SYS TEC ELECTRONIC

Introduction into openPOWERLINK

Software Manual

Edition April 2008

system house for distributed automation

	EUROPE	NORTH AMERICA
Address:	SYS TEC electronic GmbH August-Bebel-Str. 29 D-07973 Greiz GERMANY	PHYTEC America LLC 203 Parfitt Way SW, Suite G100 Bainbridge Island, WA 98110 USA
Ordering Information:	+49 (3661) 6279-0 info@systec-electronic.com	1 (800) 278-9913 info@phytec.com
Technical Support:	+49 (3661) 6279-0 support@systec-electronic.com	1 (800) 278-9913 support@phytec.com
Fax:	+49 (3661) 62 79 99	1 (206) 780-9135
Web Site:	http://www.systec-electronic.com	http://www.phytec.com

2nd Edition April 2008

1 Introduction

1.1 Ethernet POWERLINK

Ethernet POWERLINK is a Real-Time Ethernet field bus system. It is based on the Fast Ethernet Standard IEEE 802.3.

A managing node (MN), which acts as the master in the EPL network, polls the controlled nodes (CN) cyclically. This process takes place in the isochronous phase of the EPL cycle. Immediately after the isochronous phase follows an asynchronous phase for communication which is not time-critical, e.g. TCP/IP communication. The isochronous phase starts with a Start of Cyclic frame on which all nodes are synchronized. This schedule design avoids collisions, which are usually present on Standard Ethernet, and ensures the determinism of the hard real-time communication. It is implemented in the EPL data link layer. The EPL network can be connected via gateways to non real-time networks.

The communication profile of Ethernet POWERLINK is adapted from CANopen. Thus the design principles like process data object (PDO) for exchange of process variables and service data object (SDO) for configuration of remote object dictionaries are reused. All PDOs are exchanged within the isochronous phase similar to the synchronous PDOs of CANopen, because event triggered PDOs would interfere with the hard real-time requirements.

To conform to IEEE 802.3 each EPL device has got a unique MAC address. Additionally each device is assigned a logical node ID. Mostly, this node ID can be configured via node switches on the device. If a particular EPL device implements a TCP/IP stack it gets a private IP address from class C within the network 192.168.100.0 where the host part equals the EPL node ID.

It is assumed that you are familiar with the Ethernet POWERLINK V2.0 Communication Profile Specification.

1.2 Key features

- Data link layer and NMT state machine for Controlled and Managing Nodes
- SDO via UDP and EPL ASnd frames
- Dynamic PDO mapping
- User-configurable object dictionary
- Supports the EPL cycle features async-only CN and multiplexed CN
- Implemented in plain ANSI C
- Modular software structure for simple portability to different target platforms (with and without operating system)
- Event driven Communication Abstraction Layer
- Common API to application program

1.3 Software Structure

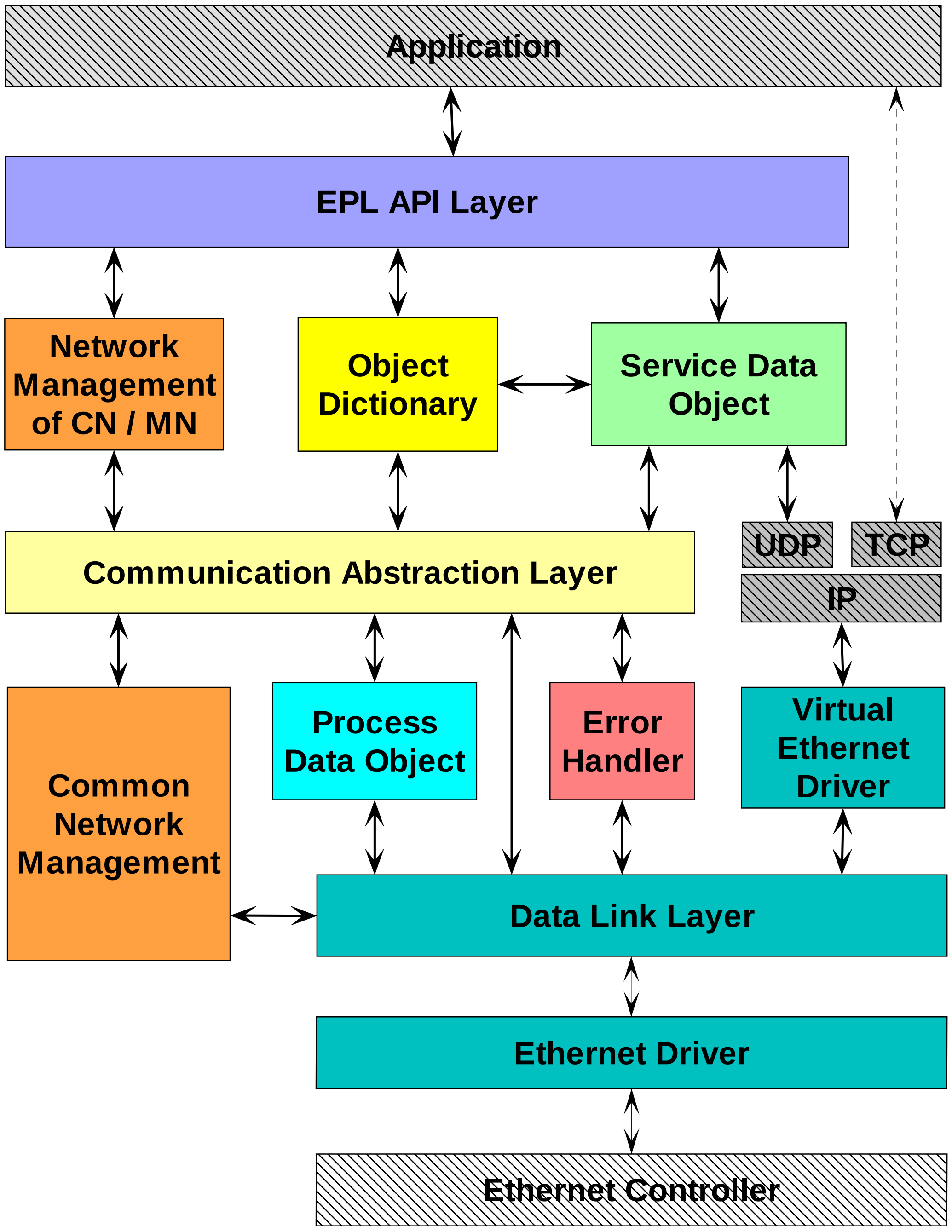

Figure 1 Software structure

1.4 Functional Survey

1.4.1 EPL API layer

The EPL API layer provides a simple interface to the application. The application uses functions to initialize the EPL stack and perform different tasks. The EPL stack informs the application via a callback function about occurred events. For example these events include NMT state changes, node state changes, object dictionary accesses, finishing of SDO transfers, EPL stack errors etc.

1.4.2 Communication Abstraction layer

The EPL stack is devided into a hard-realtime task which processes the cyclic events and a low-priority task which is responsible for asynchronous events like SDO processing. The communication between these tasks is encapsulated in the Communication Abstraction Layer which is designed after the event model. This allows easy porting and optimization to new target platforms. For example it is possible to use the highly optimized means for interprocess communication of the underlying operating system.

1.4.3 Object Dictionary and Service Data Object (SDO)

The configuration of the EPL stack takes place via the Object Dictionary. This can be performed at compile time via appropriate default entries and at run time by the application or remotely via SDO transfers. The EPL stack supports SDO via UDP and EPL ASnd frames. It uses an existing UDP/IP stack for SDO via UDP, e.g. the one which may be supplied by the operating system or a stand-alone UDP/IP stack. The virtual Ethernet driver provides means to the UDP/IP stack for communication over the EPL network. Besides SDO via UDP this enables also the application or other tasks like a Web server to perform UDP or TCP communication over the EPL network.
The application can map any variable to an object dictionary entry (see PDO).

Depending on the concerning object entry the application is informed about every read or write access. The application may reject the access before it is actually performed or trigger any action.

1.4.4 Process Data Object (PDO)

The process variables are exchanged via PDOs between the nodes in the EPL network. Therefore the application must map these variables to object dictionary entries. The PDO mapping assigns object dictionary entries to specified PDOs. The PDO mapping can be changed dynamically by the application or via SDO transfers.
Ethernet POWERLINK exchanges the PDOs cyclically.

1.4.5 Managing Node

The Managing Node is enabled by setting the node ID to 240. It performs the bootup process according to the EPL specification version 2.0 including the support of mandatory and optional controlled nodes.

2 How to integrate the EPL stack into your application

The following sections introduce the utilization of the EPL stack. It is meant to be a short introduction. For further details refer to the Software Manual “Ethernet POWERLINK Protocol Stack”.

2.1 Configuration of the EPL stack

The functionality of the EPL stack can be configured via C-defines in the header file EplCfg.h. Normally, this file resides in the project directory. There are various configuration options, but the most important ist the bit field EPL_MODULE_INTEGRATION. It defines which modules of the EPL stack are actually included in the project. For a standard controlled node with virtual Ethernet driver you can use the following definition.

```
#define EPL_MODULE_INTEGRATION (0 \
    | EPL_MODULE_OBDK           \
    | EPL_MODULE_PDOK           \
    | EPL_MODULE_SDOS           \
    | EPL_MODULE_SDOC           \
    | EPL_MODULE_SDO_ASND       \
    | EPL_MODULE_SDO_UDP        \
    | EPL_MODULE_VETH           \
    | EPL_MODULE_DLLK           \
    | EPL_MODULE_DLLU           \
    | EPL_MODULE_NMT_CN         \
    | EPL_MODULE_NMT_MN         \
    | EPL_MODULE_NMTU           \
    | EPL_MODULE_NMTK           \
    )
```

This controlled node includes an object dictionary, PDO support, SDO server and client, SDO via ASnd and UDP, virtual Ethernet driver and off course the data link layer module and the generic, the CN and the MN specific NMT modules.

2.2 Initialisation of the EPL stack.

At first it is necessary to initialize the EPL stack with several parameters like node ID, MAC address and device identification.

Most of these initialization parameters are copied to the appropriate object dictionary entries after resetting the object dictionary. For example the parameter m_dwCycleLen is copied to object 0x1006.
The parameters m_uiNodeId, m_dwIpAddress, m_fAsyncOnly, m_uiIsochrTxMaxPayload, m_uiIsochrRxMaxPayload, m_dwAsndMaxLatency and m_dwPresMaxLatency are mandatory and must be set to valid values. Additionally the parameter m_abMacAddress has to be set either to {0}, if the Ethernet driver is able to use a MAC address stored in EEPROM, or to a valid MAC address.
If the other parameters are set to the maximum value of the data type (i.e. -1 for unsigned integer types) the parameters are ignored and the corresponding default values of the object dictionary are taken.
Additionally the initialization function needs the function pointers of the application’s event and sync callback function. The latter one may be NULL. If the initialization function encounters an error it returns a value unequal kEplSuccessful. The following code fragment shows the initialization of an EPL device with node ID 240, which will operate the stack as MN.

```
tEplKernel              EplRet;
const BYTE              abMacAddr[] =
                 {0x00, 0x12, 0x34, 0x56, 0x78, 0x9A};
static tEplApiInitParam EplApiInitParam = {0};

    EplApiInitParam.m_uiSizeOfStruct =
                  sizeof (EplApiInitParam);
    EPL_MEMCPY(EplApiInitParam.m_abMacAddress,
               abMacAddr,
               sizeof(EplApiInitParam.m_abMacAddress));
    EplApiInitParam.m_uiNodeId = 240;  // MN
    EplApiInitParam.m_dwIpAddress = 0xC0A86401;
    EplApiInitParam.m_uiIsochrTxMaxPayload = 100;
    EplApiInitParam.m_uiIsochrRxMaxPayload = 100;
    EplApiInitParam.m_dwPresMaxLatency = 50000;
    EplApiInitParam.m_dwAsndMaxLatency = 150000;
    EplApiInitParam.m_fAsyncOnly = FALSE;
    EplApiInitParam.m_dwFeatureFlags = -1;
    EplApiInitParam.m_dwCycleLen = 2000;
    EplApiInitParam.m_uiPreqActPayloadLimit = 36;
    EplApiInitParam.m_uiPresActPayloadLimit = 36;
    EplApiInitParam.m_uiMultiplCycleCnt = 0;
    EplApiInitParam.m_uiAsyncMtu = 1500;
    EplApiInitParam.m_uiPrescaler = 2;
    EplApiInitParam.m_dwLossOfFrameTolerance = 500000;
    EplApiInitParam.m_dwAsyncSlotTimeout = 3000000;
    EplApiInitParam.m_dwWaitSocPreq = 150000;
    EplApiInitParam.m_dwDeviceType = -1;
    EplApiInitParam.m_dwVendorId = -1;
    EplApiInitParam.m_dwProductCode = -1;
    EplApiInitParam.m_dwRevisionNumber = -1;
    EplApiInitParam.m_dwSerialNumber = -1;
    EplApiInitParam.m_dwSubnetMask = 0xFFFFFF00;
    EplApiInitParam.m_dwDefaultGateway = 0;

    EplApiInitParam.m_pfnCbEvent = AppCbEvent;
    EplApiInitParam.m_pfnCbSync = AppCbSync;

    // initialize EPL stack
    EplRet = EplApiInitialize(&EplApiInitParam);
    if(EplRet != kEplSuccessful)
    {
        goto Exit;
    }
```

2.3 Object dictionary and process variables

Process variables from the application can be linked to the object dictionary very easily. The header file objdict.h contains the definition of the object dictionary. Just include the following lines in this file. These lines create the object 0x6000 with 2 sub indexes, where the last sub index is a user-definable unsigned 8 bit integer variable.

```
EPL_OBD_BEGIN_PART_DEVICE ()

    EPL_OBD_BEGIN_INDEX_RAM(0x6000, 0x02, NULL)

        EPL_OBD_SUBINDEX_RAM_VAR(
            0x6000, 0x00, 0x05, 0x01,
            tEplObdUnsigned8, number_of_entries, 0x1)

        EPL_OBD_SUBINDEX_RAM_USERDEF(
            0x6000, 0x01, 0x05, 0x0B,
            tEplObdUnsigned8, Sendb1, 0x0)

    EPL_OBD_END_INDEX(0x6000)

EPL_OBD_END_PART ()
```

The application declares a global variable of type BYTE (i.e. unsigned char) and links this variable via the function EplApiDefineObject() to the object dictionary. The following code fragment must be insert right after the initialization of the EPL stack (without the variable declarations).

```
BYTE            bVarIn;
unsigned int    uiVarEntries;
tEplObdSize     ObdSize;

    ObdSize = sizeof(bVarIn);
    uiVarEntries = 1;
    EplRet = EplApiDefineObject(0x6000,
                                &bVarIn,
                                &uiVarEntries,
                                &ObdSize,
                                0x01,
                                0);
    if (EplRet != kEplSuccessful)
    {
        goto Exit;
    }
```

Beside the configuration of the PDO mapping and the reaction on the sync event as shown below, the application does not need to bother how the process variables are exchanged with remote nodes. Even the PDO mapping may be configured by an external configuration tool.

2.4 Event callback function

For simplicity there exits only one callback function for events from the EPL stack (except the sync callback function which is introduced below). The application declares a function with the following prototype.

```
tEplKernel PUBLIC AppCbEvent(
    tEplApiEventType        EventType_p,
    tEplApiEventArg*        pEventArg_p,
    void GENERIC*           pUserArg_p);
```

This function is called for example on the following events: NMT state changes, node state changes, object dictionary accesses, finishing of SDO transfers, EPL stack errors etc.

NMT state changes are indicated by the event type kEplApiEventNmtStateChange. An import NMT state change is the change to kEplNmtGsOff. When this event occurs the NMT state machine was switched off and the EPL stack can be safely shut down (see below).

```
switch (EventType_p)
{
    case kEplApiEventNmtStateChange:
    {
        switch (
           pEventArg_p->m_NmtStateChange.m_NewNmtState)
        {
            case kEplNmtGsOff:
            {   // NMT state machine was shut down,
                // because of user signal (CTRL-C)
                // or critical EPL stack error
                // -> also shut down EplApiProcess()
                //    and main()
                EplRet = kEplShutdown;

                printf("AppCbEvent(kEplNmtGsOff) "
                       "originating event = 0x%X\n",
                       pEventArg_p->m_NmtStateChange.
                                        m_NmtEvent);

                // TODO: inform main process
                //       about this event

                break;
            }

            default:
            {
                break;
            }
        }
        break;
    }

    default:
    {
        break;
    }
}
```

The NMT states of EPL are represented by the enumerated type tEplNmtState. Some states require a special reaction from the application and/or the EPL stack, but others represent only a state where certain action may or may not be executed. Normally the application does not need to perform any action on NMT state changes, except on the kEplNmtGsOff state mentioned above. Another case may be the manipulation of the local object dictionary when NMT state kEplNmtGsResetCommunication is entered. An example for this is the configuration of the PDO mapping (see section 2.7).

```
typedef enum
{
    kEplNmtGsOff                    = 0x0000,
    kEplNmtGsInitialising           = 0x0019,
    kEplNmtGsResetApplication       = 0x0029,
    kEplNmtGsResetCommunication     = 0x0039,
    kEplNmtGsResetConfiguration     = 0x0079,
    kEplNmtCsNotActive              = 0x011C,
    kEplNmtCsPreOperational1        = 0x011D,
    kEplNmtCsStopped                = 0x014D,
    kEplNmtCsPreOperational2        = 0x015D,
    kEplNmtCsReadyToOperate         = 0x016D,
    kEplNmtCsOperational            = 0x01FD,
    kEplNmtCsBasicEthernet          = 0x011E,
    kEplNmtMsNotActive              = 0x021C,
    kEplNmtMsPreOperational1        = 0x021D,
    kEplNmtMsPreOperational2        = 0x025D,
    kEplNmtMsReadyToOperate         = 0x026D,
    kEplNmtMsOperational            = 0x02FD,
    kEplNmtMsBasicEthernet          = 0x021E

} tEplNmtState;
```

Constant	***Description***
kEplNmtGsOff	Generic NMT state NMT_GS_OFF.
kEplNmtGsInitialising	Generic NMT state NMT_GS_INITIALISING.
kEplNmtGsResetApplication	Generic NMT state NMT_GS_RESET_APPLICATION. The manufacturer-specific and device profile OD parts are reset to defaults.
kEplNmtGsResetCommunication	Generic NMT state NMT_GS_RESET_COMMUNICATION. The communication profile OD part is reset to defaults.

Constant	***Description***
	Additionally, the OD is updated from initialization parameters.
kEplNmtGsResetConfiguration	Generic NMT state NMT_GS_RESET_CONFIGURATION. The configuration parameters of the DLL module are updated from OD.
kEplNmtCsNotActive	CN NMT state NMT_CS_NOT_ACTIVE.
kEplNmtCsPreOperational1	CN NMT state NMT_CS_PRE_OPERATIONAL_1.
kEplNmtCsStopped	CN NMT state NMT_CS_STOPPED.
kEplNmtCsPreOperational2	CN NMT state NMT_CS_PRE_OPERATIONAL_2.
kEplNmtCsReadyToOperate	CN NMT state NMT_CS_READY_TO_OPERATE.
kEplNmtCsOperational	CN NMT state NMT_CS_OPERATIONAL.
kEplNmtCsBasicEthernet	CN NMT state NMT_CS_BASIC_ETHERNET.
kEplNmtMsNotActive	MN NMT state NMT_MS_NOT_ACTIVE.
kEplNmtMsPreOperational1	MN NMT state NMT_MS_PRE_OPERATIONAL_1.
kEplNmtMsPreOperational2	MN NMT state NMT_MS_PRE_OPERATIONAL_2.
kEplNmtMsReadyToOperate	MN NMT state NMT_MS_READY_TO_OPERATE.
kEplNmtMsOperational	MN NMT state NMT_MS_OPERATIONAL.
kEplNmtMsBasicEthernet	MN NMT state NMT_MS_BASIC_ETHERNET.

Table 1: Constants for enumerated type tEplNmtState

2.5 Sync callback function

The sync callback function will be called in NMT states PREOPERATIONAL2 or above whenever the sync event occurs. On MN this is when SoC frame is sent and on CN this is when SoC frame is received or the reception of it is anticipated.
This function is the only place where the process variables may be accessed safely, i.e. without interfering with the PDO processing. Normally, the application reads the sensors and sets the actuators in this function synchronously with all other nodes in the network.

The application shall return from this function as fast as possible. The following code fragment demonstrates how to implement this function. If this function returns kEplSuccessful like in this case, the next the next TPDOs are marked valid, i.e. the flag READY will be set. If the process variables are not yet valid, e.g. because sensors are not ready, the application can return kEplReject.

```
tEplKernel PUBLIC AppCbSync(void)
{
tEplKernel          EplRet = kEplSuccessful;

    // read digital inputs
    bVarIn = DIGITAL_INPUT;

    // TODO set digital outputs

    return EplRet;
}
```

2.6 Starting of the EPL stack

After initialization the EPL stack, i.e. the NMT state machine, needs to be started. Upon that the EPL stack does not run, which means it does not react on any EPL frame on the network for example.

```
    // start the EPL stack
    EplRet = EplApiExecNmtCommand(kEplNmtEventSwReset);
```

The above code fragment will execute the NMT command Software Reset, which starts the state machine.

2.7 Local object dictionary access and changing the PDO mapping

It is very easy to read entries from the local object dictionary. For example the following code fragment will read current cycle length.

```
DWORD           dwBuffer;
unsigned int    uiSize;

    // read cycle length (32 bit variable)
    uiSize = 4;
    EplRet = EplApiReadLocalObject(0x1006,
                                   0x00,
                                   &dwBuffer,
                                   &uiSize);
    printf("Cycle length = %u\n", dwBuffer);
```

The function EplApiReadLocalObject() takes four arguments: object index, sub index of the object, pointer to a buffer and pointer to the size of the buffer. The latter one contrains the size which was actually read, when the function has finished.
Often you want to change the PDO mapping. The following code fragment may be placed into the event callback function in NMT state kEplNmtGsResetCommunication. This ensures that the default mapping is overwritten at every reset of the NMT state machine, but the user may overwrite it again via SDO transfers. It will map object 0x6000 sub index 1 to the third byte of the PollResponse frame.

```
QWORD       qwMapping;
BYTE        bValue;

    // disable PDO,
    // i.e. set number of mapped objects to 0
    bValue = 0;
    EplRet = EplApiWriteLocalObject(0x1A00,
                                    0x00,
                                    &bValue,
                                    1);
    // set object mapping
    // (length of 8 bit, offset of 16 bit)
    //              +------------- length in bits
    //              |   +--------- offset in bits
    //              |   |   +----- sub index
    //              |   |   |   +- object index
    //             _|  _|   |  _|
    //            /  \/  \  /\/  \
    qwMapping = 0x0008000F00016000LL;
    EplRet = EplApiWriteLocalObject(0x1A00,
                                    0x01,
                                    &qwMapping,
                                    8);
    // set node ID of communication parameters
    // to 0 (PollResponse)
    bValue = 0;
    EplRet = EplApiWriteLocalObject(0x1800,
                                    0x01,
                                    &bValue,
                                    1);
    // set PDO version of communication parameters to 1
    bValue = 1;
    EplRet = EplApiWriteLocalObject(0x1800,
                                    0x02,
                                    &bValue,
                                    1);
    // enable PDO,
    // i.e. set number of mapped objects to 1
    bValue = 1;
    EplRet = EplApiWriteLocalObject(0x1A00,
                                    0x00,
                                    &bValue,
                                    1);
```

The function EplApiWriteLocalObject() takes four arguments: object index, sub index of the object, pointer to a buffer and size of the buffer.

2.8 SDO transfer

With SDO transfers it is possible to access object dictionaries of remote nodes. The current version of the EPL stack supports two types of SDO transfers: via UDP and via EPL ASnd frames. The functions EplApiReadObject() and EplApiWriteObject() are similar to the functions for local object dictionary access mentioned above. They need four additional parameters: pointer to a handle of the SDO command layer, the ID of the remote node, the SDO transfer type and a user-definable argument pointer. The node ID may be 0 or equal the local node ID. Then the local object dictionary will be accessed. If remote object dictionaries are accessed these functions will return kEplApiTaskDeferred. This indicates that the application will be informed via the event callback function when the task has finished. There is another major difference to the functions for the local object dictionary. The data in the buffer is always treated as in little endian byte order.
The following code demonstrates a call to EplApiReadObject() to read the cycle length from node 32. The last argument which is the user-definable argument pointer is set to the buffer.

```
tEplSdoComConHdl    SdoComConHdl;
unsigned int        uiSize;
BYTE                abSdoBuffer[4];

    uiSize = sizeof (abSdoBuffer);
    EplRet = EplApiReadObject(&SdoComConHdl,
                             32,
                             0x1006,
                             0x00,
                             abSdoBuffer,
                             &uiSize,
                             kEplSdoTypeAsnd,
                             abSdoBuffer);
    if (EplRet == kEplApiTaskDeferred)
    {   // SDO transfer started
        printf("SDO read started");
    }
    else if (EplRet == kEplSuccessful)
    {   // local OD access
        printf("read from local OD\n");
    }
    else
    {   // error occured
        printf("EplApiReadObject() returned 0x%02X\n",
  EplRet);
    }
```

The event callback function must contain the following code fragment to catch the SDO event.

```
    switch (EventType_p)
    {
        case kEplApiEventSdo:
        {
        tEplSdoComFinished* pSdoComFinished =
            &pEventArg_p->m_Sdo;

            if (pSdoComFinished_p->m_SdoAccessType ==
                kEplSdoAccessTypeRead)
            {
                printf("SDO read of object ");
            }
            else
            {
                printf("SDO write of object ");
            }
            printf("0x%04X/0x%02X finished with "
                   "%u bytes transfered\n",
                pSdoComFinished_p->m_uiTargetIndex,
                pSdoComFinished_p->m_uiTargetSubIndex,
                pSdoComFinished_p->m_uiTransferedByte);
            printf("(Abortcode: 0x%04x Handle: 0x%x "
                   "State: ",
                pSdoComFinished_p->m_dwAbortCode,
                pSdoComFinished_p->m_SdoComConHdl);
            switch (
                pSdoComFinished_p->m_SdoComConState)
            {
                case kEplSdoComTransferNotActive:
                    printf("Transfer not active)\n");
                    break;

                case kEplSdoComTransferRunning:
                    printf("Transfer is running)\n");
                    break;

                case kEplSdoComTransferTxAborted:
                    printf("Tx transfer aborted)\n");
                    break;
        case kEplSdoComTransferRxAborted:
            printf("Tx transfer aborted)\n");
            break;

        case kEplSdoComTransferFinished:
            printf("Transfer finished)\n");
            break;

        case kEplSdoComTransferLowerLayerAbort:
            printf("Transfer aborted "
                   "by lower layer)\n");
            break;
    }

    // Assume that we transfer always
    // numeric values,
    // if the size is less or equal 4.
    // But it can be also a VSTRING or OSTRING
    // of that size.
    switch (
        pSdoComFinished_p->m_uiTransferedByte)
    {
        case 0:
            printf("no Bytes transfered\n");
            break;

        case 1:
            printf("BYTE: 0x%02X\n",
                (WORD)AmiGetByteFromLe(
                pSdoComFinished_p->m_pUserArg)
                );
            break;

        case 2:
            printf("WORD: 0x%04X\n",
                AmiGetWordFromLe(
                pSdoComFinished_p->m_pUserArg)
                );
            break;

        case 3:
            printf("3 BYTEs: 0x%06X\n",
                AmiGetDword24FromLe(
                pSdoComFinished_p->m_pUserArg)
                );
            break;

        case 4:
            printf("DWORD: 0x%08X\n",
                    AmiGetDwordFromLe(
                    pSdoComFinished_p->m_pUserArg)
                    );
                break;

            default:
                printf("TODO: dump all bytes\n")
                break;
        }
        break;
    }
}
```

This example prints out all information of the SDO transfer which is available in the event callback function.

2.9 Shutdown of the EPL stack

As mentioned above it is necessary to switch off the NMT state machine before shutting down the EPL stack. This can be done via the NMT command Switch off.

```
  EplRet =
 EplApiExecNmtCommand(kEplNmtEventSwitchOff);

  // TODO: wait until NMT state machine is shut down

  // shut down EPL stack
  EplRet = EplApiShutdown();
```

Glossary

AMI	Abstract memory interface
ASnd	EPL frame type: Asynchronous Send, which may contain SDO or NMT messages
CAL	Communication Abstraction Layer
CN	Controlled Node, i.e. slave device in the EPL network
DCF	Device configuration file (generated by configuration tools)
DLL	Data Link Layer
EPL	Ethernet POWERLINK
EPSG	Ethernet POWERLINK Standardization Group
HMI	Human machine interface
MAC	Media Access Control
MN	Managing Node, i.e. master device in the EPL network
NMT	Network Management
node	an arbitrary EPL device. Often an EPL CN
OBD	Object dictionary module
OD	Object dictionary
PDO	Process Data Object
PReq	EPL frame type: Poll Request
PRes	EPL frame type: Poll Response
RPDO	Receive PDO
SDO	Service Data Object
SoA	EPL frame type: Start of Asynchronous

SoC	EPL frame type: Start of Cyclic
TCP	Transmision Control Protocol
TPDO	Transmit PDO
UDP	User Datagram Protocol

References

Ethernet POWERLINK V2.0 Communication Profile Specification DS 1.0.0

L-1098 openPOWERLINK: Ethernet POWERLINK Protocol Stack Software Manual

Document:	Introduction into openPOWERLINK
Document number:	L-1098e_, Edition April 2008

How would you improve this manual?

Did you find any mistakes in this manual? page

Submitted by:

Customer number:	
Name:	
Company:	
Address:	

Return to: SYS TEC electronic GmbH
August-Bebel-Str. 29
D-07973 Greiz
GERMANY
Fax : +49 (0) 36 61 / 62 79 99