YASDI

Documentation

Edition 1.0

Implementation of the SMA Data Protocol

YASDI-10NE:KS1106

Revision History

Document number YASDI	Version and Alteration Review [1])		Comments	Author
-11:NE1106	1.0	A	Translation from German Version 1.0	Prüssing

[1]) A: Alterations due to faulty documents or improvement of the documentation

B: Alterations maintaining full or upward compatibility

C: Alterations limiting or excluding compatibility

	Name	Date	Signature
Reviewed	Prüssing		

Explanation of Symbols used in this Document

To enable optimal usage of this manual and safe operation of the device during installation, operation and maintenance routines, please note the following description of symbols.

This symbol indicates information that is required for the optimal operation of the product. Read these sections carefully in order to ensure an optimal operation of the product and all its features.

This symbol indicates information that is essential for a trouble-free and safe operation of the product. Please read these sections carefully in order to avoid any damages of the equipment and for optimal personal protection.

This symbol indicates an example.

Content

1 Introduction

This document describes the structure and usage of the "YASDI" software program for communication with SMA devices. The name "YASDI" stands for "**Y**et **A**nother **S**MA **D**ata **I**mplementation".

Functioning as a driver system without its own graphical interface, the software implements communication with SMA devices (e.g. SunnyBoy inverters) using "SMA Data Protocol" via "SunnyNet" and "SMANet".

The software has been designed in such a way that it can be easily adapted to other environments (operating systems). At the time of this document's release, there exist adaptations for Windows (Win32) and Linux. All system-dependent functions are abstracted from the operating system via an interface.

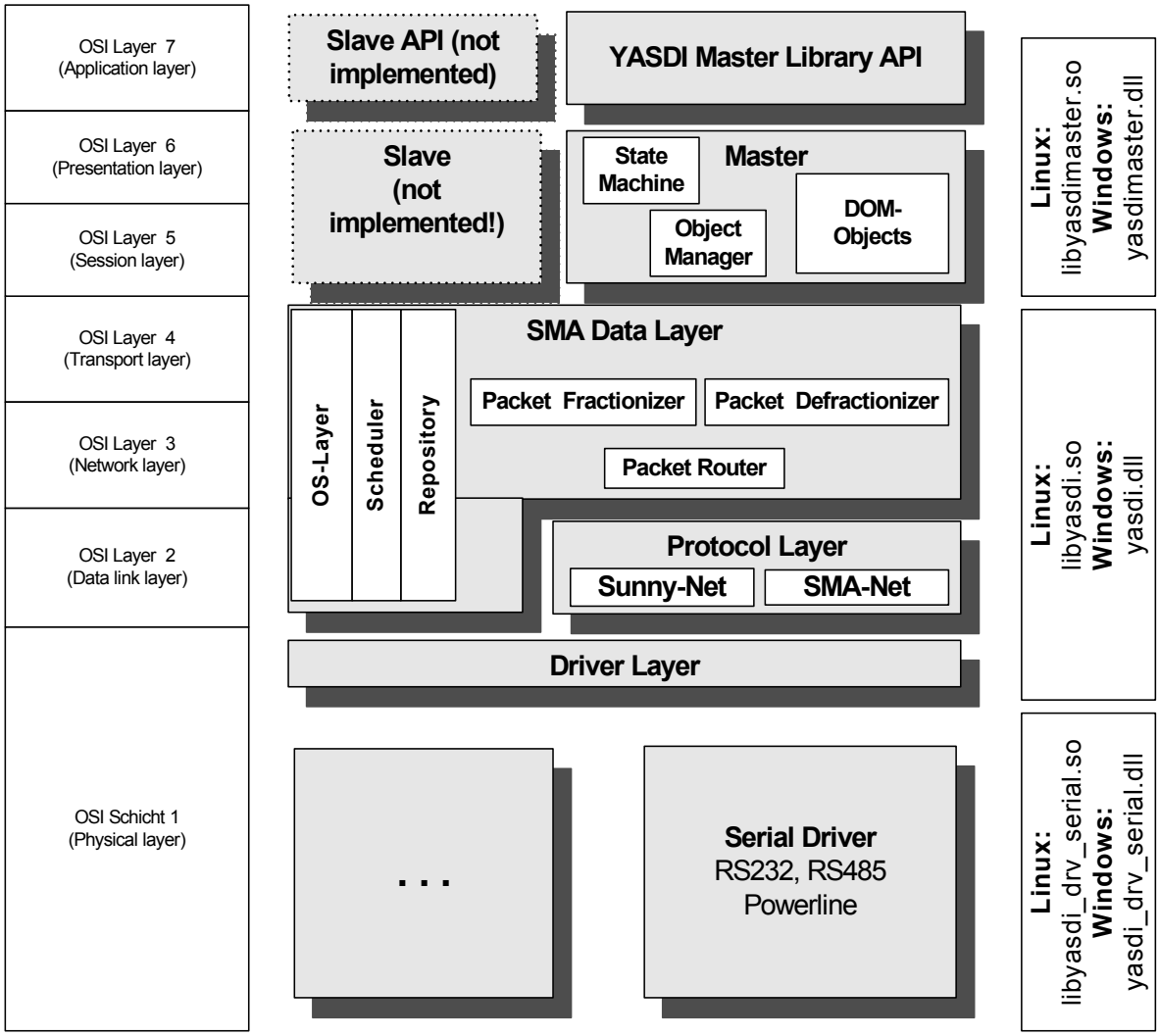
The software is written in "C", and allows maximum possible portability to other possible target platforms. Although an object-oriented language is not used, there is nevertheless an attempt made to realize an object-oriented structure with the "C" language.

The implementations for Windows and Linux are executed as libraries (Windows: DLL, Linux: SO). Another utilization, for example as part of a "monolithic" program, is also possible.

YASDI primarily implements the master functionality of the SMA Data Protocol. Slave functions can also be easily implemented by utilizing the rudimentary functions for sending and receiving packets.

2 Software – Brief Overview

During the implementation of YASDI, the program was based on the OSI layered model for network protocols. The individual layers are grouped in particular libraries, which can be seen in the following overview:



YASDI-10NE:KS1106

2.1 Description of the Layers

2.1.1 Physical Layer (Layer 1)

The software in layer 1 is primarily provided for low-level communication. It is only responsible for sending individual signals over the transmission medium and for the correct handshake on connections. It consists of the individual interface drivers, each of which corresponds to its own bus medium. At the time of this document's release, there is only one serial interface driver. In Windows, this driver can control the serial ports (COM1 to COM8) as RS232, RS485 or Powerline, in Linux, correspondingly, the interfaces "ttyS0" to "ttyS7".

At runtime, several interfaces can be active simultaneously.

2.1.2 Data Link Layer (Layer 2)

Layer 2 is responsible for processing the correct protocol frame. Here, the transport protocols "SunnyNet" and "SMANet" are supported. All framing (including checksum calculation) is available in this layer within the corresponding objects.

2.1.3 Network Layer / Transport Layer (Layer 3/4)

This layer implements the SMA Data Protocol. It has an unified application interface (API) which greatly simplifies control of the different SMA devices. The timeout behavior and any new requests for the devices are implemented. This layer also includes encapsulation of the splitting and reassembly of packet fragments (M-bit mechanism). The built-in packet router makes sure that

transmitted packets are sent to the correct interfaces. By means of dynamic route entries, the router knows the correct interface via which a device can be reached.

The API is an asynchronous packet-oriented interface (IO requests) with callback mechanisms. The only information specified is which data content is to be sent to which device, with the corresponding timeout times and repeats.

This layer also implements an interface which abstracts from operating-system-specific commands. This can be easily adapted for other target platforms.

A simple function scheduler, which allows cooperative multitasking is included as well. The scheduler also implements simple timer functions.

Slave applications should be based on this layer's API.

2.1.4 Session Layer / Presentation Layer (Layer 5/6)

In this layer, a specialized SMA data master is implemented. This master is able to detect SMA devices and to request their channel values. Alongside requesting current measurement values, it can also read and write parameters. At the time of this document's release, it is not possible to request the measurement channel records of a Sunny Boy Control.

The master is implemented as a state machine. It follows the corresponding design pattern.

The master is able to execute data requests on command. However, it can also independently send requests to devices on a cyclic basis.

The devices requested channel lists are temporarily stored on data storage devices and are dynamically loaded and interpreted at runtime.

Within this layer, a system device structure is generated corresponding to the detected SMA devices, namely device objects and channel objects which can be identified externally via a handle mechanism. The handles are generated and managed by the specialized object manager which is also included here.

2.1.5 Application Layer (Layer 7)

Layer 7 describes the actual externally utilizable interface of the SMA data master. The master can be completely controlled via this interface. The API is kept as simple as possible to easily enable requests, also from other software tools which were not written in C/C++. The API utilizes devices and channel handles, via which each of these objects can be accessed.

This interface is externally synchronous! This means, for example, that functions for the acquisition of measurement values block until they have been completely processed.

3 The Library Interfaces

3.1 Data Types Used

In the interface definition, data types are used which are listed below (defined in the file "smadef.h"):

Data type	Data type description
DWORD	32-bit unsigned
WORD	16-bit unsigned
BYTE	8-bit unsigned

3.2 YASDI Master Library API

This is the interface of the so-called master. It can actively send requests to all connected SMA devices. Handles are used to access devices and channels via the API. These are nothing more than simple 32-bit signed values (DWORD) which are used internally by YASDI.

The interface is subdivided into two halves. The first part is concerned with the initialization of the library. The actual communication with the individual devices occurs via the functions provided by the second part of the library.

3.2.1 Initialization Functions

```
void yasdiMasterInitialize( char * cIniFileName, DWORD * pDriverCount)
```

Initialization of the YASDI master library. This function must be executed exactly once, before any other library functions are executed.

Parameters:

The pointer **"cIniFileName"** points to the path and file name of the required initialization file (INI). If no name is given, "yasdi.ini" is searched for in the current directory. If it is not found, the next location examined (in Linux) is in the user directory under ".yasdi/.globalconfig.ini". Relative paths can be given (e.g. "./MyYasdiConfig.ini"). The structure of the INI file is described in chapter 3.5.

The pointer **pDriverCount** points to a variable which, when called, returns the number of interface drivers currently loaded in YASDI.

```
void yasdiMasterShutdown( void );
```

By calling this function, all storage space and resources used by the YASDI master library are once again released for general use. This function should always be executed after using YASDI. No parameters need to be passed.

```
void yasdiReset( void )
```

This function completely resets the software. Any currently detected devices are removed. The software is then in a condition much the same as when the function "yasdiMasterInitialize(...)" has just been executed.

3.2.2 Functions for Sending Requests to Measurement Channels

The following functions make it possible to send requests to available devices and to read values from the current measurement channels.

```
DWORD GetDeviceHandles(DWORD * Handles, DWORD iHandleCount)
```

This function requests all currently detected SMA devices. Exactly one handle is delivered for each device.

Parameters:

The parameter "**Handles**" points to an array which is to record the device handles. Every handle's data type is DWORD. The maximum number of handles which can be recorded in the array is passed with the "**iHandleCount**" parameter. Before calling this function, the array with iHandleCount elements must first be created!

Result:

The return value of the function is the number of handles which actually get stored in the array.

```
int GetDeviceName( DWORD DevHandle, char * DestBuffer, int len)
```

This function provides the device name pertaining to a device handle. At the time of this document's release, every device name comprises the device type, the string "SN:", and the device's serial number.

Parameters:

"DevHandle" denotes the handle of the device, of which the name is to be determined. "DestBuffer" refers to the destination buffer, in which the device name is to be written. The size of the maximum available space in bytes is passed with the parameter "**len**".

Result:

The return value of the function is the actual amount of space taken up by the string (string length).

```
int GetDeviceSN( DWORD DevHandle, DWORD * SNBuffer );
```

By calling this function, the serial number of a device can be determined.

Parameters:

The parameter "**DevHandle**" again denotes the device handle. "**SNBuffer**" refers to the storage area, in which the serial number is to be saved. The serial number is stored as a DWORD value.

Result:

A return value of "0" means OK. "-1" signifies that the handle was invalid, and that there is no corresponding device.

```
int GetDeviceType(DWORD DevHandle, char * DestBuffer, int len)
```

With this function, the device type can be determined. The device type consists of a string with currently a length of 8 characters (see SMA data specification "SMA DAT-12:ZD").

Parameters:

"**DevHandle**" again denotes the device handle. With "**DestBuffer**", the destination buffer for recording the device type is passed. "**len**" denotes the maximum available space in the storage area.

Result:

"**0**" means OK. "**-1**" means that the device handle is invalid.

```
DWORD GetChannelHandles(DWORD pdDevHandle,
                        DWORD * pdChanHandles,
                        DWORD dMaxHandleCount,
                        WORD wChanType,
                        BYTE bChanIndex);
```

With this function, all of a device's channels can be determined. Each channel again corresponds to a handle. A channel mask for pre-selection can be given.

Parameters:

The parameter "**pdDevHandle**" again denotes the device (handle), the channels of which are to be determined. "**pdChanHandles**" refers to the array which is to record the channel handles. "**dMaxHandleCount**" denotes the maximum number of handles which can be recorded in the array. With the two parameters "**wChanType**" and "**bChanIndex**", the channel masks must be passed in accordance with the SMA data specification.

Example:

WChanType = 0x040f and bChanIndex = 0 denotes all parameter channels

WChanType = 0x090f and bChanIndex = 0 denotes all spot value channels

```
DWORD FindChannelName(DWORD pdDevHandle, char * ChanName);
```

Using this function, a channel can be searched for on the basis of its name. It will return the channel handle of the first suitable channel (according to name).

Parameters:

"**pdDevHandle**" denotes the device handle. "**ChanName**" denotes the channel name to be searched for.

Result:

If "0" is returned, the channel was not found. If the value is greater than "0", the value corresponds to the channel handle.

```
int GetChannelName( DWORD dChanHandle,
                    char * ChanName,
                    DWORD ChanNameMaxBuf);
```

This function provides a device's channel name.

Parameters:

With the parameter "dChanHandle", the channel handle is passed. "ChanName" is the destination buffer area for the channel name. "ChanNameMaxBuf" denotes the size of the destination buffer's maximum available storage space.

Result:

"0" => Everything OK.

"-1" => Error: channel handle is invalid!

```
int  GetChannelValue(DWORD dChannelHandle,
                     DWORD dDeviceHandle,
                     double * dblValue,
                     char * ValText,
                     DWORD dMaxValTextSize,
                     DWORD dMaxChanValAge)
```

This function provides the value of a channel. It is possible to predefine for the function how "old" the channel value is allowed to be. The function works synchronously, i.e. it blocks until either the channel value has been returned, or an error has occurred.

Parameters:

"**dChannelHandle**" denotes the channel handle. Using the parameter "**dDeviceHandle**", the device handle to which the channel belongs is passed. "**dblValue**" refers to the storage area for recording the channel value (double value, 8 byte). If there is also a channel text for a channel, this is written to the storage area, which can be passed with "**valText**". "**dMaxValTextSize**" denotes the maximum size of the storage area for the status text. "**dMaxChanValAge**" denotes how long ago the time of measurement value determination can date back (in seconds). A value of "0" forces a retrieval of the current value. Any channel values are temporarily stored in the software. A value of 10 means that the channel value can be at most 10 seconds old. If it is older, the value is automatically retrieved anew from the device.

Result:

The function returns the following values:

0 ==> Everything OK: channel value is valid...

-1 ==> Error: channel handle was invalid...

-2 ==> Error: YASDI driver is in the "ShutDown" state

-3 ==> Error: timeout during new retrieval of channel value

-4 ==> Error: unknown error; channel value invalid

```
DWORD GetChannelValueTimeStamp( DWORD dChannelHandle )
```

Calling this function returns the time stamp of a channel value.

Parameters:

The parameter "**dChannelHandle**" denotes the channel for which the last time stamp of its channel value is to be determined. The time is passed as UNIX time with the time zone Greenwich Mean Time (GMT+0).

```
int GetChannelUnit( DWORD dChannelHandle,
                    char * cChanUnit,
                    DWORD cChanUnitMaxSize)
```

This function provides a channel's channel unit as a string.

Parameters:

The parameter "**dChannelHandle**" denotes the channel handle for which the channel unit is to be requested. "**cChanUnit**" refers to the destination buffer for recording the channel unit string. The buffer must be prepared before calling this function. The parameter "**cChanUnitMaxSize**" denotes the maximum size of the destination buffer.

```
int GetMasterStateIndex()
```

This function returns the current state of the YASDI master. Each of the master's states is represented by means of a specific constant.

Result:

The following states are defined:

```
#define MASTER_STATE_INIT           1 /* Initial state of the machine  */
#define MASTER_STATE_DETECTION      2 /* Detection of devices          */
#define MASTER_STATE_CONFIGURATION  3 /* Network address configuration */
#define MASTER_STATE_IDENTIFICATION 4 /* Channel lists request         */
#define MASTER_STATE_CONTROLLER     5 /* Master command processing     */
#define MASTER_STATE_CHANREADER     6 /* Read channels (spot/parameter)*/
#define MASTER_STATE_CHANWRITER     7 /* Write channels (parameter)    */
```

```
int SetChannelValue(DWORD dChannelHandle,
                    DWORD dDevHandle,
                    double dblValue)
```

This function sets the numerical value of a channel.

Parameters:

The parameter "dChannelHandle" denotes the handle of the channel to be set. "dDevHandle" is the device to which the channel belongs. With "dblValue", the new channel value is passed.

Result:

The function returns the following values:

0 ==> Everything OK: new channel value was taken from the device...

-1 ==> Error: channel handle was invalid...

-2 ==> Error: YASDI is already deactivated ("ShutDown")

-3 ==> Error: timeout when setting channel; value not set

```
int GetChannelStatTextCnt(DWORD dChannelHandle)
```

If status texts exist for a channel, this function returns the number of channel texts for the channel. The texts can then be easily requested with the function "GetChannelStatText(...)".

Parameters:

"dChannelHandle" denotes the channel handle.

Result:

Number of channel texts for this channel.

```
int GetChannelStatText(DWORD dChannelHandle,
                       int iStatTextIndex,
                       char * TextBuffer,
                       int BufferSize);
```

This function returns a specific status text belonging to the channel. The number of texts should first be requested with the function "GetChannelStatTextCnt(...)".

Parameters:

"dChannelHandle" again denotes the channel handle.

"iStatTextIndex" denotes the index of the channel status text which is to be requested. This begins with "0".

"TextBuffer" points to the storage area, into which the text is to be copied.

"BufferSize" is the maximum number of characters which can be copied.

Result:

"0" Everything OK (result valid).

"-1" Invalid channel handle was passed.

```
int GetChannelMask( DWORD dChannelHandle,
                    WORD * ChanType,
                    int * ChanIndex);
```

Provides the channel mask of a channel, as defined in the SMA data definition.

Parameters:

"**dChannelHandle**" => channel handle

"**ChanType**" => points to the WORD variable for recording the channel type as defined in the SMA data specification

"**ChanIndex**" => points to the int variable for recording the channel index as defined in the SMA data specification

```
int DoMasterCmdEx(char * cmd, DWORD Param1, DWORD Param2, DWORD Param3)
```

Using this function, it is possible to send commands to the YASDI master. At the time of this document's release, only the command for "device detection" exists. To send this command, the string "detection" is passed, and the number of devices to be detected is passed in parameter "**Param1**".

Example:

```
DoMasterCmdEx("detection",3,NULL,NULL);
```

The example attempts to detect exactly 3 devices. The function blocks until all devices have been found, or until an error or timeout occurs.

Parameters:

"**cmd**" points to the command string for processing.

The parameters for the master command are passed with the parameters **Param1**, **Param2**, and **Param3**. At the time of this document's release, only **Param1** is used. **Param2** and **Param3** are not used.

Result:

"0" => Command successful ("All devices detected...")

"-1" => Error during execution ("The requested number of devices could not be detected.")

3.3 The YASDI Library API

Beneath the YASDI master interface, there is another useful interface. Here, functions for low-level access to the SMA data protocol are implemented. The API contains functions for controlling the connected interface drivers (COM ports). In addition, functions for the standardized utilization of SMA data commands with usage of timeout times and repeats are provided.

Another section provides functions for the simple implementation of, for example, slave applications.

These functions should only be used by SMA data slave implementations, with the exception of a few functions for the activation of the individual interfaces (COM ports). However, implementations for device requests should primarily use the higher YASDI master API.

3.3.1 Interface Driver Control

The following functions control the availability of YASDI interfaces (RS232, RS485, Powerline,...). At runtime, YASDI can use several interfaces simultaneously.

```
DWORD yasdiGetDriver(DWORD * DriverIDArray, int maxDriverIDs);
```

This function provides all interfaces (drivers) currently available in YASDI. Each driver is represented by an ID (a DWORD value). Using this ID, the drivers can each be activated or deactivated later (see yasdiSetDriverOnline). An interface can be, for example, a serial port "COM1" in Windows. In the initialization file, it is stipulated which interfaces can be used.

Parameters:

With the parameter "**DriverIDArray**", a pointer to the storage area (array of DWORDs) is passed, in which the interface ID's are to be stored. The parameter "**maxIDs**" denotes the maximum number of interface ID's which can be saved in the array.

Result:

The function returns the number of currently available interfaces (number of handles in the array).

```
DWORD yasdiGetDriverName(DWORD DriverID,
                         char * DestBuffer,
                         DWORD MaxBufferSize);
```

With this function, the name of an individual interface can be requested. It provides, for example, the text "COM1" for the first serial port.

Parameters:

The parameter "**DriverID**" passes the ID of the interface, the name of which is to be requested. "**DestBuffer**" refers to the storage area for recording the interface name. The maximum size of the buffer (number of characters) is indicated by the parameter "**MaxBufferSize**".

Result:

The function returns the number of characters used in the destination buffer.

```
BOOL yasdiSetDriverOnline(DWORD DriverID);
```

With this function, a YASDI interface can be brought online. This means that this YASDI interface can be used immediately.

Parameters:

The parameter "**DriverID**" denotes the interface which is to be connected.

Result:

If successful, the function returns "**TRUE**" (<>0). This means that the interface was successfully connected. In the event of an error, it returns "**FALSE**" (0). In this case the interface is probably already being used by another program.

```
void yasdiSetDriverOffline(DWORD DriverID)
```

Calling this function deactivates an interface. The interface can then no longer be used by YASDI.

Parameters:

The parameter "**DriverID**" denotes the interface which is to be deactivated.

3.3.2 Monitored Transfer of SMA Data Packets

These functions enable SMA data packets to be sent and received, with attention paid to timeout times and repeats. The YASDI master, for example, uses only these functions to carry out requests.

The functions use a structure for managing monitored requests:

```
struct TIORequest
{
     TNode Node;              /* Private! Do not use!                    */
     TTimer Timer;            /* Private! Do not use!                    */

     TReqStatus Status;       /* Current status of the request           */
     TReqType   Type;         /* Type of request                         */

     BYTE Cmd;                /* SMA data command                        */
     UWORD DestAddr;          /* Destination address (device netwk. addr.)*/
     UWORD SourceAddr;        /* Source address (own address)            */
```

```
    VOID * TxData;          /* Pointer to the data area to be sent      */
    DWORD TxLength;         /* Size of the send buffer in bytes         */
    DWORD TxFlags;          /* Flags for sending (TX_BROADCAST, ...)    */

    DWORD TimeOut;          /* Timeout for receive packet(s)            */
    DWORD Repeats;          /* Send repeats if a timeout occurs         */


    /* --- Callback functions --- */

    void (*OnReceived)(
         struct _TIORequest * req, /* Pointer to corresponding IO request*/
         WORD SourceAddr,          /* Source of the answer/request      */
         BYTE * Buffer,            /* Pointer to the received answer    */
         DWORD BufferSize,         /* Size of the received answer       */
         DWORD RxFlags);           /* RxFlags:
                                      TS_BROADCAST,
                                      TS_STRING_FILTER,
                                      TS_ANSWER */

    void (*OnEnd     )(            /* Request has ended                 */
         struct _TIORequest * req );

    void (*OnTransfer)(            /* Data transfer is running          */
         struct _TIORequest * req,
         BYTE prozent );
}
```

Due to this structure, it is possible to send a packet, and to wait for the corresponding answer asynchronously, under time-controlled conditions. It is also possible to only send, without waiting for an answer, or to only wait for a specific packet:

```
Type = RT_MONORCV  = 0: /* Wait for ONE answer                   */
Type = RT_MULTIRCV = 1: /* Wait for many answers                 */
Type = RT_NORCV    = 2: /* Do not wait for answer (send only)    */
```

```
void yasdiAddIORequest( TIORequest * req )
```

This function adds a new IO request to the system. The function returns immediately. The processing of the request is completely asynchronous. Depending on what happens, the callback functions of the IO request structure may be executed. Once the request structure's callback function "OnEnd()" has been called, the request is automatically ended.

Example:

```
TIORequest Req;

void sendGetOnlineValues( TIORequest * Req, WORD DstAddr )
{
   WORD ChanMask   = 0x090f;      // All online channels
   BYTE ChanIndex  = 0;           // Channel index: 0

   Req->OnReceived = onPacketReceived; // Callback functions
   Req->OnEnd      = onRequestEnd;
   Req->TxFlags    = 0;           // No flags
   Req->DestAddr   = DstAddr;     // Destination address
   Req->SourceAddr = 0;           // Source address: 0
   Req->Cmd        = CMD_GET_DATA;// Requesting current values
   Req->TxLength   = 3;           // Packet contents consist of 3 bytes
   Req->Repeats    = 4;           // Repeat 4 times if a timeout occurs
   Req->TimeOut    = 6;           // Timeout of 6 seconds
   Req->Type       = RT_MONORCV;  // Wait for exactly ONE answer
   Req->TxData[0] = LOBYTE(ChanMask);
   Req->TxData[1] = HIBYTE(ChanMask);
   Req->TxData[2] = ChanIndex;
   yasdiAddIORequest( Req );          // Start request (asynchronously)
}

void onPacketReceived(TIORequest * req, WORD SrcAddr,
                      BYTE * Buffer, DWORD Buffersize,
                      DWORD RxFlags)
{
   printf("Packet received...\n");
   ...
}

void onRequestEnd(TIORequest * req)
{
   printf("Packet request finished...");
   ...
}
```

3.3.3 Simple Slave API

Slave implementations mainly use this interface, as it is easier to use for these requirements. It only has functions for sending and receiving packets. However, when receiving, the packets may have already been assembled from fragments, and may be split into "portions" when sending (full M-bit support).

```
void yasdiSendPacket( WORD Dest,
                      WORD Source,
                      BYTE Cmd,
                      BYTE * Data, WORD Size,
                      DWORD Flags)
```

YASDI-10NE:KS1106

This function sends an SMA data packet. There is no confirmation of receipt! Answers must be independently waited for, and reacted to.

Parameters:

"Dest" = network address of the device to which the packet is to be sent
"Source" = sender's own network address (usually just "0")
"Cmd" = the SMA data command
"Data" = if applicable, the additional data to be transferred for the SMA data command
"Size" = size of the data area for the transfer
"Flags" = packet flags:

```
TS_BROADCAST     = 1 = Broadcast packet, goes to all devices on the bus
TS_ANSWER        = 2 = This packet is an answer to a request
TS_STRING_FILTER = 4 = String filter flag: (see SMA data definition)
```

```
void yasdiAddPaketListener(TPacketRcvListener * listener)
```

Using this function, a function can be added for receiving SMA data packets (listener). The only parameter which must be passed to the system is a pointer to a structure:

```
struct TPacketRcvListener
{
   void (OnPacketReceived*)( TSMAData * smadata,
                             void * RecData,
                             DWORD Size );
}
```

The structure only possesses one pointer to a function which is activated upon receipt of a packet.

Parameters:

"smadata" = pointer to a structure for managing the received SMA data information:

```
typedef struct
{
   WORD SourceAddr;
   WORD DestAddr;
   DWORD Flags;
   BYTE Cmd;
} TSMAData;
```

```
void yasdiRemPaketListener(TPacketRcvListener * listener)
```

This function removes the listener which was added by means of the function "yasdiAddPaketListener(...)" (parameters: see afore-mentioned function).

3.4 API Usage: An Example

A typical YASDI API function call sequence to send requests to SMA data devices could look like this:

```
/* Initialize YASDI     */
yasdiMasterInitialize(...)

/* Return all interface drivers which YASDI recognizes
   and activate as necessary... */
yasdiGetDriver(...)
yasdiSetDriverOnline(...)


/* Search for all connected devices (one, in this case) */
DoMasterCmdEx("detect",1,NULL,NULL);

/* Get all device handles */
GetDeviceHandles(...)

/* Get all channel handles for this device */
GetChannelHandles(...)
```

```
/* Request or set channel values */
While( youWant )
{
     SetChannelValue(...) or  GetChannelValue(...)
}

/* Deactivate all utilized interfaces */
yasdiSetDriverOffline(...)

/* Shut down YASDI */
yasdiMasterShutdown()
```

It has to be noted, that upon beginning API usage, the function "yasdiMasterInitialize()" is called once, and upon ending, "yasdiMasterShutdown()" is also called once. All other functions of the master API can be used as often as desired, in whatever order.

3.5 Initialization File

The YASDI initialization file uses the INI format known to Windows users. This is also used in the Linux version.

The file path of the initialization file is passed to the master function "yasdiMasterInitialize(...)" (internally, this path is automatically forwarded to the "yasdiInitialize(...)" function). The file is made up of various sections:

"**DriverModules**" Section	
Entry	**Description**
"Driver0" ... "Driver9"	The interface driver to be used (e.g. **yasdi_drv_serial.dll** in Windows, or **libyasdi_drv_serial.so** in Linux), which is to be loaded at runtime.

Sections "**COM1**" to "**COM8**"	
Entry	**Description**
Device	The filename corresponding to the serial interface of the respective operating system. (For the first serial interface – Windows: "COM1", Linux: "/dev/ttyS0").
Media	The medium which the serial driver is to use. At the time of this document's release, **Powerline, RS232** and **RS485** are supported.
Baudrate	The speed of the serial interface in bits per second. The following values can be used: **110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600** (inverters use 1200 baud, and Sunny Boy Control mainly 19200 baud).
Protocol	The transport protocol to be used. The options are "**SunnyNet**" and "**SMANet**". Older Sunny Boy Control devices use only SunnyNet. The newer devices can handle both protocols.

"**Misc**" Section	
Entry	**Description**
StatisticOutput	File information (path + filename) for output of statistical values. YASDI's memory usage, as well as the number of packets sent to (and received from) the devices, is recorded in a file in XML format. This entry is optional. It is used purely for error identification purposes.

"**Master**" Section	
Entry	**Description**
NetAddress	Optional parameter for setting the network address of the SMA data master. The range of values is from "0" (default) to "65535".
ReadParamChanTimeout	Timeout time in seconds when reading parameter values from devices. This entry is optional, and has a default value of "6".
ReadParamChanRetry	Number of repeat attempts when reading from parameter channels (after timeout). This entry is optional. It has a default value of "4" (repeats).
WriteParamChanTimeout	Optional parameter for the specification of the timeout time when writing to parameter channels. The default setting

	is "6".
WriteParamChanRetry	Optional parameter for the specification of the number of repeats when setting parameters (default "4").
ReadSpotChanTimeout	Optional parameter for the specification of the timeout time when reading from spot value channels. The default setting is "6".
ReadSpotChanRetry	Number of repeat attempts when reading from spot value channels (after timeout). This entry is optional (default "4").
AutoReadOnlineChannels	This optional entry stipulates whether the spot value channels are to be automatically passed from all devices, even if at that time nobody is requesting the values from the device via the API (default value "1"). Accessing the channel values may then be somewhat faster. If set to "0", a request is only sent to the device if someone actually sends it a request via the YASDI API.
DeviceAddrRangeLow	The lower limit of the permitted network address range (device address allotment) of a detected device. This entry is optional. The range of values is from "0" (default) up to and including "255".

DeviceAddrRangeHigh	The upper limit of the permitted network address range (device address allotment) of a detected device. This entry is optional. The range of values is from "0" up to and including "255" (default).
DeviceAddrBusRangeLow	The lower limit of the permitted network address range (bus address allotment) of a detected device. This entry is optional. The range of values is from "0" up to and including "255" (default).
DeviceAddrBusRangeHigh	The upper limit of the permitted network address range (bus address allotment) of a detected device. This entry is optional. The range of values is from "0" up to and including "255" (default).
DeviceAddrStringRangeLow	The lower limit of the permitted network address range (string address allotment) of a detected device. This entry is optional. The range of values is from "0" up to and including "255" (default).
DeviceAddrStringRangeHigh	The upper limit of the permitted network address range (string address allotment) of a detected device. This entry is optional. The range of values is from "0" up to and including "255" (default).

An initialization file in Windows could look like this:

```
[DriverModules]
Driver0=yasdi_drv_serial.dll

#### Section regarding the first serial interface
[COM1]
Device=COM1
Media=Powerline
Baudrate=1200
Protocol=SMANet

[Misc]

[Master]
ReadTestChannels=1
ReadParamChanTimeout=6
ReadParamChanRetry=4
WriteParamChanTimeout=6
WriteParamChanRetry=4
ReadSpotChanTimeout=6
ReadSpotChanRetry=4
AutoReadOnlineChannels=0
```

4 Internal Structures

4.1 Packet Buffer Management

Using the TNetBuffer class, packet buffers are managed internally, in order to, for example, prevent data from being moved or copied unnecessarily upon protocol implementation. A packet (TNetBuffer) is made up of any number of fragments. Empty packets are permitted. A fragment (TNetBufferFrag) contains any number of characters (the actual packet content). Functions exist within the TNetBuffer class, which make it possible to work transparently with buffer data. The internal buffer management with fragments is invisible from outside.

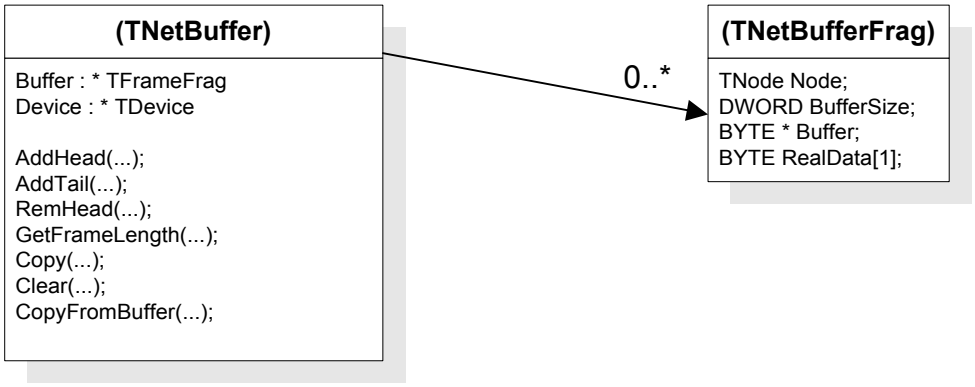


Figure 1: Packet buffer management

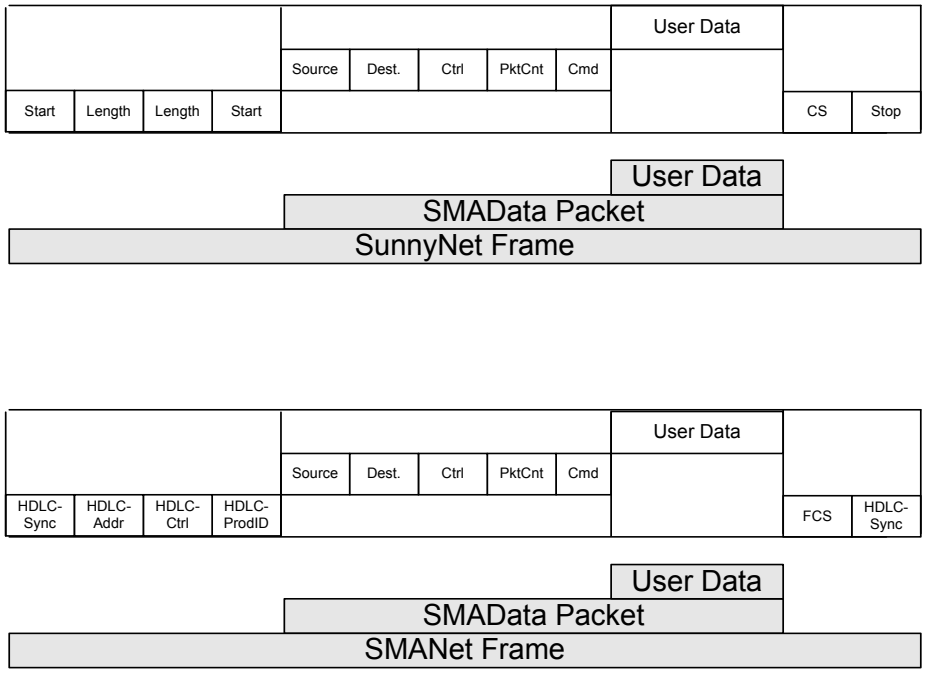


Figure 2: Packet Encapsulation of the SunnyNet and SMANet Telegram Frames

4.2 Installation of the Library Interfaces

4.2.1 Unix (Linux)

Ideally, the files ("shared objects") "libyasdi.so", "libyasdimaster.so" and "libyasdi_drv_serial.so" should be installed in the directory "/usr/local/lib". YASDI saves the temporarily stored channel lists in the subdirectory "devices". This subdirectory should be situated in the same directory as the provided initialization file:

```
"yasdi.ini"       => Initialization file
"devices/"        => Subdirectory for the device channel lists
```

4.2.2 Windows

In Windows, the files (DLL's) can be stored in a local directory. The standard filename of the initialization file is "yasdi.ini"; this file is searched for in the local directory belonging to the caller. A "devices" subdirectory should also be created here, in which the temporarily stored channel lists can be saved.

4.3 Project Directory Structures

YASDI uses a specific directory structure, which is briefly described here:

Subdirectory	Description
core	The core of YASDI.
driver	All of the drivers which YASDI supports are stored here. At the time of this document's release, these are the serial drivers for Linux and Windows.
include	Various include files.
libs	DLL/SO header files of the two library interfaces (YASDI and YASDI master).
master	Implementation of the SMA data master.
os	Operating system abstraction interfaces: all operating systems have their own implementation (at the time of this document's release, Linux and Windows).

protocol	Implementations of the transport protocols "SMANet" and "SunnyNet".
smalib	Module for reading INI files.
projects	All subprojects which are based on YASDI. These, at the time of this document's release, are the following:
projects/linuxlib	Project for the creation of Linux shared libraries (SO) for YASDI and YASDI master.
projects/windowslib	Project for the creation of Windows shared libraries (DLL) for YASDI and YASDI master.
projects/CommonShellUI	Project for the creation of a small shell application, which the YASDI master API uses. This project can run in both Windows and Linux.

5 Creation of a YASDI Application

This chapter describes the creation of the simple shell application included in the YASDI package, as well as the YASDI libraries in the Linux version.

5.1 Creation of the Linux Shared Objects

The YASDI Linux libraries can be created with the following command sequence:

```
cd projects/linuxlib
make
su
xxx    (Password of the "root" user)
make install
exit
```

If this is successful, the libraries (libyasdi.so, libyasdimaster.so, libyasdi_drv_serial.so) are installed at the file path **"/usr/local/lib"**, and can be used subsequently.

5.2 Creation of the Linux Shell Application

The Linux version of the small shell application is created with the following commands:

```
cd projects/projects/CommonShellUI
make
```

Afterwards, the program can then be started with:

```
./YasdiShellUI
```

For compilation of the program, the YASDI libraries must first be installed!