CANopen Commandline Tool

Abstract

`can_open` - Request CANopen services from a CANopen device on the command line.

Description

The *CANopen Commanline Tool* is a text-based program to request CANopen services from a CANopen device on the command line by entering commands at the program´s prompt or processing them from a batch file.

The syntax for these commands is taken from the CANopen specification CiA DS-309/3 (*Interfacing CANopen with TCP/IP – ASCII Mapping*). The program is build on the UVS CANopen Master library, which exists for several microcontrollers and even for some CAN interface boards from different vendors and for different operating systems. Here the CANopen Commandline Tool uses the SocketCAN interface on Linux operating system; it runs well with a PEAK PCAN-USB-Dongle. Furthermore the program offers a gateway function for tunneling CANopen over TCP/IP.

The program allows reading and writing of individual parameter values of any connected CANopen device. Due to the fact, that the UVS CANopen Master library is not a complete CANopen stack, some limitations must be taken into account. But the missing CANopen services can be build up manually from the CAN layer 2 commands of the program.

Usage

The *CANopen Commandline Tool* can operate in three different modes. In the *local mode* the program directly accesses a CANopen network. Commands are entered at the program´s prompt or are processed from a batch file. In *gateway mode* the program also accesses a CANopen network, but additional opens a TCP/IP port be accessed by any CANopen-over-TCP/IP client. Finally the program can operate in *remote mode*. This means commands are entered at the program´s prompt or are processed from a batch file and will be transmitted over ethernet to a CANopent-over-TCP/IP gateway which has access to a CANopen network.

For each mode of operation the program will be started with different arguments and options. As with every console program redirection of the standard input (for batch processing) and of the standard output (for logging) is possible:

```
Usage: can_open <interface> [<option>...]
Options:
 -g, --gateway=<port>         operate in gateway mode on port <port>
 -i, --id=<node-id>           node-id of CANopen Master (default=-1)
     --net=<network>          set default network number (default=1)
     --node=<node-id>         set default node-id (default=1)
     --echo                   echo input stream to output stream
     --prompt                 prefix input stream with a prompt
     --syntax                 show input syntax and exit
 -h, --help                   display this help and exit
     --version                show version information and exit
Examples:
 1. Local mode:     can_open <socket-can> --prompt
 2.1 Gateway mode:  can_open <socket-can> --gateway <port> --echo
 2.2 Remote mode:   can_open <ip-addr>:<port> --prompt
In local mode and in remote mode press ^D to leave the interactive input.
In gateway mode press ^C to close the port and exiting the program.
```

Syntax

The syntax for the CANopen commands is taken from the CANopen specification CiA DS-309/3 (*Interfacing CANopen with TCP/IP – ASCII Mapping*). In general a command consists of a 32-bit sequence number enclosed in square brackets, followed by an optional 8-bit network number, an optional 7-bit node number and the command mnemonic and optional arguments:

Input syntax (client)

<*command-request*>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] <*command*>
<*command*>	::= <*command-specifier*> {<*parameter*>}*
<*sequence*>	*::= UNSIGNED32*
<*net*>	*::= UNSIGNED8*
<*node*>	*::= UNSIGNED8*

All commands are confirmed. A confirmation consists of the repetition of the sequence number of the command enclosed in square brackets, followed by the command response or followed by an error code prefixed with the string “`Error: `”:

Output syntax (server)

<*command-response*>	::= '**[**'<*sequence*>'**]**' <*value*> |
	'**[**'<*sequence*>'**]**' "**OK**" |
	'**[**'<*sequence*>'**]**' "**Error:**" <*sdo-abort-code*> |
	'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

SDO access commands

The *SDO access commands* are used for accessing a single object entry of a connected CANopen device. This means to read (*Upload SDO command*) or to write (*Download SDO command*) an object value by means of a SDO transfer. Depending on the data type of the object entry, the segmented or the expedited SDO protocol will be used. For a list of supported data types and their encoding see below.

Upload SDO command

With the *Upload SDO command* an object value will be read from the specified node at the given object index and sub-index. The data type argument must match the data type of the addressed object entry.

Input syntax (client)

<*upload-sdo-request*>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] ("**read**"|'**r**') <*index*> <*sub-index*> <*datatype*>

Output syntax (server)

<*upload-sdo-response*>	::= '**[**'<*sequence*>'**]**' <*value*> | '**[**'<*sequence*>'**]**' "**Error:**" <*sdo-abort-code*> | '**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the object value will be written to the standard output. If an error has been encountered, then an error code prefixed with the string “`Error: `” will be written to the standard output. This error code is either a SDO abort code defined in the CANopen specification, or an internal error number. For a list of error codes see below.

Download SDO command

With the *Download SDO command* an object value will be written to the specified node at the given object index and sub-index. The data type argument must match the data type of the addressed object entry. The encoding of values is described below.

Input syntax (client)

<*download-sdo-request*>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] ("**write**"|'**w**') <*index*> <*sub-index*> <*datatype*> <*value*>

Output syntax (server)

<*download-sdo-response*>	::= '**[**'<*sequence*>'**]**' "**OK**" |

'**[**'<*sequence*>'**]**' "**Error:**" <*sdo-abort-code*> |
'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. This error code is either a SDO abort code defined in the CANopen specification, or an internal error number. For a list of error codes see below.

Configure SDO timeout command

With the *Configure SDO timeout command* the time-out value (in milliseconds) for both the SDO upload service and the SDO download service will be set.

Input syntax (client)

<*set-sdo-timeout-request*> ::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**set**" "**sdo_timeout**" <*milliseconds*>

Output syntax (server)

<*set-sdo-timeout-response*> ::= '**[**'<*sequence*>'**]**' "**OK**" |
'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

PDO access commands

The *PDO access commands* are used for PDO configuration and communication for both receive-PDOs and transmit-PDOs.

Note: The current version of the UVS CANopen Master library does not support PDO configuration and communication. Use the vendor-specific CAN layer 2 commands for this purpose; see below.

Configure RPDO command

With the *Configure RPDO command* a receive-PDO will be created.

Input syntax (client)

<*set-rpdo-request*> ::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**set**" "**rpdo**" ...

Output syntax (server)

<set-rpdo-response> ::= '**[**'*<sequence>*'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Configure TPDO command

With the *Configure TPDO command* a transmit-PDO will be created.

Input syntax (client)

<set-tpdo-request> ::= '**[**'*<sequence>*'**]**' [[*<net>*] *<node>*] "**set**" "**tpdo**" ...

Output syntax (server)

<set-tpdo-response> ::= '**[**'*<sequence>*'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Read PDO data command

With the *Read PDO data command* the data received by a RPDO will be read. If the RPDO is configured with transmission type 252 or 253, then it will be trigger by means of a RTR.

Input syntax (client)

<read-pdo-request> ::= '**[**'*<sequence>*'**]**' [*<net>*] ("**read**"|'**r**') ("**pdo**"|'**p**') ...

Output syntax (server)

<read-pdo-response> ::= '**[**'*<sequence>*'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

CANopen NMT commands

With the *CANopen NMT commands* a single CANopen node or a whole CANopen network will be controlled and managed. Associated NMT error control services like heartbeat or node guarding can be started and stopped.

Note: The current version of the UVS CANopen Master library does not implement an active NMT master with heartbeat or node guarding capabilities. So the NMT error control is not supported right now.

Start node command

With the *Start node command* a single node or a whole network will be set to NMT state OPERATIONAL; the corresponding NMT `Start_remote_node` command is trigger as a broadcast message.

Input syntax (client)

<*start-node-request*>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**start**"

Output syntax (server)

<*start-node-response*>	::= '**[**'<*sequence*>'**]**' "**OK**" |
	'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Stop node command

With the *Stop node command* a single node or a whole network will be set to NMT state STOPPED; the corresponding NMT `Stop_remote_node` command is trigger as a broadcast message.

Input syntax (client)

<*stop-node-request*>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**stop**"

Output syntax (server)

<*stop-node-response*>	::= '**[**'<*sequence*>'**]**' "**OK**" |
	'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Set node to pre-operational command

With the *Set node to pre-operational command* a single node or a whole network will be set to NMT state PRE-OPERATIONAL; the corresponding NMT `Enter_pre-operational_state` command is trigger as a broadcast message.

Input syntax (client)

<set-pre-operational-request> ::= '**[**'*<sequence>*'**]**' [[*<net>*] *<node>*]
("**preoperational**"|"**preop**")

Output syntax (server)

<set-pre-operational-response> ::= '**[**'*<sequence>*'**]**' "**OK**" |
'**[**'*<sequence>*'**]**' "**Error:**" *<error-code>*

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Reset node command

With the *Reset node command* a single node or a whole network will be set to NMT state RESET-APPLICATION; the corresponding NMT `Reset_node` command is trigger as a broadcast message.

Input syntax (client)

<reset-node-request> ::= '**[**'*<sequence>*'**]**' [[*<net>*] *<node>*] "**reset**" "**node**"

Output syntax (server)

<reset-node-response> ::= '**[**'*<sequence>*'**]**' "**OK**" |
'**[**'*<sequence>*'**]**' "**Error:**" *<error-code>*

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Reset communication command

With the *Reset communication* command a single node or a whole network will be set to NMT state RESET-COMMUNICATION; the corresponding NMT `Reset_communication` command is trigger as a broadcast message.

Input syntax (client)

<reset-communication-request> ::= '**[**'*<sequence>*'**]**' [[*<net>*] *<node>*] "**reset**"
("**communication**"|"**comm**")

Output syntax (server)

<reset-communication-response> ::= '**[**'*<sequence>*'**]**' "**OK**" |

'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Enable node guarding command

With the *Enable node guarding command* the NMT node guarding service with the parameters guard time and life time factor will be started for a specific node (NMT slave) on the NMT master.

Input syntax (client)

<*enable-guarding-request*> ::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**enable**" "**guarding**" <*guarding-time*> <*lifetime-factor*>

Output syntax (server)

<*enable-guarding-response*> ::= '**[**'<*sequence*>'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Disable node guarding command

With the *Disable node guarding command* the NMT node guarding service will be stopped for a specific node.

Input syntax (client)

<*disable-guarding-request*> ::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**disable**" "**guarding**"

Output syntax (server)

<*disable-guarding-response*> ::= '**[**'<*sequence*>'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Start heartbeat consumer command

With the *Start heartbeat consumer command* the consumption of heartbeat messages transmitted by a specific node (heartbeat producer) will be started on the NMT master (heartbeat consumer).

Input syntax (client)

<enable-heartbeat-request>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**enable**" "**heartbeat**" <*heartbeat-time*>

Output syntax (server)

<enable-heartbeat-response>	::= '**[**'<*sequence*>'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Stop heartbeat consumer command

With the *Stop heartbeat consumer command* the consumption of heartbeat messages transmitted by a specific node (heartbeat producer) will be stopped on the NMT master (heartbeat consumer).

Input syntax (client)

<disable-heartbeat-request>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] "**disable**" "**heartbeat**"

Output syntax (server)

<disable-heartbeat-response>	::= '**[**'<*sequence*>'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Device failure management commands

The *Device failure management commands* are used to manage device failures like emergency messages (EMCY message).

<u>Note</u>: The current version of the UVS CANopen Master library does not implement an active EMCY consumer. Use the vendor-specific CAN layer 2 commands for this purpose; see below.

Read device error command

With the Read device error command the emergency message information received from a specific node will be read.

Input syntax (client)

<read-error-request>	::= '**[**'<*sequence*>'**]**' [[<*net*>] <*node*>] ("**read**"|'**r**') "**error**"

Output syntax (server)

<read-error-response> ::= '**[**'*<sequence>*'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

CANopen interface configuration commands

The *CANopen interface configuration commands* are used to configure the CANopen interface and the CAN controller respectively.

Initialize gateway command

With the *Initialize gateway command* the CAN interface will be initialized, which means a reset of the CAN controller will be performed and eventually new bit-timing parameters will be set. This command is useful after bus-off conditions.

Note: On a SocketCAN interface the CAN controller can not be (re-)initialized from within a user application. So this command is just a dummy function; nor the CAN controller will be reset nor the bit-timing will be changed!

<initialize-request> ::= '**[**'*<sequence>*'**]**' [*<net>*] "**init**" *<baudrate-index>*

Output syntax (server)

<initialize-response> ::= '**[**'*<sequence>*'**]**' "**OK**" |
'**[**'*<sequence>*'**]**' "**Error:**" *<error-code>*

Upon successful execution of the command, the string “`OK`” will be written to the standard output. If an error has been encountered, then an error code prefixed with the string “`Error: `” will be written to the standard output. For a list of error codes see below.

Store configuration command

The *Store configuration command* is used to store some program settings in some kind of non-volatile memory or INI-file.

Input syntax (client)

<store-configuration-request> ::= '**[**'*<sequence>*'**]**' [*<net>*] "**store**"
["**CFG**" | "**PDO**" | "**SDO**" | "**NMT**"]

Output syntax (server)

<store-configuration-response> ::= '**[**'*<sequence>*'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Restore configuration command

The *Restore configuration command* is used to restore some program settings from some kind of non-volatile memory or INI-file.

Input syntax (client)

<restore-configuration-request> ::= '**[**'*<sequence>*'**]**' [*<net>*] "**restore**"
["**CFG**" | "**PDO**" | "**SDO**" | "**NMT**"]

Output syntax (server)

<restore-configuration-response> ::= '**[**'*<sequence>*'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Set heartbeat producer command

The Set heartbeat producer command is used to configure and start the transmission of the heartbeat message on the NMT master.

Note: The current version of the UVS CANopen Master library does not implement an active NMT master with heartbeat capability. Use the vendor-specific CAN layer 2 commands for this purpose; see below.

Input syntax (client)

<set-heartbeat-request> ::= '**[**'*<sequence>*'**]**' [*<net>*] "**set**" "**heartbeat**"
<heartbeat-time>

Output syntax (server)

<set-heartbeat-response> ::= '**[**'*<sequence>*'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Set node-id command

With *Set node-id command* the node-id of the local node on the CANopen interface will be set.

Note: The current version of the UVS CANopen Master library does not implement a local node with its own object dictionary on the CANopen interface. Use a regular CANopen stack instead;-)

Input syntax (client)

<set-id-request> ::= '**[**'<*sequence*>'**]**' [<*net*>] "**set**" "**id**" <*node-id*>

Output syntax (server)

<set-id-response> ::= '**[**'<*sequence*>'**]**' "**Error: 100**"

This command is currently not supported: error 100; see below.

Gateway management commands

The *Gateway management commands* are used to manage global settings.

Set default network command

With the *Set default network command* the default network number will be set, which will be used for all services; for the optional argument <*net*> in a command string.

*Note: The current version of the UVS CANopen Master library does only supports one network or in other words one CAN line with the network number **1**.*

Input syntax (client)

<set-network-request> ::= '**[**'<*sequence*>'**]**' "**set**" "**network**" <*network*>

Output syntax (server)

<set-network-response> ::= '**[**'<*sequence*>'**]**' "**OK**" |
'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Set default node-id command

With the *Set default node-id command* the default node number will be set, which will be used for all services; for the optional argument <*node*> in a command string.

Input syntax (client)

<set-node-request> ::= '**[**'<*sequence*>'**]**' [<*net*>] "**set**" "**node**" <*node*>

Output syntax (server)

<set-node-response>	::= '**[**'<*sequence*>'**]**' "**OK**" | '**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Get version command

With the *Get version command* information about the CANopen interface will be retrieved.

Input syntax (client)

<get-version-request>	::= '**[**'<*sequence*>'**]**' "**info**" "**version**"

Output syntax (server)

<get-version-response>	::= '**[**'<*sequence*>'**]**' <*vendor-id*> <*product-code*> <*revision-number*> <*serial-number*> <*gateway-class*> <*protocol-version*> <*implementation-class*> | '**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Rhabarbermarmelade!

Vendor-specific commands

The *Vendor-specific commands* extend the access to a CANopen network with CAN layer 2 services. This means to transmit arbitrary CAN messages onto the CAN bus and to read received CAN messages from a queue, which stores all spontaneously trigger CAN messages by any CANopen device in the network.

Send CAN message command

With the *Send CAN message command* an arbitrary CAN message with an 11-bit COB-identifier and up to 8 data bytes will be transmitted onto the CAN bus. This command can be used for the missed CANopen capabilities of the UVS CANopen master library like transmitting PDOs, SYNC and heartbeat messages.

Input syntax (client)

<send-message-request>	::= '**[**'<*sequence*>'**]**' [<*net*>] "**send**" <*cob-id*> <*length*> {<*value*>}*

Output syntax (server)

<send-message-response>	::= '**[**'<*sequence*>'**]**' "**OK**" |

'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Request CAN message command

With the *Request CAN message command* an arbitrary RTR frame with an 11-bit COB-identifier and up to 8 data bytes will be transmitted onto the CAN bus. The node guarding protocol make use of RTR frames.

Note: RTR frames are no longer recommended by the CAN user organization for some specification inaccuracies!

Input syntax (client)

<*remote-message-request*> ::= '**[**'<*sequence*>'**]**' [<*net*>] "**send**" <*cob-id*> "**rtr**" <*length*>

Output syntax (server)

<*remote-message-response*> ::= '**[**'<*sequence*>'**]**' <*length*> {<*value*>}* |
'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the message length and received data bytes the will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Read CAN message queue command

With the *Read CAN message queue command* exactly one received CAN message will be read from the message queue. All received CAN messages which are not part of a initiated peer-to-peer communication run automatically in this queue (first-in-first-out). This means all received PDOs, all received EMCY messages, all received heartbeat messages and even other CAN layer 2 messages will be stored in the message queue and can be read from it.

Input syntax (client)

<*read-message-request*> ::= '**[**'<*sequence*>'**]**' [<net>] "**recv**"

Output syntax (server)

<*read-message-response*> ::= '**[**'<*sequence*>'**]**' <*cob-id*> <*length*> {<*value*>}* |

'**[**'<*sequence*>'**]**' "**OK**" |
'**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, the received COB-identifier, the message length and the data bytes the will be written to the standard output. In case the queue was empty, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Wait command

With *Wait command* you can suspend the program execution for some time. This is useful for batch operation.

Input syntax (client)

<*wait-request*>	::= '**[**'<*sequence*>'**]**' [<*net*>] "**wait**" <*milliseconds*>

Output syntax (server)

<*wait-response*>	::= '**[**'<*sequence*>'**]**' "**OK**" | '**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

After the time to wait has expired, the string "`OK`" will be written to the standard output. If an error has been encountered, then an error code prefixed with the string "`Error: `" will be written to the standard output. For a list of error codes see below.

Get information command

With the vendor-specific *Get information command* you can retrieve detailed information from the CANopen interface like the used CAN hardware and its current firmware version, as well as some information about the UVS CANopen Master software.

Input syntax (client)

<*get-information-request*>	::= '**[**'<*sequence*>'**]**' [<*net*>] "**info**" ("**hardware**" | "**firmware**" | "**software**" | "**copyright**")

Output syntax (server)

<*get-information-response*>	::= '**[**'<*sequence*>'**]**' <*string*> | '**[**'<*sequence*>'**]**' "**Error:**" <*error-code*>

Upon successful execution of the command, a string containing the requested information will be written to the standard output. If an error has been encountered, then an error code prefixed with the string “`Error: `” will be written to the standard output. For a list of error codes see below.

Miscellaneous

Supported data types

All data types listed below are supported by the current version of the UVS CANopen Master library (Revision 18 of February 7, 2009):

"**i8**"	= 8-bit signed integer value
"**i16**"	= 16-bit signed integer value
"**i32**"	= 32-bit signed integer value
"**u8**"	= 8-bit unsigned integer value
"**u16**"	= 16-bit unsigned integer value
"**u32**"	= 32-bit unsigned integer value
"**t**"	= time of day: days milliseconds
"**td**"	= time difference: days milliseconds
"**vs**"	= visible string
"**os**"	= octet string
"**d**"	= domain

The specific data type of an individual object entry of a node must be taken from the device manual or form the EDS-file of the device.

Value encoding

Numerical values have to be encoded according to ISO/IEC 9899 (ANSI C). Visible strings with whitespaces have to be enclosed with double quotes. A double quote within a visible string have to be escaped by second double quote. Values of type domain or octet string have to be encoded according to RFC 2045 (MIME).

Error codes

The following internal errors are reported:

"**Error: 100**"	= Request not supported
"**Error: 101**"	= Syntax error
"**Error: 102**"	= Request not executed
"**Error: 103**"	= Time-out occurred
"**Error: 999**"	= Other errors

Further information

Further information can be taken from the following document:

CiA DS-301: *CANopen application layer and communication profile*, version 4.02
CiA DS-309: *Interfacing CANopen with TCP/IP,* part 1 and 3, version 1.1

These documents can be downloaded from http://www.can-cia.org/

Release notes

Version 0.2

A bug with MIME type encoding/decoding fixed.

Version 0.1

First version of the program.

Chang log

February 25, 2009

Initial revision of the document.

UV Software
Uwe Vogt
Steinäcker 28
88048 Friedrichshafen

Tel. +49 (0)75 41-60 41 530
Fax +49 (0)75 41-60 41 531

E-Mail uwe.vogt@uv-software.de
Internet http://www.uv-software.de