



Protocol API
VARAN Client (Slave)

V1.0.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC100613API03EN | Revision 3 | English | 2013-06 | Released | Public

Table of Contents

1	Introduction.....	4
1.1	About this Document.....	4
1.2	List of Revisions	4
1.3	Functional Overview.....	5
1.4	System Requirements.....	5
1.5	Intended Audience	5
1.6	Specifications	6
1.7	Terms, Abbreviations and Definitions	7
1.8	References to Documents.....	7
1.9	Legal Notes	8
1.9.1	Copyright.....	8
1.9.2	Important Notes.....	8
1.9.3	Exclusion of Liability	9
1.9.4	Export.....	9
2	Fundamentals	10
2.1	General Access Mechanisms on netX Systems	10
2.2	Accessing the Protocol Stack by Programming the AP Task's Queue.....	11
2.2.1	Getting the Receiver Task Handle of the Process Queue	11
2.2.2	Meaning of Source- and Destination-related Parameters.....	11
2.3	Accessing the Protocol Stack via the Dual Port Memory Interface.....	12
2.3.1	Communication via Mailboxes.....	12
2.3.2	Using Source and Destination Variables correctly.....	12
2.3.3	Obtaining Information about the Communication Channel	15
3	Dual-Port Memory	17
3.1	Cyclic Data (Input/Output Data)	17
3.1.1	Input Process Data	17
3.1.2	Output Process Data	18
3.2	Acyclic Data (Mailboxes).....	18
3.2.1	General Structure of Messages or Packets for Non-Cyclic Data Exchange	19
3.2.2	Status & Error Codes	21
3.2.3	Differences between System and Channel Mailboxes	21
3.2.4	Send Mailbox.....	21
3.2.5	Receive Mailbox	21
3.2.6	Channel Mailboxes (Details of Send and Receive Mailboxes)	22
3.3	Status	22
3.3.1	Common Status.....	22
3.4	Control Block.....	28
4	Getting started/Configuration	29
4.1	Overview about Essential Functionality	29
4.2	Configuration Procedures	29
4.2.1	Using a Packet	29
4.3	Configuration Parameters	30
4.3.1	Behavior when receiving a Set Configuration Packet.....	33
4.3.2	Process Data (Input and Output).....	33
5	The Application Interface	34
5.1	The VARAN AP Task – VARAN Application Layer Task.....	34
5.1.1	VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_REQ/CNF	34
5.2	The VARAN Stack Task – VARAN Protocol Layer Task	39
5.2.1	VARAN_CLIENT_CMD_INIT_REQ/CNF	39
5.2.2	VARAN_CLIENT_CMD_CHECK_CFG_REQ/CNF	45
5.2.3	VARAN_CLIENT_CMD_GET_DIAG_REQ/CNF_T.....	47
5.2.4	VARAN_CLIENT_CMD_RESET_REQ/CNF	51
5.2.5	VARAN_CLIENT_CMD_GET_OUTPUT_REQ/CNF	53
5.2.6	VARAN_CLIENT_CMD_SET_INPUT_REQ/CNF	56
5.2.7	VARAN_CLIENT_CMD_CHANGE_STATE_REQ/CNF	58
5.2.8	VARAN_CLIENT_CMD_EVENT_IND/RES.....	60
6	LED Description.....	62
7	Status/Error Codes Overview.....	63

7.1	Status/Error Codes of the AP Task.....	63
7.2	Status/Error Codes of the VARAN Stack Task	64
8	Appendix	66
8.1	List of Tables	66
8.2	Contacts	67

1 Introduction

1.1 About this Document

This manual describes the user interface of the VARAN client (slave) implementation on the netX chip. The aim of this manual is to support the integration of devices based on the netX chip into own applications based on driver functions or direct access to the dual-port memory.

The general mechanism of data transfer, for example how to send and receive a message or how to perform a warmstart is independent from the protocol. These procedures are common to all devices and are described in the 'netX DPM Interface manual'

1.2 List of Revisions

Rev	Date	Name	Chapter	Revision
1	2010-06-30	RR	all	Created
2	2011-04-08	DL HH		Added packet definitions, error codes, etc. The number of communication event indications is reduced SetInput/GetOutput packet length definitions updated Status LEDs description added
3	2011-		all	A name for command defines corrected, data structure for set configuration packet changed.

Table 1: List of Revisions

1.3 Functional Overview

VARAN is a hard real time Ethernet based industrial network, which uses master/slave communication where only one master is allowed in the network. The manager (master device) sees the whole network as a 4GB RAM, from which every client (slave device) gets its 64 KB address space. The manager administers the network, provides automatic clients addressing and indicates if any kind of failure has happened. From the network user's point of view, sending and receiving data packets is brought to simple write/read operations from the desired client's address.

In order to provide easy bus topology realization, a client device might have a splitter-out port, besides the network-in port. Such is the case and with the developed client module.

1.4 System Requirements

This software package has following system requirements to its environment:

- netX-Chip as CPU hardware platform
- operating system rcX

1.5 Intended Audience

This manual is suitable for software developers with the following background:

- Knowledge of the programming language C
- Knowledge of the use of the real-time operating system rcX
- Knowledge of the Hilscher Task Layer Reference Model

1.6 Specifications

The data below applies to VARAN Client (Slave) firmware and stack version V1.0.x.x.

Features	Parameter
Maximum number of cyclic input data	128 bytes
Maximum number of cyclic output data	128 bytes
Memory	Read Output Memory 1, Write Input Memory 1
Functions	Memory Read Memory Write
Integrated 2 Port-Splitter for daisy chain topology	supported
Baud rate	100 MBit/s
Data transport layer	Ethernet II, IEEE 802.3
VARAN protocol version	1.1.1.0
Firmware/stack available for netX	netX50, netX100, netX500

Table 2: Technical Data of the VARAN Client (Slave) Protocol Stack

Configuration

- Packet to transfer configuration parameters
- netX Configuration Tool with 'inibatch.nxd'

Limitations

- integrated EMAC for IP data exchange with client application not supported
- SPI single commands (optional feature) not supported
- Memory area 2 is not yet supported. As long Memory area 2 is not supported the max. number of cyclic input/output data is 128 bytes.

1.7 Terms, Abbreviations and Definitions

Term	Description
Manager	The master device in the network and the only device to whom any slave should respond. It administers the network and keeps it safe from undesired participants.
Client	Slave device. Up to 65,280 are possible in a single network. A slave device cannot initiate communication with the manager or other clients, they only respond to commands coming from the manager.
Splitter	Similar function to the hub devices in an Ethernet network. The difference is that it deals with VARAN packets and supports switching the ports on and off

Table 3: Terms, Abbreviations and Definitions

All variables, parameters, and data used in this manual have the LSB/MSB (“Intel”) data format. This corresponds to the convention of the Microsoft C Compiler.

All IP addresses in this document have host byte order.

1.8 References to Documents

This document is based on the following specification respectively refers to the following documents:

- [1] VARAN-BUS-USER ORGANISATION: Design specification for VARAN. Revision 0.64, english, 2009-02-17
- [2] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, netX based products. Revision 10, english, 2011-02
- [3] Task Layer Reference Manual, Hilscher GmbH, 2005

Table 4: References to Documents

1.9 Legal Notes

1.9.1 Copyright

© 2010-2011 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

The images, photographs and texts in the accompanying material (user manual, accompanying texts, documentation, etc.) are protected by German and international copyright law as well as international trade and protection provisions. You are not authorized to duplicate these in whole or in part using technical or mechanical methods (printing, photocopying or other methods), to manipulate or transfer using electronic systems without prior written consent. You are not permitted to make changes to copyright notices, markings, trademarks or ownership declarations. The included diagrams do not take the patent situation into account. The company names and product descriptions included in this document may be trademarks or brands of the respective owners and may be trademarked or patented. Any form of further use requires the explicit consent of the respective rights owner.

1.9.2 Important Notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.9.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.9.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different countries, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Fundamentals

2.1 General Access Mechanisms on netX Systems

This chapter explains the possible ways to access a Protocol Stack running on a netX system:

1. By accessing the Dual Port Memory Interface directly or via a driver.
2. By accessing the Dual Port Memory Interface via a shared memory.
3. By interfacing with the Stack Task of the Protocol Stack.

The picture below visualizes these three ways:

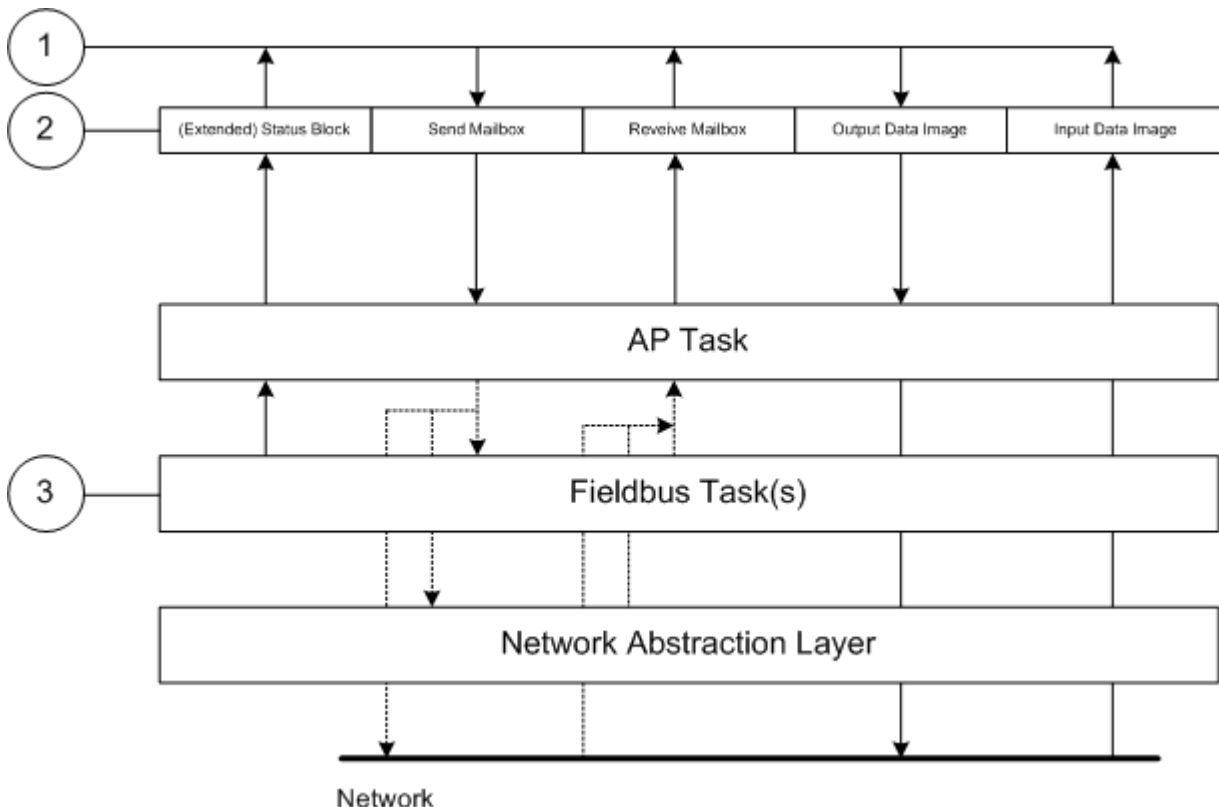


Figure 1: The 3 different ways to access a Protocol Stack running on a netX System

This chapter explains how to program the stack (alternative 3) correctly while the next chapter describes accessing the protocol stack via the dual-port memory interface according to alternative 1 (and 2, if the user application is executed on the netX chip in the context of the rcX operating system and uses the shared DPM). Finally, chapter *The Application Interface* on 34 page describes the entire interface to the protocol stack in detail.

Depending on you choose the stack-oriented approach or the Dual Port Memory-based approach; you will need either the information given in this chapter or those of the next chapter to be able to work with the set of functions described in chapter *The Application Interface* on 34 page. All of those functions use the four parameters `ulDest`, `ulSrc`, `ulDestId` and `ulSrcId`. This chapter and the next one inform about how to work with these important parameters.

2.2 Accessing the Protocol Stack by Programming the AP Task's Queue

In general, programming the AP task or the stack has to be performed according to the rules explained in the Hilscher Task Layer Reference Manual. There you can also find more information about the variables discussed in the following.

2.2.1 Getting the Receiver Task Handle of the Process Queue

To get the handle of the process queue of the AP-Task the Macro `TLR_QUE_IDENTIFY()` needs to be used. It is described in detail within section 10.1.9.3 of the Hilscher Task Layer Reference Model Manual. This macro delivers a pointer to the handle of the intended queue to be accessed (which is returned within the third parameter, `phQue`), if you provide it with the name of the queue (and an instance of your own task). The correct VARAN-queue names for accessing the VARAN client (Slave) AP-Task which you have to use as current value for the first parameter (`pszIdn`) is

VARAN Queue name	Description
"QUE_VRN_APP"	Name of the VARAN AP Task process queue
"QUE_VRN_CLIENT"	Name of the VARAN Stack Task process queue

Table 5: Names of Queues in ARAN Client (Slave) Firmware

The returned handle has to be used as value `ulDest` in all initiator packets the VARAN AP-Task intends to send to the VARAN DL-Task. This handle is the same handle that has to be used in conjunction with the macros like `TLR_QUE_SENDBUFFER_FIFO/LIFO()` for sending a packet to the respective task.

2.2.2 Meaning of Source- and Destination-related Parameters

The meaning of the source- and destination-related parameters is explained in the following table:

Variable	Meaning
<code>ulDest</code>	Application mailbox used for confirmation
<code>ulSrc</code>	Queue handle returned by <code>TLR_QUE_IDENTIFY()</code> as described above.
<code>ulSrcId</code>	Used for addressing at a lower level

Table 6: Meaning of Source- and Destination-related Parameters

For more information about programming the AP task's stack queue, please refer to [3]. Especially the following sections might be of interest in this context:

1. Chapter 7 "Queue-Packets"
2. Section 10.1.9 "Queuing Mechanism"

2.3 Accessing the Protocol Stack via the Dual Port Memory Interface

This chapter defines the application interface of the VARAN Client (Slave) Stack.

2.3.1 Communication via Mailboxes

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer to and from the netX.

- Send Mailbox
Packet transfer from host system to netX firmware
- Receive Mailbox
Packet transfer from netX firmware to host system

For more details about acyclic data transfer via mailboxes see section *Acyclic Data (Mailboxes)* on page 18 in this context, is described in detail in section *General Structure of Messages or Packets for Non-Cyclic Data Exchange* on page 19.

However, this section concentrates on correct addressing the mailboxes.

2.3.2 Using Source and Destination Variables correctly

2.3.2.1 How to use `ulDest` for Addressing rcX and the netX Protocol Stack by the System and Channel Mailbox

The preferred way to address the netX operating system rcX is through the system mailbox; the preferred way to address a protocol stack is through its channel mailbox. All mailboxes, however, have a mechanism to route packets to a communication channel or the system channel, respectively. Therefore, the destination identifier `ulDest` in a packet `tHeader` has to be filled in according to the targeted receiver. See the following example:

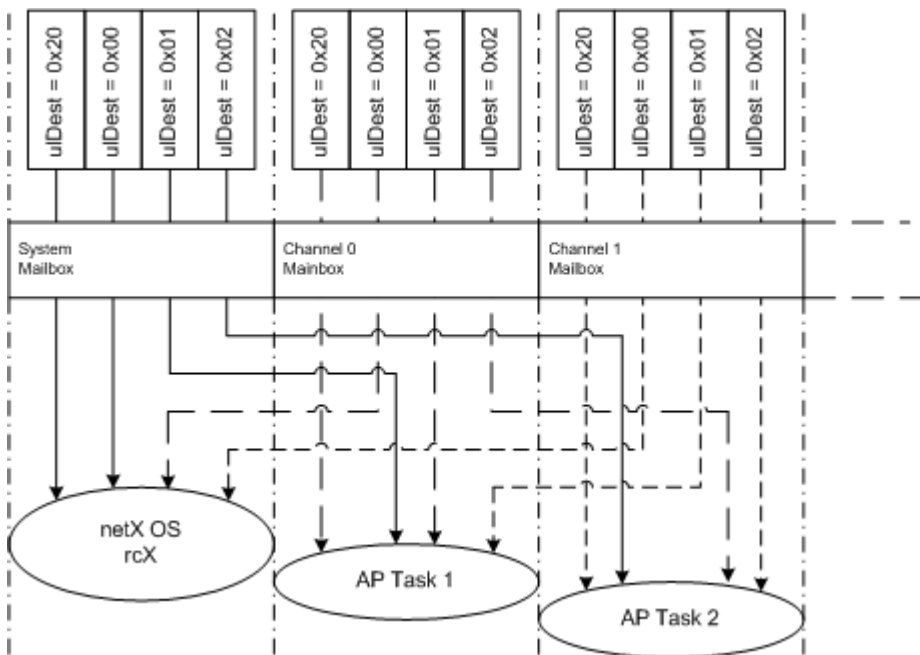


Figure 2: Use of `ulDest` in Channel and System Mailbox

For use in the destination queue handle, the tasks have been assigned to hexadecimal numerical values as described in the following table:

ulDest	Description
0x00000000	Packet is passed to the netX operating system rcX
0x00000001	Packet is passed to communication channel 0
0x00000002	Packet is passed to communication channel 1
0x00000003	Packet is passed to communication channel 2
0x00000004	Packet is passed to communication channel 3
0x00000020	Packet is passed to communication channel of the mailbox
else	Reserved, do not use

Table 7: Meaning of Destination-Parameter *ulDest*

The figure and the table above both show the use of the destination identifier *ulDest*.

A remark on the special channel identifier 0x00000020 (= *Channel Token*). The Channel Token is valid for any mailbox. That way the application uses the same identifier for all packets without actually knowing which mailbox or communication channel is applied. The packet stays 'local'. The system mailbox is a little bit different, because it is used to communicate to the netX operating system rcX. The rcX has its own range of valid commands codes and differs from a communication channel.

Unless there is a reply packet, the netX operating system returns it to the same mailbox the request packet went through. Consequently, the host application has to return its reply packet to the mailbox the request was received from.

2.3.2.2 How to use *ulSrc* and *ulSrcId*

Generally, a netX protocol stack can be addressed through its communication channel mailbox. The example below shows how a host application addresses a protocol stack running in the context of a netX chip. The application is identified by a number (#444 in this example). The application consists of three processes identified by the numbers #11, #22 and #33. These processes communicate through the channel mailbox with the AP task of the protocol stack. Have a look at the following figure:

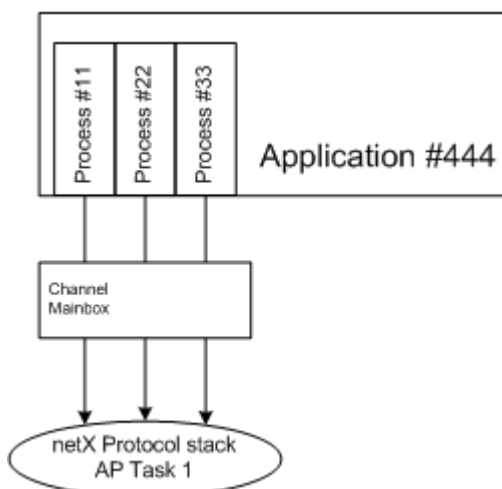


Figure 3: Using *ulSrc* and *ulSrcId*

Example:

This example applies to command messages initiated by a process in the context of the host application. If the process #22 sends a packet through the channel mailbox to the AP task, the packet tHeader has to be filled in as follows:

Object	Variable Name	Numeric Value	Description
Destination Queue Handle	ulDest	= 32 (0x00000020)	This value needs always to be set to 0x00000020 (the channel token) when accessing the protocol stack via the local communication channel mailbox.
Source Queue Handle	ulSrc	= 444	Denotes the host application (#444).
Destination Identifier	ulDestId	= 0	In this example it is not necessary to use the destination identifier.
Source Identifier	ulSrcId	= 22	Denotes the process number of the process within the host application and needs therefore to be supplied by the programmer of the host application.

Table 8: Example for correct Use of Source- and Destination-related Parameters

For packets through the channel mailbox, the application uses 32 (= 0x20, *Channel Token*) for the destination queue handler *ulDest*. The source queue handler *ulSrc* and the source identifier *ulSrcId* are used to identify the originator of a packet. The destination identifier *ulDestId* can be used to address certain resources in the protocol stack. It is not used in this example. The source queue handler *ulSrc* has to be filled in. Therefore its use is mandatory; the use of *ulSrcId* is optional.

The netX operating system passes the request packet to the protocol stack's AP task. The protocol stack then builds a reply to the packet and returns it to the mailbox. The application has to make sure that the packet finds its way back to the originator (process #22 in the example).

2.3.2.3 How to Route rcX Packets

To route an rcX packet the source identifier *ulSrcId* and the source queues handler *ulSrc* in the packet tHeader hold the identification of the originating process. The router saves the original handle from *ulSrcId* and *ulSrc*. The router uses a handle of its own choices for *ulSrcId* and *ulSrc* before it sends the packet to the receiving process. That way the router can identify the corresponding reply packet and matches the handle from that packet with the one stored earlier. Now the router replaces its handles with the original handles and returns the packet to the originating process.

2.3.3 Obtaining Information about the Communication Channel

A communication channel represents a part of the Dual Port Memory and usually consists of the following elements:

- Output Data Image - is used to transfer cyclic process data to the network (normal or high-priority)
- Input Data Image - is used to transfer cyclic process data from the network (normal or high-priority)
- Send Mailbox - is used to transfer non-cyclic data to the netX
- Receive Mailbox - is used to transfer non-cyclic data from the netX
- Control Block - allows the host system to control certain channel functions
- Common Status Block - holds information common to all protocol stacks
- Extended Status Block - holds protocol specific network status information

This section describes a procedure how to obtain useful information for accessing the communication channel(s) of your netX device and to check if it is ready for correct operation.

Proceed as follows:

- Start with reading the channel information block within the system channel (usually starting at address 0x0030).
- Then you should check the hardware assembly options of your netX device. They are located within the system information block following offset 0x0010 and stored as data type UINT16. The following table explains the relationship between the offsets and the corresponding xC Ports of the netX device:

Value	Description
0x0010	Hardware Assembly Options for xC Port[0]
0x0012	Hardware Assembly Options for xC Port[1]
0x0014	Hardware Assembly Options for xC Port[2]
0x0016	Hardware Assembly Options for xC Port[3]

Table 9: Address Assignment of Hardware Assembly Options

Check each of the hardware assembly options whether its value has been set to `RCX_HW_ASSEMBLY_ETHERNET = 0x0080`. If true, this denotes that this xC Port is suitable for running the protocol stack. Otherwise, this port is designed for another communication protocol. In most cases, xC Port[2] will be used for field-bus systems, while xC Port[0] and xC Port[1] are normally used for Ethernet communication.

- You can find information about the corresponding communication channel (0...3) under the following addresses:

Value	Description
0x0050	Communication Channel 0
0x0050	Communication Channel 0
0x0060	Communication Channel 1
0x0070	Communication Channel 2
0x0080	Communication Channel 3

Table 10: Addressing Communication Channel 0-3

In devices which support only one communication system which is usually the case (either a single field-bus system or a single standard for Industrial-Ethernet communication), always communication channel 0 will be used. In devices supporting more than one communication system you should also check the other communication channels.

- There you can find such information as the ID (containing channel number and port number) of the communication channel, the size and the location of the handshake cells, the overall number of blocks within the communication channel and the size of the channel in bytes. Evaluate this information precisely in order to access the communication channel correctly.

The information is delivered as follows:

Address	Data Type	Description
0x0050	UINT8	Channel Type = COMMUNICATION (must have the fixed value define RCX_CHANNEL_TYPE_COMMUNICATION = 0x05)
0x0051	UINT8	ID (Channel Number, Port Number)
0x0052	UINT8	Size / Position Of Handshake Cells
0x0053	UINT8	Total Number Of Blocks Of This Channel
0x0054	UINT32	Size Of Channel In Bytes
0x0058	UINT8[8]	Reserved (set to zero)

Table 11: Address Assignment of Communication Channels demonstrated at Communication Channel 0

These addresses correspond to communication channel 0, for communication channels 1, 2 and 3 you have to add an offset of 0x0010, 0x0020 or 0x0030 to the address values, respectively.

- Finally, you can access the communication channel using the addresses you determined previously. For more information how to do this, please refer to the netX DPM Manual, especially section 3.2 "Communication Channel".

3 Dual-Port Memory

All data in the dual-port memory is structured in blocks. According to their functions, these blocks use different data transfer mechanisms. For example, data transfer through mailboxes uses a synchronized handshake mechanism between host system and netX firmware. The same is true for IO data images, when a buffered handshake mode is configured. Other blocks, like the status block, are read by the host application and use no synchronization mechanism.

Types of blocks in the dual-port memory are outlined below:

- Mailbox - transfer non-cyclic messages or packages with a tHeader for routing information
- Data Area - holds the process image for cyclic IO data or user defined data structures
- Control Block - is used to signal application related state to the netX firmware
- Status Block - holds information regarding the current network state
- Change of State - collection of flags, that initiate execution of certain commands or signal a change of state

3.1 Cyclic Data (Input/Output Data)

The input block holds the process data image received **from** the network whereas the output block holds data sent **to** the network.

For the controlled / buffered mode, the protocol stack updates the process data in the internal input buffer for each valid bus cycle. Each IO block uses handshake bits for access synchronization. Input and output data block handshake operates independently from each other. When the application toggles the input handshake bit, the protocol stack copies the data from the internal buffer into the input data image of the dual-port memory. Now the application can copy data from the dual-port memory and then give control back to the protocol stack by toggling the appropriate input handshake bit. When the application/driver toggles the output handshake bit, the protocol stack copies the data from the output data image of the dual-port memory into the internal buffer. From there the data is transferred to the network. The protocol stack toggles the handshake bits back, indicating to the application that the transfer is finished and a new data exchange cycle may start. This mode guarantees data consistency over both input and output area.

3.1.1 Input Process Data

The input data block is used by Fieldbus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The input data image is used to receive cyclic data **from** the network.

The default size of the input data image is 5760 byte. However, not all available space is actually used by the protocol stack. Depending on the specific protocol, the area actually available for user data might be much smaller than 5760 byte. An input data block may or may not be available in the dual-port memory. It is always available in the default memory map (see the *netX Dual-Port Memory Manual*).

Input Data Image			
Offset	Type	Name	Description
0x2680	UINT8	abPd0Input[5760]	Input Data Image: Cyclic Data From The Network

Table 12: Input Data Image

3.1.2 Output Process Data

The output data block is used by Fieldbus and industrial Ethernet protocols that utilize a cyclic data exchange mechanism. The output data Image is used to send cyclic data from the host **to** the network.

The default size of the output data image is 5760 byte. However, not all available space is actually used by the protocol stack. Depending on the specific protocol, the area actually available for user data might be much smaller than 5760 byte. An output data block may or may not be available in the dual-port memory. It is always available in the default memory map (see the *netX DPM Manual*).

Output Data Image			
Offset	Type	Name	Description
0x1000	UINT8	abPd0Output[5760]	Output Data Image Cyclic Data To The Network

Table 13: Output Data Image

3.2 Acyclic Data (Mailboxes)

The mailbox of each communication channel has two areas that are used for non-cyclic message transfer.

- Send Mailbox - packet transfer from host system to firmware
- Receive Mailbox - packet transfer from firmware to host system

The send and receive mailbox areas are used by field bus protocols providing a non-cyclic data exchange mechanism. Another use of the mailbox system is to allow access to the firmware running on the netX chip itself for diagnostic and identification purposes. The send mailbox is used to transfer cyclic data **to** the network or **to** the firmware. The receive mailbox is used to transfer cyclic data **from** the network or **from** the firmware.

A send/receive mailbox may or may not be available in the communication channel. It depends on the function of the firmware whether or not a mailbox is needed. The location of the system mailbox and the channel mailbox is described in the *netX DPM Interface Manual*.

Note: Each mailbox can hold one packet at a time. The netX firmware stores packets that are not retrieved by the host application in a packet queue. This queue has limited space and may fill up so new packets maybe lost. To avoid these data loss situations, it is strongly recommended to empty the mailbox frequently, even if packets are not expected by the host application. Unexpected command packets should be returned to the sender with an Unknown Command in the status field; unexpected reply messages can be discarded.

3.2.1 General Structure of Messages or Packets for Non-Cyclic Data Exchange

The non-cyclic packets through the netX mailbox have the following structure:

Structure Information				Type of packet
Variable	Type	Value / Range	Description	
tHead - Structure Information				
ulDest	UINT32		Destination Queue Handle	
ulSrc	UINT32		Source Queue Handle	
ulDestId	UINT32		Destination Queue Reference	
ulSrcId	UINT32		Source Queue Reference	
ulLen	UINT32		Packet Data Length (In Bytes)	
ulId	UINT32		Packet Identification As Unique Number	
ulSta	UINT32		Status / Error Code	
ulCmd	UINT32		Command / Response	
ulExt	UINT32		Reserved	
ulRout	UINT32		Routing Information	
tData - Structure Information				
...	...		User Data Specific To The Command	

Table 14: General Structure of Packets for non-cyclic Data Exchange.

Some of the fields are mandatory; some are conditional; others are optional. However, the size of a packet is always at least 10 double-words or 40 bytes. Depending on the command, a packet may or may not have a data field. If present, the content of the data field is specific to the command, respectively the reply.

Destination Queue Handle

The *ulDest* field identifies a task queue in the context of the netX firmware. The task queue represents the final receiver of the packet and is assigned to a protocol stack. The *ulDest* field has to be filled out in any case. Otherwise, the netX operating system cannot route the packet. This field is mandatory.

Source Queue Handle

The *ulSrc* field identifies the sender of the packet. In the context of the netX firmware (inter-task communication) this field holds the identifier of the sending task. Usually, a driver uses this field for its own handle, but it can hold any handle of the sending process. Using this field is mandatory. The receiving task does not evaluate this field and passes it back unchanged to the originator of the packet.

Destination Identifier

The *ulDestId* field identifies the destination of an unsolicited packet from the netX firmware to the host system. It can hold any handle that helps to identify the receiver. Therefore, its use is mandatory for unsolicited packets. The receiver of unsolicited packets has to register for this.

Source Identifier

The *ulSrcId* field identifies the originator of a packet. This field is used by a host application, which passes a packet from an external process to an internal netX task. The *ulSrcId* field holds the handle of the external process. When netX operating system returns the packet, the application can identify the packet and returns it to the originating process. The receiving task on the netX does not evaluate this field and passes it back unchanged. For inter-task communication, this field is not used.

Length of Data Field

The *ulLen* field holds the size of the data field in bytes. It defines the total size of the packet's payload that follows the packet's tHeader. The size of the tHeader is not included in *ulLen*. So the total size of a packet is the size from *ulLen* plus the size of packet's tHeader. Depending on the command, a data field may or may not be present in a packet. If no data field is included, the length field is set to zero.

Identifier

The *ulId* field is used to identify a specific packet among others of the same kind. That way the application or driver can match a specific reply or confirmation packet to a previous request packet. The receiving task does not change this field and passes it back to the originator of the packet. Its use is optional in most of the cases. But it is mandatory for sequenced packets. Example: Downloading big amounts of data that does not fit into a single packet. For a sequence of packets the identifier field is incremented by one for every new packet.

Status / Error Code

The *ulState* field is used in response or confirmation packets. It informs the originator of the packet about success or failure of the execution of the command. The field may be also used to hold status information in a request packet.

Command / Response

The *ulCmd* field holds the command code or the response code, respectively. The command/response is specific to the receiving task. If a task is not able to execute certain commands, it will return the packet with an error indication. A command is always even (the least significant bit is zero). In the response packet, the command code is incremented by one indicating a confirmation to the request packet.

Extension

The extension field *ulExt* is used for controlling packets that are sent in a sequenced manner. The extension field indicates the first, last or a packet of a sequence. If sequencing is not required, the extension field is not used and set to zero.

Routing Information

The *ulRout* field is used internally by the netX firmware only. It has no meaning to a driver type application and therefore set to zero.

User Data Field

This field contains data related to the command specified in *ulCmd* field. Depending on the command, a packet may or may not have a data field. The length of the data field is given in the *ulLen* field.

3.2.2 Status & Error Codes

The following status and error codes can be returned in *ulState*. List of codes see manual named *netX Dual-Port Memory Interface*.

3.2.3 Differences between System and Channel Mailboxes

The mailbox system on netX provides a non-cyclic data transfer channel for field bus and industrial Ethernet protocols. Another use of the mailbox is allowing access to the firmware running on the netX chip itself for diagnostic purposes. There is always a send and a receive mailbox. Send and receive mailboxes utilize handshake bits to synchronize these data or diagnostic packages through the mailbox. There is a pair of handshake bits for both the send and receive mailbox.

The netX operating system rcX only uses the system mailbox.

The *system mailbox*, however, has a mechanism to route packets to a communication channel.

A *channel mailbox* passes packets to its own protocol stack only.

3.2.4 Send Mailbox

The send mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **send** mailbox is used to transfer non-cyclic data **to** the network or **to** the protocol stack.

The size is 1596 bytes for the send mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of packages that can be accepted.

3.2.5 Receive Mailbox

The receive mailbox area is used by protocols utilizing a non-cyclic data exchange mechanism. Another use of the mailbox system is to provide access to the firmware running on the netX chip itself. The **receive** mailbox is used to transfer non-cyclic data **from** the network or **from** the protocol stack.

The size is 1596 bytes for the receive mailbox in the default memory layout. The mailbox is accompanied by counters that hold the number of waiting packages (for the receive mailbox).

3.2.6 Channel Mailboxes (Details of Send and Receive Mailboxes)

Master Status			
Offset	Type	Name	Description
0x0200	UINT16	usPackagesAccepted	Packages Accepted Number of packages that can be accepted
0x0202	UINT16	usReserved	Reserved Set to 0
0x0204	UINT8	abSendMbx[1596]	Send Mailbox Non cyclic data to the network or to the protocol stack
0x0840	UINT16	usWaitingPackages	Packages waiting Counter of packages that are waiting to be processed
0x0842	UINT16	usReserved	Reserved Set to 0
0x0844	UINT8	abRecvMbx[1596]	Receive Mailbox Non cyclic data from the network or from the protocol stack

Table 15: Channel Mailboxes

Channel Mailboxes Structure

```
typedef struct tagNETX_SEND_MAILBOX_BLOCK
{
    UINT16 usPackagesAccepted;
    UINT16 usReserved;
    UINT8 abSendMbx[ 1596 ];
} NETX_SEND_MAILBOX_BLOCK;

typedef struct tagNETX_RECV_MAILBOX_BLOCK
{
    UINT16 usWaitingPackages;
    UINT16 usReserved;
    UINT8 abRecvMbx[ 1596 ];
} NETX_RECV_MAILBOX_BLOCK;
```

3.3 Status

A status block is present within the communication channel. It contains information about network and task related issues. In some respects, status and control block are used together in order to exchange information between host application and netX firmware. The application reads a status block whereas the control block is written by the application. Both status and control block have registers that use the *Change of State* mechanism (see also section 2.2.1 of the *netX Dual-Port-Memory manual*).

3.3.1 Common Status

The Common Status Block contains information that is the same for all communication channels. The start offset of this block depends on the size and location of the preceding blocks. The status block is always present in the dual-port memory.

All Implementations

The structure outlined below is common to all protocol stacks.

Common Status Structure Definition

Offset	Type	Name	Description
0x0010	UINT32	ulCommunicationCOS	Communication Change of State READY, RUN, RESET REQUIRED, NEW, CONFIG AVAILABLE, CONFIG LOCKED
0x0014	UINT32	ulCommunicationState	Communication State NOT CONFIGURED, STOP, IDLE, OPERATE
0x0018	UINT32	ulCommunicationError	Communication Error Unique Error Number According to Protocol Stack
0x001C	UINT16	usVersion	Version Version Number of this Diagnosis Structure
0x001E	UINT16	usWatchdogTime	Watchdog Timeout Configured Watchdog Time
0x0020	UINT16	usHandshakeMode	Handshake Mode Process Data Transfer Mode (see netX DPM Interface Manual)
0x0022	UINT16	usReserved	Reserved Set to 0
0x0024	UINT32	ulHostWatchdog	Host Watchdog Joint Supervision Mechanism: Protocol Stack Writes, Host System Reads
0x0028	UINT32	ulErrorCount	Error Count Total Number of Detected Error Since Power-Up or Reset
0x002C	UINT32	ulErrorLogInd	Error Log Indicator Total Number Of Entries In The Error Log Structure (not supported yet)
0x0030	UINT32	ulReserved[2]	Reserved Set to 0

Table 16: Common Status Structure Definition

Common Status Block Structure Reference

```
typedef struct NETX_COMMON_STATUS_BLOCK_Ttag
{
    UINT32    ulCommunicationCOS;
    UINT32    ulCommunicationState;
    UINT32    ulCommunicationError;
    UINT16    usVersion;
    UINT16    usWatchdogTime;
    UINT16    ausReserved[2];
    UINT32    ulHostWatchdog;
    UINT32    ulErrorCount;
    UINT32    ulErrorLogInd;
    UINT32    ulReserved[2];
    union
    {
        {
            NETX_MASTER_STATUS_T    tMasterStatus;    /* for master implementation */
            UINT32                    aulReserved[6];    /* otherwise reserved */
        } unStackDepended;
    }
} NETX_COMMON_STATUS_BLOCK_T;
```

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the *netX Change of State Command* flag in the netX communication flags register (see section 3.2.2.1 of the netX DPM Interface Manual). The application then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state (see section 3.2.2.2 of the netX DPM Interface Manual).

ulCommunicationCOS - netX writes, Host reads		
Bit	Short name	Name
31..7	unused, set to zero	
6	Restart Required Enable	RCX_COMM_COS_RESTART_REQUIRED_ENABLE
5	Restart Required	RCX_COMM_COS_RESTART_REQUIRED
4	Configuration New	RCX_COMM_COS_CONFIG_NEW
3	Configuration Locked	RCX_COMM_COS_CONFIG_LOCKED
2	Bus On	RCX_COMM_COS_BUS_ON
1	Running	RCX_COMM_COS_RUN
0	Ready	RCX_COMM_COS_READY

Table 17: Communication State of Change

Communication Change of State Flags (netX System ⇌ Application)

Bit	Definition / Description
0	Ready (RCX_COMM_COS_READY) 0 - ... 1 - The Ready flag is set as soon as the protocol stack is started properly. Then the protocol stack is awaiting a configuration. As soon as the protocol stack is configured properly, the Running flag is set, too.
1	Running (RCX_COMM_COS_RUN) 0 - ... 1 - The Running flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the Ready flag and the Running flag are set.
2	Bus On (RCX_COMM_COS_BUS_ON) 0 - ... 1 - The Bus On flag is set to indicate to the host system whether or not the protocol stack has the permission to open network connections. If set, the protocol stack has the permission to communicate on the network; if cleared, the permission was denied and the protocol stack will not open network connections.
3	Configuration Locked (RCX_COMM_COS_CONFIG_LOCKED) 0 - ... 1 - The Configuration Locked flag is set, if the communication channel firmware has locked the configuration database against being overwritten. Re-initializing the channel is not allowed in this state. To unlock the database, the application has to clear the Lock Configuration flag in the control block (see page 28).
4	Configuration New (RCX_COMM_COS_CONFIG_NEW) 0 - ... 1 - The Configuration New flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated. This flag may be set together with the Restart Required flag.

5	Restart Required (RCX_COMM_COS_RESTART_REQUIRED) 0 - ... 1 - The Restart Required flag is set when the channel firmware requests to be restarted. This flag is used together with the Restart Required Enable flag below. Restarting the channel firmware may become necessary, if a new configuration was downloaded from the host application or if a configuration upload via the network took place.
6	Restart Required Enable (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) 0 - ... 1 - The Restart Required Enable flag is used together with the Restart Required flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the Enable mechanism see section 2.3.2 of the netX DPM Interface Manual)).
7 ... 31	Reserved, set to 0

Table 18: Meaning of Communication Change of State Flags

Communication State (All Implementations)

The communication state field contains information regarding the current network status of the communication channel. Depending on the implementation, all or a subset of the definitions below is supported.

Value	Definition	Description
0x00000000	RCX_COMM_STATE_UNKNOWN	UNKNOWN
0x00000001	RCX_COMM_STATE_NOT_CONFIGURED	NOT_CONFIGURED
0x00000002	RCX_COMM_STATE_STOP	STOP
0x00000003	RCX_COMM_STATE_IDLE	IDLE
0x00000004	RCX_COMM_STATE_OPERATE	OPERATE

Table 19: Meaning of Communication State

Communication Channel Error (All Implementations)

This field holds the current error code of the communication channel. If the cause of error is resolved, the communication error field is set to zero (= RCX_SYS_SUCCESS) again. Not all of the error codes are supported in every implementation. Protocol stacks may use a subset of the error codes below.

Value	Definition	Description
0x00000000	RCX_SYS_SUCCESS	SUCCESS
Runtime Failures		
0xC000000C	RCX_E_WATCHDOG_TIMEOUT	WATCHDOG TIMEOUT
Initialization Failures		
0xC0000100	RCX_E_INIT_FAULT	(General) INITIALIZATION FAULT
0xC0000101	RCX_E_DATABASE_ACCESS_FAILED	DATABASE ACCESS FAILED
Configuration Failures		
0xC0000119	RCX_E_NOT_CONFIGURED	NOT CONFIGURED
0xC0000120	RCX_E_CONFIGURATION_FAULT	(General) CONFIGURATION FAULT
0xC0000121	RCX_E_INCONSISTENT_DATA_SET	INCONSISTENT DATA SET
0xC0000122	RCX_E_DATA_SET_MISMATCH	DATA SET MISMATCH
0xC0000123	RCX_E_INSUFFICIENT_LICENSE	INSUFFICIENT LICENSE
0xC0000124	RCX_E_PARAMETER_ERROR	PARAMETER ERROR
0xC0000125	RCX_E_INVALID_NETWORK_ADDRESS	INVALID NETWORK ADDRESS

Value	Definition	Description
0xC0000126	RCX_E_NO_SECURITY_MEMORY	NO SECURITY MEMORY

Table 20: Meaning of Communication Channel Error

Network Failures

Value	Definition	Description
0xC0000140	RCX_COMM_NETWORK_FAULT	(General) NETWORK FAULT
0xC0000141	RCX_COMM_CONNECTION_CLOSED	CONNECTION CLOSED
0xC0000142	RCX_COMM_CONNECTION_TIMEOUT	CONNECTION TIMED OUT
0xC0000143	RCX_COMM_LONELY_NETWORK	LONELY NETWORK
0xC0000144	RCX_COMM_DUPLICATE_NODE	DUPLICATE NODE
0xC0000145	RCX_COMM_CABLE_DISCONNECT	CABLE DISCONNECT

Table 21: Meaning of Network failures

Version (All Implementations)

The version field holds version of this structure. It starts with one; zero is not defined.

Value	Definition	Description
0x0001	RCX_STATUS_BLOCK_VERSION	STRUCTURE VERSION

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see section 4.13 of the netX DPM Interface Manual.

Host Watchdog (All Implementations)

The protocol stack supervises the host system using the watchdog function. If the application fails to copy the value from the device watchdog location (section 3.2.5 of the netX DPM Interface Manual) to the host watchdog location (section 3.2.4 of the netX DPM Interface Manual), the protocol stack assumes that the host system has some sort of problem and shuts down all network connections. For details on the watchdog function, refer to section 4.13 of the netX DPM Interface Manual.

Error Count (All Implementations)

This field holds the total number of errors detected since power-up, respectively after reset. The protocol stack counts all sorts of errors in this field no matter if they were network related or caused internally.

Error Log Indicator (All Implementations)

Not supported yet: The error log indicator field holds the number of entries in the internal error log. If all entries are read from the log, the field is set to zero.

Master Implementation

In addition to the common status block as outlined in the previous section, a master firmware maintains the additional structures for the administration of all slaves which are connected to the master. These are not discussed here as they are not relevant for the slave.

Slave Implementation

The slave firmware uses only the common structure as outlined in section 3.2.5.1. of the *netX DPM Interface Manual*. This is true for all protocol stacks.

3.4 Control Block

A control block is always present within the communication channel. In some respects, control and status block are used together in order to exchange information between host application and netX firmware. The control block is written by the application, whereas the application reads a status block. Both control and status block have registers that use the *Change of State* mechanism (see also section 2.2.1 of the *netX Dual-Port-Memory Manual*.)

The following gives an example of the use of control and status block. The host application wishes to lock the configuration settings of a communication channel to protect them against changes. The application sets the *Lock Configuration* flag in the control block to the communication channel firmware. As a result, the channel firmware sets the *Configuration Locked* flag in the status block (see below), indicating that the current configuration settings cannot be deleted, altered, overwritten or otherwise changed.

The control block of a dual-port memory features a watchdog function to allow the operating system running on the netX supervise the host application and vice versa. The control area is always present in the dual-port memory.

Control Block			
Offset	Type	Name	Description
0x0008	UINT32	ulApplicationCOS	Application Change Of State State Of The Application Program INITIALIZATION, LOCK CONFIGURATION
0x000C	UINT32	ulDeviceWatchdog	Device Watchdog: Host System Writes, Protocol Stack Reads

Table 22: Communication Control Block

Communication Control Block Structure

```
typedef struct NETX_CONTROL_BLOCK_Ttag
{
    UINT32 ulApplicationCOS;
    UINT32 ulDeviceWatchdog;
} NETX_CONTROL_BLOCK_T;
```

For more information concerning the Control Block please refer to the *netX DPM Interface Manual*.

4 Getting started/Configuration

This chapter gives an overview where to find some important information for starters and it explains the parameters of the VARAN Client firmware and the different ways how you can set them.

4.1 Overview about Essential Functionality

You can find the most commonly used functionality of the VARAN Client protocol API within the following sections of this document:

Topic	Section No.	Section Name
Configuration		VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_REQ/CNF
		VARAN_CLIENT_CMD_INIT_REQ/CNF
Cyclic data transfer		VARAN_CLIENT_CMD_GET_OUTPUT_REQ/CNF
		VARAN_CLIENT_CMD_SET_INPUT_REQ/CNF

4.2 Configuration Procedures

The following ways are available to configure the VARAN Client:

- By set configuration parameters packet
- By netX configuration and diagnostic utility

4.2.1 Using a Packet

In order to send the configuration parameters to the interface and to perform a warmstart subsequently, the packet can be sent to the protocol stack. For more information how to accomplish this, please refer to section VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_REQ/CNF of this manual.

4.3 Configuration Parameters

The following table contains relevant information about the parameters for the VARAN Client firmware such as an explanation of the meaning of the parameter and ranges of allowed values:

These parameters are especially relevant in the context of packet

VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_REQ/CNF – Initializing the VARAN Client Stack

Parameter	Meaning	Range of Value / Value
Bus Startup	<p>This parameter is represented by bit 0 of the system flags. The start of the device can be performed either application controlled or automatically:</p> <p>Automatic (0): Network connections are opened automatically without taking care of the state of the host application. Communication with a controller after a device start is allowed without BUS_ON flag, but the communication will be interrupted if the BUS_ON flag changes state to 0</p> <p>Application controlled (1): The channel firmware is forced to wait for the host application to wait for the Application Ready flag in the communication change of state register (see section 3.2.5.1 of the netX DPM Interface Manual). Communication with controller is allowed only with the BUS_ON flag.</p> <p>For more information concerning this topic see section "Controlled or Automatic Start" of the netX DPM Interface Manual.</p>	Application controlled, Automatic
Watchdog Time	<p>Watchdog time (in milliseconds)</p> <p>Time for the application program for retriggering the device watchdog. The application program monitoring has to be activated. A value of 0 indicates that the watchdog timer has been switched off and the application program monitoring is therefore deactivated. (value 0x00 turns off the watchdog timer)</p>	<p>Allowed values:</p> <p>20 ... 65535 (0xFFFF)</p> <p>Default value: 100 (0x64)</p>
<p>The parameters below are VARAN specific. Some of them are stored into the RAM as an image, emulating the presence of a SPI memory. After a successful initialization, the Manager tries to read/write/erase this SPI image by sending specific VARAN commands. At this stage read only commands are processed.</p> <p>The usage of memory area 2 is not implemented yet in the firmware layer.</p>		
Vendor Identifier	This value is specific for every single vendor.	<p>Allowed values:</p> <p>0 ... 0xFFFFFFFF</p> <p>Default value: 0x1D</p>
Device Identifier	This value is specific for every single device type. In case of an unknown device Id the manager rejects the client	<p>Allowed values:</p> <p>0 ... 0xFFFFFFFF</p> <p>Default value: 0x3FD</p>
License Number	The license number of the device	<p>Allowed values:</p> <p>0 ... 0xFFFFFFFF</p> <p>Default value: 0x0</p>

Parameter	Meaning	Range of Value / Value
Product Revision	The product revision of the device	Allowed values: 0 ... 0xFFFFFFFF Default value: 0x0
VendorName	This is a 64-byte ASCII array, which should contain the vendor name	Allowed values: ASCII characters range Default value: "Hilscher GmbH"
DeviceName	This is a 64-byte ASCII array, which should contain the device name	Allowed values: ASCII characters range Default value: "netX"
Serial lNumber	The serial number of the device	Allowed values: 0 ... 0xFFFFFFFF Default value: 0x0
Order Number	The order number of the device	Allowed values: 0 ... 0xFFFFFFFF Default value: 0x0
MemArealReadOffset	Reading offset of memory area 1	Allowed values: 0 ... 0xFFFF Default value: 0x2000
MemArealReadSize	Reading size of memory area 1	Allowed values: 0 ... 128 Default value: 128
MemArealWriteOffset	Writing offset of memory area 1	Allowed values: 0 ... 0xFFFF Default value: 0x2000
MemArealWriteSize	Writing size of memory area 1	Allowed values: 0 ... 128 Default value: 128
MemArea2ReadOffset (not supported, set to default value)	Reading offset of memory area 2	Allowed values: 0... 0xFFFF Default value: 0xFFFF
MemArea2ReadSize (not supported, set to default value)	Reading size of memory area 2	Allowed values: 0... 128 Default value: 0
MemArea2WriteOffset (not supported, set to default value)	Writing offset of memory area 2	Allowed values: 0 ... 0xFFFF Default value: 0xFFFF
MemArea2WriteSize (not supported, set to default value)	Writing size of memory area 2	Allowed values: 0 ... 128 Default value: 0

Parameter	Meaning	Range of Value / Value
ConfigFlags (not supported, set to default value)	HAL configuration flags Bit 0: Enables EMAC Bit 1: Enable Memory Area 2	Reserved bits: 0 ... 0xFFFFFFFFC Default value: 0
ClientWdgTime	Client watchdog time in ms. This parameter concerns the communication over the VARAN bus.	Allowed values: 0 ... 130 0 – Disables watchdog triggering Default value: 130
SyncOutPulsLen	Length of the sync out pulse in 10ns steps. (e.g. value 100 results as 10ns*100=1000ns=1µs pulse)	Allowed values: 10 ... n Default value: 100
SyncOut0Mode *)	This parameter controls wether SYNC OUT 0 ID 0 – This output is not available 3 – Time for data IN valid 4 – Time for data OUT valid 5 – Time for data IN/OUT valid	Allowed values: 0,3,4,5 Default value: 5
SyncOut0Flags	Sync out 0 flags: Bit 0: Enable/Disable output Bit 1: Polarity active high/low	Bit 0: 0 = Output disable Bit 0: 1 = Output enable Bit 1: 0 = active low Bit 1: 1 = active high Bit 2..31 reserved, set to 0 Default value: 0x00000001
SyncOut1Mode *)	This parameter controls wether SYNC OUT 1 ID 0 – This output is not available 3 – Time for data IN valid 4 – Time for data OUT valid 5 – Time for data IN/OUT valid	Allowed values: 0,3,4,5 Default value: 0
SyncOut1Flags	Sync out 1 flags: Bit 0: Enable/Disable output Bit 1: Polarity active high/low	Bit 0: 0 = Output disable Bit 0: 1 = Output enable Bit 1: 0 = active low Bit 1: 1 = active high Bit 2..31 reserved, set to 0 Default value: 0x00000000

Table 23: Meaning and allowed Values for Configuration Parameters

Note: *) Only following 3 combinations between SyncOut0Mode + SyncOut1Mode are reasonable:

"Time for IN/OUT valid (TIO)" (5)	+	"Disable" (0)
"Time for IN valid (TI)" (3)	+	"Time for OUT valid (TO)" (4)
"Time for OUT valid (TO)" (4)	+	"Time for IN valid (TI)" (3)

4.3.1 Behavior when receiving a Set Configuration Packet

The following rules apply for the behavior of the VARAN Client protocol stack when receiving a set configuration command:

- The configuration packets name is VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_REQ for the request packet and VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_CNF for the confirmation packet.
- The configuration data are checked for consistency and integrity.
- In case of failure no data are accepted.
- In case of success the configuration parameters are stored internally (within the RAM).
- The parameterized data will be activated only after a channel init has been performed.
- No automatic registration of the application at the stack happens.
- The confirmation packet VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_CNF only transfers simple status information, but does not repeat the whole parameter set.

4.3.2 Process Data (Input and Output)

The input and output data area is divided into the following sections:

- Input and Output Data for VARAN Client (Slave)

I/O Offset	Memory Area 1	Length (Byte)	Type
0	Output block	128	Write
0	Input block	128	Read

5 The Application Interface

This chapter defines the application interface of the VARAN stack.

The AP-Task's process queue is keeping track of all its incoming packets. It provides the communication channel for the underlying VARAN stack. Once, the stack communication is established, mailbox packets are sent to the AP task's process queue. On one hand every packet has to be evaluated in the AP-Task's context and corresponding actions be executed. On the other hand, Initiator-Services that are requested by the AP-Task itself are sent via predefined queue macros to the underlying stack queues via packets as well.

The following chapters are describing the packets that may be received from the host application or may be sent by the AP-Task.

5.1 The VARAN AP Task – VARAN Application Layer Task

5.1.1 VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_REQ/CNF

Once the application queue has been established, the VARAN Stack must be initialized with the appropriate bus related parameters. After their successful validation, they are buffered into the memory. Firmware starts using these settings barely after a channel init is done.

Packet Structure Reference

```

/** used @ ulConfigFlags */
#define VARAN_CLIENT_MSK_CFG_FLAG_EN_EMAC      (0x00000001)
#define VARAN_CLIENT_MSK_CFG_FLAG_EN_MEM2      (0x00000002)
#define VARAN_CLIENT_MSK_CFG_FLAG_RESERVED    (0xFFFFFFFF)

/** used @ ulSyncOut0Mode & ulSyncOut1Mode */
#define VARAN_CLIENT_SYNC_MODE_DISABLE         (0) /* This output is not available */
#define VARAN_CLIENT_SYNC_MODE_TI              (3) /* Time for data in valid */
#define VARAN_CLIENT_SYNC_MODE_T0              (4) /* Time for data out valid */
#define VARAN_CLIENT_SYNC_MODE_TIO             (5) /* Time for data in/out valid */

/** used @ ulSyncOut0Flags & ulSyncOut1Flags */
#define VARAN_CLIENT_MSK_SYNC_OUT_OE           (0x00000001)
#define VARAN_CLIENT_MSK_SYNC_OUT_ACTIVE_HIGH (0x00000002)
#define VARAN_CLIENT_MSK_SYNC_OUT_RESERVED    (0xFFFFFFFF)

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_STACK_DATA_INIT_REQ_Ttag{

    /** Device Identity configuration */
    TLR_UINT32  ulVendorId;           /* VendorID */
    TLR_UINT32  ulDeviceId;           /* DeviceID */
    TLR_UINT32  ulLicenceNumber;      /* LicNumber */
    TLR_UINT32  ulProductRevision;    /* Revision */
    TLR_UINT8   abVendorName[64];     /* Vendor Name */
    TLR_UINT8   abDeviceName[64];     /* Device Name */
    TLR_UINT32  ulSerialNumber;       /* Serial Number */
    TLR_UINT32  ulOrderNumber;        /* Order Number */
    TLR_UINT8   abIdReserved[8];      /* Resreved */

    /**Protocol Process Data configuration */
    TLR_UINT32  ulMemArealReadOffset;  /* MemAreal Read Offset */
    TLR_UINT32  ulMemArealReadSize;    /* MemAreal Read Size */
    TLR_UINT32  ulMemArealWriteOffset; /* MemAreal Write Offset */
    TLR_UINT32  ulMemArealWriteSize;   /* MemAreal Write Size */

    TLR_UINT32  ulMemArea2ReadOffset;  /* MemArea2 Read Offset */
    TLR_UINT32  ulMemArea2ReadSize;    /* MemArea2 Read Size */
    TLR_UINT32  ulMemArea2WriteOffset; /* MemArea2 Write Offset */

```

```

TLR_UINT32  ulMemArea2WriteSize;      /* MemArea2 Write Size */

/** Protocol Misc configuration */
TLR_UINT32  ulConfigFlags;             /* Configuration flags */
TLR_UINT32  ulClientWdgTime;          /* Client watchdog time */
TLR_UINT8   abReserved[32];           /* Reserved */

/** Protocol SYNC configuration */
TLR_UINT32  ulSyncOutPulsLen;         /* SYNC OUT puls length */
TLR_UINT32  ulSyncOut0Mode;           /* SYNC OUT 0 ID */
TLR_UINT32  ulSyncOut0Flags;          /* SYNC OUT 0 Flags */
TLR_UINT32  ulSyncOut1Mode;           /* SYNC OUT 1 ID */
TLR_UINT32  ulSyncOut1Flags;          /* SYNC OUT 1 Flags */

/** Protocol EMAC configuration */
TLR_UINT8   abEmacReserved[16];       /* Reserved for EMAC cfg */

}VARAN_CLIENT_STACK_DATA_INIT_REQ_T;

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_APP_DATA_INIT_REQ_Ttag{
    /** Application Interface Initialization */
    TLR_UINT32  ulSystemFlags;          /* System flags */
    TLR_UINT32  ulWdgTime;              /* App. Watchdog timeout */
    TLR_UINT32  ulAppMode;              /* App. Mode */
    TLR_UINT8   abReserved[20];        /* App. Param Reserved */
}VARAN_CLIENT_APP_DATA_INIT_REQ_T;

typedef __PACKED_PRE struct __PACKED_POST
VARAN_CLIENT_APP_DATA_SET_CONFIGURATION_REQ_Ttag{
    VARAN_CLIENT_APP_DATA_INIT_REQ_T  tAppInitData;
    VARAN_CLIENT_STACK_DATA_INIT_REQ_T tStackInitData;
}VARAN_CLIENT_APP_DATA_SET_CONFIGURATION_REQ_T;

#define VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_REQ_LEN
(sizeof(VARAN_CLIENT_APP_DATA_SET_CONFIGURATION_REQ_T))

typedef __PACKED_PRE struct __PACKED_POST
VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_REQ_Ttag{
    TLR_PACKET_HEADER_T      tHead;
    VARAN_CLIENT_APP_PCK_INIT_REQ_T  tData;
}VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_REQ_T;

```

Packet Description

Structure VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_REQ_T			Type: Request
Variable	Type	Value / Range	Description
Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of Application Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	300	VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_REQ_LEN - Packet data length in bytes
ulId	UINT32		Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0	
ulCmd	UINT32	0x00007000	VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
tData - Structure VARAN_CLIENT_APP_DATA_SET_CONFIGURATION_REQ_T			
tData.tApplInitData - Structure VARAN_CLIENT_APP_DATA_INIT_REQ_T			
ulSystemFlags	UINT32 (Bit field)	0 – automatic 1 – application controlled	System flags area
ulWdgTime	UINT32	Default value: 1000 Allowed values: 20..0xFFFF	The parameter ulWdgTime sets the watchdog time in multiples of 1msec. Value 0 turns off watchdog
ulAppMode	UINT32	0	Application mode 0 = VARAN_CLIENT_APP_MODE_IO
abReserved	UINT8[]	0	Reserved bytes, set to 0
tData.tStackInitData - Structure VARAN_CLIENT_STACK_DATA_INIT_REQ_T			
ulVendorId	UINT32	0x0 – 0xFFFFFFFF	Vendor ID
ulDeviceId	UINT32	0x0 – 0xFFFFFFFF	Device ID
ulLicenseNumber	UINT32	0x0 – 0xFFFFFFFF	Licencse number
ulProductRevision	UINT32	0x0 – 0xFFFFFFFF	Product revision number
abVendorName	UINT8[]	ASCII char	Vendor name
abDeviceName	UINT8[]	ASCII char	Device name
ulSerialNumber	UINT32	0x0 – 0xFFFFFFFF	Serial number
ulOrderNumber	UINT32	0x0 – 0xFFFFFFFF	Order number
abIdReserved	UINT8[]	0	Reserved

Structure VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_REQ_T			Type: Request
Variable	Type	Value / Range	Description
ulMemArea1ReadOffset	UINT32	0x0 – 0xFFFF Def: 0x2000	Memory Area 1 read offset
ulMemArea1ReadSize	UINT32	0 – 128 Def: 128	Memory Area 1 read size
ulMemArea1WriteOffset	UINT32	0x0 – 0xFFFF Def: 0x2000	Memory Area 1 write offset
ulMemArea1WriteSize	UINT32	0 – 128 Def: 128	Memory Area 1 write size
ulMemArea2ReadOffset	UINT32	0x0 – 0xFFFF Def: 0xFFFF	Memory Area 2 read offset
ulMemArea2ReadSize	UINT32	0 – 128 Def: 0	Memory Area 2 read size
ulMemArea2WriteOffset	UINT32	0x0 – 0xFFFF Def: 0xFFFF	Memory Area 2 write offset
ulMemArea2WriteSize	UINT32	0 – 128 Def: 0	Memory Area 2 write size
ulConfigFlags	UINT32	0x00 - 0x03	Configuration flags : Bit 0 – Enable EMAC (not supported yet) Bit 1 – Enable Memory area 2 (not supported yet) Bit 2..31 reserved, set to 0
ulClientWdgTime	UINT32	0 – 130 Def: 130	Client watchdog time in ms (0 -> watchdog off)
abReserved	UINT8[]	0	Reserved
ulSyncOutPulseLen	UINT32	10 ... n Def: 100	SYNC OUT pulse length in 10ns steps (Default value: 100 * 10ns = 1000ns = 1µs)
ulSyncOut0Mode	UINT32	0,3,4,5 Default value: 5	SYNC OUT 0 ID 0 – This output is not available 3 – Time for data IN valid 4 – Time for data OUT valid 5 – Time for data IN/OUT valid
ulSyncOut0Flags	UINT32	0x00 – 0x03	SYNC OUT 0 Flags Bit 0 – Syncout 0 output enable(1)/disable(0) Bit 1 – Syncout 0 polarity active high(1)/low(0) Bit 2..31 reserved, set to 0
ulSyncOut1Mode	UINT32	0,3,4,5 Default value: 0	SYNC OUT 1 ID 0 – This output is not available 3 – Time for data IN valid 4 – Time for data OUT valid 5 – Time for data IN/OUT valid
ulSyncOut1Flags	UINT32	0x00 – 0x03	SYNC OUT 1 Flags Bit 0 – Syncout 1 output enable(1)/disable(0) Bit 1 – Syncout 1 polarity active high(1)/low(0) Bit 2..31 reserved, set to 0

Table 24: VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_T – Set configuration Request

Packet Structure Reference

```
#define  VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_SIZE      (0)

typedef __PACKED_PRE struct __PACKED_POST
VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_Ttag{
    TLR_PACKET_HEADER_T          tHead;
}VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_T;
```

Packet Description

Structure VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0 ... $2^{32}-1$	Source End Point Identifier, untouched
ulLen	UINT32	0	VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_SIZE - Packet data length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet identification, untouched
ulSta	UINT32		See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00007001	VARAN_CLIENT_APP_CMD_SET_CONFIGURATION_CNF - Command
ulExt	UINT32	0	Extension, untouched
ulRout	UINT32	0	Routing, do not touch

Table 25: VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_T – Set configuration confirmation Packet

5.2 The VARAN Stack Task – VARAN Protocol Layer Task

5.2.1 VARAN_CLIENT_CMD_INIT_REQ/CNF

The VARAN protocol task is initialized by sending a VARAN_CLIENT_CMD_INIT_REQ_T packet to the stack queue. After its successful processing, the stack does not start cyclic communication automatically, but waits for receiving a VARAN_CLIENT_PCK_CHANGE_STATE_REQ_T request.

Packet Structure Reference

```

/** used @ ulConfigFlags */
#define VARAN_CLIENT_MSK_CFG_FLAG_EN_EMAC      (0x00000001)
#define VARAN_CLIENT_MSK_CFG_FLAG_EN_MEM2     (0x00000002)
#define VARAN_CLIENT_MSK_CFG_FLAG_RESERVED    (0xFFFFFFFFC)

/** used @ ulSyncOut0Mode & ulSyncOut1Mode */
#define VARAN_CLIENT_SYNC_MODE_DISABLE        (0) /* This output is not available */
#define VARAN_CLIENT_SYNC_MODE_TI            (3) /* Time for data in valid */
#define VARAN_CLIENT_SYNC_MODE_TO            (4) /* Time for data out valid */
#define VARAN_CLIENT_SYNC_MODE_TIO           (5) /* Time for data in/out valid */

/** used @ ulSyncOut0Flags & ulSyncOut1Flags */
#define VARAN_CLIENT_MSK_SYNC_OUT_OE          (0x00000001)
#define VARAN_CLIENT_MSK_SYNC_OUT_ACTIVE_HIGH (0x00000002)
#define VARAN_CLIENT_MSK_SYNC_OUT_RESERVED    (0xFFFFFFFFC)

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_STACK_DATA_INIT_REQ_Ttag{

    /** Device Identity configuration */
    TLR_UINT32  ulVendorId;           /* VendorID */
    TLR_UINT32  ulDeviceId;           /* DeviceID */
    TLR_UINT32  ulLicenseNumber;      /* LicNumber */
    TLR_UINT32  ulProductRevision;    /* Revision */
    TLR_UINT8   abVendorName[64];     /* Vendor Name */
    TLR_UINT8   abDeviceName[64];     /* Device Name */
    TLR_UINT32  ulSerialNumber;       /* Serial Number */
    TLR_UINT32  ulOrderNumber;        /* Order Number */
    TLR_UINT8   abIdReserved[8];      /* Resreved */

    /**Protocol Process Data configuration */
    TLR_UINT32  ulMemArealReadOffset;  /* MemAreal Read Offset */
    TLR_UINT32  ulMemArealReadSize;    /* MemAreal Read Size */
    TLR_UINT32  ulMemArealWriteOffset; /* MemAreal Write Offset */
    TLR_UINT32  ulMemArealWriteSize;   /* MemAreal Write Size */

    TLR_UINT32  ulMemArea2ReadOffset;  /* MemArea2 Read Offset */
    TLR_UINT32  ulMemArea2ReadSize;    /* MemArea2 Read Size */
    TLR_UINT32  ulMemArea2WriteOffset; /* MemArea2 Write Offset */
    TLR_UINT32  ulMemArea2WriteSize;   /* MemArea2 Write Size */

    /** Protocol Misc configuration */
    TLR_UINT32  ulConfigFlags;         /* Configuration flags */
    TLR_UINT32  ulClientWdgTime;       /* Client watchdog time */
    TLR_UINT8   abReserved[32];        /* Reserved */

    /** Protocol SYNC configuration */
    TLR_UINT32  ulSyncOutPulsLen;      /* SYNC OUT puls length */
    TLR_UINT32  ulSyncOut0Mode;        /* SYNC OUT 0 ID */
    TLR_UINT32  ulSyncOut0Flags;       /* SYNC OUT 0 Flags */
    TLR_UINT32  ulSyncOut1Mode;        /* SYNC OUT 1 ID */
    TLR_UINT32  ulSyncOut1Flags;       /* SYNC OUT 1 Flags */

    /** Protocol EMAC configuration */
    TLR_UINT8   abEmacReserved[16];    /* Reserved for EMAC cfg */

}VARAN_CLIENT_STACK_DATA_INIT_REQ_T;

```

```
/* *****  
 *   VARAN Slave - Packet - Init.Req  
 * ***** */  
  
#define VARAN_CLIENT_PCK_INIT_REQ_T_LEN    (sizeof(VARAN_CLIENT_STACK_DATA_INIT_REQ_T))  
typedef __PACKED_PRE struct __PACKED_POST  VARAN_CLIENT_PCK_INIT_REQ_Ttag  
{  
    TLR_PACKET_HEADER_T          tHead;  
    VARAN_CLIENT_STACK_DATA_INIT_REQ_T  tData;  
}VARAN_CLIENT_PCK_INIT_REQ_T;
```


Packet Description

Structure VARAN_CLIENT_PCK_INIT_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of Application Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	268	VARAN_CLIENT_PCK_INIT_REQ_T_LEN - Packet data length in bytes
ulId	UINT32		Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0	
ulCmd	UINT32	0x00006F00	VARAN_CLIENT_CMD_INIT_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
tData – Structure VARAN_CLIENT_STACK_DATA_INIT_REQ_T			
ulVendorId	UINT32	0x0 – 0xFFFFFFFF	Vendor ID
ulDeviceId	UINT32	0x0 – 0xFFFFFFFF	Device ID
ulLicenseNumber	UINT32	0x0 – 0xFFFFFFFF	Licencse number
ulProductRevision	UINT32	0x0 – 0xFFFFFFFF	Product revision number
abVendorName	UINT8[]	ASCII char	Vendor name
abDeviceName	UINT8[]	ASCII char	Device name
ulSerialNumber	UINT32	0x0 – 0xFFFFFFFF	Serial number
ulOrderNumber	UINT32	0x0 – 0xFFFFFFFF	Order number
abIdReserved	UINT8[]	0	Reserved
ulMemArealReadOffset	UINT32	0x0 – 0xFFFF Def: 0x2000	Memory Area 1 read offset
ulMemArealReadSize	UINT32	0 – 128 Def: 128	Memory Area 1 read size
ulMemArealWriteOffset	UINT32	0x0 – 0xFFFF Def: 0x2000	Memory Area 1 write offset
ulMemArealWriteSize	UINT32	0 – 128 Def: 128	Memory Area 1 write size

Structure VARAN_CLIENT_PCK_INIT_REQ_T			Type: Request
Variable	Type	Value / Range	Description
ulMemArea2ReadOffset	UINT32	0x0 – 0xFFFF Def: 0xFFFF	Memory Area 2 read offset
ulMemArea2ReadSize	UINT32	0 – 128 Def: 0	Memory Area 2 read size
ulMemArea2WriteOffset	UINT32	0x0 – 0xFFFF Def: 0xFFFF	Memory Area 2 write offset
ulMemArea2WriteSize	UINT32	0 – 128 Def: 0	Memory Area 2 write size
ulConfigFlags	UINT32	0x00 - 0x00000003	Configuration flags : Bit 0 – Enable EMAC (not supported yet) Bit 1 – Enable Memory area 2 (not supported yet)
ulClientWdgTime	UINT32	0 – 130 Def: 130	Client watchdog time in ms (0 -> watchdog off)
abReserved	UINT8[]	0	Reserved
ulSyncOutPulseLen	UINT32	10 ... n Def: 100	SYNC OUT pulse length in 10ns steps (Default value: 100 * 10ns = 1000ns = 1µs)
ulSyncOut0Mode	UINT32	0,3,4,5	SYNC OUT 0 ID 0 – This output is not available 3 – Time for data IN valid 4 – Time for data OUT valid 5 – Time for data IN/OUT valid
ulSyncOut0Flags	UINT32	0x00 – 0x03	SYNC OUT 0 Flags Bit 0 – Syncout 0 output enable/disable Bit 1 – Syncout 0 polarity active high/low
ulSyncOut1Mode	UINT32	0,3,4,5	SYNC OUT 1 ID 0 – This output is not available 3 – Time for data IN valid 4 – Time for data OUT valid 5 – Time for data IN/OUT valid
ulSyncOut1Flags	UINT32	0x00 – 0x03	SYNC OUT 1 Flags Bit 0 – Syncout 1 output enable/disable Bit 1 – Syncout 1 polarity active high/low

Table 26: VARAN_CLIENT_PCK_INIT_REQ_T – Init Request

Packet Structure Reference

```
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_INIT_CNF_Ttag{
    TLR_UINT32 ulTriBufHandleMemArea1Send; /* Tittle buffer handle MemArea1Send */
    TLR_UINT32 ulTriBufHandleMemArea1Recv; /* Tittle buffer handle MemArea1Recv */
    TLR_UINT32 ulTriBufHandleMemArea2Send; /* Tittle buffer handle MemArea2Send */
    TLR_UINT32 ulTriBufHandleMemArea2Recv; /* Tittle buffer handle MemArea2Recv */
}VARAN_CLIENT_PCK_INIT_CNF_T;

/*****
 *  VARAN Slave - Packet - Init.Cnf
 *****/

#define VARAN_CLIENT_PCK_INIT_CNF_T_LEN    (sizeof(VARAN_CLIENT_PCK_INIT_CNF_T))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_INIT_CNF_Ttag{
    TLR_PACKET_HEADER_T          tHead;
    VARAN_CLIENT_PCK_INIT_CNF_T  tData;
}VARAN_CLIENT_PCK_INIT_CNF_T;
```

Packet Description

Structure VARAN_CLIENT_PCK_INIT_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of Application Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	16	VARAN_CLIENT_PCK_INIT_CNF_T_LEN - Packet data length in bytes
ulId	UINT32		Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0 ... $2^{32}-1$	See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F01	VARAN_CLIENT_CMD_INIT_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
tData - Structure VARAN_CLIENT_STACK_DATA_INIT_CNF_T			
ulTriBufHandleMemArea1Send	UINT32		Handle to the triple buffer for sending data of memory area1
ulTriBufHandleMemArea1Recv	UINT32		Handle to the triple buffer for receiving data of memory area1
ulTriBufHandleMemArea2Send	UINT32		Handle to the triple buffer for sending data of memory area2
ulTriBufHandleMemArea2Recv	UINT32		Handle to the triple buffer for receiving data of memory area2

Table 27: VARAN_CLIENT_PCK_INIT_CNF_T – Confirmation Packet

5.2.2 VARAN_CLIENT_CMD_CHECK_CFG_REQ/CNF

This packet validates the content of VARAN_CLIENT_STACK_DATA_INIT_REQ_T structure, before using it as an argument of the VARAN_CLIENT_CMD_INIT_REQ_T request.

Packet Structure Reference

```
#define VARAN_CLIENT_PCK_CHECK_CONFIGURATION_REQ_SIZE
(sizeof(VARAN_CLIENT_STACK_DATA_INIT_REQ_T))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_CHECK_CONFIGURATION_REQ_Ttag{
    TLR_PACKET_HEADER_T          tHead;
    VARAN_CLIENT_STACK_DATA_INIT_REQ_T  tData;
}VARAN_CLIENT_PCK_CHECK_CONFIGURATION_REQ_T;
```

Packet Description

Structure VARAN_CLIENT_PCK_CHECK_CONFIGURATION_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of Application Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	268	VARAN_CLIENT_PCK_CHECK_CONFIGURATION_REQ_SIZE - Packet data length in bytes
ulId	UINT32		Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0	
ulCmd	UINT32	0x00006F0E	VARAN_CLIENT_CMD_CHECK_CFG_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
tData – Structure VARAN_CLIENT_STACK_DATA_INIT_REQ_T			

Table 28: VARAN_CLIENT_PCK_CHECK_CONFIGURATION_REQ_T– Check Configuration Request

Packet Structure Reference

```
#define VARAN_CLIENT_PCK_CHECK_CONFIGURATION_CNF_SIZE      (0)
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_CHECK_CONFIGURATION_CNF_Ttag{
    TLR_PACKET_HEADER_T          tHead;
}VARAN_CLIENT_PCK_CHECK_CONFIGURATION_CNF_T;
```

Packet Description

Structure VARAN_CLIENT_PCK_CHECK_CONFIGURATION_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of Application Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	VARAN_CLIENT_PCK_CHECK_CONFIGURATION_CNF_SIZE - Packet data length in bytes
ulId	UINT32		Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0 ... $2^{32}-1$	See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F0F	VARAN_CLIENT_CMD_CHECK_CFG_CNF - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons

Table 29: VARAN_CLIENT_PCK_CHECK_CONFIGURATION_CNF_T – Packet Status/Error

5.2.3 VARAN_CLIENT_CMD_GET_DIAG_REQ/CNF_T

The command VARAN_CLIENT_CMD_GET_DIAG_REQ is used to obtain diagnostic data from the VARAN Client.

Packet Structure Reference

```
#define DIAG_EXT (1)
#define DIAG_HAL (2)

typedef __PACKED_PRE struct __PACKED_POST VARANAPP_PCK_DIAGTYPE_Ttag{
    TLR_UINT32 ulDiagID;
}VARAN_CLIENT_PCK_GET_DIAG_ID_T;

#define VARAN_CLIENT_PCK_GET_DIAG_REQ_T_SIZE (sizeof(VARAN_CLIENT_PCK_GET_DIAG_ID_T))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_GET_DIAG_REQ_Ttag{
    TLR_PACKET_HEADER_T tHead;
    VARAN_CLIENT_PCK_GET_DIAG_ID_T tData;
}VARAN_CLIENT_PCK_GET_DIAG_REQ_T;
```

Packet Description

Structure VARAN_CLIENT_PCK_GET_DIAG_REQ			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of Application Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32		Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0x00000004	VARAN_CLIENT_PCK_GET_DIAG_REQ_T_SIZE - Packet data length in bytes
ulId	UINT32		Packet identification as unique number generated by the source process of the packet
ulSta	UINT32	0	
ulCmd	UINT32	0x00006F10	VARAN_CLIENT_CMD_GET_DIAG_REQ - Command
ulExt	UINT32	0	Extension not in use, set to zero for compatibility reasons
ulRout	UINT32	0	Routing not in use, set to zero for compatibility reasons
tData - Structure VARAN_CLIENT_PCK_GET_DIAG_ID_T			
ulDiagID	UINT32	1, 2	1 – Get extended diagnostic info 2 – Get HAL related diagnostic info

Table 30: VARAN_CLIENT_PCK_GET_DIAG_REQ_T - Get Diagnostic Request

Packet Structure Reference

```
typedef __PACKED_PRE struct __PACKED_POST  VARAN_CLIENT_EXT_DIAG_Ttag{

    /** VARAN_CLIENT_PCK_GET_DIAG_REQ */
    unsigned long ulGetDiagReq;
    unsigned long ulGetDiagCnfPos;
    unsigned long ulGetDiagCnfNeg;

    /** VARAN_CLIENT_PCK_GET_OUTPUT_REQ */
    unsigned long ulGetOutputReq;
    unsigned long ulGetOutputCnfPos;
    unsigned long ulGetOutputCnfNeg;

    /** VARAN_CLIENT_PCK_SET_INPUT_REQ */
    unsigned long ulSetInputReq;
    unsigned long ulSetInputCnfPos;
    unsigned long ulSetInputCnfNeg;

    /** VARAN_CLIENT_PCK_CHECK_CFG_REQ */
    unsigned long ulCheckCfgReq;
    unsigned long ulCheckCfgCnfPos;
    unsigned long ulCheckCfgCnfNeg;

    /** VARAN_CLIENT_PCK_INIT_REQ */
    unsigned long ulIntiReq;
    unsigned long ulInitCnfPos;
    unsigned long ulInitCnfNeg;

    /** VARAN_CLIENT_PCK_CHANGE_STATE_REQ */
    unsigned long ulChangeStateReq;
    unsigned long ulChangeStateCnfPos;
    unsigned long ulChangeStateCnfNeg;

}VARAN_CLIENT_EXT_DIAG_T;

typedef __PACKED_PRE struct __PACKED_POST  VARAN_DL_CP_DIAG_Ttag{

    unsigned long ulVRNCP_RXDV_UP_CNT_P0;
    unsigned long ulVRNCP_RAW_FRAMES_RCVD_OK_P0;
    unsigned long ulVRNCP_NESTED_FRAMES_RCVD_OK_P0;
    unsigned long ulVRNCP_FRAMES_RCVD_ERR_P0;
    unsigned long ulVRNCP_FRAME_FIN_OUT_CNT_P0;
    unsigned long ulVRNCP_URX_OVERFLOW_CNT_P0;
    unsigned long ulVRNCP_RX_ERR_STATISTIC_P0;
    unsigned long ulVRNCP_RXDV_UP_CNT_P1;
    unsigned long ulVRNCP_RAW_FRAMES_RCVD_OK_P1;
    unsigned long ulVRNCP_NESTED_FRAMES_RCVD_OK_P1;
    unsigned long ulVRNCP_FRAMES_RCVD_ERR_P1;
    unsigned long ulVRNCP_FRAME_FIN_OUT_CNT_P1;
    unsigned long ulVRNCP_URX_OVERFLOW_CNT_P1;
    unsigned long ulVRNCP_RX_ERR_STATISTIC_P1;
    unsigned long ulVRNCP_FRAMES_SEND_OK_CNT;
    unsigned long ulVRNCP_UTX_UFL_CNT_P0;
    unsigned long ulVRNCP_UTX_UFL_CNT_P1;
    unsigned long ulVRNCP_XPEC_NOT_RDY;
    unsigned long ulVRNCP_DEBUG_CNT_P0;
    unsigned long ulVRNCP_DEBUG_CNT_P1;
    unsigned long ulVRNCP_IP_FRAGMENTS_RECEIVED_OK;
    unsigned long ulVRNCP_IP_FRAGMENTS_DROPPED_DUE_LOW_RES;
    unsigned long ulVRNCP_IP_FRAGMENTS_DROPPED_DUE_DMA_NOT_RDY;
    unsigned long ulVRNCP_IP_FRAGMENTS_TRANSMITTED_OK;
    unsigned long ulVRNCP_IP_FRAGMENTS_NOT_SEND_DUE_LOW_RES;
    unsigned long ulVRNCP_IP_FRAMES_RECEIVED_BY_XPEC;
    unsigned long ulVRNCP_IP_FRAMES_RECEIVED_BY_HOST;
    unsigned long ulVRNCP_IP_FRAMES_TRANSMITTED_BY_HOST;
    unsigned long ulVRNCP_IP_FRAMES_TRANSMITTED_BY_XPEC;

}VARAN_DL_CP_DIAG_T;
```



```

typedef __PACKED_PRE struct __PACKED_POST  VARAN_DL_CS_DIAG_Ttag{

    unsigned long ulVRNCS_PLL_RESET_REQ;
    unsigned long ulVRNCS_SYNC0_RESET_REQ;
    unsigned long ulVRNCS_SYNC1_RESET_REQ;
    unsigned long ulVRNCS_COLLISION;
    unsigned long ulVRNCS_RX_NIBBLE_FIFO_ERR;
    unsigned long ulVRNCS_TX_NIBBLE_FIFO_ERR;
    unsigned long ulVRNCS_TIMER_A_EXP;
    unsigned long ulVRNCS_TIMER_B_EXP;
    unsigned long ulVRNCS_PLL_CYCLE_SEQ_ERR_CNT;

}VARAN_DL_CS_DIAG_T;

typedef __PACKED_PRE struct __PACKED_POST  VARAN_HAL_DIAG_Ttag{
    VARAN_DL_CP_DIAG_T tCPHalDiag;
    VARAN_DL_CS_DIAG_T tCSHalDiag;
}VARAN_CLIENT_HAL_DIAG_T;

typedef union{
    VARAN_CLIENT_EXT_DIAG_T      tExtDiag;
    VARAN_CLIENT_HAL_DIAG_T      tHALDiag;
}VARAN_CLIENT_DIAG_DATA_U;

typedef struct{
    TLR_UINT32                ulDiagID;
    VARAN_CLIENT_DIAG_DATA_U  uDiagnosis;
}VARAN_CLIENT_PCK_GET_DIAG_DATA_T;

#define VARAN_CLIENT_PCK_GET_DIAG_EXT_CNF_T_SIZE      (sizeof(VARAN_CLIENT_EXT_DIAG_T) +
sizeof(TLR_UINT32))
#define VARAN_CLIENT_PCK_GET_DIAG_HAL_CNF_T_SIZE      (sizeof(VARAN_CLIENT_HAL_DIAG_T) +
sizeof(TLR_UINT32))

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_GET_DIAG_CNF_Ttag{
    TLR_PACKET_HEADER_T      tHead;
    VARAN_CLIENT_PCK_GET_DIAG_DATA_T      tData;
}VARAN_CLIENT_PCK_GET_DIAG_CNF_T;

```

Packet Description

Structure Information VARAN_CLIENT_PCK_GET_DIAG_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0	Source End Point Identifier, untouched
ulLen	UINT32	Case ext diag : 76 Case HAL diag : 156	Case ext diag : VARAN_CLIENT_PCK_GET_DIAG_EXT_CNF_T_SIZE Case HAL diag : VARAN_CLIENT_PCK_GET_DIAG_HAL_CNF_T_SIZE - Packet Header Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F11	VARAN_CLIENT_CMD_GET_DIAG_CNF - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - structure VARAN_CLIENT_PCK_GET_DIAG_DATA_T			
ulDiagID	UINT32	x	Routing Information
In case there has been a get extended diagnostic request (ulDiagType = 1)			
tExtDiag	VARAN_CLIENT_EXT_DIAG_T		Extended diagnostic structure
In case there has been a get HAL related diagnostic request (ulDiagType = 2)			
tHALDiag	VARAN_CLIENT_HAL_DIAG_T		HAL diagnostic structure

Table 31: VARAN_CLIENT_PCK_GET_DIAG_CNF_T – Get diagnostic confirmation

5.2.4 VARAN_CLIENT_CMD_RESET_REQ/CNF

This request reinitalizes VARAN HAL with its current configuration and nullates HAL diagnostic. In case sent to the application task, it also nullates its diagnostic structure.

Packet Structure Reference

```
#define VARAN_CLIENT_PCK_RESET_REQ_SIZE (0)
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_RESET_REQ_Ttag{
    TLR_PACKET_HEADER_T tHead;
}VARAN_CLIENT_PCK_RESET_REQ_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_RESET_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	VARAN_CLIENT_PCK_RESET_REQ_SIZE - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		
ulCmd	UINT32	0x00006F02	VARAN_CLIENT_CMD_RESET_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information

Table 32: VARAN_CLIENT_PCK_RESET_REQ_T – Reset VARAN Client Request

Packet Structure Reference

```
#define  VARAN_CLIENT_PCK_RESET_CNF_SIZE      (0)
typedef __PACKED_PRE struct __PACKED_POST  VARAN_CLIENT_PCK_RESET_CNF_Ttag{
    TLR_PACKET_HEADER_T      tHead;
}VARAN_CLIENT_PCK_RESET_CNF_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_RESET_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination queue handle, untouched
ulSrc	UINT32		Source queue handle, untouched
ulDestId	UINT32	0	Destination End Point Identifier, untouched
ulSrcId	UINT32	0	Source End Point Identifier, untouched
ulLen	UINT32	0	VARAN_CLIENT_PCK_RESET_CNF_SIZE - Packet Header Length in bytes
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F03	VARAN_CLIENT_CMD_RESET_CNF - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information

Table 33: VARAN_CLIENT_PCK_RESET_CNF_T – Reset VARAN Client Confirmation

5.2.5 VARAN_CLIENT_CMD_GET_OUTPUT_REQ/CNF

The command `VARAN_CLIENT_CMD_GET_OUTPUT_REQ_T` can be used by the user application to get acyclically data from the output data image of the protocol task. But the preferred and most performed method which should be used to exchange input and output data is the triple buffer mechanism.

Packet Structure Reference

```
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_DATA_GET_OUTPUT_REQ_Ttag{
    TLR_UINT32 ulMemArea;
    TLR_UINT32 ulReserved;
    TLR_UINT32 ulLen;
}VARAN_CLIENT_PCK_DATA_GET_OUTPUT_REQ_T;

#define VARAN_CLIENT_PCK_GET_OUTPUT_REQ_SIZE
(sizeof(VARAN_CLIENT_PCK_DATA_GET_OUTPUT_REQ_T))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_GET_OUTPUT_REQ_Ttag{
    TLR_PACKET_HEADER_T tHead;
    VARAN_CLIENT_PCK_DATA_GET_OUTPUT_REQ_T tData;
}VARAN_CLIENT_PCK_GET_OUTPUT_REQ_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_GET_OUTPUT_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	VARAN_CLIENT_PCK_GET_OUTPUT_REQ_SIZE - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		
ulCmd	UINT32	0x00006F06	VARAN_CLIENT_CMD_GET_OUTPUT_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure VARAN_CLIENT_PCK_DATA_GET_OUTPUT_REQ_T			
ulMemArea	UINT32	0, 1 Default : 0	Number of the output memory area
ulReserved	UINT32	0	Reserved
ulLen	UINT32	0... 128	Number of bytes to be read from the output memory area

Table 34: VARAN_CLIENT_PCK_GET_OUTPUT_REQ_T – Get input packet request

Packet Structure Reference

```
#define VARAN_CLIENT_DATA_STATE_OK      (1)
#define VARAN_CLIENT_DATA_STATE_NOK    (0)
#define VARAN_CLIENT_BUFFER_SIZE       (128)

/** used @ ulMemArea */
#define VARAN_CLIENT_MEMORY_AREA_1     (0)
#define VARAN_CLIENT_MEMORY_AREA_2     (1)

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_DATA_GET_OUTPUT_CNF_Ttag{
    TLR_UINT32 ulMemArea;
    TLR_UINT32 ulDataState;
    TLR_UINT32 ulLen;
    TLR_UINT8  abData[VARAN_CLIENT_BUFFER_SIZE];
}VARAN_CLIENT_PCK_DATA_GET_OUTPUT_CNF_T;

#define VARAN_CLIENT_PCK_GET_OUTPUT_CNF_SIZE
(sizeof(VARAN_CLIENT_PCK_DATA_GET_OUTPUT_CNF_T)-VARAN_CLIENT_BUFFER_SIZE) /* + 1.128 */

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_GET_OUTPUT_CNF_Ttag{
    TLR_PACKET_HEADER_T          tHead;
    VARAN_CLIENT_PCK_DATA_GET_OUTPUT_CNF_T tData;
}VARAN_CLIENT_PCK_GET_OUTPUT_CNF_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_GET_OUTPUT_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12... 140	VARAN_CLIENT_PCK_GET_OUTPUT_CNF_SIZE + the length of the read data - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F07	VARAN_CLIENT_CMD_GET_OUTPUT_CNF – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure VARAN_CLIENT_PCK_DATA_GET_OUTPUT_CNF_T			
ulMemArea	UINT32	0, 1 Default : 0	Number of the output memory area
ulDataState	UINT32	0, 1	0 – Data is invalid 1 – Data is valid
ulLen	UINT32	0... 128	Number of bytes read from the output memory area
abData	UINT8[]		Data read from the output memort area

Table 35: VARAN_CLIENT_PCK_GET_OUTPUT_CNF_T – Get output packet confirmation

5.2.6 VARAN_CLIENT_CMD_SET_INPUT_REQ/CNF

The command `VARAN_CLIENT_CMD_SET_INPUT_REQ_T` is used by the user application to set acyclically data in the input data image of the protocol task. But the preferred and most performed method which should be used to exchange input and output data is the triple buffer mechanism.

Packet Structure Reference

```

/** used @ ulMemArea */
#define VARAN_CLIENT_MEMORY_AREA_1          (0)
#define VARAN_CLIENT_MEMORY_AREA_2          (1)

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_DATA_SET_INPUT_REQ_Ttag{
    TLR_UINT32 ulMemArea;
    TLR_UINT32 ulReserved;
    TLR_UINT32 ulLen;
    TLR_UINT8  abData[VARAN_CLIENT_BUFFER_SIZE];
}VARAN_CLIENT_PCK_DATA_SET_INPUT_REQ_T;

#define VARAN_CLIENT_PCK_SET_INPUT_REQ_SIZE
(sizeof(VARAN_CLIENT_PCK_DATA_SET_INPUT_REQ_T)-VARAN_CLIENT_BUFFER_SIZE) /* + 1.128 */
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_GET_INPUT_REQ_Ttag{
    TLR_PACKET_HEADER_T          tHead;
    VARAN_CLIENT_PCK_DATA_SET_INPUT_REQ_T  tData;
}VARAN_CLIENT_PCK_SET_INPUT_REQ_T;

```

Packet Description

Structure Information VARAN_CLIENT_PCK_SET_INPUT_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12...140	VARAN_CLIENT_PCK_SET_INPUT_REQ_SIZE + the number of bytes to be written(ulLen) - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		
ulCmd	UINT32	0x00006F04	VARAN_CLIENT_CMD_SET_INPUT_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure VARAN_CLIENT_PCK_DATA_SET_INPUT_REQ_T			
ulMemArea	UINT32	0, 1 Default : 0	Number of the output memory area
ulReserved	UINT32	0	Reserved
ulLen	UINT32	0... 128	Number of bytes written in the input memory area
abData	UINT8[]		Data to be written in the input memory area

Table 36: VARAN_CLIENT_PCK_SET_INPUT_REQ_T– Set input packet request

Packet Structure Reference

```
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_SET_INPUT_CNF_Ttag{
    TLR_UINT32 ulMemArea;
    TLR_UINT32 ulReserved;
    TLR_UINT32 ulLen;
}VARAN_CLIENT_PCK_DATA_SET_INPUT_CNF_T;

#define VARAN_CLIENT_PCK_SET_INPUT_CNF_SIZE
(sizeof(VARAN_CLIENT_PCK_DATA_SET_INPUT_CNF_T))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_GET_INPUT_CNF_Ttag{
    TLR_PACKET_HEADER_T          tHead;
    VARAN_CLIENT_PCK_DATA_SET_INPUT_CNF_T tData;
}VARAN_CLIENT_PCK_SET_INPUT_CNF_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_SET_INPUT_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	12	VARAN_CLIENT_PCK_SET_INPUT_CNF_SIZE - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F05	VARAN_CLIENT_CMD_SET_INPUT_CNF – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure VARAN_CLIENT_PCK_DATA_SET_INPUT_CNF_T			
ulMemArea	UINT32	Default : 0	Number of the input memory area
ulReserved	UINT32	0	Reserved
ulLen	UINT32	0... 128	Number of bytes written in the input memory area

Table 37: VARAN_CLIENT_PCK_SET_INPUT_CNF_T – Set input confirmation packet

5.2.7 VARAN_CLIENT_CMD_CHANGE_STATE_REQ/CNF

The VARAN_CLIENT_CMD_CHANGE_STATE_REQ is sent by the user to the protocol stack, when a change of the communication state is needed.

Packet Structure Reference

```
#define VARAN_CLIENT_STOP_COMM      (1)
#define VARAN_CLIENT_START_COMM    (2)

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_CHANGE_STATE_DATAtag{
    TLR_UINT32 ulState;
}VARAN_CLIENT_PCK_CHANGE_STATE_DATA;

#define VARAN_CLIENT_PCK_CHANGE_STATE_REQ_SIZE
(sizeof(VARAN_CLIENT_PCK_CHANGE_STATE_DATA))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_CHANGE_STATE_REQ_Ttag{
    TLR_PACKET_HEADER_T          tHead;
    VARAN_CLIENT_PCK_CHANGE_STATE_DATA  tData;
}VARAN_CLIENT_PCK_CHANGE_STATE_REQ_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_CHANGE_STATE_REQ_T			Type: Request
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS AP Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	VARAN_CLIENT_PCK_CHANGE_STATE_REQ_SIZE - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		
ulCmd	UINT32	0x00006F08	VARAN_CLIENT_CMD_CHANGE_STATE_REQ – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure VARAN_CLIENT_PCK_CHANGE_STATE_DATA			
ulState	UINT32	1, 2	The state, in which the Varan stack should go into : 1 – Stop communication 2 – Start communication

Table 38: VARAN_CLIENT_PCK_CHANGE_STATE_REQ_T – Change state packet request

Packet Structure Reference

```
#define VARAN_CLIENT_PCK_CHANGE_STATE_CNF_SIZE
(sizeof(VARAN_CLIENT_PCK_CHANGE_STATE_DATA))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_CHANGE_STATE_CNF_Ttag{
    TLR_PACKET_HEADER_T          tHead;
    VARAN_CLIENT_PCK_CHANGE_STATE_DATA tData;
}VARAN_CLIENT_PCK_CHANGE_STATE_CNF_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_CHANGE_STATE_CNF_T			Type: Confirmation
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS AP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	VARAN_CLIENT_PCK_CHANGE_STATE_CNF_SIZE - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F09	VARAN_CLIENT_CMD_CHANGE_STATE_CNF - Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure VARAN_CLIENT_PCK_CHANGE_STATE_DATA			
ulState	UINT32	1, 2	The current Varan stack communication state: 1 – Stopped communication 2 – Active communication

Table 39: VARAN_CLIENT_PCK_CHANGE_STATE_CNF_T- Set input confirmation packet

5.2.8 VARAN_CLIENT_CMD_EVENT_IND/RES

The command VARAN_CLIENT_CMD_EVENT_IND is sent by the protocol task to the user, in case a change of state event has happened.

Packet Structure Reference

```
#define VRN_STATE_COMM_ACTIVE                (0x00000001)
#define VRN_STATE_COMM_INACTIVE              (0x00000002)
#define VRN_STATE_COMM_ERROR                 (0x00000003)

typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_EVENT_DATA_Ttag{
    TLR_UINT32 ulState;
}VARAN_CLIENT_EVENT_DATA_T;

#define VARAN_CLIENT_EVENT_IND_SIZE          (sizeof(VARAN_CLIENT_EVENT_DATA_T))
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_PCK_EVENT_IND_Ttag{
    TLR_PACKET_HEADER_T                tHead;
    VARAN_CLIENT_EVENT_DATA_T          tData;
}VARAN_CLIENT_PCK_EVENT_IND_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_EVENT_IND_T			Type: Indication
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of APP Task Process Queue
ulSrc	UINT32		Source Queue-Handle of Stack Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	4	VARAN_CLIENT_EVENT_IND_SIZE - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		
ulCmd	UINT32	0x00006F0A	VARAN_CLIENT_CMD_EVENT_IND – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information
tData - Structure VARAN_CLIENT_EVENT_DATA_T			
ulState	UINT32		0x1 – Clients has gone online 0x2 – Clients has gone offline 0x3 – An error has occurred

Table 40: VARAN_CLIENT_PCK_EVENT_IND_T – Change state packet request

Packet Structure Reference

```
#define VARAN_CLIENT_EVENT_RES_SIZE (0)
typedef __PACKED_PRE struct __PACKED_POST VARAN_CLIENT_EVENT_RES_Ttag{
    TLR_PACKET_HEADER_T          tHead;
}VARAN_CLIENT_PCK_EVENT_RES_T;
```

Packet Description

Structure Information VARAN_CLIENT_PCK_EVENT_RES_T			Type: Response
Variable	Type	Value / Range	Description
tHead - Structure TLR_PACKET_HEADER_T			
ulDest	UINT32		Destination Queue-Handle of VRS Stack Task Process Queue
ulSrc	UINT32		Source Queue-Handle of VRS App Task Process Queue
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process
ulSrcId	UINT32	0	Source End Point Identifier, specifying the origin of the packet inside the Source Process
ulLen	UINT32	0	VARAN_CLIENT_EVENT_RES_SIZE - Packet data size
ulId	UINT32	0 ... $2^{32}-1$	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32		See section <i>Status/Error Codes Overview</i> from page 63.
ulCmd	UINT32	0x00006F0B	VARAN_CLIENT_CMD_EVENT_RES – Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	x	Routing Information

Table 41: VARAN_CLIENT_PCK_EVENT_RES_T – Set input confirmation packet

6 LED Description

The VARAN client uses two status LEDs (green and red), which indicate the user the current stack state. Below is a description table of the most common states and their corresponding LED status.

Varan Client state	Status	Description
RUN LED	On (green)	Configured, communication is active
	Blinking (green)	Configured, communication inactive
	Off	Not configured
ERR LED	On (red)	Error has occurred
	Blinking (red)	Not configured
	Off	Configured

Table 42: Description of the VARAN Client Status LEDs

7 Status/Error Codes Overview

7.1 Status/Error Codes of the AP Task

Value	Definition
	Description
0x00000000L	TLR_S_OK Status ok
0xC0000007L	TLR_E_INVALID_PACKET_LEN Packet length is invalid.
0xC0000009L	TLR_E_INVALID_PARAMETER Invalid ulState value
0xC0000181L	TLR_E_CONFIG_LOCK Configuration locked
0xC09D0000L	TLR_E_VARAN_CLIENT_AP_COMMAND_INVALID Invalid command received.

Table 43: Status/Error Codes VARAN AP-Task

7.2 Status/Error Codes of the VARAN Stack Task

Value	Definition Description
0x00000000L	TLR_S_OK Status ok
0xC09C0001	TLR_E_VRS_INVALID_CLIENT_WATCHDOG_VALUE Invalid Varan client watchdog timeout
0xC09C0002	TLR_E_VRS_INVALID_APP_WATCHDOG_VALUE Invalid DPM watchdog timeout
0xC09C0003	TLR_E_VRS_INVALID_MEMAREA1_R_OFFSET Invalid memory area 1 read offset
0xC09C0004	TLR_E_VRS_INVALID_MEMAREA1_W_OFFSET Invalid memory area 1 write offset
0xC09C0005	TLR_E_VRS_INVALID_MEMAREA1_R_SIZE Invalid memory area 1 read size
0xC09C0006	TLR_E_VRS_INVALID_MEMAREA1_W_SIZE Invalid memory area 1 write size
0xC09C0007	TLR_E_VRS_INVALID_MEMAREA2_R_OFFSET Invalid memory area 2 read offset
0xC09C0008	TLR_E_VRS_INVALID_MEMAREA2_W_OFFSET Invalid memory area 2 write offset
0xC09C0009	TLR_E_VRS_INVALID_MEMAREA2_R_SIZE Invalid memory area 2 read size
0xC09C000A	TLR_E_VRS_INVALID_MEMAREA2_W_SIZE Invalid memory area 2 write size
0xC09C000B	TLR_E_VRS_INVALID_SYNCOUT_PULSE_LEN Invalid syncout0 pulse length
0xC09C000C	TLR_E_VRS_INVALID_SYNCOUT_0_MODE Invalid syncout0 mode
0xC09C000D	TLR_E_VRS_INVALID_SYNCOUT_1_MODE Invalid syncout1 mode
0xC09C000E	TLR_E_VRS_INVALID_VENDOR_ID Invalid vendor ID
0xC09C000F	TLR_E_VRS_INVALID_DEVICE_ID Invalid device ID
0xC09C0010	TLR_E_VRS_INVALID_LICENSE_NUM Invalid license number
0xC09C0011	TLR_E_VRS_INVALID_SERIAL_NUM Invalid serial number
0xC09C0012	TLR_E_VRS_INVALID_ORDER_NUM Invalid order number
0xC09C0013	TLR_E_VRS_INVALID_MEM_AREA_ID Invalid memory area identifier
0xC09C0014	TLR_E_VRS_STACK_NOT_CONFIGURED Client is not configured yet
0xC09C0015	TLR_E_VRS_INIT_TRI_BUFFER Indicates inability to create triple buffers

Value	Definition Description
0xC09C0017	TLR_E_VRS_CYCLIC_COMM_SUSPENDED Cyclic communication is stopped or the client was unable to start communication, due to lost connection with the Manager
0xC09C0018	TLR_E_VRS_INVALID_DIAG_ID Invalid diagnostic ID
0xC09C0019	TLR_E_VRS_DIAG_STRUCT_SIZE_MISMATCH HAL diagnostic is not up to date with the local HAL related diagnostic structures
0xC09C001A	TLR_E_VRS_RESERVED_FIELD_NOT_NULLED Reserved field is not set to NULL
0xC09C001B	TLR_E_VRS_INVALID_APP_MODE Invalid application mode
0xC09C001C	TLR_E_VRS_INVALID_CONFIG_FLAGS Invalid configuration flags

Table 44: Status/Error Codes VARAN Stack-Task

8 Appendix

8.1 List of Tables

Table 1: List of Revisions	4
Table 2: Technical Data of the VARAN Client (Slave) Protocol Stack.....	6
Table 3: Terms, Abbreviations and Definitions.....	7
Table 4: References to Documents	7
Table 5: Names of Queues in ARAN Client (Slave) Firmware	11
Table 6: Meaning of Source- and Destination-related Parameters.....	11
Table 7: Meaning of Destination-Parameter ulDest.....	13
Table 8: Example for correct Use of Source- and Destination-related Parameters	14
Table 9: Address Assignment of Hardware Assembly Options	15
Table 10: Addressing Communication Channel 0-3.....	15
Table 11: Address Assignment of Communication Channels demonstrated at Communication Channel 0.....	16
Table 12: Input Data Image.....	17
Table 13: Output Data Image	18
Table 14: General Structure of Packets for non-cyclic Data Exchange.....	19
Table 15: Channel Mailboxes.....	22
Table 16: Common Status Structure Definition	23
Table 17: Communication State of Change.....	24
Table 18: Meaning of Communication Change of State Flags	25
Table 19: Meaning of Communication State	25
Table 20: Meaning of Communication Channel Error	26
Table 21: Meaning of Network failures	26
Table 22: Communication Control Block	28
Table 23: Meaning and allowed Values for Configuration Parameters.....	32
Table 24: VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_T – Set configuration Request.....	37
Table 25: VARAN_CLIENT_APP_PCK_SET_CONFIGURATION_CNF_T – Set configuration confirmation Packet.....	38
Table 26: VARAN_CLIENT_PCK_INIT_REQ_T – Init Request	42
Table 27: VARAN_CLIENT_PCK_INIT_CNF_T – Confirmation Packet	44
Table 28: VARAN_CLIENT_PCK_CHECK_CONFIGURATION_REQ_T– Check Configuration Request.....	45
Table 29: VARAN_CLIENT_PCK_CHECK_CONFIGURATION_CNF_T – Packet Status/Error	46
Table 30: VARAN_CLIENT_PCK_GET_DIAG_REQ_T – Get Diagnostic Request.....	47
Table 31: VARAN_CLIENT_PCK_GET_DIAG_CNF_T – Get diagnostic confirmation.....	50
Table 32: VARAN_CLIENT_PCK_RESET_REQ_T – Reset VARAN Client Request.....	51
Table 33: VARAN_CLIENT_PCK_RESET_CNF_T – Reset VARAN Client Confirmation	52
Table 34: VARAN_CLIENT_PCK_GET_OUTPUT_REQ_T – Get input packet request	53
Table 35: VARAN_CLIENT_PCK_GET_OUTPUT_CNF_T – Get output packet confirmation	55
Table 36: VARAN_CLIENT_PCK_SET_INPUT_REQ_T– Set input packet request.....	56
Table 37: VARAN_CLIENT_PCK_SET_INPUT_CNF_T – Set input confirmation packet	57
Table 38: VARAN_CLIENT_PCK_CHANGE_STATE_REQ_T – Change state packet request	58
Table 39: VARAN_CLIENT_PCK_CHANGE_STATE_CNF_T– Set input confirmation packet	59
Table 40: VARAN_CLIENT_PCK_EVENT_IND_T – Change state packet request	60
Table 41: VARAN_CLIENT_PCK_EVENT_RES_T – Set input confirmation packet	61
Table 42: Description of the VARAN Client Status LEDs	62
Table 43: Status/Error Codes VARAN AP-Task.....	63
Table 44: Status/Error Codes VARAN Stack-Task.....	65

8.2 Contacts

Headquarters

Germany

Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support

Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China

Hilscher Ges.f.Systemaut. mbH
Shanghai Representative Office
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support

Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France

Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support

Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India

Hilscher India Pvt. Ltd.
New Delhi - 110 025
Phone: +91 11 40515640
E-Mail: info@hilscher.in

Italy

Hilscher Italia srl
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support

Phone: +39/02 25007068
E-Mail: it.support@hilscher.com

Japan

Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support

Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea

Hilscher Korea Inc.
Suwon-Si, 443-810
Phone: +82-31-204-6190
E-Mail: info@hilscher.kr

Switzerland

Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support

Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA

Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support

Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com