Veseo

MaCaco

Light Data Protocol

User and Developers Guide

VT-D04

Originally written by Dario Di Maio

Printed in Italy

Veseo

Napoli

Revision Control

Description	**Author**	**Date**	**Rev.**
First draft	Dario Di Maio	2013 May 10	0

Index

1 Purpose and Intended Audience

This document describe the functionality and the structure of the MaCaco Data Protocol, is intended for people with a basic knowledge of programming and telecommunication.

1.1 Download Source Code

Source code is included in Souliss project available for download at website http://www.souliss.net

1.2 Credits

The MaCaco code is developed by Veseo and contributors, is distributed as open source, you can use and modify it as per General Public License GPL v3.

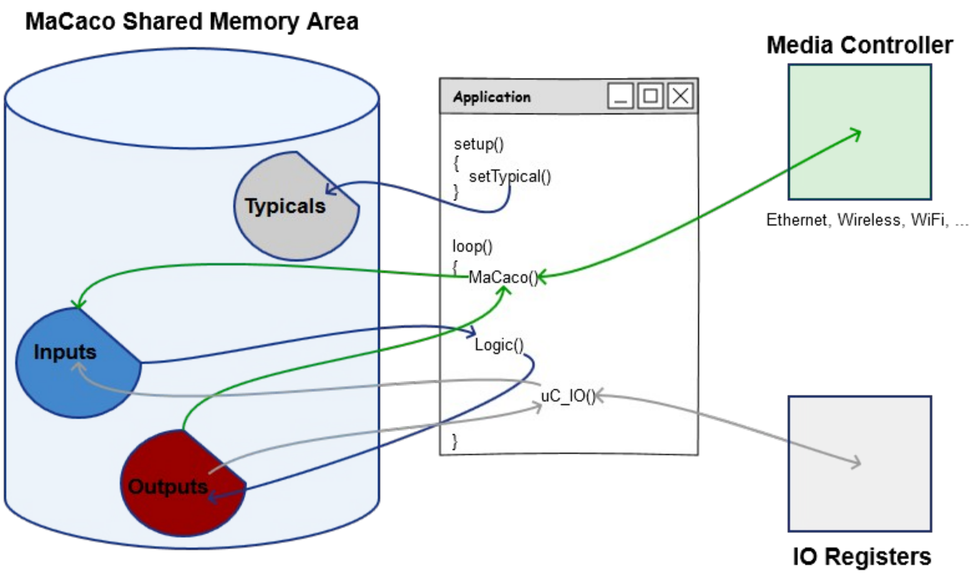
2 Overview

MaCaco is a binary event based data protocol designed for microcontrollers with few bytes of RAM, it define the data structure and the protocol to exchange the data. It has a peer-to-peer structure and request to be transported over a peer-to-peer layer, however it can be used a master/slave protocol at the application level.

MaCaco is almost a stateless protocol, means that while receiving a frame, it will include all the information to parse it and execute (if any) an action, is not required to store the history of the previous interaction.

2.1 Data Structure

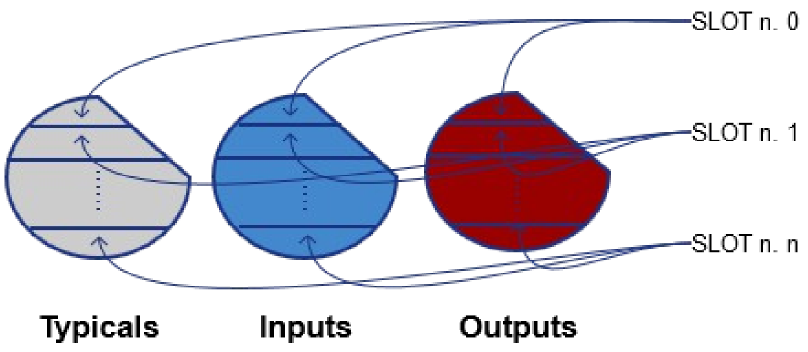
The Data Structure is an area of the whole RAM memory of the microcontroller used for sharing data with other devices that use the same protocol, in MaCaco the data structure and the protocol are merged together to give direct communication between devices without relay on an application layer.



The following definitions apply to the MaCaco Data Structure:

- Typical Logic (in short Typical) : Is a logic or a functionality that is standard for such type of devices, is addressed with a unique identification number. The identification numbers and the logic associated to a Typical are not defined in MaCaco, but at the application level. In the Data Structure is allocated an area for holding the defined Typicals.

- Input : Is the area of the Data Structure dedicated to holding the incoming information from other nodes or application within the same node, data included in the inputs are processed at the application level.
- Output : Is the area of the Data Structure dedicated to holding the output information to other nodes or application within the same node, output data are inserted and handled by the application level.
- SLOT : A slot is a partition of the Data Structure, at the application level slots are assigned to logics or functionality, addressing for each one of these one or more slots.

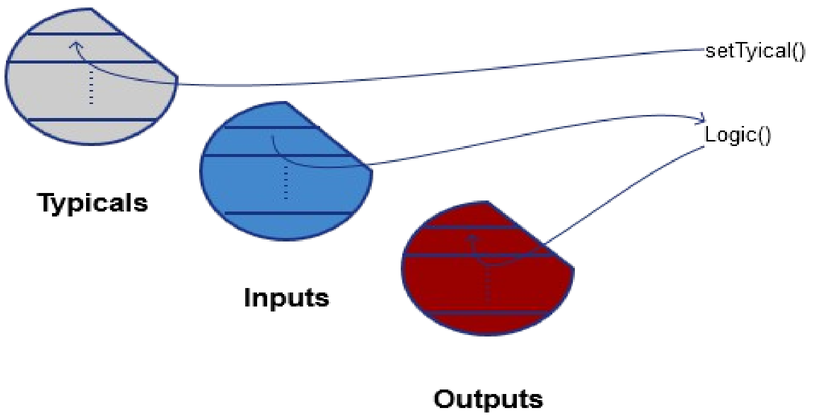


The MaCaco protocol relay on a common data structure that must share the same structure and dimension in all nodes involved in the communication, that data structure is common referred as “Shared Memory Area” because can be directly accessed and manipulated by all nodes.

2.2 Data Structure used at Application Level

The application level is a software that runs over MaCaco, it use MaCaco APIs to communicate with other nodes and use the MaCaco data structure for processing data and outputs.

At the application level a logic is defined with the assignment of a SLOT, each slot contains three location that include: Typical, Input and Output. In the area of memory used for Typical is stored an identification number defined at the application level, this number can be retrieved from any node in the network and give a direct understanding of the functionality performed by that node. The Input and Output are used to handle the in/out data for the relevant logic or functionality.



From the application point of view, a logic or functionality can be designed to read the inputs of the assigned slot and write the result in relevant outputs. Values into the inputs are automatically transferred while processing MaCaco based on interaction with other nodes in the network.

2.3 Protocol Structure

The MaCaco protocol relay on a transportation protocol (vNet for the Souliss project) and so hasn't addressing a port assignment, each frame has a fixed header and a variable payload.

Partially the structure of MaCaco is similar to Modbus, but include modification to support peer-to-peer, Modbus devices aren't compatible with MaCaco ones.

The header of a MaCaco frame has 5 bytes length grouped as follow, due its peer-to-peer nature there is no difference between the header of a request and an answer.

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)

The Functional Code is an identification code, while parsing the frame give information on what have to be done, example of functional codes are : Read request for digital values, Read answer for digital values, Ping request, Ping answer, Subscription request, Subscription answer. Full list of functional codes is later in this document.

The Put in is an identification code for the frame it-self, the node that starts the communication shall fill in that code and the answer will just echo that value. When receiving an incoming frame, the Put is used to link the received frame to the original request issued. Each device can define the Put in independently, in the Souliss implementation of MaCaco the Put in is pointer to the data area where the answer data shall be placed. In this way once received the answer frame, the data shall just be placed at the pointer specified at request time, a consistency check on the pointer is mandatory to avoid writing operation on not allowed data area.

The Start Offset is the first slot of the data area of the node that will receive the request, is used to identify data to read or write on the node, the number of data is specified with Number Of. A request frame is the header itself, an answer is the header plus a payload that contains an amount of data equal to Number Of.

Due to it peer-to-peer nature, each node could handle both answer and request, mostly of functional codes are paired and there is a boolean relationship between paired answer and request functional codes. As example, pair (Read request for digital values, Read answer for digital values) is (0x01, 0x11).

2.4 Communication Data Flow

Potentially any node can start a communication, MaCaco is connection-less and each iteration is build of just two messages : the answer and the request. There is no acknowledge and consistency check, if needed a proper transportation protocol can be used. Transporting MaCaco over TCP/IP gives both acknowledge and consistency check at that level. In Souliss MaCaco is transported over vNet.

There are three types of data flow in MaCaco : Polling, Pushing and Subscribing. In polling mode a node start a communication with a request, then it will receive the answer once. In pushing mode a node start a communication with a force request, and will not receive any answer. In subscribing mode a node start a communication with a subscribe request, that notify the interest in some data, after a request an answer follow immediately and later every time that an event occur on the subscribed node a new frame is sent automatically without any new request.

The subscribing allow reduction of data flow, data are transmitted only if necessary and this free resources on nodes. At same time the subscriber may result in long time without any data, so is not know if the subscribed node is still healty, since subscription are run-time and stored in RAM a reset of a subscribed node result in a broken subscription.

For this reason the subscriber node shall care of periodically renew of it subscriptions, there are no restriction on type of renew method to be used and each subscriber can use its own. In MaCaco the subscription is handled with an adaptive polling, for each subscription is defined an healthy value that start from a predefined value, lower is the healthy greater is the polling rate and lower is the timeout time.

This technique allow an hybrid data flow, at very begin is similar to a polling and if answers are received in time then the timeout is increased, for an healthy node the polling rate is low to hours. In case of dirty communication channel, the healthy value goes to a middle value that result in a greater rate that helps to recover missed frames due to channel errors.

In case of devices that need a short fault recognition time a chain watchdog is available, every node force a value into a timer of it neighbor, if this value is not forced in time the timer expire and an alarm is raised. Using the chain watchdog there is no direct handling of answer and request to understand if a node is healthy, because while pushing data with a force request there is no back answer.

2.5 *Extended Data Structure*

Is some cases is needed a collector node that gather data from all others in the network and make these information available to an endpoint, this solution is used to simplify configuration for user interfaces that shall not access the whole network but just a gateway node. For this reason in MaCaco is available an extended data structure and dedicated functional codes, the gateway node subscribe all nodes in its network and make that data available. This solution is not used in node-to-node communication, where is mandatory a direct communication to relevant node.

The Extended Data Structure is build as sum of N Data Structures where are located the data coming from the nodes in the network. The number N is a configuration parameter an represent the number of nodes in the network.

3 Functional Codes and Frames

In MaCaco there are several way to transfer data, these are represented by relevant Functional Codes, as described in previous chapter all frames has the same structure but each field could have a different use based on the relevant functional code.

The parsing of a MaCaco frame is unique and independent of the functional code, while the processing is functional code relevant.

3.1 Functional codes

In this paragraph are listed all functional codes that can be used for the communication between two MaCaco compliant nodes, functional codes are paired as per request and answer.

3.1.1 Read digital values

In reading mode the only data type accepted is the byte, so is not possible to receive only some bits from a node. Read request for digital values is addressed per byte.

Functional Codes :

- 0x01 // Read request for digital values
- 0x11 // Read answer for digital values

The most significant four bits of the functional code bytes are (if applicable) paired with a increased digit, all event digit in the most significant four bits of the functional code are used for request functional code, rather odds are for answers.

A request for three bytes starting from the first (0), with Put in identification 0xABCD looks like:

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x01	0xABCD	0x00	0x03

The byte referred in Start Offset is the first in the OUTPUT side of the data structure (or memory map).

The answer has the same header with the paired functional code and the requested data, in the example the data are 0x0A, 0xA0, 0xAA.

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x11	0xABCD		0x00	0x03
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x0A	0xA0	0xAA	-	-

The answer is parsed and received data are placed into the memory area addressed by 0xABCD for three bytes, the Put in can be directly the RAM pointer or can be related to it using a conversion table.

3.1.2 Read analog values

The analog values are handles as digital in reading requests, the use of different functional code is foreseen to allow direct identification of the data typology at the application level.

3.1.3 Subscription

The subscription is used to get data when events (data change) occur in the source of data, rather the previous functional code are time based and data are received for every request. The structure of the frame is the same of previous functional codes.

Functional codes:

- 0x05 // Subscription request
- 0x15 // Subscription answer

Subscription request :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x05	0xABCD	0x00	0x05

Subscription answer :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x15	0xABCD		0x00	0x05
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x0A	0xA0	0xAA	0x0A	0xA0

The answer is sent immediately after the request and at every change of the source data, in the Souliss implementation of MaCaco there is just one trigger for the whole data structure, so all subscribed are sent once if there is a data change also if this is not relevant the subscription. Is anyway possible use a trigger for each slot.

Subscription cannot be multiple, so a node cannot have two different subscription into the same subscribed node, the newest subscription always override the oldest.

3.1.4 Force

Force functional codes are push methods, that impose a value into a register of a node there are four type of push. Forces has no answer and contains data, so are odds codes. This functional code write into the inputs of a node, so the Start Offset is relevant the INPUT of the data structure (or memory map).

Functional codes :

- 0x13 // Force back a register value
- 0x14 // Force register value
- 0x16 // Force register value (bit-wise AND)
- 0x17 // Force register value (bit-wise OR)

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x14	0x0000		0x00	0x05
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x01	0x10	0x11	0x01	0x10

The functional code 0x16 and 0x17 allow only one byte in the payload, that byte is written using a bit-wise AND/OR boolean operation.

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x16, 0x17	0x0000		0x00	0x01
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x0A	-	-	-	-

The value to force is 0x0A or 00001010b, if using 0x16 (bit-wise AND) assuming that current value in first input register is 01010101b the result is 00000000b, while a 0x17 (bit-wise OR) result is 01011111b and a 0x14 result is 00001010b.

The functional code 0x13 is a force back, it has the same structure of other forces but the node that receive a force back doesn't write in its data structure the values, but send a force equal to the one received. A force back can be used for watchdog.

3.1.5 Ping

Ping is a basically network test functionality, can be used in run-time to identify if the communication with a node is properly working, it has no effect on the data structure (or memory map) and has no data.

Functional codes :

- 0x08 // Ping request
- 0x18 // Ping answer

Ping request :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x08	0x0000	0x00	0x00

Ping answer :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x18	0x0000	0x00	0x00

3.1.6 Error codes

In case of not recognized functional codes, illegal address and data out of range or subscription refused (table full), the node reply with an error. The frame is an echo of the request with one of these functional codes:

- 0x83 // Functional code not supported
- 0x84 // Data out of range,
- 0x85 // Subscription refused

3.2 Functional codes for gateway

A gateway is a node able to collect information from all others in the network and make these data available to external user interfaces, giving a unique collector point for the whole data in the network. A network of MaCaco nodes can have one or more gateways or can have no gateway nodes.

Getting information through a gateway is called “buffered” communication, because the gateway acts as a buffer in between the node and the rest of network. The buffered communication is developed to give an easy access for user interfaces and shall not be used for other type of nodes. When data are reached directly from the node that ows the required information the communication is referred as “direct”, this mode is the one described in the previous chapter.

3.2.1 Reason to use a buffered communication

The use of a gateway in between has as main advantage the abstraction of the network behind the gateway make easier the access of data and the configuration, this simplify the development of user interfaces that get the benefit of a binary event-based protocol.

A binary protocol use less RAM and is more suitable for devices with low performances, the event-based nature free resources and power that is extremely important on battery operated devices.

3.3 Functional codes for buffered communication

In this paragraph are listed all functional codes that can be used for the buffered communication with a MaCaco compliant user interface, functional codes are paired as per request and answer.

As per the direct communication, also the buffered assumes that the data structure of the nodes involved in the communication are equals. The buffered communication give access to the Extended Data Structure of the gateway node.

This type of interaction is similar to a master/slave one because a node doesn't need to gather data from an user interface either a node never start the communication with an user interface, is rather the user interface to start the communication and gather data from the node.

While is assumed that nodes that runs MaCaco share the same Data Structure, the user interfaces are supposed to have a dynamic structure of its Data Structure that shall be adapted to the node structure. At first run the user interface shall request the data structure dimensions and then subscribe data.

Starting from this point is assumed a data structure with 10 nodes each one composed of 8 slots (so 8+8+8 bytes to represent typicals, inputs and outputs).

3.3.1 State

The state is a subscription of data from a gateway node, this is similar to the subscription request.

Functional Codes :

- 0x21 // Read state request with subscription
- 0x31 // Read state answer

Only one subscription per user interface is allowed, every new request override the previous one.

State Request :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x21	0xABCD	0x00	0x01

State Answer :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x31	0xABCD		0x00	0x08
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x0A	0xA0	0xAA	0x0A	0xA0
Payload (byte 5)	Payload (byte 6)	Payload (byte 7)	Payload (byte 8)	Payload (byte 9)
0xAA	0xA0	0x0A	-	-

The Number Of values in the request is 0x01 and the number of bytes in the answer frame is 0x08, the buffered communication hasn't a fixed payload and the Start Offset and Number Of are no longer related to bytes.

In case of buffered communication the data transferred are no longer related to the bytes but to the nodes in the Extended Data Structure, the Start Offset is the first node to consider and the Number Of is the number of nodes to retrieve.

In the request of the example, is requested one node starting from the first one and the answer contains a number of bytes equal to the number of slots multiplied by the number of requested nodes, in the example is assumed that a node contains slots for 8 bytes. In the answer Number Of is the number of bytes in the payload.

3.3.2 Typicals

The typical is a polling request of the typicals contained into the extended data structure, as per State functional code the request is relevant the whole nodes contained into the extended data structure.

Functional Codes :

- 0x22 // Read typical logic request
- 0x32 // Read typical logic answer

Typicals Request :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x22	0xABCD	0x00	0x02

Are requested two nodes, the answer contains 16 bytes in the payload where the first 8 bytes are related to the first node (defined by the Start Offset) and the following eight are relevant the next node.

While parsing this frame shall be known the number of slots (and so of bytes) received for each requested node. This information, as clarified above, shall be retrieved by the user interface at first interaction before subscribing data.

Typicals Answer :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x32	0xABCD		0x00	0x10
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x11	0x11	0x12	0x12	0x00
Payload (byte 5)	Payload (byte 6)	Payload (byte 7)	Payload (byte 8)	Payload (byte 9)
0x00	0x00	0x00	0x13	0x13
Payload (byte 10)	Payload (byte 11)	Payload (byte 12)	Payload (byte 13)	Payload (byte 14)
0x13	0x13	0x31	0x11	0x11
Payload (byte 15)	Payload (byte 16)	Payload (byte 17)	Payload (byte 18)	Payload (byte 19)
0x00	-	-	-	-

3.3.3 Force

The force request (0x33) is used to write values into the data structure of the node, only one node per time can be addressed, the Start Offset parameter is referred to the node that shall be addressed rather the Number Of is relevant the bytes involved in.

The payload of the message is written into the memory of the addressed node always starting from the first byte of its inputs.

Force Request :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x33	0xABCD		0x03	0x08
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x00	0x01	0x00	0x00	0x00
Payload (byte 5)	Payload (byte 6)	Payload (byte 7)	Payload (byte 8)	Payload (byte 9)
0x00	0x00	0x00	-	-

In the example the second byte for node number three is forced to one, all the others are at zero (that is usually referred as neutral command).

An additional functional code (0x34) is used to force values based on the assigned typicals, the payload can be forced in all inputs slots that are assigned to the typical number specified in the Start Offset of the request.

As example, forcing all typicals 21 (0x15) to value 0x04 result in this Force by Typical Request :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x34	0xABCD		0x15	0x01
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x04	-	-	-	-

The force functional code of the buffered mode are not designed for node-to-node interaction, but can be used as same because nodes can be able to parse these functional codes also if are not gateways (depend on configuration). So is possible to force all values related to a typical with a single frame and without known the typicals in a node.

3.3.4 Healthy

The healthy functional code lets retrieve the healthy values for the nodes that are gathered into the gateway. The gateway shall build a subscription channel for each node that needs to be collected, so for each node there is an healthy parameters defining the quality of the communication and rate of polling for the subscription.

For each node there is a unique healthy values sized for one byte, so for this functional code bytes and nodes has the same meaning and data size.

Functional Codes:

- 0x25 // Nodes healthy request
- 0x35 // Nodes healthy answer

Healthy Request :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x25	0xABCD	0x00	0x08

Healthy Answer :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x35	0xABCD		0x00	0x08
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0xFF	0xF0	0xF4	0xFA	0xFE
Payload (byte 5)	Payload (byte 6)	Payload (byte 7)	Payload (byte 8)	Payload (byte 9)
0xDF	0xF0	0xFA	-	-

These values can be used in the user interface for displaying the communication performances or to ignore data from channel with low healthy (because that data could be old or wrong).

3.3.5 Database Structure

As discussed at begin of this chapter, the user interface need to know the sizing of the data structure of the nodes contained into the gateway, these information are used to size the internal database of the user interface and are also used for proper parsing of functional codes related to buffered mode.

Functional Codes :

- 0x26 // Database structure request
- 0x36 // Database structure answer

The answer has a fixed structure and contains the number of nodes configured in the gateway, the maximum number of allowed nodes, the number of slots for each node and the number of maximum subscription allowed by the gateway.

Database Structure Request :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x26	0xABCD	any	any

Database Structure Answer :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x36	0xABCD		0x00	0x04
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x05	0x0A	0x08	0x05	-

This functional code shall be used to size in run-time the database of the user interface, and together with the ping functional code gives the basic communication before the subscribing.

3.3.6 Data

The data functional code has same structure of State one but doesn't build a subscription, basically the user interface shall subscribe the nodes of its interest and periodically renew that subscription. Every time that the subscription is renewed the nodes answer with the full state and later with an answer for each data change.

Some times the user interface may require only partial data, not the whole database, also if there is no data change at the node. These data cannot be requested via State because it will override the previous subscription with a smallest one, in that cases the Data functional code can be used.

Data Request :

Functional Code (1 byte)	Put in (2 bytes)	Start Offset (1 byte)	Number Of (1 byte)
0x27	0xABCD	0x00	0x01

Data Answer :

Functional Code (1 byte)	Put in (2 bytes)		Start Offset (1 byte)	Number Of (1 byte)
0x37	0xABCD		0x00	0x08
Payload (byte 0)	Payload (byte 1)	Payload (byte 2)	Payload (byte 3)	Payload (byte 4)
0x0A	0xA0	0xAA	0x0A	0xA0
Payload (byte 5)	Payload (byte 6)	Payload (byte 7)	Payload (byte 8)	Payload (byte 9)
0xAA	0xA0	0x0A	-	-

This functional code shall not be used as periodically data refresh, but to get data out of events or to cover miss communication. For example, if an user is uncertain about the state of a given command, it can request a one shot refresh function that shall be based on this functional code.

The periodical refresh shall be a subscription, as explained before the subscription must be periodically refreshed in order to cover periodic data alignment out of events.

3.4 *Missed event communication*

A missed event communication shall be taken in account while developing a MaCaco nod or generally an event-based protocol. The event notification could be lost in several cases, also if using an acknowledged transport layer (such TCP).

For this reason MaCaco offers polling style functionality and a subscription mechanism that dynamically change the transmission type from polling to event-based, is mandatory a subscription renew but there is no constrain on the used method.

4 MaCaco over vNet

MaCaco protocol need a transportation protocol to reach the nodes of the network, it hasn't an addressing policy and can be transferred over every transportation protocol without restrictions.

In Souliss project a dedicated virtualization layer is used for data exchange between nodes, is called vNet and acts as bridging and routing layer between nodes, addressing all the communication jobs. In order to build a MaCaco compliant node is not required to transport it over vNet, but a connection between nodes can be established only if all use the same transportation layer. An application that would like to interact with a Souliss node, shall include MaCaco and vNet in order to be in communication.

In order to make simpler the development process for external application that would communicate with Souliss nodes, a lighter vNet implementation is allowed, refer to "vNet User and Developers Guide" for more information about.

Is not scope of this document to discuss about vNet protocol structure, so only a brief overview will be given here. The header of vNet is a 6 byte fixed structure:

Lenght (1 byte)	Port (1 byte)	Final Destination Address (2 bytes)	Original Destination Address (2 bytes)	Payload (n bytes)
0x06 + n	0x17	0x0011	0x0012	payload

Briefly as Lenght is referred the total size in byte of the frame, as Port is referred the identification number of the protocol that goes over vNet (0x17 for MaCaco), the addresses identify the nodes into the vNet network. The MaCaco protocol is contained into the payload of the frame.

4.1 MaCaco over vNet/IP

The IP version of vNet allow direct communication with any IP based device that run MaCaco over vNet, this type of interaction is used for Android smartphone and can be extended to any device that is located inside our outside the network.

As a virtualization layer, vNet include many drivers that can have their own structure. Frames are then unpacked and provided to the vNet layer as a standard frame, is always possible identify the driver to use from the address.

In case of IP drivers, and in most of vNet drivers, is added one extra byte with the length of the frame that is then embedded into and UDP/IP frame.

The example frame seen before, over IP will became as follow:

Lenght (1 byte)	Lenght (1 byte)	Port (1 byte)	Final Destination Address (2 bytes)	Original Destination Address (2 bytes)	Payload (n bytes)
0x06 + n + 1	0x06 + n	0x17	0x0011	0x0012	payload

Building a frame like that and including over an UDP/IP one, gives direct connection to a Souliss node.

From an IP node is possible to reach any other node, also if using a different communication media, to get this there are rules that apply for the addressing and linking the IP address to the vNet address, these shall be followed to allow a proper routing of the message over the nodes.

A capture of MaCaco over vNet/IP is available as reference for structure of data exchange.