\_RegularROP\_Unload(theemstxrx, &ropdesc);

}

} break;

default:

{

} break;

}

}

**Figure 38**: The update() function executes the code required to make effective the command stored inside the RAM. At first it retrieve value of the eOmn\_ropsigcfg\_command\_t and then it configures the regular ROPs of EOtransceiver. It can clear them all or operate with an array of ROP descriptors contained in the command itself to: (a) append them to the existing, (b) substitute them to the existing, and (c) remove them from the existing.

### Build a ROP sig<>

It is possible to signal a given value by forming ROP sig<>. In the following we show the signalling of the entire configuration of joint 1 of board 1.

How to add a sig<> ROP to EOtheBOARDtransceiver

void \*var\_data = NULL; // we don’t set thus the ROP does not have any payload

uint16\_t var\_size = sizeof(eOmc\_joint\_config\_t);

eOnvID32\_t var\_id = eoprot\_ID\_get( eoprot\_endpoint\_management, // the endpoint: mc

eoprot\_entity\_mc\_joint, // the entity: joint

1, // the index: the second one

eoprot\_tag\_mc\_joint\_config // the tag: the config

);

// then we need to build a rop descriptor

eOropdescriptor\_t ropdescr =

{

.control =

{

.confinfo = eo\_ropconf\_none,

.plustime = 0,

.plussign = 0,

.rqsttime = 0,

.rqstconf = 0,

.version = EOK\_ROP\_VERSION\_0

},

.ropcode = eo\_ropcode\_sig,

.size = 0, // there is no need of a size as it is internally searched for

.id32 = var\_id, // of the target variable

.data = NULL, // there is no need of a pointer as it is internally searched for

.signature = 0,

.time = 0

};

// now we can add the rop as an occasional transmission

eOresult\_t res = eo\_transceiver\_OccasionalROP\_Load(txrx1, &ropdescr);

**Figure 39**: The occasional ROP sig<> is added using a method of EOtransceiver.

### Manage the reception of a ROP ask<>

The ROP ask<> is managed automatically for fully local variables and need user customisation only for those variables which are proxied.

The EOreceiver parses the received EOropframe and for each ask<ID> ROP with ID corresponding to a fully local variable, it builds the say<ID, data> ROP by filling data with the RAM of the EOnv associated to the ID. In case the ROP ask<> contains a signature the ROP say<> shall contain the same signature. Then the ROP is passed to the EOtransmitter for insertion in the next EOropframe to be sent to the PC104.

If the variable is proxied, the EOreceiver puts the EOrop and the EOnv inside the EOproxy and it does not prepare any say<> to send back. The EOproxy stores the request in an internal queue and then calls the update() function of the EOnv. It is responsibility of the user to propagate to CAN bus (or whatever) the request for the value.

When the reply arrives, the user must call the function eo\_transceiver\_LoadReplyInProxy() passing the received data and the relevant ID. This function calls the EOproxy which takes care of forming the correct say<> with ID, data and if requested the time and the same signature as the ROP say<> had. Then the ROP is passed to the EOtransmitter for insertion in the next EOropframe to be sent to the PC104

Configuration of a board

Here is described how a board is configured to use a number of endpoints with a given amount of entities.

## The eOprot\_Bxx.h file

The file eOprot\_Bxx.h contains the board number, the number of managed endpoints and for each them it gives description of how many entities they have. It also contains the declaration of a constant object of type eOnvset\_DEVcfg\_t which is the one used by the EOtransceiver to configure the EOnvSet.

The eOprot\_B01.h file

// - external dependencies ---------------------------------------------------------------------------

#include "EoCommon.h"

#include "EOnvSet.h"

#include "EoProtocol.h"

#include "EoProtocolAS.h"

#include "EoProtocolMC.h"

#include "EoProtocolMN.h"

// - declaration of public user-defined types --------------------------------------------------------

enum { eoprot\_b01\_boardnumber = 0 };

enum { eoprot\_b01\_endpoints\_numberof = 3 };

**Figure 40**: The general part of eOprot\_B01.h file.

The eOprot\_B01.h file

// - management

enum { eoprot\_b01\_mn\_comms\_numberof = 1,

eoprot\_b01\_mn\_appls\_numberof = 1 };

/\*\* @typedef typedef struct eOprot\_b01\_management\_t;

@brief It is the container of entities comm and app in the management endpoint of board b01.

\*\*/

typedef struct

{

eOmn\_comm\_t communication;

eOmn\_appl\_t application;

} eOprot\_b01\_management\_t;

**Figure 41**: The management part of file eOprot\_B01.h file.

The eOprot\_B01.h file

// - motion control

enum { eoprot\_b01\_mc\_joints\_numberof = 4,

eoprot\_b01\_mc\_motors\_numberof = 4,

eoprot\_b01\_mc\_controllers\_numberof = 1 };

/\*\* @typedef typedef struct eOprot\_b01\_motioncontrol\_t;

@brief It is the container of joints, motors, controllers in the motion control

endpoint of board eb1.

\*\*/

typedef struct

{

eOmc\_joint\_t joints[eoprot\_b01\_mc\_joints\_numberof];

eOmc\_motor\_t motors[eoprot\_b01\_mc\_motors\_numberof];

eOmc\_controller\_t thecontroller;

} eOprot\_b01\_motioncontrol\_t;

**Figure 42**: The motion control part of file eOprot\_B01.h file.

The eOprot\_B01.h file

// - analog sensors

enum { eoprot\_b01\_as\_strains\_numberof = 1,

eoprot\_b01\_as\_maises\_numberof = 0, // THIS BOARD DOES NOT HAVE A MAIS

eoprot\_b01\_as\_extorque\_numberof = 4 };

/\*\* @typedef typedef struct eOprot\_b01\_analogsensors\_t;

@brief It is the container of strain, extortque in the analog sensors endpoint of board eb1.

\*\*/

typedef struct

{

eOas\_strain\_t strain;

eOas\_extorque\_t extorque[eoprot\_b01\_as\_extorque\_numberof];

} eOprot\_b01\_analogsensors\_t;

**Figure 43**: The analog sensors part of file eOprot\_B01.h file.

The eOprot\_B01.h file

// - declaration of extern public variables, ... but better using use \_get/\_set instead --------------

// the configuration for the EOnvset object for protocol management

extern const eOnvset\_DEVcfg\_t eoprot\_b01\_nvsetDEVcfg;

// the number of entities for each endpoint organised as an array

extern const uint8\_t eoprot\_b01\_mn\_entities\_numberofeach[];

extern const uint8\_t eoprot\_b01\_mc\_entities\_numberofeach[];

extern const uint8\_t eoprot\_b01\_as\_entities\_numberofeach[];

// - declaration of extern public functions ----------------------------------------------------------

/\*\* @fn extern eOresult\_t eoprot\_b01\_Initialise(eObool\_t islocal)

@brief Initialises the endpoints of this board by loading the number of

entities for each of them in the related endpoint file. As a result of that,

the functions which require a brd argument will return the correct value if called

with brd = eoprot\_b01\_boardnumber.

This function is called by the EOnvset because the eOnvset\_DEVcfg\_t contains a

pointer to it. However, it is made public so that it can be called independently

from the use of EOnvset.

@return eores\_OK if successful or eores\_NOK\_generic upon failure.

\*\*/

extern eOresult\_t eoprot\_b01\_Initialise(eObool\_t islocal);

/\*\* @fn extern eObool\_t eoprot\_b01\_isvariableproxied(eOnvID32\_t id)

@brief Tells if a variable is proxied.

This function is called by the EOnvset because the eOnvset\_DEVcfg\_t contains a

pointer to it. However, it is made public so that it can be called independently

from the use of EOnvset.

@return eobool\_true if the variable described by ID is proxied, eobool\_false if it is fully

local.

\*\*/

extern eObool\_t eoprot\_b01\_isvariableproxied(eOnvID32\_t id);

**Figure 44**: The exported variables and functions of file eOprot\_B01.h file.

## The eOprot\_Bxx.c file

The file eOprot\_Bxx.c contains definition of all data structures declared in eOprot\_Bxx.h.

Change of an endpoint

Here it is described how to modify an existing endpoint by adding a tag to an existing entity or to add a new entity to an existing endpoint.

## Adding a tag on an existing entity

At first, it is required to identify which is the entity where to add a tag. Suppose it is a joint in endpoint motion control and the variable inside eOjoint is abcdef.

1. If the struct eOjoint\_t in file EoMotionControl.h does not have the variable abcdef, then it is necessary to add it.
2. In file EOprotocolMC.h it is necessary to add the relevant tag in enum eOprot\_tag\_mc\_joint\_t. Then increment the constant eoprot\_tags\_mc\_joint\_numberof.
3. In file EOprotocolMC.h it is necessary to add the relevant rw mode in enum eOprot\_rwm\_mc\_joint\_t. Then increment the constant eoprot\_rwms\_mc\_joint\_numberof.
4. In file EOprotocolMC.h it is necessary to add declaration of the INIT and UPDT functions.
5. In file EOprotocolMC\_fun.c add the definition of the INIT and UPDT functions.
6. In file EOprotocolMC\_rom.c add the static constant descriptor for the variable as: static EOnv\_rom\_t eoprot\_mc\_rom\_descriptor\_joint\_abcdef variable.
7. In file EOprotocolMC\_rom.c add inside the array eoprot\_mc\_rom\_folded\_descriptors[] the entry with & eoprot\_mc\_rom\_descriptor\_joint\_abcdef in the section of the joint and with the same order as its tag.

## Adding a new entity

At first, it is required add the entity and all the wanted tags. Let’s use as an example the addition of an external torque measure in analog sensors endpoint. Let’s call it extorque.

1. In file EoAnalogSensors.h, add eoas\_entity\_extorque inside eOas\_entity\_t with value equal to the previous incremented by one, increment value of eoas\_entities\_numberof. Then define the entity as struct eOas\_extorque\_t.
2. In file EoProtocolAS.h add a section about extorque with: the tags inside eOprot\_tag\_as\_extorque\_t, their number in eoprot\_tags\_as\_extorque\_numberof, the rw modes in eOprot\_rwm\_as\_extorque\_t and their number in eoprot\_rwms\_as\_extorque\_numberof. Then in struct eOprot\_template\_as\_t add a field extorque using the same order as values in eOas\_entity\_t (add after last position).
3. Add declaration and definition of all the INIT and UPDT functions in files EoProtocolAS.h and EoProtocolAS\_fun.c.
4. In file EoProtocolAS\_rom.c add: verification of tags using the EO\_VERIFYproposition() macro, a static const default value for eOas\_extorque\_t, the definition descriptors for all the tags of an extorque.
5. In file EoProtocolAS\_rom.c fill the array eoprot\_as\_rom\_folded\_descriptors[]. The place where to add pointers to the descriptors is: after all of previous entity with the same order of the tag values in eOprot\_tag\_as\_extorque\_t.
6. In file EoProtocolAS\_rom.c add an entry in following arrays: eoprot\_as\_rom\_tags\_numberof[], eoprot\_as\_rom\_entities\_sizeof[], eoprot\_as\_rom\_entities\_defval[].

Finally one must change the files eOprot\_Bxx.[h, c] to tell how many entities a given board supports. As an example see file eOprot\_B01.h.

The eOprot\_B01.h file

// - analog sensors

enum { eoprot\_b01\_as\_strains\_numberof = 1,

eoprot\_b01\_as\_maises\_numberof = 0,

eoprot\_b01\_as\_extorque\_numberof = 4 };

/\*\* @typedef typedef struct eOprot\_b01\_analogsensors\_t;

@brief It is the container of strain, extortque in the analog sensors endpoint of board eb1.

\*\*/

typedef struct

{

eOas\_strain\_t strain;

eOas\_extorque\_t extorque[eoprot\_b01\_as\_extorque\_numberof];

} eOprot\_b01\_analogsensors\_t;

**Figure 45**: The analog sensors part of file eOprot\_B01.h file with changes due to addition of entity extorque.

The eOprot\_B01.c file

const uint8\_t eoprot\_b01\_as\_entities\_numberofeach[eoas\_entities\_numberof] =

{

eoprot\_b01\_as\_strains\_numberof,

eoprot\_b01\_as\_maises\_numberof,

eoprot\_b01\_as\_extorque\_numberof

};

**Figure 46**: The analog sensors part of file eOprot\_B01.c file with changes due to addition of entity extorque.

Adding an entire endpoint

In here it is described what to if one wants to add a new endpoint helloworld to a board.

1. Create a new file EoHelloWorld.h with definition of all types required by the endpoint. In this file you must define also all the entities. See EoMotionControl.h as an example.
2. In file EoProtocol.h add eoprot\_endpoint\_helloworld in eOprot\_endpoint\_t and increment eoprot\_endpoints\_numberof. Then add in eOprot\_entity\_t all the entities defined in the endpoint and adjust value of eoprot\_entities\_numberof.
3. Create a new file EoProtocolHW.h with a value for eoprot\_entities\_hw\_numberof, add all the tags and rw mode for every entity, add declarations for a eoprot\_fun\_INITIALISE\_hw() and for all the INIT and UPDT.
4. Create e new file EoProtocolHW\_fun.c with the definition of all the above functions.
5. Create a new file EoProtocolHW\_rom.c with all is required. See EoProtocolMC\_rom.c as a template.

Finally one must change the files eOprot\_Bxx.[h, c] to tell if a given board support the endpoint and with how many entities. As reference example see a new fake file eOprot\_B01.[h, c].

The eOprot\_B01.h file

// - external dependencies ---------------------------------------------------------------------------

#include "EoCommon.h"

#include "EOnvSet.h"

#include "EoProtocol.h"

#include "EoProtocolAS.h"

#include "EoProtocolMC.h"

#include "EoProtocolMN.h"

#include "EoProtocolHW.h"

// - declaration of public user-defined types --------------------------------------------------------

enum { eoprot\_b01\_boardnumber = 0 };

enum { eoprot\_b01\_endpoints\_numberof = 4 };

**Figure 47**: The general part of new fake file eOprot\_B01.h file with hello world endpoint.

The eOprot\_B01.h file

// - helloworld

enum { eoprot\_b01\_hw\_entity00\_numberof = 3,

eoprot\_b01\_hw\_entity01\_numberof = 1 };

/\*\* @typedef typedef struct eOprot\_b01\_helloworld\_t;

@brief It is the container of entities helloworld endpoint of board b01.

\*\*/

typedef struct

{

eOhw\_entity00\_t entity00[eoprot\_b01\_hw\_entity00\_numberof];

eOhw\_entity01\_t entity01;

} eOprot\_b01\_helloworld\_t;

**Figure 48**: The hello world part of new fake file eOprot\_B01.h file with hello world endpoint.

The eOprot\_B01.h file

// - declaration of extern public variables, ... but better using use \_get/\_set instead --------------

// the configuration for the EOnvset object for protocol management

extern const eOnvset\_DEVcfg\_t eoprot\_b01\_nvsetDEVcfg;

// the number of entities for each endpoint organised as an array

extern const uint8\_t eoprot\_b01\_mn\_entities\_numberofeach[];

extern const uint8\_t eoprot\_b01\_mc\_entities\_numberofeach[];

extern const uint8\_t eoprot\_b01\_as\_entities\_numberofeach[];

extern const uint8\_t eoprot\_b01\_hw\_entities\_numberofeach[];

// - declaration of extern public functions ----------------------------------------------------------

/\*\* @fn extern eOresult\_t eoprot\_b01\_Initialise(eObool\_t islocal)

@brief Initialises the endpoints of this board by loading the number of

entities for each of them in the related endpoint file. As a result of that,

the functions which require a brd argument will return the correct value if called

with brd = eoprot\_b01\_boardnumber.

This function is called by the EOnvset because the eOnvset\_DEVcfg\_t contains a

pointer to it. However, it is made public so that it can be called independently

from the use of EOnvset.

@return eores\_OK if successful or eores\_NOK\_generic upon failure.

\*\*/

extern eOresult\_t eoprot\_b01\_Initialise(eObool\_t islocal);

**Figure 49**: The exported variables and functions new fake file eOprot\_B01.h file with hello world endpoint.

The eOprot\_B01.c file

static const eOnvset\_EPcfg\_t s\_eoprot\_b01\_theEPcfgs[] =

{

{ // management

EO\_INIT(.endpoint) eoprot\_endpoint\_management,

EO\_INIT(.dummy) 0,

EO\_INIT(.epram\_sizeof) sizeof(eOprot\_b01\_management\_t),

EO\_INIT(.fptr\_ram\_initialise) eoprot\_fun\_INITIALISE\_mn,

EO\_INIT(.protif) (eOnvset\_protocol\_Interface\_t\*)&eoprot\_eonvset\_Interface

},

{ // motion-control

EO\_INIT(.endpoint) eoprot\_endpoint\_motioncontrol,

EO\_INIT(.dummy) 0,

EO\_INIT(.epram\_sizeof) sizeof(eOprot\_b01\_motioncontrol\_t),

EO\_INIT(.fptr\_ram\_initialise) eoprot\_fun\_INITIALISE\_mc,

EO\_INIT(.protif) (eOnvset\_protocol\_Interface\_t\*)&eoprot\_eonvset\_Interface },

{ // analog-sensors

EO\_INIT(.endpoint) eoprot\_endpoint\_analogsensors,

EO\_INIT(.dummy) 0,

EO\_INIT(.epram\_sizeof) sizeof(eOprot\_b01\_analogsensors\_t),

EO\_INIT(.fptr\_ram\_initialise) eoprot\_fun\_INITIALISE\_as,

EO\_INIT(.protif) (eOnvset\_protocol\_Interface\_t\*)&eoprot\_eonvset\_Interface

},

{ // hello-world

EO\_INIT(.endpoint) eoprot\_endpoint\_helloworld,

EO\_INIT(.dummy) 0,

EO\_INIT(.epram\_sizeof) sizeof(eOprot\_b01\_helloworld\_t),

EO\_INIT(.fptr\_ram\_initialise) eoprot\_fun\_INITIALISE\_hw,

EO\_INIT(.protif) (eOnvset\_protocol\_Interface\_t\*)&eoprot\_eonvset\_Interface

}

}; EO\_VERIFYsizeof(s\_eoprot\_b01\_theEPcfgs, sizeof(eOnvset\_EPcfg\_t)\*(eoprot\_b01\_endpoints\_numberof));

**Figure 50**: The new fake file eOprot\_B01.c file with hello world endpoint.

The eOprot\_B01.c file

const uint8\_t eoprot\_b01\_hw\_entities\_numberofeach[eohw\_entities\_numberof] =

{

eoprot\_b01\_hw\_entity00\_numberof,

eoprot\_b01\_hw\_entity01\_numberof

};

**Figure 51**: The new fake file eOprot\_B01.c file with hello world endpoint.

The eOprot\_B01.c file

extern eOresult\_t eoprot\_b01\_Initialise(eObool\_t islocal)

{

// must initialise the mc, the mn, the ...

eoprot\_config\_endpoint\_entities(eoprot\_b01\_boardnumber, eoprot\_endpoint\_management,

eoprot\_b01\_mn\_entities\_numberofeach);

eoprot\_config\_endpoint\_entities(eoprot\_b01\_boardnumber, eoprot\_endpoint\_motioncontrol,

eoprot\_b01\_mc\_entities\_numberofeach);

eoprot\_config\_endpoint\_entities(eoprot\_b01\_boardnumber, eoprot\_endpoint\_analogsensors,

eoprot\_b01\_as\_entities\_numberofeach);

eoprot\_config\_endpoint\_entities(eoprot\_b01\_boardnumber, eoprot\_endpoint\_helloworld,

eoprot\_b01\_hw\_entities\_numberofeach);

if(eobool\_true == islocal)

{

eoprot\_config\_board\_local(eoprot\_b01\_boardnumber);

}

return(eores\_OK);

}

**Figure 52**: The new fake file eOprot\_B01.c file with hello world endpoint.

The eOprot\_B01.c file

EO\_VERIFYproposition(s\_eoprot\_b01\_mn\_val, 0 == eoprot\_endpoint\_management);

EO\_VERIFYproposition(s\_eoprot\_b01\_mc\_val, 1 == eoprot\_endpoint\_motioncontrol);

EO\_VERIFYproposition(s\_eoprot\_b01\_as\_val, 2 == eoprot\_endpoint\_analogsensors);

EO\_VERIFYproposition(s\_eoprot\_b01\_sk\_val, 3 == eoprot\_endpoint\_skin);

EO\_VERIFYproposition(s\_eoprot\_b01\_hw\_val, 4 == eoprot\_endpoint\_helloworld);

**Figure 53**: The new fake file eOprot\_B01.c file with hello world endpoint.