Application Note

EtherNet/IP Adapter

CIP Sync

V2.9/V3.2 and higher

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC130104AN05EN | Revision 5 | English | 2016-09 | Released | Public

Table of Contents

1 Introduction

1.1 About this Document

This document describes the usage of the EtherNet/IP Adapter protocol stack as loadable firmware for netX with the aspect to CIP Sync. CIP Sync of EtherNet/IP uses the Precision Time Protocol (PTP) for Time Synchronized Distributed Control in motion applications. The EtherNet/IP Adapter protocol supports the CIP Sync protocol and provides a synchronized CIP Sync system time and a Sync Interrupt.

1.2 List of Revisions

Rev	Date	Name	Chapter	Revision
5	2016-09-30	KM/RG	all	United new version for V3.2.x and V2.x

Table 1: List of Revisions

1.3 Terms, Abbreviations and Definitions

Term	Description
BMCA	Best Master Clock Algorithm
CIP	Common Industrial Protocol
comX	Communication Module based on netX
COS	Change of State
DPM	Dual-Port Memory
LFW	Loadable Firmware
LOM	Linkable object modules
netX	Next generation of communication processors
ODVA	Open DeviceNet Vendor Association
PTP	Precision Time Protocol
TC	Transparent Clock

Table 2: Terms, Abbreviations and Definitions

1.4 References

This document based on the following specification:

[1] CIP Specification Vol1 Edition 3.15 (Common Industrial Protocol)

[2] CIP Specification Vol2 Edition 1.16 (EtherNet/IP Adaptation of CIP)

[3] IEEE 1588 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, Revision 2, 2008

[4] Hilscher Gesellschaft für Systemautomation mbH: Specification netX IO Synchronization V1.0, 2012

[5] Hilscher Gesellschaft für Systemautomation mbH: Driver Manual cifX Device Driver, Document ID: DOC060701DRV22EN, Revision 22, 2013

[6] Hilscher Gesellschaft für Systemautomation mbH: EtherNet/IP Adapter Protocol API Manual, Document ID: DOC060301API15EN, Revision 15, 2015 (for stack version V2.x.x) or Hilscher Gesellschaft für Systemautomation mbH: EtherNet/IP Adapter Protocol API Manual, Document ID: DOC150401API02EN, Revision 2, 2015 (for stack version V3.x.x)

[7] Hilscher Gesellschaft für Systemautomation mbH: netX Dual-Port Memory Interface Manual, Document ID: DOC060302DPM12EN, Revision 12, 2012

[8] Hilscher Gesellschaft für Systemautomation mbH: comX User Manual, Document ID: DOC100903UM04EN, Revision 4, 2014

1.5 Legal Notes

1.5.1 Copyright

1.5.2 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.

1.5.3 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.

1.5.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, 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.

1.5.5 Registered Trademarks

Windows® 2000, Windows® XP, Windows® Vista und Windows® 7 are registered trademarks of Microsoft Corporation.

EtherNet/IP® is a trademark of ODVA (Open DeviceNet Vendor Association, Inc).

All other mentioned trademarks are property of their respective legal owners.

2 Introduction to CIP Sync

CIP Sync is the time synchronization technology for the Common Industrial Protocol (CIP). This technology allows accurate real-time synchronization of devices and controllers connected over CIP networks that require

- time stamping,
- recording sequences of events,
- distributed motion control,
- increased control coordination.

CIP Sync uses the time synchronization technology as defined in the IEEE 1588 - Precision Clock Synchronization Protocol for Networked Measurement and Control Systems -standard (reference [3]).

The main components of CIP Sync are:

- The Precision Time Protocol defined in IEEE 1588:2008. It is a network protocol providing a standard mechanism for time synchronization of communicating clocks across a network of distributed devices.
- The CIP Sync Specification including the Time Sync object (CIP class ID 0x43) (see reference [1]) providing a CIP interface to the IEEE 1588 standard.

2.1 IEEE 1588 Precision Time Protocol (PTP) Overview

The IEEE 1588 standard specifies a protocol to synchronize independent clocks running on separate nodes of a distributed measurement and control system to a high degree of accuracy and precision. This protocol is commonly denominated as the Precision Time Protocol (PTP). Contrary to other time synchronization protocols, PTP is focused at achieving high precision within locally limited networks. It works based on these principles:

- The clocks communicate with each other over a communication network.
- In its basic form, the protocol is intended to be administration free.
- The protocol generates a master slave relationship among the clocks in the system.
- Within a given subnet of a network there will be a single master clock.
- All clocks ultimately derive their time from a clock denominated as the grandmaster clock.

It is not the aim of PTP to adjust the local time of the slave to that of the associated master as one might think at first. Instead of this, only the clock interval of the slave is adjusted to that of the associated master and the offset between the time of associated master and slave is determined with the highest possible precision. This allows calculating a very precise CIP Sync system time after the initial adjustment phase. (Indeed, adjustment is permanently continued within steady state operation).

In general, the guide line is to generate the time stamps

- as late as possible prior to transmission.
- as early as possible after reception.

The achievable accuracy is in the microsecond range for a software-based solution without any hardware support. With hardware support such as the Sync0 and Sync1 signals of the netX, an accuracy in the range of some nanoseconds is achievable.

2.1.1 Network Medium

The protocol is originally designed for, but not limited to, local area networks like Ethernet. Practically, PTP is often encapsulated within UDP (PTP over UDP) which in turn is encapsulated in IPv4 or IPv6 (UDP/IP using IPv4 or IPv6). Another possibility would be to use genuine Ethernet (IEEE 802.3).

Other possible networks besides Ethernet include DeviceNet or ControlNet.

2.1.2 Fault Tolerance

PTP tolerates occasionally occurring:

- Missing messages
- Duplicate messages
- Disordered messages

if these do not occur too often.

2.1.3 Message Types

IEEE 1588 defines two different kinds of messages: event messages and general messages.

Event Messages

Event messages are timed messages i.e. time stamps are generated both at their transmission and reception. The following types of event messages are relevant in the scope of this document.

- Sync messages
- Delay_Req messages

General Messages

General messages do not produce time stamps. The following types of general messages are relevant in the scope of this document.

- Announce messages
- Follow-up messages
- Delay_Resp messages

2.1.4 Sending Modes

The choice of the sending mode depends on the available hardware capabilities.

There are two sending modes defined in IEEE 1588.

2.1.4.1 One-step sending mode

A mode for sending a sync message including direct „on the fly“ time stamp transmission to the recipient. This requires a hardware fast enough to perform the „on the fly“ processing of time stamps. No follow-up message transmission is required then.

2.1.4.2 Two-step sending mode

A mode for sending a sync message without time stamp transmission to the recipient. In this case no „on the fly“ processing of time stamps is necessary any more. However, an additional follow-up message transmission is required in this case in order to inform the slave about the time stamp.

This follow-up message should be sent as soon as possible after the associated sync message in order to achieve best accuracy. In any case, the transmission of the follow-up message has to be completed before the transmission of the next sync message takes place.

This is the mode typically used in CIP Sync networks based on EtherNet/IP, so this document only discusses two-step mode in the following.

2.1.5 Types of Clocks

A PTP system of distributed clocks consists of ordinary, boundary, and transparent clocks:

- An ordinary clock may be a time master (master clock) or a time slave (slave clock).
- A boundary clock is a clock with more than one port and separates two or more distinct communication paths.
- A transparent clock also has more than one port, but differs from a boundary clock by transparently passing PTP messages between its ports.

Typically boundary and transparent clocks are implemented in switches connecting network segments.

One clock on each subnet in the system is selected as the master clock. The selection of a master is made by each of the other clocks by examining information contained in PTP Announce messages. An announce message is sent periodically by any port claiming to be the master clock. All ports use the same algorithm, which is denominated the Best Master Clock Algorithm (BMCA). If a port of a master clock receives an Announce message from a better clock then that port will cease to be a master port and will become a slave port. Likewise if a clock with a slave port determines that it would make a better master than the current master clock or if there is no current master clock, the port becomes a master port and begins to send Announce messages. Some nodes may be implemented as slave only nodes and never assume mastership (e.g. an I/O device). Other nodes may be implemented as master only nodes and never become slaves (e.g. primary clock source).

2.1.6 Sequence of Messages

The following sequence of messages applies:

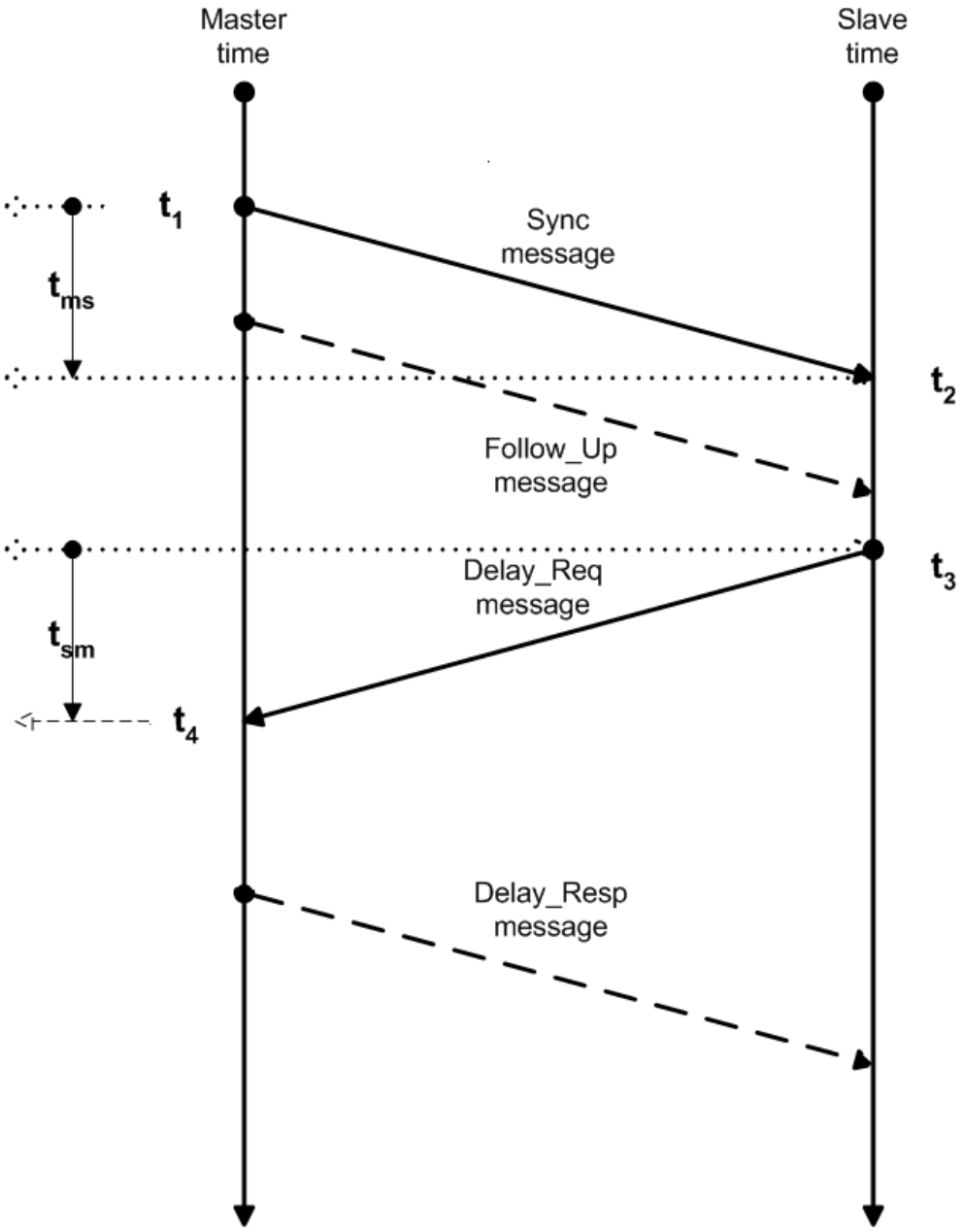

Figure 1: Protocol for Steady State Operation

Determination of Offset and Delay

From the 4 values t_0, t_1, t_2, t_3 the offset and the transmission delay can be calculated.

The offset O is calculated according to the following formula:

$O = (t_1+t_2-t_0-t_3)/2$

The delay D is calculated according to the following formula:

$O = (t_1+t_3-t_0-t_2)/2$

The following table explains the six steps required for offset determination, especially:

which action the master or slave performs, the involved time value and the acquired values meanwhile available in storage.

Step	Action	Time value to be stored	Available values in storage	Comment
1	Master sends Sync signal to slave	t_1 (Time of sending Sync signal at the master) is stored		Precise measurement of accurate sending time of Sync signal
2	Slave receives Sync signal from master	t_2 (Time of receiving Sync signal at the slave) is stored	t_2	Precise measurement of accurate reception time of Sync signal
3	Master sends follow-up message containing t1 to the slave		t_1 t_2	Happens when message arrives: One-step: Sync message Two-step: Follow-up message
4	Slave sends Delay_Req signal to master	t_3 (Time of sending Delay_Req signal at the slave) is stored	t_1 t_2 t_3	Precise measurement of accurate sending time of Delay_Req signal
5	Master receives Delay_Req signal from slave	t_4 (Time of receiving Delay_Req signal at the master) is stored	t_1 t_2 t_3	Precise measurement of accurate reception time of Delay_Req signal
6	Master sends Delay_Resp t4 to the slave		t_1 t_2 t_3 t_4	Happens when Delay_Resp message arrives

Table 3: Actions of Master and Slave and storage of time stamps

- After receiving the Follow-up message the offset can be calculated.
- After receiving the Delay_Resp message the delay can be calculated.

2.2 CIP Time Sync Object (0x43) Overview

The CIP Time Sync Object provides a CIP interface to the IEEE 1588 standard. For any device supporting the CIP Sync specification it is mandatory to provide support for the CIP Time Sync Object.

The object provides attributes and services for:

1. Retrieving the clock status and properties such as synchronized state, current offset to master, and identity of the grandmaster.
2. Accessing PTP clock management functions.

The CIP Time Sync Object is described in detail in Protocol API Manual [6].

3 Synchronization – General Aspects

3.1 Host-controlled vs. Device controlled Mode

The process data exchange between the netX-based protocol stack (Device) and host based application (Host) can be controlled either by the one or by the other side over Process Data Handshakes in DPM:

- Device controlled mode should be used with Slave protocol stacks.
- The host application can exchange the process data with the protocol stack over DPM in host controlled mode.

In both host controlled and device controlled mode the process data exchange can occur either in buffered or synchronized mode.

In buffered mode the data exchange between netX based protocol stack and host based application is decoupled from each other (not synchronized). In this case the stack handles the receiving and transmission of the data from/to the bus/network automatically and uses buffers to handle data consistently. Independently of the bus/network state (i.e. cycle start, incoming data, requests etc.) the host application can access the already completely received data or provide to netX the data required to send, which will be processed with the next bus cycle.

In synchronized mode the process data exchange between netX based protocol stack and host based application is linked (synchronized) to the bus communication. In this case the stack handles the receiving and transmission of the data from/to the bus/network and process data exchange with the host application is synchronized with protocol specific bus/network event.

The EtherNet/IP Adapter protocol stack uses:

- buffered host controlled mode for process data
- synchronized device controlled mode for synchronization

As the following affects synchronization only, we concentrate on synchronized device controlled mode here.

3.2 Synchronization Mechanisms on netX

Real time Ethernet systems have been defined to realize synchronous exchange of the input and output data. So, precise time measurement is getting more and more important.

If precise time measurement is not possible the quality of control depends on the jitter of synchronization cycles. In most cases slaves derive their internal clocks from receiving output or sync frames. Due to variable latency of the protocol stack or task switching of the OS the jitter can be unacceptable for precise control tasks involving bus communication. Formerly, the netX dual-port memory interface only supported the so called buffered mode, which does not fulfill synchronization requirements.

Two synchronized modes have generally been defined;

- synchronized device controlled mode for slave implementations.
- synchronized host controlled mode for master implementations (as mentioned above this mode is not relevant for the slave implementation discussed in this document)

Depending on accuracy requirements, there are two major mechanisms of host application synchronization possible:

- Software assisted synchronization using synchronization handshakes in DPM
- Hardware assisted synchronization using special hardware pin of netX.

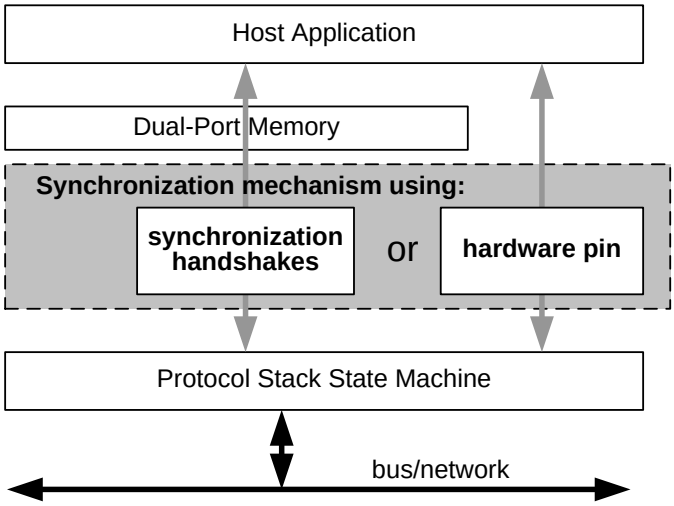

Figure 2: Process Data Exchange state machines: Host controlled mode

3.2.1 Synchronization Handshakes

In order to achieve more exact (less jitter) software assisted synchronization via DPM and to provide the additional synchronous operation modes the special synchronization handshakes are introduced. These can be used either to control the bus cycle by the host application (Master Stack in host controlled mode) or to indicate a selected protocol specific bus event to the host (Slave Stack in device controlled mode).

Synchronization handshakes are located at the positions, which correspond to the handshake channel itself [Reference [1], section 3.3] at offset 0x204 and 0x206. The following Handshake Flags `usHostSyncFlags` and `usNetxSyncFlags` are defined:

`usHostSyncFlags` Host writes, netX reads DPM offset: 0x206		`usNetxSyncFlags` netX writes, Host reads DPM offset: 0x204

Flag	Bit		Bit	Flag
`HSYNCF_CH0`	0		0	`NSYNCF_CH0`
`HSYNCF_CH1`	1		1	`NSYNCF_CH1`
`HSYNCF_CH2`	2		2	`NSYNCF_CH2`
`HSYNCF_CH3`	3		3	`NSYNCF_CH3`
`Reserved`	4:15		4:15	`reserved`

Table 4: Sync Flags of the Sync Handshake register

Features

- Handshake bits are located at dedicated handshake channel.
- Due to dedicated handshake channel, it is possible to handle Sync flags with different IRQ without any influence on the main stack functionality.
- Lowest possible latency and jitter can be achieved (about 500ns) since this can be handled by the netX CPU. A fast and simple interrupt service routine is available. Unnecessary checks are avoided.
- Synchronization functionality can be enhanced later since more handshake bits available.
- Synchronization of more than one protocol can be achieved in a consistent way with only one handshake register

3.2.1.1 Synchronization Mechanism

The synchronization mechanism is described with the simple state sequence running on the host and on the netX side connected via sync handshake toggles in the DPM (see Fig.4).

In either host or device controlled mode the sync state machine remains the same. The desired control mode is defined by the initial (start) condition of the state machine only.

The sequencing of the sync state machine is triggered by the Sync Event on the netX side. Dependent on the synchronization mode the host application has either to command the start of the bus cycle by toggling the synchronization handshake (host controlled mode) or to acknowledge an indication of the selected bus event from netX.

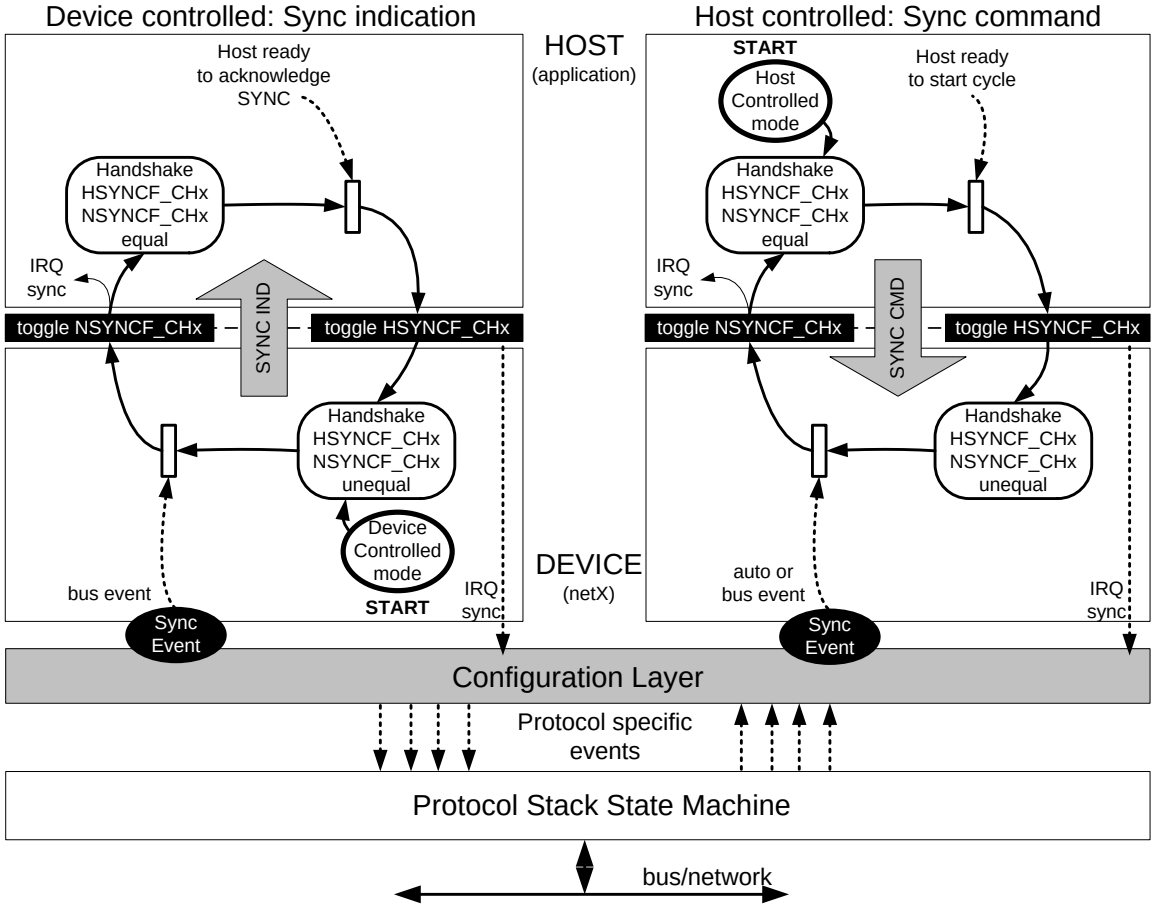

Figure 3: Synchronization Handshake Mechanism

3.2.2 Hardware assisted Synchronization by Pins Sync 0 and Sync 1

In order to achieve the best possible synchronization a precise hardware assisted synchronization mechanism should be considered. The netX provides special input/output pins, which can be used as a clock input or output. The use of hardware synchronization pins doesn't involve an ARM CPU and is very precise.

Basically, the generation of the sync signals is based on the netX internal hardware timers. The support of the hardware assisted synchronization is strongly related to the kind of the communication protocol If applicable, the corresponding protocol API manual defines the supported modes of hardware assisted synchronization.

4 CIP Sync Implementation in Hilscher's EtherNet/IP Adapter Protocol Stack for netX

4.1 Configuration Aspects (V3.x)

Note: This section only applies to EtherNet/IP Adapter stack version 3.x. For version 2.x see the next section of this chapter.

In addition to the general configuration of the stack's parameters (as described in sections "Getting Started/Configuration" and "`EIP_APS_SET_CONFIGURATION_PARAMETERS_REQ/CNF` – Configure Device with Warmstart Parameter" of the EtherNet/IP Adapter Protocol API Manual (reference [6]), the following configuration steps are required to switch on and configure the CIP Sync functionality:

1. Create the Time Sync Object

   (Command EIP_OBJECT_MR_REGISTER_REQ - 0x00001A02)

2. Configure the Synchronization Mode

   (Command EIP_OBJECT_CIP_SERVICE_REQ - 0x00001AF8)

3. Enable PTP Synchronization

   (Command EIP_OBJECT_CIP_SERVICE_REQ - 0x00001AF8)

4. Configure the Handshake mode for System Time Exchange

   (Command RCX_SET_HANDSHAKE_CONFIG_REQ - 0x00002F34)

4.1.1 Create Time Sync Object

In order to activate the Time Sync object within the EtherNet/IP stack, the command `EIP_OBJECT_MR_REGISTER_REQ` (0x00001A02) needs to be sent to the stack.

4.1.2 Configure the Synchronization Mode

In order to configure the synchronizations mode, attribute #300 of the Time Sync object (class ID 0x43) needs to be set. (Have a look into the Protocol API Manual [6] for further information regarding attribute 300)

Executing this request includes setting some required synchronization-related parameters. These are used to adjust the interval and offset times for the hardware synchronization signals Sync 0 and Sync 1.

Basically, the Sync 0 signal is the interrupt that the host application will receive in order to retrieve the current system time. On each event the EtherNet/IP stack writes the current system time into the extended data area of the Dual Port Memory interface.

Note: Currently, only Sync 0 can be used.

Attribute #300 of the Time Sync object has default values (see [6]). Therefore, it must only be configured if the host application requires a different configuration.

4.1.3 Enable the PTP Synchronization

In order to enable the synchronization attribute #1 (PTP Enable) of the Time Sync Object (class ID 0x43) needs to be set to value 0x01.

This can be achieved by sending the request `EIP_OBJECT_CIP_SERVICE_REQ (0x00001AF8)` to the stack. Have a look at the corresponding API manual for a detailed description of the packet (reference [5]).

To set attribute #1 of the Time Sync object request needs to be filled as stated below:

`ulService`	`= 0x10`	→ Set Attribute Single
`ulClass`	`= 0x43`	→ Time Sync Object
`ulInstance`	`= 0x01`	→ Instance 1 of the Time Sync Object
`ulAttribute`	`= 0x01`	→ Attribute #1 of the Time Sync Object
`abData[0]`	`= 0x01`	→ Data to be set to attribute 1

Note: Attribute #1 of the Time Sync Object can be accessed/ configured via the Ethernet Network. The host application will then receive the packet EIP_OBJECT_CIP_OBJECT_CHANGE_IND which includes the data this attribute was set to.

4.1.4 Configure the Handshake Mode for System Time Exchange

In order to be able to retrieve the current CIP Sync system time from the EtherNet/IP Adapter stack (which is done via the Dual Port Interface), the handshake mode needs to be configured properly. Therefore, the host application must send the following packet to the EtherNet/IP Adapter stack.

4.1.4.1 Set Handshake Configuration Request

The application uses the RCX_SET_HANDSHAKE_CONFIG_REQ packet in order to set the handshake mode. The packet is send to the protocol stack through the channel mailbox.

For the proper usage in the context of EtherNet/IP and CIP Sync have a look at the example section *Software-assisted Synchronization of System Time between Adapter Clock and Host Application of the* Adapter on page 26.

Packet Structure Reference

```
/* SET HANDSHAKE CONFIGURATION REQUEST */
#define RCX_SET_HANDSHAKE_CONFIG_REQ         0x00002F34

typedef struct RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_Ttag
{
  UINT8   bPDInHskMode;   /* input process data handshake mode           */
  UINT8   bPDInSource;    /* input process data trigger source           */
  UINT16  usPDInErrorTh;  /* threshold for input data handshake handling errors     */
  UINT8   bPDOutHskMode;  /* output process data handshake mode          */
  UINT8   bPDOutSource;   /* output process data trigger source          */
  UINT16  usPDOutErrorTh; /* threshold for output data handshake handling errors    */
  UINT8   bSyncHskMode;   /* synchronization handshake mode              */
  UINT8   bSyncSource;    /* synchronization source                      */
  UINT16  usSyncErrorTh;  /* threshold for sync handshake handling errors*/
  UINT32  aulReserved[2]; /* reserved for future use                     */
} RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_T;

typedef struct RCX_SET_HANDSHAKE_CONFIG_REQ_Ttag
{
  RCX_PACKET_HEADER                       tHead;   /* packet header      */
  RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_T     tData;   /* packet data        */
} RCX_SET_HANDSHAKE_CONFIG_REQ_T;
```

Packet Description

Structure RCX_SET_HANDSHAKE_CONFIG_REQ_T			Type: Request
Variable	**Type**	**Value / Range**	**Description**
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0, 0x20	Destination Queue-Handle. Set to 0: Destination is operating system rcX 32 (0x20): Destination is the protocol stack
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue-Handle. Set to: 0: when working with loadable firmware. Queue handle returned by TLR_QUE_IDENTIFY()}: when working with loadable firmware.
ulDestId	UINT32	0	Destination End Point Identifier, specifying the final receiver of the packet within the Destination Process. Set to 0, will not be changed
ulSrcId	UINT32	0 ... 2^{32}-1	Source End Point Identifier, specifying the origin of the packet inside the Source Process. This variable may be used for low-level addressing purposes.
ulLen	UINT32	16	Packet Data Length (In Bytes);
ulId	UINT32	0 ... 2^{32}-1	Packet Identification as unique number generated by the Source Process of the Packet
ulSta	UINT32	0	Status/Error
ulCmd	UINT32	0x2F34	RCX_SET_HANDSHAKE_CONFIG_REQ Command
ulExt	UINT32	0	Reserved
ulRout	UINT32	0	Routing Information
tData - Structure RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_T			
bPDInHskMode	UINT8		Input process data handshake mode
bPDInSource	UINT8	0	Input process data trigger source; unused, set to zero
usPDInErrorTh	UINT16		Threshold for input process data handshake handling errors
bPDOutHskMode	UINT8		Output process data handshake mode
bPDOutSource	UINT8	0	Output process data trigger source; unused, set to zero
usPDOut ErrorTh	UINT16	0 … 0xFFFF	Threshold for output process data handshake handling errors
bSyncHskMode	UINT8		Synchronization handshake mode
bSyncSource	UINT8		Synchronization source
usSyncErrorTh	UINT16	0 … 0xFFFF	Threshold for synchronization handshake handling errors
aul Reserved[2]	UINT32	0	Reserved for future use; set to zero

Table 5: RCX_SET_HANDSHAKE_CONFIG_REQ - Set Handshake Configuration Request

4.1.4.2 Set Handshake Configuration Confirmation

The following packet is returned by the protocol stack.

Packet Structure Reference

```
/* SET HANDSHAKE CONFIGURATION CONFIRMATION */
#define RCX_SET_HANDSHAKE_CONFIG_CNF         RCX_SET_HANDSHAKE_CONFIG_REQ+1

typedef struct RCX_SET_HANDSHAKE_CONFIG_CNF_Ttag
{
  TLR_PACKET_HEADER_T                    tHead;   /* packet header      */
} RCX_SET_HANDSHAKE_CONFIG_CNF_T;
```

Packet Description

Structure RCX_SET_HANDSHAKE_CONFIG_CNF_T			**Type: Confirmation**
Variable	**Type**	**Value / Range**	**Description**
tHead – Structure TLR_PACKET_HEADER_T			
ulDest	UINT32	0, 0x20	Destination Queue Handle
ulSrc	UINT32	0 ... 2^{32}-1	Source Queue Handle
ulDestId	UINT32	0	Destination Queue Reference
ulSrcId	UINT32	0 ... 2^{32}-1	Source Queue Reference
ulLen	UINT32	0	Packet Data Length (In Bytes);
ulId	UINT32	From Request	Packet Identification as Unique Number
ulSta	UINT32	=0 <>0	means: correct execution means: an error has occurred, for listings of possible error codes see netX Dual-port Memory Manual, chapter 6 *"Status & Error Codes"*
ulCmd	UINT32	0x00002F35	RCX_SET_HANDSHAKE_CONFIG_CNF Command
ulExt	UINT32	0x00000000	Reserved
ulRout	UINT32	x	Routing Information, Do not use!

Table 6: RCX_SET_HANDSHAKE_CONFIG_CNF_T- Confirmation to Set Handshake Configuration Request

4.2 Configuration Aspects (V2.x)

Note: This section only applies to EtherNet/IP Adapter stack version 2.x. For stack version 3.x see the previous section.

In addition to the general configuration of the stack's parameters (as described in sections "Getting Started/Configuration" and "`EIP_APS_SET_CONFIGURATION_REQ/CNF` – Configure Device with Warmstart Parameter" of the EtherNet/IP Adapter Protocol API Manual (reference [6]), the following configuration steps are required to switch on and configure the CIP Sync functionality:

1. Create the Time Sync Object and Configure the Synchronization Mode

   (Command EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ - 0x00001A58)

2. Enable PTP Synchronization

   (Command EIP_OBJECT_CIP_SERVICE_REQ - 0x00001AF8)

3. Configure the Handshake mode for System Time Exchange

   (Command RCX_SET_HANDSHAKE_CONFIG_REQ - 0x00002F34)

4.2.1 Create Time Sync Object and Configure the Synchronization Mode

In order to activate the Time Sync object within the EtherNet/IP stack, the command `EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ` (0x00001A58) needs to be sent to the stack. For further information related to that packet see the stack's API manual.

Executing this request includes setting some required synchronization-related parameters. These are used to adjust the interval and offset times for the hardware synchronization signals Sync 0 and Sync 1.

Basically, the Sync 0 signal is the interrupt that the host application will receive in order to retrieve the current system time. On each event the EtherNet/IP stack writes the current system time into the extended data area of the Dual Port Memory interface.

If the confirmation packet is received with `ulSta=0` then the Create Time Sync Object Request has been processed correctly.

Note: Currently, only Sync 0 can be used.

4.2.1.1 CIP Time Sync Object (Class Code: 0x43)

The Time Sync Object of CIP is implemented in the following manner in the EtherNet/IP Adapter protocol stack:

Instance	Name	Attribute ID	Name	NV	Supported Services	
					Get Attribute Single	Set Attribute Single
0	Class	1	Revision		Yes	No
		2	Max Instance		Yes	No
1	Instance Attributes	1	PTPEnable	NV	Yes	Yes
		2	IsSynchronized		Yes	No
		3	SystemTimeMicroseconds		Yes	No
		4	SystemTimeNanoseconds		Yes	No
		5	OffsetFromMaster		Yes	No
		6	MaxOffsetFromMaster		Yes	Yes
		7	MeanPathDelayToMaster		Yes	No
		8	GrandMasterClockInfo		Yes	No
			ClockIdentity			
			ClockClass			
			TimeAccuracy			
			OffsetScaledLogVariance			
			CurrentUtcOffset			
			TimePropertyFlags			
			TimeSource			
			Priority1			
			Priority2			
		9	ParentClockInfo		Yes	No
			ClockIdentity			
			PortNumber			
			ObservedOffsetScaledLogVariance			
			ObservedPhaseChangeRate			
		10	LocalClockInfo		Yes	No
			ClockIdentity			
			ClockClass			
			TimeAccuracy			
			OffsetScaledLogVariance			
			CurrentUtcOffset			
			TimePropertyFlags			
			TimeSource			
		11	NumberOfPorts		Yes	No
		12	PortStateInfo		Yes	No
			NumberOfPorts			
			(ARRAY of STRUCT)			
			PortNumber			
			PortState			
		13	PortEnableCfg		Yes	No

Instance	Name	Attribute ID	Name	NV	Supported Services	
			NumberOfPorts			
			(ARRAY of STRUCT)			
			PortNumber			
			PortEnable			
		14	PortLogAnnounceIntervalC fg	NV	Yes	Yes
			NumberOfPorts			
			(ARRAY of STRUCT)			
			PortNumber			
			PortLogAnnounceInterval			
		15	PortLogSyncIntervalCfg	NV	Yes	Yes
			NumberOfPorts			
			(ARRAY of STRUCT)			
			PortNumber			
			PortLogSyncInterval			
		18	DomainNumber	NV	Yes	Yes
		19	ClockType		Yes	No
		20	ManufactureIdentity		Yes	No
		21	ProductDescription		Yes	No
			Size			
			Description			
		22	RevisionData		Yes	No
			Size			
			Revision			
		23	UserDescription		Yes	No
			Size			
			Description			
		24	PortProfileIdentityInfo		Yes	No
			NumberOfPorts			
			(ARRAY of STRUCT)			
			PortNumber			
			PortProfileIdentity			
		25	PortPhysicalAddressInfo		Yes	No
			NumberOfPorts			
			(ARRAY of STRUCT)			
			PortNumber			
			PhysicalProtocol			
			SizeOfAddress			
			PortPhysicalAddress			
		26	PortProtocolAddressInf		Yes	No
			NumberOfPorts			
			(ARRAY of STRUCT)			

Instance	Name	Attribute ID	Name	NV	Supported Services	
			PortNumber			
			NetworkProtocol			
			SizeOfAddress			
			PortProtocolAddress			
		27	StepsRemoved		Yes	No
		28	SystemTimeAndOffset		Yes	No
			SystemTime			
			SystemOffset			

Table 7: Time Sync Object Supported Features

Remarks: The conditional attributes #16 (Priority1) and #17 (Priority2) are not implemented within the EtherNet/IP Adapter Protocol Stack.

4.2.2 Enable the PTP Synchronization

In order to enable the synchronization attribute #1 (PTP Enable) of the Time Sync Object (class ID 0x43) needs to be set to value 1.

This can be achieved by sending the request `EIP_OBJECT_CIP_SERVICE_REQ (0x00001AF8)` to the stack. Have a look at the corresponding API manual for a detailed description of the packet (reference [5]).

To set attribute #1 of the Time Sync object request needs to be filled as stated below:

`ulService = 0x10` → Set Attribute Single

`ulClass = 0x43` → Time Sync Object

`ulInstance = 0x01` → Instance 1 of the Time Sync Object

`ulAttribute = 0x01` → Attribute #1 of the Time Sync Object

`abData[0] = 0x01` → Data to be set to attribute 1

Additionally, have a look at the example code in section *Enabling the PTP in the CIP Time Sync Object (0x43)* on page 37.

4.2.3 Configure the Handshake Mode for System Time Exchange

See section *Configure the Handshake Mode for System Time Exchange* on page 19.

4.3 Software-assisted Synchronization of System Time between Adapter Clock and Host Application of the Adapter

This section explains how the host application can retrieve the current CIP Sync system time from the EtherNet/IP Adapter stack via the DPM. This process takes place by software assisted synchronization using the handshake flags of the netX communication channel.

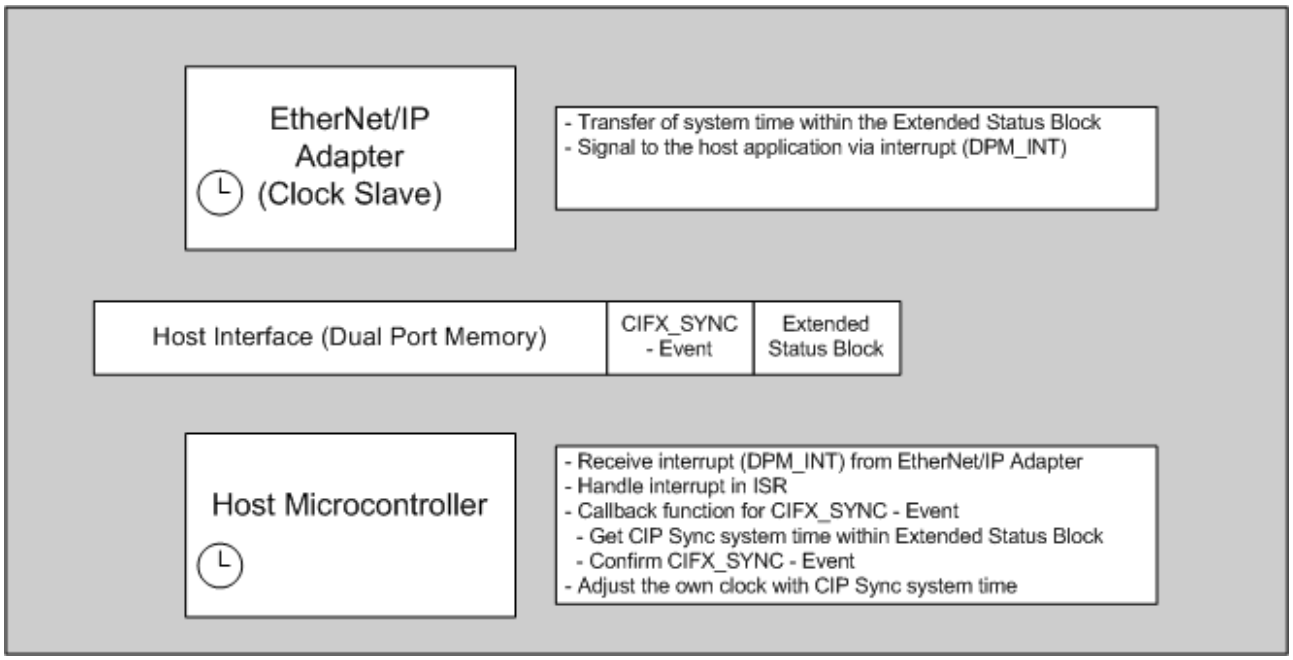

Figure 4: Overview: Synchronization of System Time between Adapter Clock and Host Application of the Adapter

The EtherNet/IP Adapter sends an interrupt signal to its host (**CIFX_NOTIFY_SYNC**) in order to provide the time information. These interrupts have to be handled at the).

The system time is transferred to the host via the Extended Status Block, (offset 0xA4 within the Ext. Status Block). Furthermore on a software level, CIFX_SYNC Events are generated and sent to the host.

The host must install a callback function for these CIFX_SYNC Events which has to perform two tasks:

- It must read out the CIP Sync System Time written to the Extended Status Block (Offset 0xA4) by the EtherNet/IP Adapter.
- It must confirm the CIFX_SYNC Events.

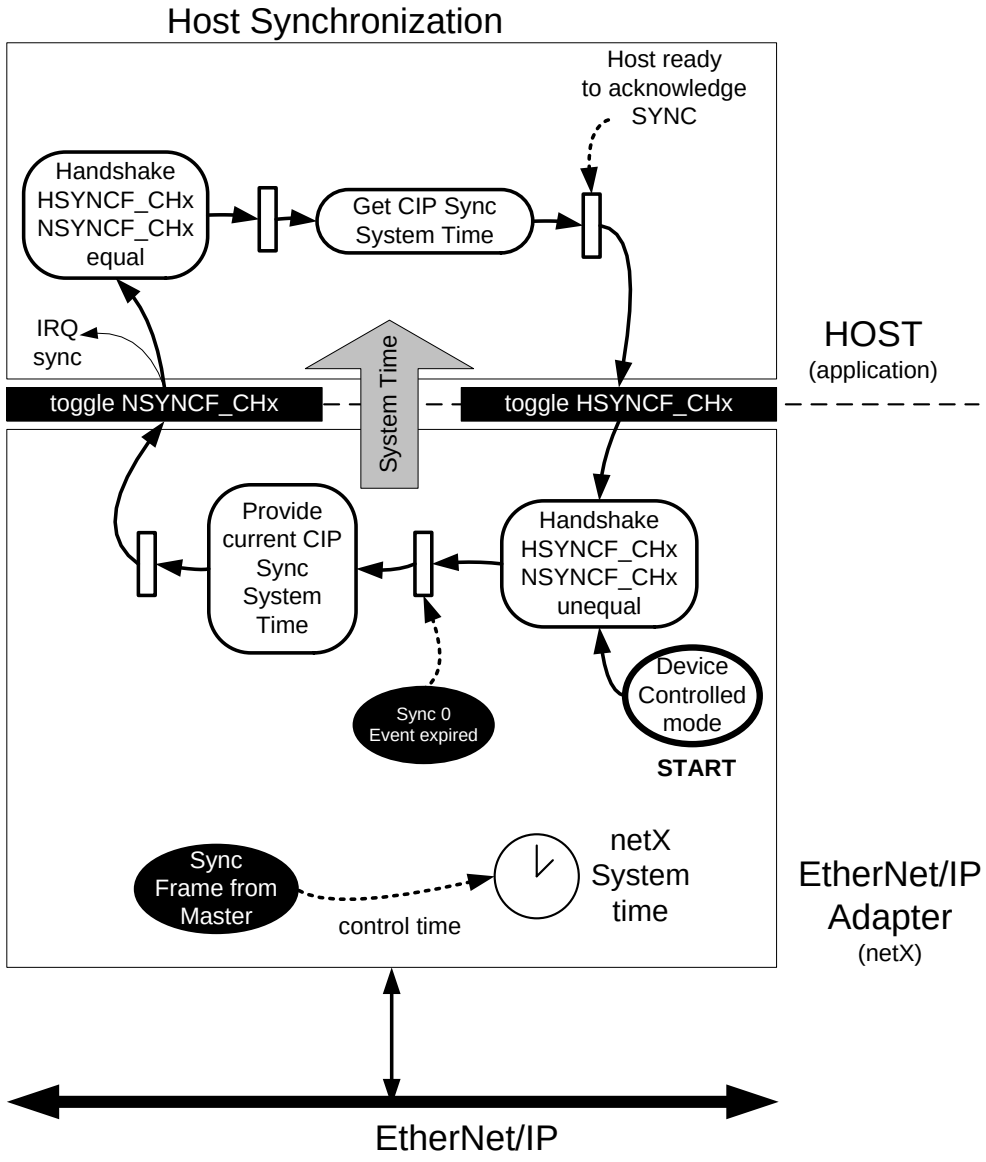

Figure 5: Synchronization of System Time between Adapter Clock and Host Application

Figure 5 and Figure 6 show the synchronization process in more detail. Every time the EtherNet/IP Adapter receives a sync frame from the clock master it controls the internal system time of the netX. This time controlling process is completely independent of the exchange of the CIP Sync System Time between the adapter and the host application.

In order to provide the CIP Sync System Time, the Sync0 event triggers the EtherNet/IP Adapter to write the current system time into the Extended Status Block of the DPM (Offset 0xA4).

The timing (interval) of the Sync0 event can be configured upon the creation of the Time Sync object (see section *Create Time Sync Object* on page 17)

After writing the System Time into the DPM the EtherNet/IP Adapter triggers the DPM interrupt via the sync handshakes (NSYNCF_Chx). At this point the host application reads the current System Time and acknowledges the interrupt (HSYNCF_Chx).

This mechanism is started by the EtherNet/IP Adapter only if it has synchronized its time to the Sync Master. So, if the master stops sending sync frames or is powered down, the Adapter will stop generating DPM interrupts.

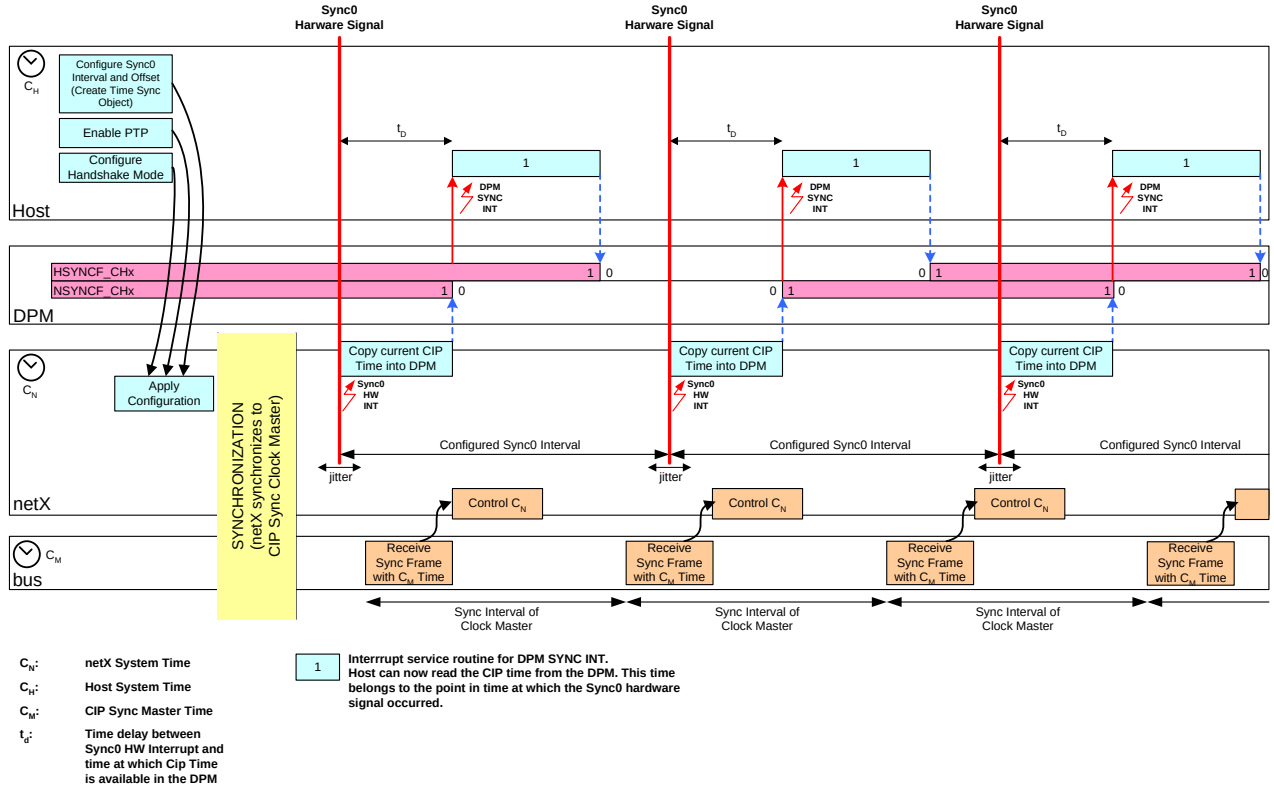

Figure 6: Software Assisted Synchronization of System Time between Adapter Clock and Host Application

4.4 Hardware-assisted Synchronization of System Time between Adapter Clock and Host Application

The synchronization of system time with hardware assistance is basically the same as the software assisted method. The system time must be read after the time has been written into the DPM by the adapter. So, the DPM interrupt must still be used.

Additionally, the host application can use an interrupt routine that is triggered directly by the hardware sync pin of the netX. This way, the host application knows about the exact point of time at which the Sync0 event occurs and can then compensate the time delay that comes with the DPM interrupt (see delay t_D in Figure 7).

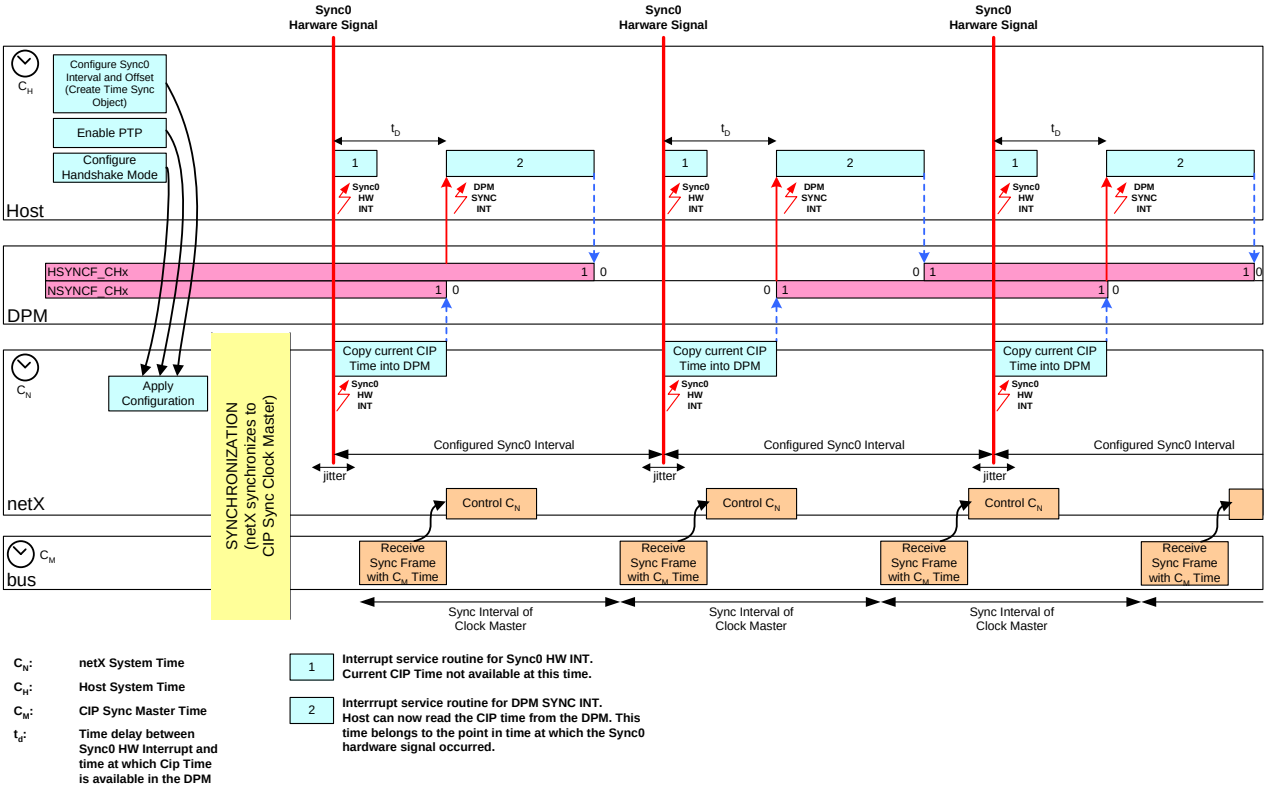

Figure 7: Hardware Assisted Synchronization of System Time between Adapter Clock and Host Application

5 Behavior in special Situations

This subsection discusses the aspects of some special situations such as

- Start-up
- Missing SYNC Signal
- Sync 0 and Sync 1 Signal occur at the same Time

5.1.1 Start-up

Important: It might happen that the CIP Sync System Time value in the Extended Status Block of the DPM is 0. This can occur during startup or if the Adapter lost synchronization and comes back into it.

Therefore, 0 values should be ignored.

The reason for this is that after establishing synchronization there is no time information available at the first time the Sync0 signal is triggered. At all subsequent signals the system time will then be valid.

5.1.2 Missing SYNC Signal

If for a period of 10 seconds no Sync Signal is received, the EtherNet/IP Adapter will fall back into the listening state. This means the Adapter is not synchronized anymore and no Sync0 signal will be created.

5.1.3 Sync 0 and Sync 1 Signal occur at the same Time

Clock synchronization is significantly deteriorated if the Sync 0 and the Sync 1 signal occur at (nearly) exactly the same time. In order to achieve maximum precision, do not configure equal or nearly equal offset settings for Sync 0 and Sync 1.

6 Example

This section explains how to achieve a clock synchronous operation of EtherNet/IP Scanner and Adapter.

Important: The example presented in this chapter is only applicable for a Loadable Firmware scenario (LFW).

The following figure illustrates the relationship between the EtherNet/IP Scanner and Adapter and the host microcontroller connected to the EtherNet/IP Adapter. Some details have already been described generally in chapters *Synchronization – General Aspects* on page 13 and *CIP Sync Implementation in Hilscher's EtherNet/IP Adapter Protocol Stack for netX* on page 17.

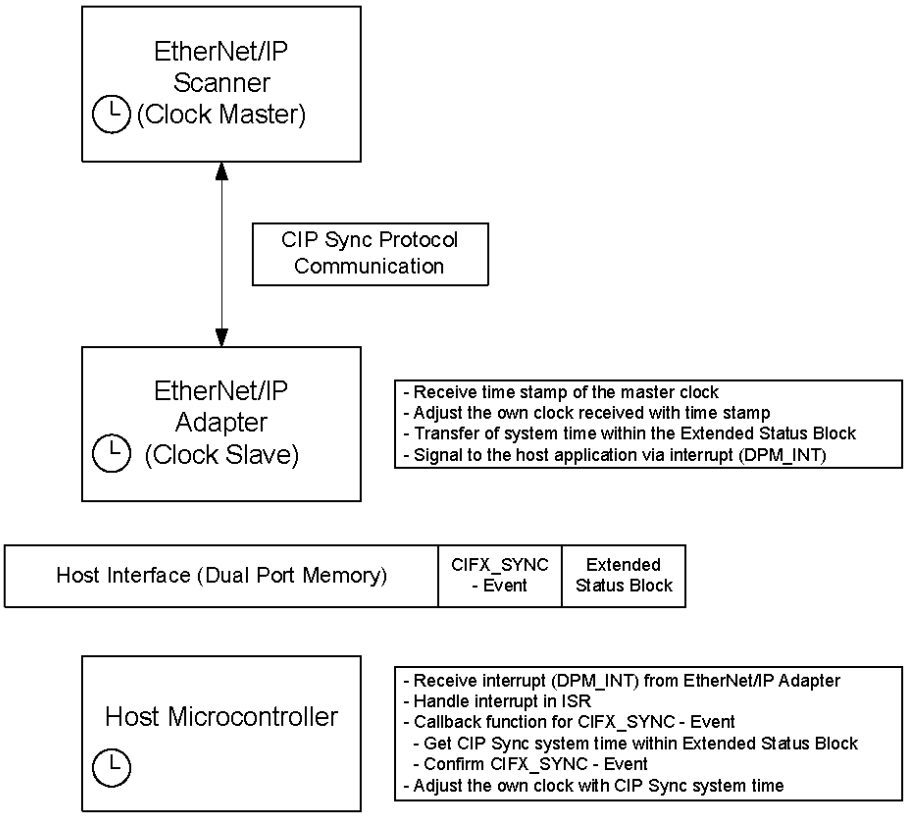

Figure 8: Communication Scenario for Host Example/EtherNet/IP Adapter CIP Sync

Standard function calls for setting up communication

The host application example uses the following standard function calls:

1. Open the driver
2. Open a connection to a system device on the passed board
3. Open a connection to a communication channel
4. Create set configuration packet (for EtherNet/IP Adapter)
5. Send this packet to the channel's mailbox for processing
6. Retrieve a pending packet from the channel mailbox
7. Reset the given communication channel, after set configuration packet

8. Set the bus state flag in the application COS state flags
9. Wait for communication to be established

Additionally, the Host application example performs the following additional actions:

1. Create and configure CIP Time Sync Object by sending the packet `EIP_OBJECT_MR_REGISTER_REQ` and `EIP_OBJECT_CIP_SERVICE_REQ` to the channel mailbox (allows setting of `ulSync0Interval`, `ulSync0Offset`, `ulSync1Interval`, `ulSync1Offset`)
2. Enable PTP by sending a `EIP_OBJECT_CIP_SERVICE_REQ` packet to the channel mailbox
3. Configure Sync handshake mode and send packet `RCX_SET_HANDSHAKE_CONFIG_REQ` to the channel mailbox
4. Register a notification and setting up a callback function for reading out the System Time from the Extended Status Block, then wait for synchronization event or trigger/acknowledge sync

6.1 Practical Usage of Example

If more that one netX card is available on the PC, the host example uses the first netX card that is running an EtherNet/IP Adapter Firmware.

The following prerequisites are necessary for successfully testing the example on an NXHX board:

- Usage of the netX based hardware e.g. NXHX board.
- NXPCA-PCI board.
- cifX Device Driver with interrupt support (V1.1.1.0 and later).

Steps to be performed for testing the example on an NXHX board

In order to practically perform the example, proceed as follows:

1. Install the provided 2nd Stage Bootloader into the NXHX board with Boot wizard application.
2. Install the NXPCA-PCI board into your PC.
3. Establish the connection between the NXHX and the NXPCA-PCI board with the ribbon cable.
4. Download the firmware with the cifX Device Driver.Activate the interrupt support within the cifX Device Driver Setup Utility. See *Figure 9: Option "Use Interrupt with cifX Device Driver"* below.
5. Execute the Host Example Project.

Note: For stack version 3.x: The example project has CIP Sync disabled by default. To enable the CIP Sync feature set the preprocessor define “_SUPPORT_CIP_SYNC_”.

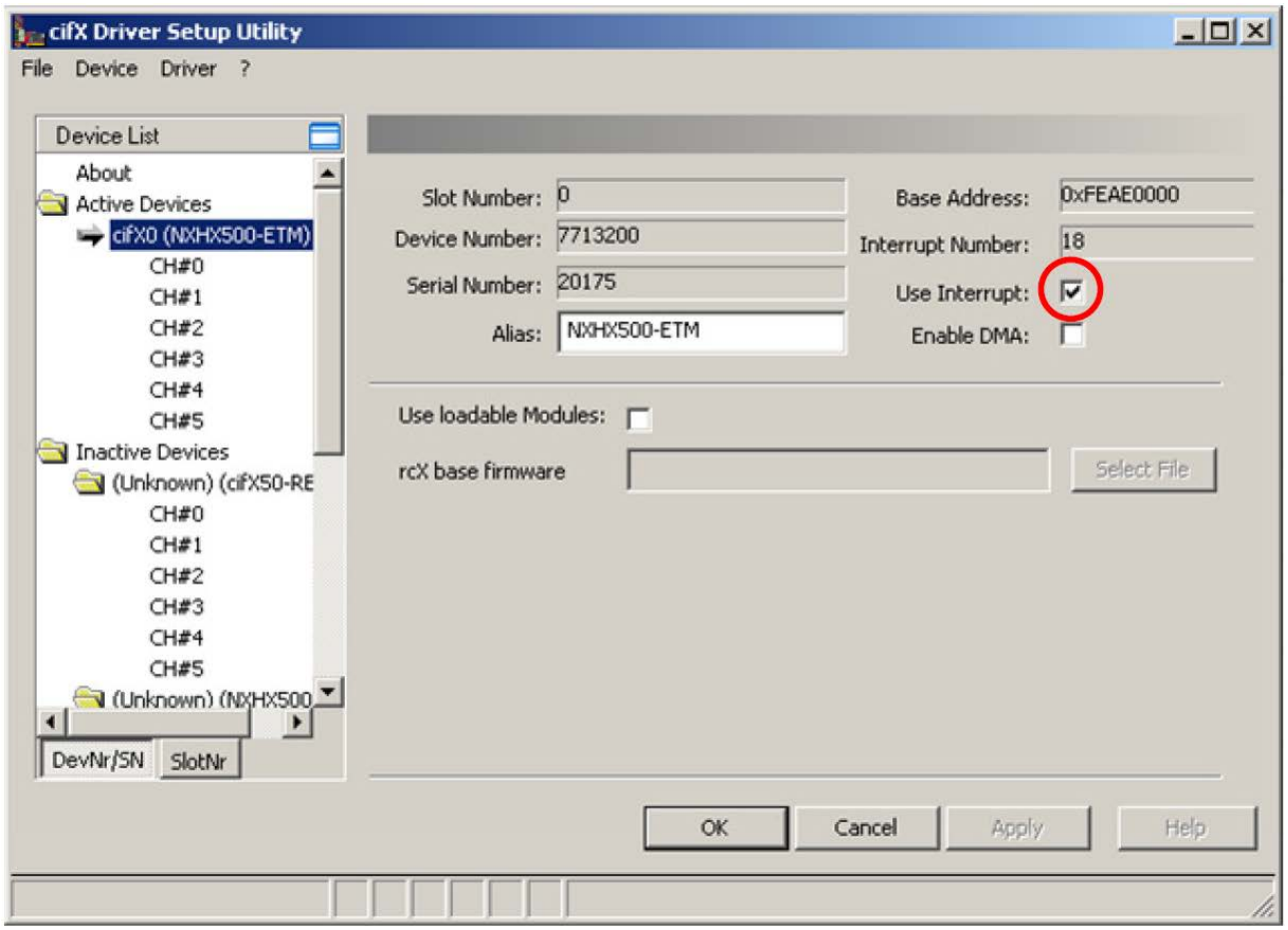

Figure 9: Option "Use Interrupt with cifX Device Driver"

6.2 Main Actions to be performed by the Host Application Program

Note: This section only applies for the EtherNet/IP Adapter protocol stack V2.x.

The following actions have to be performed by the Host application program:

- Configuration of the EtherNet/IP protocol stack must be done properly.
- Synchronization of System Time between Master Clock and Slave Clocks
- Software-assisted Synchronization of System Time between Adapter Clock and Host Application of the Adapter

Within the example host application program, these are performed by the routines of file `SetCIPSync.c`. for a full listing of this file, see section *SetCIPSync.c* on page 48.

6.2.1 Configuration Aspects

The configuration is done by sending an `EIP_APS_SET_CONFIGURATION_REQ` request packet to the EtherNet/IP Adapter protocol stack. This is performed by function `SetConfigParamsPkt` in file `SetConfigParams.c`.

For a function definition and a set of applicable parameters for proper configuration, see section *SetConfigParams.h* on page 62.

6.2.2 Synchronization of System Time between Master Clock and Slave Clocks

In order to achieve clock synchronous operation of EtherNet/IP Scanner and Adapter, the following steps have to be performed by the Host application program:

- Creating the Object Time Sync Packet
- Enabling the PTP in the CIP Time Sync Object

6.2.2.1 Creating the Object Time Sync Packet

First the object for time synchronization has to be created by sending the `EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ` request packet to the channel mailbox. This packet enables generating the timestamps and writing these into the Extended Status Block within the Dual-Port Memory.

In the host application example, this is done by the function

```
CreateCIPSyncObj(( void* )&tSendPkt );
```

For a description of the `EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ` request packet.

The following data have to be set there:

- `ulSync0Interval`:
  Interval of Sync0 hardware signal (XMAC3_IO0) and CIFX_SYNC
  - set to the value 1000000000 for an interval of 1 s.
- `ulSync0Offset`:
  Offset for Sync0 hardware signal
  - set to the value 0 for an offset of 0 ns.
- `ulSync1Interval`:
  Interval of Sync1 hardware signal (XMAC3_IO1)
  - set to the value 1000000000 for an interval of 1 s.
- `ulSync1Offset`:
  Offset for Sync1 hardware signal
  - set to the value 150 for an offset of 150 ns.

This is illustrated by the following code excerpt from file `Main.c`:

```
/* Check for CIP Sync currently complete configured */
if( tSyncResParms.ulConfigured == FALSE )
{
  /* ## First packet */
  /* Create CIP Time Sync Object */
  CreateCIPSyncObj( ( void* )&tSendPkt );

  /* Send a packet to the channel's mailbox */
  lRet = xChannelPutPacket( hChannel, &tSendPkt, ulTimeout );

  if( lRet != CIFX_NO_ERROR )
  {
    printf( "Error sending packet to device: 0x%08X !\r\n", lRet );
  }
  else
  {
    printf( "Send Packet:\r\n" );

    /* Dumps a rcX packet to debug console */
    DumpPacket( &tSendPkt );

    /* Retrieve a pending packet from the channel mailbox */
    lRet = xChannelGetPacket( hChannel, sizeof( tRecvPkt ), ( void* )&tRecvPkt, ulTimeout
);

    if( lRet != CIFX_NO_ERROR )
    {
      printf( "Error getting packet from device: 0x%08X !\r\n", lRet );
    }
    else
    {
      printf( "Received Packet:\r\n" );

      /* Dumps a rcX packet to debug console */
      DumpPacket( &tRecvPkt );
    }
  }
```

The following code excerpt from file SetCIPSync.c contains the definition of function CreateCIPSyncObj((void*)&tSendPkt);

```
/*****************************************************************************
 function:    CreateCIPSyncObj
 description: Create CIP Time Sync Object.

 global:      none
 input:       void* pvPck                                    - pointer to the packet

 output:      none
 return:      none
*****************************************************************************/
void CreateCIPSyncObj( void* pvPck )
{
  /* Get struct of task packet union */
  EIP_OBJECT_PACKET_CREATE_OBJECT_TIMESYNC_REQ_T *ptCIPSyncObj = (
EIP_OBJECT_PACKET_CREATE_OBJECT_TIMESYNC_REQ_T* )pvPck;

  ptCIPSyncObj->tHead.ulDest    = 0x20;                                                 /*
Destination of packet, process queue */
  ptCIPSyncObj->tHead.ulSrc     = 0x10;                                                 /*
Source of packet, process queue      */
  ptCIPSyncObj->tHead.ulDestId  = 0;                                                    /*
Destination reference of packet      */
  ptCIPSyncObj->tHead.ulSrcId   = 0;                                                    /*
Source reference of packet           */
  ptCIPSyncObj->tHead.ulLen     = EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ_SIZE;           /*
Length of packet data without header */
  ptCIPSyncObj->tHead.ulId      = 0;                                                    /*
Identification handle of sender      */
  ptCIPSyncObj->tHead.ulSta     = 0;                                                    /*
Status code of operation             */
  ptCIPSyncObj->tHead.ulCmd     = EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ;                /*
Packet command                       */
  ptCIPSyncObj->tHead.ulExt     = 0;                                                    /*
Extension                            */
  ptCIPSyncObj->tHead.ulRout    = 0;                                                    /*
Router                               */

  ptCIPSyncObj->tData.ulSync0Interval = 1000000000;                                     /*
Sync 0 Time Interval: 1s */
  ptCIPSyncObj->tData.ulSync0Offset   = 0;                                              /*
Sync 0 Offset: 0s        */
  ptCIPSyncObj->tData.ulSync1Interval = 1000000000;                                     /*
Sync 1 Time Interval: 1s */
  ptCIPSyncObj->tData.ulSync1Offset   = 150;                                            /*
Sync 0 Offset: 150ns     */

} /* CreateCIPSyncObj */
```

For the structure definition of EIP_OBJECT_PACKET_CREATE_OBJECT_TIMESYNC_REQ_T.

6.2.2.2 Enabling the PTP in the CIP Time Sync Object (0x43)

Then the PTP has to be enabled within the CIP Time Sync Object. In order to do so, the application has to send the `EIP_OBJECT_CIP_SERVICE_REQ` request packet to the channel mailbox. In the host application example, this is done by the function `EnableCIPService( ( void* )&tSendPkt );`.

For a description of the `EIP_OBJECT_CIP_SERVICE_REQ` request packet, see section *Enable the PTP Synchronization* on page 18 or the EtherNet/IP Adapter Protocol API Manual (reference [5]).

The following data have to be set there

- `ulService`: Has to be set to 0x10 indicating „Set attribute single“
- `ulClass`: Has to be set to 0x43 indicating „Time Sync Object“
- `ulInstance`: Has to be set to 1.
- `ulAttribute`: Has to be set to 1 indicating „*Enable PTP*“.

This is illustrated by the following code excerpt from file `Main.c`:

```
/* ## Second packet */
/* Enable PTP */
EnableCIPService( ( void* )&tSendPkt );

/* Send a packet to the channel's mailbox */
lRet = xChannelPutPacket( hChannel, &tSendPkt, ulTimeout );

if( lRet != CIFX_NO_ERROR )
{
  printf( "Error sending packet to device: 0x%08X !\r\n", lRet );
}
else
{
  printf( "Send Packet:\r\n" );

  /* Dumps a rcX packet to debug console */
  DumpPacket( &tSendPkt );

  /* Retrieve a pending packet from the channel mailbox */
  lRet = xChannelGetPacket( hChannel, sizeof( tRecvPkt ), ( void* )&tRecvPkt, ulTimeout
);

  if( lRet != CIFX_NO_ERROR )
  {
    printf( "Error getting packet from device: 0x%08X !\r\n", lRet );
  }
  else
  {
    printf( "Received Packet:\r\n" );

    /* Dumps a rcX packet to debug console */
    DumpPacket( &tRecvPkt );
  }
}
```

The following code excerpt from file `SetCIPSync.c` contains the definition of function `CreateCIPSyncObj(( void* )&tSendPkt );`

```
/*****************************************************************************
 function:    EnableCIPService
 description: Enable PTP.

 global:      none
 input:       void* pvPck                                    - pointer to the packet

 output:      none
 return:      none
*****************************************************************************/
void EnableCIPService( void* pvPck )
{
  /* Get struct of task packet union */
  EIP_OBJECT_PACKET_CIP_SERVICE_REQ_T *ptCIPServ = ( EIP_OBJECT_PACKET_CIP_SERVICE_REQ_T*
)pvPck;

  ptCIPServ->tHead.ulDest   = 0x20;                                                 /*
Destination of packet, process queue */
  ptCIPServ->tHead.ulSrc    = 0x10;                                                 /*
Source of packet, process queue      */
  ptCIPServ->tHead.ulDestId = 0;                                                    /*
Destination reference of packet      */
  ptCIPServ->tHead.ulSrcId  = 0;                                                    /*
Source reference of packet           */
  ptCIPServ->tHead.ulLen    = EIP_OBJECT_CIP_SERVICE_REQ_SIZE;                      /*
Length of packet data without header */
  ptCIPServ->tHead.ulId     = 0;                                                    /*
Identification handle of sender      */
  ptCIPServ->tHead.ulSta    = 0;                                                    /*
Status code of operation             */
  ptCIPServ->tHead.ulCmd    = EIP_OBJECT_CIP_SERVICE_REQ;                           /*
Packet command                       */
  ptCIPServ->tHead.ulExt    = 0;                                                    /*
Extension                            */
  ptCIPServ->tHead.ulRout   = 0;                                                    /*
Router                               */

  ptCIPServ->tData.ulService    = 0x10;                                             /*
Set attribute single */
  ptCIPServ->tData.ulClass      = 0x43;                                             /*
Time Sync Object     */
  ptCIPServ->tData.ulInstance   = 1;                                                /*
Instance             */
  ptCIPServ->tData.ulAttribute  = 1;                                                /*
Enable PTP           */

  ptCIPServ->tData.abData[0]    = 1;                                                /*
Time Sync Object data        */
  ptCIPServ->tHead.ulLen       += 1;                                                /*
Time Sync Object data length */

} /* EnableCIPService */
```

6.2.3 Software-assisted Synchronization of System Time between Adapter Clock and Host Application of the Adapter

This process takes place by software-assisted synchronization using the handshake flags of the netX communication channel.

6.2.3.1 RCX Set Handshake Packet

The Sync handshake mode has to be configured. In order to do so, the application has to send the RCX_SET_HANDSHAKE_CONFIG_REQ request packet to the channel mailbox.

In the host application example, this is done by the function ConfSyncHdsk((void*)&tSendPkt);.

The handshake registers to be used are 0x204, 0x206, 0x208 and 0x20A. For more details see the netX Dual-Port Memory Interface Manual (Reference #7).

For a description of the RCX_SET_HANDSHAKE_CONFIG_REQ request packet, see section *Set Handshake Configuration Request* on page 19.

The following data have to be set there

- bSyncHskMode: Synchronization handshake mode in Device Controlled mode
  - set to the value RCX_SYNC_MODE_DEV_CTRL for device-controlled synchronization mode.
- bSyncSource: Synchronization Source 1
  - set to the value RCX_SYNC_SOURCE_1 for source 1.

```
/* ## Third packet */
/* Configure Sync handshake mode */
ConfSyncHdsk( ( void* )&tSendPkt );

/* Send a packet to the channel's mailbox */
lRet = xChannelPutPacket( hChannel, &tSendPkt, ulTimeout );

if( lRet != CIFX_NO_ERROR )
{
  printf( "Error sending packet to device: 0x%08X !\r\n", lRet );
}
else
{
  printf( "Send Packet:\r\n" );

  /* Dumps a rcX packet to debug console */
  DumpPacket( &tSendPkt );

  /* Retrieve a pending packet from the channel mailbox */
  lRet = xChannelGetPacket( hChannel, sizeof( tRecvPkt ), ( void* )&tRecvPkt, ulTimeout
);

  if( lRet != CIFX_NO_ERROR )
  {
    printf( "Error getting packet from device: 0x%08X !\r\n", lRet );
  }
  else
  {
    printf( "Received Packet:\r\n" );

    /* Dumps a rcX packet to debug console */
    DumpPacket( &tRecvPkt );
  }
}
```

The following code excerpt from file `SetCIPSync.c` contains the definition of function `CreateCIPSyncObj(( void* )&tSendPkt );`

```
/*****************************************************************************
 function:    ConfSyncHdsk
 description: Configure Sync handshake mode.

 global:      none
 input:       void* pvPck                                  - pointer to the packet

 output:      none
 return:      none
*****************************************************************************/
void ConfSyncHdsk( void* pvPck )
{
  /* Get struct of task packet union */
  RCX_SET_HANDSHAKE_CONFIG_REQ_T *ptSyncHdsk = ( RCX_SET_HANDSHAKE_CONFIG_REQ_T* )pvPck;

  ptSyncHdsk->tHead.ulDest    = 0x20;                                                  /*
Destination of packet, process queue */
  ptSyncHdsk->tHead.ulSrc     = 0x10;                                                  /*
Source of packet, process queue      */
  ptSyncHdsk->tHead.ulDestId  = 0;                                                     /*
Destination reference of packet      */
  ptSyncHdsk->tHead.ulSrcId   = 0;                                                     /*
Source reference of packet           */
  ptSyncHdsk->tHead.ulLen     = sizeof(RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_T);           /*
Length of packet data without header */
  ptSyncHdsk->tHead.ulId      = 0;                                                     /*
Identification handle of sender      */
  ptSyncHdsk->tHead.ulSta     = 0;                                                     /*
Status code of operation             */
  ptSyncHdsk->tHead.ulCmd     = RCX_SET_HANDSHAKE_CONFIG_REQ;;                         /*
Packet command                       */
  ptSyncHdsk->tHead.ulExt     = 0;                                                     /*
Extension                            */
  ptSyncHdsk->tHead.ulRout    = 0;                                                     /*
Router                               */

  ptSyncHdsk->tData.bPDInHskMode    = RCX_IO_MODE_BUFF_HST_CTRL;                       /*
Input process data handshake mode                                    */
  ptSyncHdsk->tData.bPDInSource     = 0;                                             /*
Input process data trigger source, Currently unused, set to zero */
  ptSyncHdsk->tData.usPDInErrorTh   = 0;                                             /*
Threshold for input process data handshake handling errors       */

  ptSyncHdsk->tData.bPDOutHskMode   = RCX_IO_MODE_BUFF_HST_CTRL;                     /*
Output process data handshake mode                                     */
  ptSyncHdsk->tData.bPDOutSource    = 0;                                             /*
Output process data trigger source, Currently unused, set to zero  */
  ptSyncHdsk->tData.usPDOutErrorTh  = 0;                                             /*
Threshold for output process data handshake handling errors        */

  ptSyncHdsk->tData.bSyncHskMode    = RCX_SYNC_MODE_DEV_CTRL;                        /*
Synchronization handshake mode, Device controlled */
  ptSyncHdsk->tData.bSyncSource     = RCX_SYNC_SOURCE_1;                             /*
Synchronization source */
  ptSyncHdsk->tData.usSyncErrorTh   = 0;                                             /*
Threshold for synchronization handshake handling errors */

  ptSyncHdsk->tData.aulReserved[0]  = 0;                                             /*
Reserved for future use. Set to zero. */
  ptSyncHdsk->tData.aulReserved[1]  = 0;                                             /*
Reserved for future use. Set to zero. */

} /* ConfSyncHdsk */
```

6.2.3.2 Register a Notification and fetch System Time from Extended Status

The application first has to register the notification

`xChannelRegisterNotification(hChannel, CIFX_NOTIFY_SYNC, fnSyncClb, &pvUser )`

See the following code excerpt from file `Main.c`:

```
/* Delay */
Sleep( 1000 );

/* Register a notification callback */
if( /*( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_RX_MBX_FULL,  fnEventClb, (void*)1))) ||
    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_TX_MBX_EMPTY, fnEventClb, (void*)2))) ||
    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_PD0_IN,       fnEventClb, (void*)3))) ||
    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_PD0_OUT,      fnEventClb, (void*)5))) ||*/
    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel, CIFX_NOTIFY_SYNC,
fnSyncClb,  (void*)&tSyncResParms ) ) ) )
{
  /* Failed to register one of the events */
  printf( "Error to register SYNC event: 0x%08X !\r\n", lRet );
}
else
{
  /* Configuration completed successfully */
  tSyncResParms.ulConfigured = TRUE;

  /* Get actual host state */
  if( ( lRet = xChannelHostState( hChannel, CIFX_HOST_STATE_READ, &ulState, 0L ) ) !=
CIFX_NO_ERROR )
  {
    /* Read driver error description */
    printf( "Error HOST State: 0x%08X !\r\n", lRet );
  }

  /* Set host ready */
  if( ( lRet = xChannelHostState( hChannel, CIFX_HOST_STATE_READY, NULL, 2000L ) ) !=
CIFX_NO_ERROR )
  {
    /* Read driver error description */
    printf( "Error HOST State: 0x%08X !\r\n", lRet );
  }

  /*--------------------*/
  /* Wait for user      */
  /*--------------------*/
  printf("\n\rEVENT-Handling aktiv.......!!!!!\r\n");
}
```

Then it has to set up a callback function `fnSyncClb` to be called each time when CIFX SYNC event occurs

This callback function does the following:

- Read out the CIP Sync Time Stamp (stored value of System Time) from the Extended Status Block using
  - `xChannelExtendedStatusBlock`: Read data with offset `0xA4` within the Extended Status Block with length of 8 byte
- Wait for synchronization event or trigger/acknowledge sync using
  - `xChannelSyncState`: Acknowledge the CIFX_SYNC-Event

Important: If you read out a CIP Sync Time Stamp of exactly 0, this signals a preceding loss of the synchronization signal. In this case you should wait until the next sync signal as very likely the CIP Sync Time Stamp will be correct at the next time. The reason for this is that after resuming processing synchronization signals at the first time no valid system time is available, but the signal is produced anyway. At all subsequent signals the system time will then be valid.

The following code excerpt from file `Main.c` contains the definition of function `fnSyncClb`:

```
/*****************************************************************************
 function:    fnSyncClb
 description: Function to be called if CIFX SYNC event occurs.

 global:      none
 input:       none

 output:      none
 return:      none
*****************************************************************************/
void APIENTRY fnSyncClb(uint32_t ulNotification, uint32_t ulDataLen, void* pvData, void*
pvUser)
{
  long                lRet            = 0;                                        /*
Return value for common error codes          */
  CIP_SYNC_RES_PARAM* ptSyncResParams = {0};                                      /*
CIP Sync Demo resourcen parameter structure  */

  UNREFERENCED_PARAMETER( pvData );
  UNREFERENCED_PARAMETER( ulDataLen );
  UNREFERENCED_PARAMETER( ulNotification );

  (CIP_SYNC_RES_PARAM*) ptSyncResParams = (CIP_SYNC_RES_PARAM*)pvUser;

  /* Read the CIP Sync Time Stamp */
  lRet = xChannelExtendedStatusBlock( ptSyncResParams->hChannel, CIFX_CMD_READ_DATA,
0xA4, 8, &ptSyncResParams->tCIPSyncTimeStamp);

  /* Get System device specific information */
  lRet = xSysdeviceInfo( ptSyncResParams->hSysdevice,
                         CIFX_INFO_CMD_SYSTEM_STATUS_BLOCK,
                         sizeof( SYSTEM_CHANNEL_SYSTEM_STATUS_BLOCK ),
                         &ptSyncResParams->tSysStatusBlock );

  /* Wait for synchronization event or trigger/acknowledge sync */
  lRet = xChannelSyncState( ptSyncResParams->hChannel, CIFX_SYNC_ACKNOWLEDGE_CMD, 0,
&ptSyncResParams->ulErrorCount );

  /* Copy time since start in seconds to local CIP Sync resourcen parameter */
  memcpy( &ptSyncResParams->tHostStartTime.ulTimeS, &ptSyncResParams-
>tSysStatusBlock.ulTimeSinceStart, sizeof( ptSyncResParams-
>tSysStatusBlock.ulTimeSinceStart ) );

  /* Display the CIP Syn Time Stamp */
  printf( "CIP Sync Time Stamp" );
  DumpData( &ptSyncResParams->tCIPSyncTimeStamp, sizeof( ptSyncResParams-
>tCIPSyncTimeStamp ) );

  /* Display the local Time Stamp since start in seconds */
  printf( "Local Time Stamp" );
  DumpData( &(unsigned char)ptSyncResParams->tHostStartTime.ulTimeS, sizeof(
ptSyncResParams->tHostStartTime.ulTimeS ) );

} /* fnSyncClb */
/*****************************************************************************
```

7 Appendix

7.1 Extended Status Block of the EtherNet/IP Adapter Protocol Stack

The content of the channel specific extended status block is specific to the implementation. Depending on the protocol, a status area may or may not be present in the dual-port memory. It is always available in the default memory map (see section 3.2.1 of netX Dual-Port Memory Manual).

Note: Each offset is always related to the begin of correspondent channel start.

The definition of the first structure remains specific to correspondent protocol and contains additional information about network status (i.e. flags, error counters, events etc.).

The Extended Status Block for EtherNet/IP Adapter describes the last error which has occurred. It is stored at the location of the offset **0x0050** and structured as follows:

Extended Status Block – First part (EIP_CODE_DIAG_T)			
Offset	**Type**	**Name**	**Description**
0x0050	UINT32	ulInfoCount	Info count
0x0054	UINT32	ulWarningCount	Warning count
0x0058	UINT32	ulErrorCount	Error count
0x005C	UINT32	ulLevel	Error Level (Info / Warning or Error)
0x0060	UINT32	ulCode	Error Code
0x0064	UINT32	ulParameter	Parameter of the error code
0x0068	UINT32	ulLine	Line in the source code where the error happened.
0x006C	UINT8[12]	abModulName[12]	Source identifier
0x0078	UINT8[]	bReserved[44]	Reserved for further status information
0xA4	UINT32	ulSystemTimeNs	SystemTime (nanoseconds part)
0xA8	UINT32	ulSystemTimeSeconds	SystemTime (seconds part)

Table 8: Extended Status Block (for EtherNet/IP Scanner Protocol Stack)

Within the reserved area, meanwhile the time information is stored at 0xA4.

The second structure begins at offset 0x00FC and provides the definition of the up to 32 independent State Fields. These state fields can be defined to represent a kind of bit-list, byte-list etc. with up to 65535 entities. In this way a common access mechanism for different state definitions and quantities can be provided independent of protocol implementation.

The second part of the Extended Status Block is structured as follows:

Extended Status Block for EtherNet/IP Scanner – Second part (State Field Definition Block)			
Offset	**Type**	**Name**	**Description/Value**
0x00FC	unsigned char	bReserved[3]	Reserved. Do not use.
0x00FF	unsigned char	bNumStateStructs	Number of State Structures defined below = 3
↓	**NETX_EXTENDED_STATE_STRUCT_T**	**atStateStruct[0]**	**Structure to define State field properties**
0x0100	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=1. Corresponds to a bit list (one bit per node) of configured nodes
	unsigned short	usNumOfStateEntries	=128. Corresponds to 128 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the slave configuration area
↓	**NETX_EXTENDED_STATE_STRUCT_T**	**atStateStruct[1]**	**Structure to define State field properties**
0x0110	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=2. Corresponds to a bit list (one bit per node) of active nodes
	unsigned short	usNumOfStateEntries	=128. Corresponds to 128 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the slave state information area
↓	**NETX_EXTENDED_STATE_STRUCT_T**	**atStateStruct[2]**	**Structure to define State field properties**
0x0120	unsigned char	bStateArea	=0. State field is located in standard input area of channel 0
	unsigned char	bStateTypeID	=3. Corresponds to a bit list (one bit per node) of diagnostic nodes
	unsigned short	usNumOfStateEntries	=128. Corresponds to 128 bits, each representing a slave
	unsigned long	ulStateOffset	Contains an offset pointer to a state field inside input data area 0, which contains the slave diagnostic area

Table 9: Extended Status Block for EtherNet/IP Scanner – Second part (State Field Definition Block)

If the location of the state fields is defined to be inside of input data area 0 block (as it is shown in generic example above), the corresponding bit lists will be updated by the stack consistently to the data in this area. Moreover, the data and corresponding state fields can be read out by the host application as one data block i.e. with DMA support.

7.2 Example Files (Full Listings)

Note: This section only applies to EtherNet/IP Adapter stack version 2.x.

7.2.1 SetConfigParams.c

```
/**************************************************************************************

Copyright (c) Hilscher GmbH. All Rights Reserved.

**************************************************************************************

  Filename:
    $Id: SetConfigParams.c 31696 2013-01-31 09:42:12Z gordon $:
  Last Modification:
    $Revision:: 31696           $:  Revision of last commit
    $Author:: gordon            $:  Author of last commit
    $Date:: 2013-01-31 10:42:12#$:  Date of last commit

  Targets:
    Win32/ANSI   : yes
    Win32/Unicode: no (define _UNICODE)
    WinCE        : no
    rcX          : no

  Description:
    Host PC Example
    SetConfigParams
    Set configuration parameters setting for Host PC example.

  Changes:
    Name    Date          Version     Description
    ----------------------------------------------------------------------------------
    GG      2013-01-31    1.000       created
**************************************************************************************/


/**************************************************************************************/
/* Includes */
#include <windows.h>                                                                  /*
Include Windows-specific header file for the C */

#include "rX_Types.h"                                                                 /*
rcX basic typedefinifions                          */
#include "rcX_User.h"                                                                 /*
rcX definition file for the rcX operating system */

#include "TLR_Types.h"                                                                /*
Task Layer Reference commands definitions  */
#include "TLR_Packet.h"                                                               /*
Task Layer Reference common definitions    */

#include "SetConfigParams.h"                                                          /*
Inlucde Host PC example specifics header */

#include "EipAps_Public.h"                                                            /*
EtherNetIP Application Task includes */
#include "EipObject_Public.h"                                                         /*
EthernetIP Object Task includes      */
#include "TcpipTcpTask_Public.h"                                                      /*
Include TCP IP specific header       */


/**************************************************************************************/
```

```
/* Host PC Example, SetConfigParams */


/*****************************************************************************
 function:    SetConfigParamsPkt
 description: Create protocol specific set configuration packet.

 global:      none
 input:       void* pvPck                                  - pointer to the packet

 output:      none
 return:      none
*****************************************************************************/
void SetConfigParamsPkt( void* pvPck )
{
  /* Get struct of task packet union */
  EIP_APS_PACKET_SET_CONFIGURATION_REQ_T *ptSetConfigReq = (
EIP_APS_PACKET_SET_CONFIGURATION_REQ_T* )pvPck;

  ptSetConfigReq->tHead.ulDest    = 0x20;                                        /*
Destination of packet, process queue */
  ptSetConfigReq->tHead.ulSrc     = 0x10;                                        /*
Source of packet, process queue      */
  ptSetConfigReq->tHead.ulDestId  = 0;                                           /*
Destination reference of packet      */
  ptSetConfigReq->tHead.ulSrcId   = 0;                                           /*
Source reference of packet           */
  ptSetConfigReq->tHead.ulLen     = EIP_APS_SET_CONFIGURATION_REQ_SIZE;          /*
Length of packet data without header */
  ptSetConfigReq->tHead.ulId      = 0;                                           /*
Identification handle of sender      */
  ptSetConfigReq->tHead.ulSta     = 0;                                           /*
Status code of operation             */
  ptSetConfigReq->tHead.ulCmd     = EIP_APS_SET_CONFIGURATION_REQ;               /*
Packet command                       */
  ptSetConfigReq->tHead.ulExt     = 0;                                           /*
Extension                            */
  ptSetConfigReq->tHead.ulRout    = 0;                                           /*
Router                               */

  ptSetConfigReq->tData.ulSystemFlags   = WSTART_SYSFLAG_START_AUTO;             /*
System flags                        */
  ptSetConfigReq->tData.ulWdgTime       = 1000;                                  /*
Watchdog time                       */
  ptSetConfigReq->tData.ulInputLen      = 504;                                   /*
Length of input data (Instance 100)  */
  ptSetConfigReq->tData.ulOutputLen     = 504;                                   /*
Length of Output data (Instance 101) */

  /* Flags for TCP stack configuration */
  ptSetConfigReq->tData.ulTcpFlag       = IP_CFG_FLAG_IP_ADDR |                  /*
Fixed IP address                    */
                                          IP_CFG_FLAG_NET_MASK |                 /*
Fixed network mask                  */
                                          //IP_CFG_FLAG_GATEWAY |                /*
Fixed gateway address               */
                                          //IP_CFG_FLAG_BOOTP |                  /*
Enable BOOTP                        */
                                          //IP_CFG_FLAG_DHCP |                   /*
Enable DHCP                         */
                                          //IP_CFG_FLAG_FULL_DUPLEX |            /*
Set fixed to full duplex            */
                                          //IP_CFG_FLAG_SPEED_100MBIT |          /*
Set fixed to 100MBit                */
                                          IP_CFG_FLAG_AUTO_NEGOTIATE ;           /*
Enable auto negotiate               */

  ptSetConfigReq->tData.ulIpAddr        = 0xC0A8D20A;                            /*
IP address:   192.168.210.10        */
```

```
  ptSetConfigReq->tData.ulNetMask       = 0xFFFFFF00;                                  /*
Network mask: 255.255.255.0          */
  ptSetConfigReq->tData.ulGateway       = 0;                                           /*
Gateway address                      */

  ptSetConfigReq->tData.usVendId        = 283;                                         /*
Vendor identification                */
  ptSetConfigReq->tData.usProductType   = 12;                                          /*
Product type                         */
  ptSetConfigReq->tData.usProductCode   = 257;                                         /*
Product code                         */
  ptSetConfigReq->tData.ulSerialNumber  = 0;                                           /*
Serial number of the device          */
  ptSetConfigReq->tData.bMinorRev       = 1;                                           /*
Minor revision                       */
  ptSetConfigReq->tData.bMajorRev       = 1;                                           /*
Major revision                       */
  ptSetConfigReq->tData.abDeviceName[0] = 19;                                          /*
Product name length                  */
  memcpy( &ptSetConfigReq->tData.abDeviceName[1], "EtherNet/IP Adapter", 19 );        /*
Product name                         */

  ptSetConfigReq->tData.ulInputAssInstance  = 100;                                     /*
Instance number of input assembly    */
  ptSetConfigReq->tData.ulInputAssFlags     = 0;                                       /*
Input assembly flags                 */
  ptSetConfigReq->tData.ulOutputAssInstance = 101;                                     /*
Instance number of output assembly   */
  ptSetConfigReq->tData.ulOutputAssFlags    = 0;                                       /*
Output assembly flags                */

  /* Quality of Service configuration structure */
  ptSetConfigReq->tData.tQoS_Config.ulQoSFlags        =
EIP_OBJECT_QOS_FLAGS_DISABLE_802_1Q;
                                                                                       /*
Enables or disables sending 802.1Q frames on CIP messages          */
  ptSetConfigReq->tData.tQoS_Config.bTag802Enable     = 0;                             /*
Enables or disables sending 802.1Q frames on CIP messages          */
  ptSetConfigReq->tData.tQoS_Config.bDSCP_PTP_Event   = 0;                             /*
Not supported                                                      */
  ptSetConfigReq->tData.tQoS_Config.bDSCP_PTP_General = 0;                             /*
Not supported                                                      */
  ptSetConfigReq->tData.tQoS_Config.bDSCP_Urgent      = 0;                             /*
DSCP value for CIP transport class 0/1 Urgent priority messages    */
  ptSetConfigReq->tData.tQoS_Config.bDSCP_Scheduled   = 0;                             /*
DSCP value for CIP transport class 0/1 Scheduled priority messages */
  ptSetConfigReq->tData.tQoS_Config.bDSCP_High        = 0;                             /*
DSCP value for CIP transport class 0/1 High priority messages      */
  ptSetConfigReq->tData.tQoS_Config.bDSCP_Low         = 0;                             /*
DSCP value for CIP transport class 0/1 low priority messages       */
  ptSetConfigReq->tData.tQoS_Config.bDSCP_Explicit    = 0;                             /*
DSCP value for CIP explicit messages                               */

  ptSetConfigReq->tData.ulNameServer    = 0;                                           /*
Name Server 1  */
  ptSetConfigReq->tData.ulNameServer_2  = 0;                                           /*
Name Server 2  */

  /* Domain name */
  memset( &ptSetConfigReq->tData.abDomainName[0], 0, sizeof( ptSetConfigReq-
>tData.abDomainName ) );
  /* Host name */
  memset( &ptSetConfigReq->tData.abHostName[0], 0, sizeof( ptSetConfigReq-
>tData.abHostName ) );

  ptSetConfigReq->tData.bSelectAcd = 1;                                                /*
Select ACD */

  /* Structure containing information on the last detected conflict */
```

```
  ptSetConfigReq->tData.tLastConflictDetected.bAcdActivity = 0;                        /*
State of ACD activity when last conflict detected */

  /* MAC address of remote node from the ARP PDU in which a conflict was detected */
  memset( &ptSetConfigReq->tData.tLastConflictDetected.abRemoteMac[0], 0, sizeof(
ptSetConfigReq->tData.tLastConflictDetected.abRemoteMac ) );

  /* Copy of the raw ARP PDU in which a conflict was detected */
  memset( &ptSetConfigReq->tData.tLastConflictDetected.abArpPdu[0], 0, sizeof(
ptSetConfigReq->tData.tLastConflictDetected.abArpPdu ) );

} /* SetConfigParamsPkt */
```

7.2.2 SetCIPSync.c

```
/**************************************************************************************

Copyright (c) Hilscher GmbH. All Rights Reserved.

**************************************************************************************

  Filename:
    $Id: SetCIPSync.c 31696 2013-01-31 09:42:12Z gordon $:
  Last Modification:
    $Revision:: 31696           $:  Revision of last commit
    $Author:: gordon            $:  Author of last commit
    $Date:: 2013-01-31 10:42:12#$:  Date of last commit

  Targets:
    Win32/ANSI   : yes
    Win32/Unicode: no (define _UNICODE)
    WinCE        : no
    rcX          : no

  Description:
    Host PC Example
    SetCIPSync
    Set CIP Sync configuration parameters setting for Host PC example.

  Changes:
    Name    Date          Version     Description
    ----------------------------------------------------------------------------------
    GG      2013-01-31    1.000       created
**************************************************************************************/


/**************************************************************************************/
/* Includes */
#include <windows.h>                                                                   /*
Include Windows-specific header file for the C */

#include "rX_Types.h"                                                                  /*
rcX basic typedefinifions                       */
#include "rcX_User.h"                                                                  /*
rcX definition file for the rcX operating system */
#include "rcX_Public.h"                                                                /*
rcX basic typedefinifions                       */

#include "TLR_Types.h"                                                                 /*
Task Layer Reference commands definitions  */
#include "TLR_Packet.h"                                                                /*
Task Layer Reference common definitions    */

#include "SetCIPSync.h"                                                                /*
Inlucde Host PC example specifics header */

#include "EipObject_Public.h"                                                          /*
EthernetIP Object Task includes          */
```

```
#include "CIFxErrors.h"                                                         /*
Include cifX driver API error definition */


/*****************************************************************************/
/* Host PC Example, Additional packets for CIP Sync */


/*****************************************************************************
 function:    CreateCIPSyncObj
 description: Create CIP Time Sync Object.

 global:      none
 input:       void* pvPck                                - pointer to the packet

 output:      none
 return:      none
*****************************************************************************/
void CreateCIPSyncObj( void* pvPck )
{
  /* Get struct of task packet union */
  EIP_OBJECT_PACKET_CREATE_OBJECT_TIMESYNC_REQ_T *ptCIPSyncObj = (
EIP_OBJECT_PACKET_CREATE_OBJECT_TIMESYNC_REQ_T* )pvPck;

  ptCIPSyncObj->tHead.ulDest    = 0x20;                                         /*
Destination of packet, process queue */
  ptCIPSyncObj->tHead.ulSrc     = 0x10;                                         /*
Source of packet, process queue      */
  ptCIPSyncObj->tHead.ulDestId  = 0;                                            /*
Destination reference of packet      */
  ptCIPSyncObj->tHead.ulSrcId   = 0;                                            /*
Source reference of packet           */
  ptCIPSyncObj->tHead.ulLen     = EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ_SIZE;   /*
Length of packet data without header */
  ptCIPSyncObj->tHead.ulId      = 0;                                            /*
Identification handle of sender      */
  ptCIPSyncObj->tHead.ulSta     = 0;                                            /*
Status code of operation             */
  ptCIPSyncObj->tHead.ulCmd     = EIP_OBJECT_CREATE_OBJECT_TIMESYNC_REQ;        /*
Packet command                       */
  ptCIPSyncObj->tHead.ulExt     = 0;                                            /*
Extension                            */
  ptCIPSyncObj->tHead.ulRout    = 0;                                            /*
Router                               */

  ptCIPSyncObj->tData.ulSync0Interval = 1000000000;                             /*
Sync 0 Time Interval: 1s */
  ptCIPSyncObj->tData.ulSync0Offset   = 0;                                      /*
Sync 0 Offset: 0s        */
  ptCIPSyncObj->tData.ulSync1Interval = 1000000000;                             /*
Sync 1 Time Interval: 1s */
  ptCIPSyncObj->tData.ulSync1Offset   = 150;                                    /*
Sync 0 Offset: 150ns     */

} /* CreateCIPSyncObj */


/*****************************************************************************
 function:    EnableCIPService
 description: Enable PTP.

 global:      none
 input:       void* pvPck                                - pointer to the packet

 output:      none
 return:      none
*****************************************************************************/
void EnableCIPService( void* pvPck )
{
  /* Get struct of task packet union */
```

```
  EIP_OBJECT_PACKET_CIP_SERVICE_REQ_T *ptCIPServ = ( EIP_OBJECT_PACKET_CIP_SERVICE_REQ_T*
)pvPck;

  ptCIPServ->tHead.ulDest   = 0x20;                                                    /*
Destination of packet, process queue */
  ptCIPServ->tHead.ulSrc    = 0x10;                                                    /*
Source of packet, process queue      */
  ptCIPServ->tHead.ulDestId = 0;                                                       /*
Destination reference of packet      */
  ptCIPServ->tHead.ulSrcId  = 0;                                                       /*
Source reference of packet           */
  ptCIPServ->tHead.ulLen    = EIP_OBJECT_CIP_SERVICE_REQ_SIZE;                         /*
Length of packet data without header */
  ptCIPServ->tHead.ulId     = 0;                                                       /*
Identification handle of sender      */
  ptCIPServ->tHead.ulSta    = 0;                                                       /*
Status code of operation             */
  ptCIPServ->tHead.ulCmd    = EIP_OBJECT_CIP_SERVICE_REQ;                              /*
Packet command                       */
  ptCIPServ->tHead.ulExt    = 0;                                                       /*
Extension                            */
  ptCIPServ->tHead.ulRout   = 0;                                                       /*
Router                               */

  ptCIPServ->tData.ulService    = 0x10;                                                /*
Set attribute single */
  ptCIPServ->tData.ulClass      = 0x43;                                                /*
Time Sync Object     */
  ptCIPServ->tData.ulInstance   = 1;                                                   /*
Instance             */
  ptCIPServ->tData.ulAttribute  = 1;                                                   /*
Enable PTP           */

  ptCIPServ->tData.abData[0]    = 1;                                                   /*
Time Sync Object data        */
  ptCIPServ->tHead.ulLen       += 1;                                                   /*
Time Sync Object data length */

} /* EnableCIPService */


/*****************************************************************************
 function:    ConfSyncHdsk
 description: Configure Sync handshake mode.

 global:      none
 input:       void* pvPck                                  - pointer to the packet

 output:      none
 return:      none
*****************************************************************************/
void ConfSyncHdsk( void* pvPck )
{
  /* Get struct of task packet union */
  RCX_SET_HANDSHAKE_CONFIG_REQ_T *ptSyncHdsk = ( RCX_SET_HANDSHAKE_CONFIG_REQ_T* )pvPck;

  ptSyncHdsk->tHead.ulDest    = 0x20;                                                  /*
Destination of packet, process queue */
  ptSyncHdsk->tHead.ulSrc     = 0x10;                                                  /*
Source of packet, process queue      */
  ptSyncHdsk->tHead.ulDestId  = 0;                                                     /*
Destination reference of packet      */
  ptSyncHdsk->tHead.ulSrcId   = 0;                                                     /*
Source reference of packet           */
  ptSyncHdsk->tHead.ulLen     = sizeof(RCX_SET_HANDSHAKE_CONFIG_REQ_DATA_T);           /*
Length of packet data without header */
  ptSyncHdsk->tHead.ulId      = 0;                                                     /*
Identification handle of sender      */
  ptSyncHdsk->tHead.ulSta     = 0;                                                     /*
Status code of operation             */
```

```
  ptSyncHdsk->tHead.ulCmd     = RCX_SET_HANDSHAKE_CONFIG_REQ;;                          /*
Packet command                        */
  ptSyncHdsk->tHead.ulExt     = 0;                                                      /*
Extension                             */
  ptSyncHdsk->tHead.ulRout    = 0;                                                      /*
Router                                */

  ptSyncHdsk->tData.bPDInHskMode    = RCX_IO_MODE_BUFF_HST_CTRL;                        /*
Input process data handshake mode                                      */
  ptSyncHdsk->tData.bPDInSource     = 0;                                              /*
Input process data trigger source, Currently unused, set to zero */
  ptSyncHdsk->tData.usPDInErrorTh   = 0;                                              /*
Threshold for input process data handshake handling errors       */

  ptSyncHdsk->tData.bPDOutHskMode   = RCX_IO_MODE_BUFF_HST_CTRL;                      /*
Output process data handshake mode                                       */
  ptSyncHdsk->tData.bPDOutSource    = 0;                                              /*
Output process data trigger source, Currently unused, set to zero  */
  ptSyncHdsk->tData.usPDOutErrorTh  = 0;                                              /*
Threshold for output process data handshake handling errors        */

  ptSyncHdsk->tData.bSyncHskMode    = RCX_SYNC_MODE_DEV_CTRL;                         /*
Synchronization handshake mode, Device controlled */
  ptSyncHdsk->tData.bSyncSource     = RCX_SYNC_SOURCE_1;                              /*
Synchronization source */
  ptSyncHdsk->tData.usSyncErrorTh   = 0;                                              /*
Threshold for synchronization handshake handling errors */

  ptSyncHdsk->tData.aulReserved[0]  = 0;                                              /*
Reserved for future use. Set to zero. */
  ptSyncHdsk->tData.aulReserved[1]  = 0;                                              /*
Reserved for future use. Set to zero. */

} /* ConfSyncHdsk */
```

7.2.3 Main.c

```
/*****************************************************************************

Copyright (c) Hilscher GmbH. All Rights Reserved.

*****************************************************************************

  Filename:
    $Id: Main.c 31696 2013-01-31 09:42:12Z gordon $:
  Last Modification:
    $Revision:: 31696           $:  Revision of last commit
    $Author:: gordon            $:  Author of last commit
    $Date:: 2013-01-31 10:42:12#$:  Date of last commit

  Targets:
    Win32/ANSI   : yes
    Win32/Unicode: no (define _UNICODE)
    WinCE        : no
    rcX          : no

  Description:
    Host PC Example
    Main
    This simple example show programming and configuration of Host side.

  Changes:
    Name    Date          Version     Description
    ----------------------------------------------------------------------------
    GG      2013-01-31    1.000       created
*****************************************************************************/

/*****************************************************************************/
/* Includes */
```

```
#include <windows.h>                                                            /*
Include Windows-specific header file for the C */
#include <stdio.h>                                                              /*
Include C standard library input/output header */

#include "CIFxuser.h"                                                           /*
Include cifX driver API definition       */
#include "CIFxErrors.h"                                                         /*
Include cifX driver API error definition */

#include "SetConfigParams.h"                                                    /*
Inlucde Host PC example specifics header */
#include "SetCIPSync.h"                                                         /*
Inlucde Host PC example specifics header */


/*****************************************************************************/
/* Host PC Example, Main */


/*****************************************************************************/
/* Definition Area for CIP Sync Demo */

/* Systime TimeStamp structure contains Nanoseconds and Seconds */
typedef struct SYSTIME_TIMESTAMP_Ttag
{
  UINT32 ulTimeS;
  UINT32 ulTimeNs;
} SYSTIME_TIMESTAMP_T;

/* CIP Sync Demo resourcen parameter structure */
typedef struct CIP_SYNC_RES_PARAMtag
{
  HANDLE                                hChannel;
  HANDLE                                hSysdevice;
  uint32_t                              ulErrorCount;
  BOOL                                  ulConfigured;
  SYSTIME_TIMESTAMP_T                   tHostStartTime;
  SYSTIME_TIMESTAMP_T                   tCIPSyncTimeStamp;
  SYSTEM_CHANNEL_SYSTEM_STATUS_BLOCK    tSysStatusBlock;
} CIP_SYNC_RES_PARAM;

CIP_SYNC_RES_PARAM tSyncResParms = {0};


/*****************************************************************************
 function:    DumpData
 description: Displays a hex dump on the debug console (16 bytes per line).

 global:      none
 input:       unsigned char* pbData                        - pointer to dump data
              unsigned long ulDataLen                      - length of data dump

 output:      none
 return:      none
*****************************************************************************/
void DumpData( unsigned char* pbData, unsigned long ulDataLen )
{
  unsigned long ulIdx;

  for( ulIdx = 0; ulIdx < ulDataLen; ++ulIdx )
  {
    if( 0 == ( ulIdx % 16 ) )
      printf( "\r\n" );

    printf( "%02X ", pbData[ulIdx] );
  }
  printf( "\r\n" );

} /* DumpData */
```

```
/*****************************************************************************
 function:    DumpPacket
 description: Dumps a rcX packet to debug console.

 global:      none
 input:       CIFX_PACKET* ptPacket                        - pointer to packed being
                                                             dumped

 output:      none
 return:      none
*****************************************************************************/
void DumpPacket( CIFX_PACKET* ptPacket )
{
  printf( "Dest   : 0x%08X      ID   : 0x%08X\r\n", ptPacket->tHeader.ulDest,   ptPacket-
>tHeader.ulId );
  printf( "Src    : 0x%08X      Sta  : 0x%08X\r\n", ptPacket->tHeader.ulSrc,    ptPacket-
>tHeader.ulState );
  printf( "DestID : 0x%08X      Cmd  : 0x%08X\r\n", ptPacket->tHeader.ulDestId, ptPacket-
>tHeader.ulCmd );
  printf( "SrcID  : 0x%08X      Ext  : 0x%08X\r\n", ptPacket->tHeader.ulSrcId,  ptPacket-
>tHeader.ulExt );
  printf( "Len    : 0x%08X      Rout : 0x%08X\r\n", ptPacket->tHeader.ulLen,    ptPacket-
>tHeader.ulRout );

  printf( "Data:" );

  /* Displays a hex dump on the debug console (16 bytes per line) */
  DumpData( ptPacket->abData, ptPacket->tHeader.ulLen );

} /* DumpPacket */


/*****************************************************************************
 function:    fnSyncClb
 description: Function to be called if CIFX SYNC event occurs.

 global:      none
 input:       none

 output:      none
 return:      none
*****************************************************************************/
void APIENTRY fnSyncClb(uint32_t ulNotification, uint32_t ulDataLen, void* pvData, void*
pvUser)
{
  long                lRet            = 0;                                          /*
Return value for common error codes          */
  CIP_SYNC_RES_PARAM* ptSyncResParams = {0};                                        /*
CIP Sync Demo resourcen parameter structure  */

  UNREFERENCED_PARAMETER( pvData );
  UNREFERENCED_PARAMETER( ulDataLen );
  UNREFERENCED_PARAMETER( ulNotification );

  (CIP_SYNC_RES_PARAM*) ptSyncResParams = (CIP_SYNC_RES_PARAM*)pvUser;

  /* Read the CIP Sync Time Stamp */
  lRet = xChannelExtendedStatusBlock( ptSyncResParams->hChannel, CIFX_CMD_READ_DATA,
0xA4, 8, &ptSyncResParams->tCIPSyncTimeStamp);

  /* Get System device specific information */
  lRet = xSysdeviceInfo( ptSyncResParams->hSysdevice,
                         CIFX_INFO_CMD_SYSTEM_STATUS_BLOCK,
                         sizeof( SYSTEM_CHANNEL_SYSTEM_STATUS_BLOCK ),
                         &ptSyncResParams->tSysStatusBlock );

  /* Wait for synchronization event or trigger/acknowledge sync */
```

```
  lRet = xChannelSyncState( ptSyncResParams->hChannel, CIFX_SYNC_ACKNOWLEDGE_CMD, 0,
&ptSyncResParams->ulErrorCount );

  /* Copy time since start in seconds to local CIP Sync resourcen parameter */
  memcpy( &ptSyncResParams->tHostStartTime.ulTimeS, &ptSyncResParams-
>tSysStatusBlock.ulTimeSinceStart, sizeof( ptSyncResParams-
>tSysStatusBlock.ulTimeSinceStart ) );

  /* Display the CIP Syn Time Stamp */
  printf( "CIP Sync Time Stamp" );
  DumpData( &ptSyncResParams->tCIPSyncTimeStamp, sizeof( ptSyncResParams-
>tCIPSyncTimeStamp ) );

  /* Display the local Time Stamp since start in seconds */
  printf( "Local Time Stamp" );
  DumpData( &(unsigned char)ptSyncResParams->tHostStartTime.ulTimeS, sizeof(
ptSyncResParams->tHostStartTime.ulTimeS ) );

} /* fnSyncClb */


/*****************************************************************************
 function:    fnEventClb
 description: Function to be called if event occurs.

 global:      none
 input:       none

 output:      none
 return:      none
*****************************************************************************/
void APIENTRY fnEventClb( uint32_t ulNotification, uint32_t ulDataLen, void* pvData,
void* pvUser )
{
  UNREFERENCED_PARAMETER( pvUser);
  UNREFERENCED_PARAMETER( pvData);
  UNREFERENCED_PARAMETER( ulDataLen);

  switch (ulNotification)
  {
  case CIFX_NOTIFY_RX_MBX_FULL:
    printf("fnEventClb(): CIFX_NOTIFY_RX_MBX_FULL\n" );
    break;

  case CIFX_NOTIFY_TX_MBX_EMPTY:
    printf("fnEventClb(): CIFX_NOTIFY_TX_MBX_EMPTY\n" );
    break;

  case CIFX_NOTIFY_PD0_IN:
    printf("fnEventClb(): CIFX_NOTIFY_PD0_IN\n" );
    break;

  case CIFX_NOTIFY_PD0_OUT:
    printf("fnEventClb(): CIFX_NOTIFY_PD0_OUT\n" );
    break;

  case CIFX_NOTIFY_PD1_IN:
    printf("fnEventClb(): CIFX_NOTIFY_PD1_IN\n" );
    break;

  case CIFX_NOTIFY_PD1_OUT:
    printf("fnEventClb(): CIFX_NOTIFY_PD1_OUT\n" );
    break;

  default:
    printf("fnEventClb(): UNKNOWN Event, Event number %u\n", ulNotification);
    break;
  }
}
```

```
/*****************************************************************************
 function:    main
 description: Main entry function.

 global:      none
 input:       int argc                                    - argument counter from the
                                                            console
              char *argv[]                                - pointer field to the
                                                            arguments from the console

 output:      none
 return:      int ( CIFX_NO_ERROR == succeeded )
*****************************************************************************/
int main( int argc, char *argv[] )
{
  CIFXHANDLE    hDriver         = NULL;                                       /*
Handle of cifX driver                    */
  long          lRet            = 0;                                          /*
Return value for common error codes      */
  unsigned long ulState         = 0;                                          /*
Actual state returned                    */
  unsigned long ulTimeout       = 1000;                                       /*
Timeout in milliseconds                  */
  CIFX_PACKET   tSendPkt        = {0};                                        /*
Packet to be send                        */
  CIFX_PACKET   tRecvPkt        = {0};                                        /*
Buffer to returned packet                */
  BOOL          fRunning        = TRUE;                                       /*
Cyclic exchange running flag             */

  unsigned char abSendData[32]  = {0};                                        /*
Defining the send process data field     */
  unsigned char abRecvData[32]  = {0};                                        /*
Defining the receive process data field  */
  unsigned long ulCycle         = 0;

  printf( "---------- Host Demo ----------\r\n" );

  /* Opens the driver, allowing access to every driver function */
  lRet = xDriverOpen( &hDriver );

  if( lRet == CIFX_NO_ERROR )
  {
    CIFXHANDLE hChannel = NULL;
    CIFXHANDLE hSys     = NULL;

    /* Opens a connection to a communication channel */
    lRet = xChannelOpen( NULL, "cifX0", 0, &hChannel );

    if( lRet != CIFX_NO_ERROR )
    {
      printf( "Error opening Channel: 0x%08X", lRet );
    }
    else
    {
      /* Store handle of the channel */
      tSyncResParms.hChannel = hChannel;

      /* Open a connection to a system device on the passed board */
      lRet = xSysdeviceOpen(hDriver, "cifX0", &hSys);

      if( lRet != CIFX_NO_ERROR )
      {
        printf( "Error opening boards system device: 0x%08X", lRet );
      }
      else
      {
        CHANNEL_INFORMATION tChannelInfo = {0};

        /* Store handle to boards system device */
```

```
        tSyncResParms.hSysdevice = hSys;

        /* Retrieve the global communication channel information */
        if( CIFX_NO_ERROR != ( lRet = xChannelInfo( hChannel, sizeof( CHANNEL_INFORMATION
), &tChannelInfo ) ) )
        {
          printf( "Error querying system information block: 0x%08X !\r\n", lRet );
        }
        else
        {
          printf( "Communication Channel Info:\r\n" );
          printf( "Device Number    : %u\r\n", tChannelInfo.ulDeviceNumber );
          printf( "Serial Number    : %u\r\n", tChannelInfo.ulSerialNumber );
          printf( "Firmware         : %s\r\n", tChannelInfo.abFWName );
          printf( "FW Version       : %u.%u.%u build %u\r\n",
                  tChannelInfo.usFWMajor,
                  tChannelInfo.usFWMinor,
                  tChannelInfo.usFWRevision,
                  tChannelInfo.usFWBuild );
          printf( "FW Date          : %02u/%02u/%04u\r\n",
                  tChannelInfo.bFWMonth,
                  tChannelInfo.bFWDay,
                  tChannelInfo.usFWYear );
          printf( "Mailbox Size     : %u\r\n", tChannelInfo.ulMailboxSize );
        }

        /* Set the Application state flag in the application COS flags */
        lRet = xChannelHostState( hChannel, CIFX_HOST_STATE_READY, &ulState, 1000 );

        /* Create protocol specific set configuration packet */
        SetConfigParamsPkt( ( void* )&tSendPkt );

        /* Send a packet to the channel's mailbox */
        lRet = xChannelPutPacket( hChannel, &tSendPkt, ulTimeout );

        if( lRet != CIFX_NO_ERROR )
        {
          printf( "Error sending packet to device: 0x%08X !\r\n", lRet );
        }
        else
        {
          printf( "Send Packet:\r\n" );

          /* Dumps a rcX packet to debug console */
          DumpPacket( &tSendPkt );

          /* Retrieve a pending packet from the channel mailbox */
          lRet = xChannelGetPacket( hChannel, sizeof( tRecvPkt ), ( void* )&tRecvPkt,
ulTimeout );

          if( lRet != CIFX_NO_ERROR )
          {
            printf( "Error getting packet from device: 0x%08X !\r\n", lRet );
          }
          else
          {
            printf( "Received Packet:\r\n" );

            /* Dumps a rcX packet to debug console */
            DumpPacket( &tRecvPkt );
          }
        }

        /* Reset the given communication channel, after Set Configuration packet */
        lRet = xChannelReset( hChannel, CIFX_CHANNELINIT, 3000 );

        /* Set the bus state flag in the application COS state flags, to start
communication */
        do
        {
```

```
          lRet = xChannelBusState( hChannel, CIFX_BUS_STATE_ON, &ulState, 10 );
        }
        while( lRet == CIFX_DEV_NOT_RUNNING );

        /* Check the netX state */
        if( ( lRet == CIFX_NO_ERROR ) || ( lRet == CIFX_DEV_NO_COM_FLAG ) )
        {
          /* Cyclic exchange IO data until fRunning is set to false */
          while( fRunning )
          {
            /* Wait for communication to be established */
            if( CIFX_DEV_NO_COM_FLAG == ( lRet = xChannelBusState( hChannel,
CIFX_BUS_STATE_ON, &ulState, 10 ) ) )
            {
              Sleep( 10 );
              continue;
            }
            else
            {
              /* Check for CIP Sync currently complete configured */
              if( tSyncResParms.ulConfigured == FALSE )
              {
                /* ## First packet */
                /* Create CIP Time Sync Object */
                CreateCIPSyncObj( ( void* )&tSendPkt );

                /* Send a packet to the channel's mailbox */
                lRet = xChannelPutPacket( hChannel, &tSendPkt, ulTimeout );

                if( lRet != CIFX_NO_ERROR )
                {
                  printf( "Error sending packet to device: 0x%08X !\r\n", lRet );
                }
                else
                {
                  printf( "Send Packet:\r\n" );

                  /* Dumps a rcX packet to debug console */
                  DumpPacket( &tSendPkt );

                  /* Retrieve a pending packet from the channel mailbox */
                  lRet = xChannelGetPacket( hChannel, sizeof( tRecvPkt ), ( void*
)&tRecvPkt, ulTimeout );

                  if( lRet != CIFX_NO_ERROR )
                  {
                    printf( "Error getting packet from device: 0x%08X !\r\n", lRet );
                  }
                  else
                  {
                    printf( "Received Packet:\r\n" );

                    /* Dumps a rcX packet to debug console */
                    DumpPacket( &tRecvPkt );
                  }
                }

                /* ## Second packet */
                /* Enable PTP */
                EnableCIPService( ( void* )&tSendPkt );

                /* Send a packet to the channel's mailbox */
                lRet = xChannelPutPacket( hChannel, &tSendPkt, ulTimeout );

                if( lRet != CIFX_NO_ERROR )
                {
                  printf( "Error sending packet to device: 0x%08X !\r\n", lRet );
                }
                else
                {
```

```
                  printf( "Send Packet:\r\n" );

                  /* Dumps a rcX packet to debug console */
                  DumpPacket( &tSendPkt );

                  /* Retrieve a pending packet from the channel mailbox */
                  lRet = xChannelGetPacket( hChannel, sizeof( tRecvPkt ), ( void*
)&tRecvPkt, ulTimeout );

                  if( lRet != CIFX_NO_ERROR )
                  {
                    printf( "Error getting packet from device: 0x%08X !\r\n", lRet );
                  }
                  else
                  {
                    printf( "Received Packet:\r\n" );

                    /* Dumps a rcX packet to debug console */
                    DumpPacket( &tRecvPkt );
                  }
                }

                /* ## Third packet */
                /* Configure Sync handshake mode */
                ConfSyncHdsk( ( void* )&tSendPkt );

                /* Send a packet to the channel's mailbox */
                lRet = xChannelPutPacket( hChannel, &tSendPkt, ulTimeout );

                if( lRet != CIFX_NO_ERROR )
                {
                  printf( "Error sending packet to device: 0x%08X !\r\n", lRet );
                }
                else
                {
                  printf( "Send Packet:\r\n" );

                  /* Dumps a rcX packet to debug console */
                  DumpPacket( &tSendPkt );

                  /* Retrieve a pending packet from the channel mailbox */
                  lRet = xChannelGetPacket( hChannel, sizeof( tRecvPkt ), ( void*
)&tRecvPkt, ulTimeout );

                  if( lRet != CIFX_NO_ERROR )
                  {
                    printf( "Error getting packet from device: 0x%08X !\r\n", lRet );
                  }
                  else
                  {
                    printf( "Received Packet:\r\n" );

                    /* Dumps a rcX packet to debug console */
                    DumpPacket( &tRecvPkt );
                  }
                }

                /* Delay */
                Sleep( 1000 );

                /* Register a notification callback */
                if( /*( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_RX_MBX_FULL,  fnEventClb, (void*)1))) ||
                    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_TX_MBX_EMPTY, fnEventClb, (void*)2))) ||
                    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_PD0_IN,        fnEventClb, (void*)3))) ||
                    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_PD0_OUT,       fnEventClb, (void*)5))) ||*/
```

```
                    ( CIFX_NO_ERROR != (lRet = xChannelRegisterNotification( hChannel,
CIFX_NOTIFY_SYNC,          fnSyncClb,  (void*)&tSyncResParms ) ) ) )
                {
                  /* Failed to register one of the events */
                  printf( "Error to register SYNC event: 0x%08X !\r\n", lRet );
                }
                else
                {
                  /* Configuration completed succesfully */
                  tSyncResParms.ulConfigured = TRUE;

                  /* Get actual host state */
                  if( ( lRet = xChannelHostState( hChannel, CIFX_HOST_STATE_READ,
&ulState, 0L ) ) != CIFX_NO_ERROR )
                  {
                    /* Read driver error description */
                    printf( "Error HOST State: 0x%08X !\r\n", lRet );
                  }

                  /* Set host ready */
                  if( ( lRet = xChannelHostState( hChannel, CIFX_HOST_STATE_READY, NULL,
2000L ) ) != CIFX_NO_ERROR )
                  {
                    /* Read driver error description */
                    printf( "Error HOST State: 0x%08X !\r\n", lRet );
                  }

                  /*--------------------*/
                  /* Wait for user      */
                  /*--------------------*/
                  printf("\n\rEVENT-Handling aktiv.......!!!!!\r\n");
                }
              }

              /* Instructs the device to place the latest data into the DPM and passes
them to the user */
              if( CIFX_NO_ERROR != ( lRet = xChannelIORead( hChannel, 0, 0, sizeof(
abRecvData ), abRecvData, 10 ) ) )
              {
                printf( "Error reading IO Data area: 0x%08X !\r\n", lRet );
                break;
              }
              else
              {
                /*printf( "IORead Data:" );*/

                /* Displays a hex dump on the debug console (16 bytes per line) */
                /*DumpData( abRecvData, sizeof( abRecvData ) );*/

                /* Copies the data to the DPM and waits for the firmware to retrieve them
*/
                if( CIFX_NO_ERROR != ( lRet = xChannelIOWrite( hChannel, 0, 0, sizeof(
abSendData ), abSendData, 10 ) ) )
                {
                  printf( "Error writing to IO Data area: 0x%08X !\r\n", lRet );
                  break;
                }
                else
                {
                  /*printf( "IOWrite Data:" );*/

                  /* Displays a hex dump on the debug console (16 bytes per line) */
                  /*DumpData( abSendData, sizeof( abSendData ) );*/

                  memset( abSendData, ++ulCycle, sizeof( abSendData ) );
                }
              }
              /* Delay */
              Sleep( 200 );
            }
```

```
          }
        }
        else
        {
          printf( "Error setting application COS bus on flag: 0x%08X !\r\n", lRet );
        }

        /* Check configuration completed succesfully */
        if( tSyncResParms.ulConfigured == TRUE )
        {
          /* Unregister a notification callback */
          if( /*( CIFX_NO_ERROR != (lRet = xChannelUnregisterNotification( hChannel,
CIFX_NOTIFY_RX_MBX_FULL  ) ) ) ||
              ( CIFX_NO_ERROR != (lRet = xChannelUnregisterNotification( hChannel,
CIFX_NOTIFY_TX_MBX_EMPTY ) ) ) ||
              ( CIFX_NO_ERROR != (lRet = xChannelUnregisterNotification( hChannel,
CIFX_NOTIFY_PD0_IN       ) ) ) ||
              ( CIFX_NO_ERROR != (lRet = xChannelUnregisterNotification( hChannel,
CIFX_NOTIFY_PD0_OUT      ) ) ) ||*/
              ( CIFX_NO_ERROR != (lRet = xChannelUnregisterNotification( hChannel,
CIFX_NOTIFY_SYNC         ) ) ) )
          {
            /* Failed to register one of the events */
            printf( "Error to unregister SYNC event: 0x%08X !\r\n", lRet );
          }
        }

        /* Set the bus state flag in the application COS state flags, to stop
communication */
        xChannelBusState( hChannel, CIFX_BUS_STATE_OFF, &ulState, 10 );

        /* Set Host not ready to stop bus communication */
        xChannelHostState( hChannel, CIFX_HOST_STATE_NOT_READY, &ulState, ulTimeout );

        /* Close a connection to a system device */
        xSysdeviceClose( hSys );
      }
      /* Close a connection to a communication channel */
      xChannelClose( hChannel );
    }
  }
  /* Closes an open connection to the driver */
  xDriverClose( hDriver );

  printf( " State = 0x%08X !\r\n", lRet );
  printf( "-------------------------------------------------\r\n" );

  return lRet;

} /* main */
```

7.3 Example File Headers

Note: This section only applies to EtherNet/IP Adapter stack version 2.x.

7.3.1 SetCIPSync.h

```
/**************************************************************************************

Copyright (c) Hilscher GmbH. All Rights Reserved.

*/

#ifndef __SETCIPSYNC_H__
#define __SETCIPSYNC_H__


/*****************************************************************************************/
/* Prototypes */

/* Create CIP Time Sync Object */
void
CreateCIPSyncObj
(
 void* pvPck
);

/* Enable PTP */
void
EnableCIPService
(
 void* pvPck
);

/* Configure Sync handshake mode */
void
ConfSyncHdsk
(
 void* pvPck
);

/* Function to be called if CIFX SYNC event occurs */
void APIENTRY
fnSyncClb
(
  uint32_t ulNotification,
  uint32_t ulDataLen,
  void* pvData,
  void* pvUser
);

/* Function to be called if event occurs */
void APIENTRY
fnEventClb
(
  uint32_t ulNotification,
  uint32_t ulDataLen,
  void* pvData,
  void* pvUser
);


/*****************************************************************************************/

#endif /* __SETCIPSYNC_H__ */
```

7.3.2 SetConfigParams.h

```
/**************************************************************************************

Copyright (c) Hilscher GmbH. All Rights Reserved.

*/

#ifndef __SETCONFIG_PARAMETERS_H__
#define __SETCONFIG_PARAMETERS_H__


/**************************************************************************************/
/* Prototypes */

/* Create protocol specific set configuration packet */
void
SetConfigParamsPkt
(
 void* pvPck
);

/**************************************************************************************/

#endif /* __SETCONFIG_PARAMETERS_H__ */
```

7.3.3 Other Headers

Further used Headers are (see example):

- cifXErrors.h
- cifXUser.h
- EipAps_Public.h
- EipObject_Public.h
- rcx_error.h
- rcX_Public.h
- rcX_User.h
- rX_Results.h
- rX_Types.h
- stdint.h
- TcpipConfig.h
- TcpipTcpTask_Public.h
- TLR_Packet.h
- TLR_Results.h
- TLR_Types.h

7.4 Sync State Machine

There is a Synchronization State Machine for CIP Sync with the following states:

1. Power on
2. Disabled (Deactivate)
3. Init
4. Listening
5. Uncalibrate
6. Slave
7. Fault

For more information on these states see IEEE 1588:2008, Sect. 9.2.5 “State machines”.

The current state can be read out via read access to the CIP Sync Object.

8 Glossary

Announce Message

According to IEEE 1588 an announce message is used to establish the synchronization hierarchy.

An announce message contains various data including a timestamp and information about the grandmaster clock. For instance, if a new grandmaster clock takes over this role, it will send an announce message.

Best Master Clock Algorithm

This is the most crucial part of the IEEE 1588 time synchronization technology and the Precision Time Protocol. It is an algorithm for the determination of the most precise clock on the entire network or a single network segment. All slaves use the same algorithm.

BMCA

See Best Master Clock Algorithm

Boundary Clock

A boundary clock is a clock with more than one port that separates two or more distinct communication paths. It is able to transfer time synchronization messages over network boundaries.

Change of State

Collection of flags that initiate execution of certain commands or signal a change of state.

CIP Sync

Technology for time synchronization to be applied within CIP networks.

COS

See Change of State

Delay_Req Message

The Delay_Req message is sent from the slave to the master in order to determine the transmission time in this direction (t_{sm}). It contains an origin timestamp i.e. a timestamp taken at t_3.

Delay_Req messages should be processed as soon as possible after having been received.

Delay_Resp Message

The Delay_Resp message is sent from the master to the slave in order to inform the slave about the determined transmission time from the slave to the master (t_3). It contains the receive timestamp t_4 required to calculate the transmission time t_{sm} for message passing between slave and master and the identity of the requesting port..

Event messages

Event messages are timed messages i.e. time stamps are generated both at their transmission and reception. The following types of event messages are defined in IEEE 1588:

- Sync messages
- Delay_Req messages
- Pdelay_Req messages
- Pdelay_Resp messages

Only the first two kinds of messages are relevant in the scope of this document.

Follow-up messages

In two-step sending mode, the follow-up message is sent from the master to the slave in order to determine the transmission time in this direction (t_{ms}). It contains an precision timestamp taken at t_1.

Follow-up messages should be sent as fast as possible after the associated sync message.

They have to be sent prior to the subsequent sync message.

General messages

General messages do not produce time stamps. The following types of general messages are defined in IEEE 1588:

- Announce messages
- Follow-up messages
- Delay_Resp messages
- Pdelay_Resp follow-up messages
- Management messages
- Signaling messages

Of these, only announce messages, follow-up messages and Delay_Resp messages are relevant in the scope of this document.

Grandmaster Clock

The clock having been determined as the most precise one on the entire network is denominated as grandmaster clock.

IEEE 1588

Standard defining a widely usable technology for time synchronization based on the Precision Time Protocol (PTP).

LFW

Loadable Firmware

Local Time

Time of a local clock, typically a slave clock (here: EtherNet/IP Adapter)

LOM

Linkable Object Modules

Master Clock

A clock that sends time information to other clocks within its network segment in order to synchronize these with itself.

Offset

Calculated time difference between Local Time and System Time. The offset is used to correct deviations of the local time.

Ordinary Clock

An ordinary clock may be either a time master or a time slave. It is not able to transfer time synchronization messages over network boundaries.

Precision Time Protocol

Protocol for high precision time synchronization defined within IEEE 1588.

PTP

See Precision Time Protocol

Slave Clock

A clock that receives time information from a master or grandmaster clock in order to synchronize itself with this (grand)master clock.

Sync 0

Hardware signal of the netX designed for synchronization purposes

Sync 1

Another hardware signal of the netX designed for synchronization purposes

Sync Message

The Sync message is periodically sent from the master to the slave in order to determine the transmission time in this direction (t_{ms}). It contains either an estimated timestamp taken at t_1 or the value 0.

System Time

Time of the grandmaster (or master if only a network segment is taken into account)

TC

Transparent Clock

Time stamp

A time stamp contains the time information when a defined event (such as a Sync 0/ Sync 1 signal, for instance) occurred. In PTP, time stamps are send within some message types, for instance within the Follow_up message.

Time Sync Object

The Time Sync Object provides an interface between CIP on one hand and PTP as defined in IEEE 1588:2008 on the other hand. It allows easy use of PTP from CIP-based applications.

Transparent Clock

A participant in the PTP network which receives time signals and sends them to another receiver after performing corrections.

Transparent clocks are only defined in the IEEE 1588:2008 standard, but not in the elder standard IEEE 1588:2002.

UDP

User Datagram Protocol

9 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 Systemautomation (Shanghai) Co. Ltd.
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.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

Italy
Hilscher Italia S.r.l.
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.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
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