CANopen Library
DS401 & DS402

CANbus CANopen I/O Support Library for Analog, Digital, Serial, ASI-bridge fieldbus devices

User’s & Porting Manual

Ready to work with QNX / Neutrino® O.S
Ready to work with Windows-2000® O.S.
Ready to work with Linux-RTAI O.S.

Easy to port to any other O.S.
Object & Source Available

© C.J.B. Computer Job srl ~ 2012

C.J.B. Computer Job srl - Via Ghislandi, 24 25125 Brescia – Italy
Tel.: +39 0303531883 ~ Fax.: +39 030349557
WEB: www.cjb.it ~ E-mail: sales@cjb.it
1. Introduction

CJB’s CANopen support Library enables you to easily manage Digital and Analog I/O for CANopen-compliant fieldbus modules, without charging the designer of a complicate management of the CANopen protocol.

More, the CANopen Library allows you to access (both read and write) the service data objects of any unit, and also allows you to manage serial communication channels of such units as from Wago & Beckhoff.

The Library is supplied ready to work for any PC platform hosting the QNX®, or Windows® provided there is a master CANbus interface based upon the SJA1000 or the (now discontinued) 82527 CAN controller. For historical reasons since the library has been developed when i82527 was the preferred CANbus interface, most if not all files regarding the Interface Controller are named “82527”. This does NOT mean that SJA1000 is not supported. Nowadays most of our interfaces and supported modules are built upon the SJA1000 CAN controller.

The Library itself may be quite easily ported onto any other PC platform hosting any other OS, since it is highly modularized and the architecture is already designed to make a porting very easy. The following modules are included:

- **Canopen.cpp**: Protocol management, hardware independent and also operating system independent. This module MUST BE NEVER modified, whichever the porting you have to design.
- **Canosi.cpp**: Interface routines to the OS, for the canopen.c module.
- **Can82527.cpp**: i82527 access routines, used by canopen.c module. This module should be changed and/or substituted if you have another CANbus interface device in your h/w platform.
- **Gprofile.cpp**: Application program interface for DS301 useful to realize a generic profile.
- **Gpdemo.cpp**: Demo program for generic profile application program interface.
- **Candemo.cpp**: Test program to stimulate the most important features of the fieldbus: you may use it a very good sample of usage of the primitive functions our Library gives to you.

2. Basic Structure of Software: the Threads

CAN management is based upon three threads:

- **Periodic Thread**: This is a periodic (periodically timed) thread which updates outputs and asks for inputs (if they are not interrupt-managed upon changes). It also serves the node-guarding procedure, so checking the over-all status of the units.

- **Receive Thread**: It wakes up when the CAN controller receive routine interrupt is activated. It manages the translation & understanding of the received message, then updating the related data areas.

- **Background Thread**: If there are remoted serial communication lines (like those provided by beckhoff and Wago modules) it manages the data transfer to/from these modules.
Software Architecture Schematic

APPLICATION

CANopen Protocol interface
- openCANUnits
- getCANPeriod
- getCANStatistics
- writeCANDigitalOutputs
- CANserialSend
- enableRCE
- closeCAN
- readServiceDataObject
- readCANDigitalInputs
- readCANAnalogInput
- CANserialRecv
- disableRCE
- setCANPeriod
- writeServiceDataObject
- readCANDigitalOutputs
- readCANAnalogOutput
- CANserialStatus
- restartUnit
- getRCERecord

Operating system interface
- createCANPeriodicThread
- createCANReceiveThread
- killCANPeriodicThread
- killCANReceiveThread
- createCANMutex
- postCANMutex
- CANDelay
- installCANInterruptHandler
- getCANTime
- closeCANDevice
- CANSend
- CANReceive
- setCANBaseAddress
- setCANInterruptLevel

CAN-controller interface
4. Application program interface

Herebelow you will find the description of all the primitive function of the CANopen Support Library from CJB.

4.1 int openCANUnits(int, baudRateCode, UNITINFO *ui)

Programmes the units as they are defined in the received array: \textit{ui}, then generates the threads to manage CAN, then puts all the units into RUN status. Before making this call, the application program must write the numbers of the units to be configured into the field: \textit{unit} of each structure of the array: \textit{ui}.

The array is ended when the first item with unit number = 0 has been found.

The fields to be written in the structure: \textit{UNITINFO} before calling the function from the application program are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{unit}</td>
<td>It must include the unit number</td>
</tr>
<tr>
<td>\textit{numSER}</td>
<td>Must include the number of serial comm channels in the module (if WAGO), or 0</td>
</tr>
<tr>
<td>\textit{Diperiod}</td>
<td>Polling period for Digital Inputs, expressed in milli-seconds (Notice: if Digital Inputs are managed in “asynch mode”, i.e. the unit itself communicates the status of its inputs, this field must be set at 0)</td>
</tr>
<tr>
<td>\textit{DofaultMode}</td>
<td>Specifies the status of the Digital Outputs when faulty. If 0 the Digital Outputs get the status defined by DofaultState. If 1 they retain their actual value.</td>
</tr>
<tr>
<td>\textit{DofaultState}</td>
<td>If DofaultMode was set to 0, then if this field is set to 0 the outputs will be switched off, instead if this field is set to 1 the outputs will be switched on.</td>
</tr>
<tr>
<td>\textit{Aiperiod}</td>
<td>Specifies the polling period for Analog Inputs, in milli-seconds. If 0 then the polling period is the default 100 ms.</td>
</tr>
<tr>
<td>\textit{Aitype}</td>
<td>Specifies the type and, consequently, the coding for the Analog Inputs. If 0 the field gets the default type (2 = +/-10Vdc)</td>
</tr>
<tr>
<td>\textit{Aotype}</td>
<td>Specifies the type and, consequently, the coding for the Analog Outputs. If 0 the field gets the default type (2 = +/-10Vdc)</td>
</tr>
<tr>
<td>\textit{AofaultMode}</td>
<td>Defines the status of the Analog Outputs when faulty. If 0 then the Analog Outputs will be forced to 0 Vdc, if 1 the Analog Outputs will retain their actual value.</td>
</tr>
</tbody>
</table>

\textit{All fields (except the unit number field) may be set to 0}. 

After calling the function, for each unit you will have the following informations updated in these field of the \textit{UNITINFO} structure:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{linkDown}</td>
<td>If different from 0 then the unit is NOT correctly connected or it does not give any answer to the system</td>
</tr>
<tr>
<td>\textit{wordsDI}</td>
<td>Number of groups of 16 inputs each, managed by the unit</td>
</tr>
<tr>
<td>\textit{wordsDO}</td>
<td>Number of groups of 16 outputs each, managed by the unit</td>
</tr>
<tr>
<td>\textit{numAI}</td>
<td>Number of Analog Inputs managed by the unit</td>
</tr>
<tr>
<td>\textit{numAO}</td>
<td>Number of Analog Outputs managed by the unit</td>
</tr>
<tr>
<td>\textit{numSER}</td>
<td>Number of Serial comm lines managed by the unit</td>
</tr>
<tr>
<td>\textit{deviceName}</td>
<td>It is a string with the unit name</td>
</tr>
</tbody>
</table>
The argument: \texttt{baudRateCode} which must be passed when calling the function includes a code which defines the operating selected baud-rate for the CAN controller. Codes are described in the defines: \texttt{CAN1MBIT, CAN500KBIT \ldots} in file: \texttt{can82527.h}. In depth:

\begin{itemize}
\item 1 = 1 MB
\item 2 = 500 KB
\item 3 = 250 KB
\item 4 = 125 KB
\end{itemize}

The \texttt{openCANUnits} function returns a value which is different from 0 if there is an error (then error codes may be looked at in the file: \texttt{canopen.h} with the defines: \texttt{CANOPEN_ERR\_\ldots} and so on). After calling the \texttt{openCANUnits} function, even if there is an error, you MUST always call the \texttt{closeCAN} function before exiting the application program. See the following paragraph.

4.2 \textbf{void closeCAN()}

Deletes the threads which manage CAN, switches off the interface device, stops any CAN related activity.

4.3 \textbf{void setCANPeriod(int per)}

Sets the time period of the periodic thread which manages CANopen. The received parameter is the period value in milli-seconds. At startup the default period value is 10 milli-seconds.

4.4 \textbf{int getCANPeriod()}

Returns the present value of the execution period of the periodic thread, in milli-seconds.

4.5 \textbf{int readServiceDataObject(int unit, int idx, int subidx, int *len, void *dst, int maxLen)}

Lets you read the value of one Service Data Object (see CANopen documentation as supplied by CIA; in CAN, an Object is like a message). Both this function and the following one (writeServiceDataObject) will be used in the Application program only in some instances, under your personal responsibility, if you need to modify the standard behaviour and/or implement new management layers for the remote units.

\textbf{Received parameters:}

\begin{tabular}{|l|l|}
\hline
\text{unit} & Unit number \\
\text{idx} & Object index (identifier number) \\
\text{subidx} & Object sub-index \\
\text{len} & Pointer to the variable into which the function will write the length (in bytes) of the received object \\
\text{dst} & Destination buffer, where the function will write the object value \\
\text{maxLen} & Max length of the receive buffer \\
\hline
\end{tabular}

\textbf{Returned parameters:}

Integer value, which is $<> 0$ if error.
4.6 \textbf{int writeServiceDataObject(int unit, int idx, int subidx, int len, long value)}

Lets you write the value of one Service Data Object (see CANopen documentation as supplied by CIA; in CAN, an Object is like a message) with a length up to 4 bytes. Both this function and the preceding one (readServiceDataObject) will be used in the Application program only in some instances, under your personal responsibility, if you need to modify the standard behaviour and/or implement new management layers for the remote units.

\textit{Received parameters:}

\begin{tabular}{|l|l|}
  \hline
  unit & Unit number \\
  idx & Object index (identifier number) \\
  subidx & Object sub-index \\
  len & Length of the object you want to write \\
  value & Value to be written into the object \\
  \hline
\end{tabular}

\textit{Returned parameters:}

Integer value, which is $<>$ 0 if error.

4.7 \textbf{int getCANStatistics(int unit, UNITSTAT *us)}

Returns communication statistics for the unit received into the structure: UNITSTAT. Herebelow we’ll describe the fields of the UNITSTAT structure:

\begin{tabular}{|l|l|}
  \hline
  GlobalError & 0 = unit OK, all OK \\
 & 1 = faulty unit (i.e.: the unit has not answered the node-guarding message for 3 sequential times, or there is an emergency issued by the unit with a proper message, or the unit is off/not powered) \\
  RecCycle & Number of node-guarding messages sent \\
  RecErr & Number of non-answered node-guarding messages \\
  CommError & Flag, communication error \\
  ErrorCode & Error code received with the emergency message \\
  NodeState & Status of the unit (must be: NODE OPERATIONAL) \\
  ManufacturerStatusRegister & Other non-standard info, if there is \\
  \hline
\end{tabular}

\textit{Returned parameters:}

The function returns 0 if all OK, or an error-code. See the file: canopen.h.

4.8 \textbf{int readCANDigitalInputs(int unit, int group, short *pval)}

Returns the value of 16 Digital Inputs. The value is returned in the variable whose pointer was received as calling parameter.

\textit{Received parameters:}

\begin{tabular}{|l|l|}
  \hline
  unit & unit number \\
  group & value (from 0 to 7) which tells the group of Digital Inputs to be returned. Each unit, in fact, can manage up to 128 Digital Inputs, i.e. up to 8 groups of 16 Inputs each (from 0 to 7) \\
  pval & pointer to the short variable where the read value will be found \\
  \hline
\end{tabular}

\textit{Returned parameters:}

The function returns 0 if all OK, or an error-code.
### 4.9 int readCANDigitalOutputs(int unit, int group, short *pval)

Returns the present value of 16 Digital Outputs. The value is returned in the variable whose pointer was received as calling parameter.

**Received parameters:**

<table>
<thead>
<tr>
<th>unit</th>
<th>unit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>value (from 0 to 7) which tells the group of Digital Outputs to be returned. Each unit, in fact, can manage up to 128 Digital Inputs, i.e. up to 8 groups of 16 Outputs each (from 0 to 7)</td>
</tr>
<tr>
<td>pval</td>
<td>pointer to the short variable where the read value will be found</td>
</tr>
</tbody>
</table>

**Returned parameters:**

The function returns 0 if all OK, or an error-code.

### 4.10 int writeCANDigitalOutputs(int unit, int group, short val)

Sets the value for 16 Digital Outputs.

**Received parameters:**

<table>
<thead>
<tr>
<th>unit</th>
<th>unit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>value (from 0 to 7) which tells the group of Digital Outputs to be written. Each unit, in fact, can manage up to 128 Digital Outputs, i.e. up to 8 groups (from 0 to 7)</td>
</tr>
<tr>
<td>val</td>
<td>value to be written into the 16 Digital Outputs.</td>
</tr>
</tbody>
</table>

**Returned parameters:**

The function returns 0 if all OK, or an error-code.

### 4.11 int readCANAnalogInput(int unit, int group, short *pval)

Returns the present value of an Analog Input. The value is in a variable whose pointer was received as calling parameter.

**Received parameters:**

<table>
<thead>
<tr>
<th>unit</th>
<th>unit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>Relative number, inside the unit, of the Analog Input Channel to be read.</td>
</tr>
<tr>
<td>pval</td>
<td>pointer to the short variable where the read value will be found</td>
</tr>
</tbody>
</table>

**Returned parameters:**

The function returns 0 if all OK, or an error-code.

### 4.12 int readCANAnalogOutput(int unit, int group, short *pval)

Returns the present value of an Analog Output. The value is in a variable whose pointer was received as calling parameter.

**Received parameters:**

<table>
<thead>
<tr>
<th>unit</th>
<th>unit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>relative number, inside the unit, of the Analog Output Channel to be read.</td>
</tr>
<tr>
<td>pval</td>
<td>pointer to the short variable where the read value will be found</td>
</tr>
</tbody>
</table>

**Returned parameters:**

The function returns 0 if all OK, or an error-code.
4.13 *int writeCANAnalogOutput(int unit, int group, short val)*

Sets the value for an Analog Output Channel.

**Received parameters:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>unit number</td>
</tr>
<tr>
<td>group</td>
<td>relative number, inside the unit, of the Analog Output Channel to be written.</td>
</tr>
<tr>
<td>val</td>
<td>value to be written</td>
</tr>
</tbody>
</table>

**Returned parameters:**
The function returns 0 if all OK, or an error-code.

4.14 *int CANserialRecv(int unit, int relSer, int *pch)*

Gets a character from an RS-232 or RS-422/485 remoted serial comm line, remoted in the CAN fieldbus, according the (non-standard) BeckHoff or WAGO protocol. If the RX buffer (associated to the serial comm line) is not empty, this function will extract the first character and returns; if the buffer is empty, waits until a character is received.

**Received parameters:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>unit number</td>
</tr>
<tr>
<td>relSer</td>
<td>relative number, inside the unit, of the serial comm line from which the character will be received</td>
</tr>
<tr>
<td>pch</td>
<td>pointer to the variable where the received character will be put.</td>
</tr>
</tbody>
</table>

**Returned parameters:**
The function returns 0 if all OK, or an error-code.

4.15 *int CANserialStatus(int unit, int relSer, int *num)*

Returns the number of characters existing in the RX buffer for the serial comm line. The number is returned in the *num* variable, whose pointer is received as calling parameter.

**Received parameters:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>unit number</td>
</tr>
<tr>
<td>relSer</td>
<td>relative number, inside the unit, of the serial comm line from which the character will be received</td>
</tr>
<tr>
<td>num</td>
<td>pointer to the variable where the number of characters found will be put.</td>
</tr>
</tbody>
</table>

**Returned parameters:**
The function returns 0 if all OK, or an error-code.

4.16 *int CANserialSend(int unit, int relSer, int cc)*

Puts one character into the TX buffer for the serial comm line

**Received parameters:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>unit number</td>
</tr>
<tr>
<td>relSer</td>
<td>relative number, inside the unit, of the serial comm line to which the character will be written</td>
</tr>
<tr>
<td>cc</td>
<td>character to be sent</td>
</tr>
</tbody>
</table>

**Returned parameters:**
The function returns 0 if all OK, or an error-code.
4.17 **int inspectCANConfiguration(int baudRateCode, UNITINFO *pu, int maxUnits)**

Checks the connected units and their configuration.

**Received parameters:**
- **baudRateCode**
  - Code which defines the CAN BaudRate (see: openCANUnits)
- **Pu**
  - Pointer to the array of structures: UNITINFO which will include the data of the connected units. The array will be terminated with an item which is a unit number set to 0.
- **MaxUnits**
  - Array dimension in the application program

**Returned parameters:**
The function returns 0 if all OK, or an error-code.

4.18 **int restartUnit( int unit)**

Stops, reprogrammes, then restarts the unit whose number is received as calling parameter.

**Received parameters:**
- **unit**
  - Unit number

**Returned parameters:**
The function returns 0 if all OK, or an error-code.

4.19 **int enableRCE( int unit)**

Enables the recording of variations for Digital Inputs of the unit whose number is received as calling parameter.

**Received parameters:**
- **unit**
  - Unit number

**Returned parameters:**
The function returns 0 if all OK, or an error-code.

4.20 **int disableRCE( int unit)**

Disables the recording of variations for Digital Inputs of the unit whose number is received as calling parameter.

**Received parameters:**
- **unit**
  - Unit number

**Returned parameters:**
The function returns 0 if all OK, or an error-code.

4.21 **int getRCERecord( RCERECORD *pr)**

Extracts the oldest Digital Inputs variation from the variation buffer, then copies it into the structure: RCERECORD whose pointer is received as calling parameter. The structure: RCERECORD includes the time stamp when the variation was recorded, old Inputs values and new ones, the unit number to which those Inputs belong.

**Received parameters:**
- **pr**
  - Pointer to destination structure

**Returned parameters:**
The function returns 0 if the buffer is empty, or 1 if the structure was successfully extracted.

4.22 **void setCANBaseAddress(long address)**

Sets the base-address of the CAN Controller (if memory-mapped its the real-mode address in memory). The function must be called, in the application program, BEFORE calling the openCANUnits.
**4.23 void setCANInterruptLevel(int intLev)**
Sets the interrupt level for the CAN Controller. The function must be called, in the application program, BEFORE calling the openCANUnits.

**4.24 void setCANDeviceType(int devt)**
Sets the hardware type. At present there are six types of hardware ready made, associated to the defines: CANDEV_… inside the file: can82527.h,. They are:

<table>
<thead>
<tr>
<th>CANDEV_CODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANDEV_CAN527D</td>
<td>ECAN-527 PC-104 CAN Controller Board</td>
</tr>
<tr>
<td>CANDEV_ORCOMPUTER</td>
<td>“OR” PC Industrial PC, with one 82527 mapped inside the integrated I/O.</td>
</tr>
<tr>
<td>CANDEV_SYSTEM_ELECTRONICS</td>
<td>PC104 board with 2 82527 CAN-controller</td>
</tr>
<tr>
<td>CANDEV_AX2000</td>
<td>PCI board with 2 or 4 SJA1000 CAN controller</td>
</tr>
<tr>
<td>CANDEV_CJ370</td>
<td>Integrated 62527 CAN controller in CJ370 CPU board</td>
</tr>
<tr>
<td>CANDEV_IOSJA1000</td>
<td>Generic SJA1000 ISA I/O mapped</td>
</tr>
</tbody>
</table>

The function must be called, in the application program, BEFORE calling the openCANUnits

**4.25 void enableCANScan(int programUnits)**
Re-enables the node-guarding management, output management, serial comm lines management after you have disabled all using the corresponding disableCANScan function. If the received flag was different from 0, the SDO of all the units will be reprogrammed.

**4.26 void disableCANScan(void)**
Disables the node-guarding management, output management, serial comm lines management.

**4.27 void disableCANDOwrite(void)**
Disables the Digital Outputs management, so that Digital Outputs get frozen at their current status.

**4.28 void enableCANDOwrite(void)**
Resumes the Digital Outputs management.

**4.29 void disableCANAOwrite(void)**
Disables the Analog Outputs management, so that Analog Outputs get frozen at their current value.

**4.30 void enableCANAOwrite(void)**
Resumes the Analog Outputs management.
5. Porting to another Operating System

To make a porting to another OS you must redesign the functions used by the CANopen module. All OS-dependant functions are absolutely grouped in the module: canosi.c. Herebelow we’ll describe the functions which you have to modify and customize to your OS.

5.1 int createCANPeriodicThread(void)
Creates a thread which executes the routine: CANHandler (inside the module: canopen) with a period which must match the period value as returned by the function: getCANPeriod. Returns 0 if the creation was made without problems, otherwise it returns 1 if error.

5.2 int createCANReceiveThread(void)
Creates a thread which will wake up when receiving the signal from the interrupt routine which on its turn executes the: CANgetMessage routine (inside the module: canopen). It must be the highest priority thread of the three needed threads. Returns 0 if OK (creation done), 1 if error.

5.3 int createCANBackgroundThread(void)
Creates a thread which on its turn executes (in background) the routine: CANserialHandler (inside the module: canopen). It must be the lowest priority thread of the three needed threads. Returns 0 if OK (creation done), 1 if error.

5.4 void killCANPeriodicThread(void)
Deletes the periodic thread.

5.5 void killCANReceiveThread(void)
Deletes the Receive thread.

5.6 void killCANBackgroundThread(void)
Deletes the Background thread for the serial comm lines.

5.7 int createCANMutex(void)
Creates an object for accessing – in a mutually exclusive way – the CAN controller. Returns 0 if created without errors, 1 if error.

5.8 void waitCANMutex(void)
Makes the wait for accessing mutually exclusive.

5.9 void postCANMutex(void)
Makes the post for accessing mutually exclusive.

5.10 void destroyCANMutex(void)
Deletes object for mutual exclusion.

5.11 void CANDelay(int millisec)
Sets the current task to sleep, for millisec milli-seconds.

5.12 long CANclock(void)
Returns the system time in milli-seconds.
5.13 int installInterruptHandler(int irqnum)
Must work so that the interrupt, whose number (level) it receives, will be associated to the routine: ISR8257(). This latter returns a value different from 0 if you have to wake up the Receiving Thread.

5.14 void deinstallInterruptHandler(void)
Deletes the link to the interrupt routine.

5.15 void getCANTime(long *sec, long *nsec)
Must include, in the two received variables, the system time in seconds and nano-seconds. Will be called when there are variations in the Digital Inputs (if the variation management has been enabled).

6. Porting to another CAN-controller
To make the porting for usage with another CAN controller you must modify the module can82527.c. The I/O code should include the following routines:

6.1 int initCANDevice(int baudrate)
Initializes the controller, receiving the Baud Rate code (see: canopen.h). Returns 0 if ok, 1 if error.

6.2 int closeCANDevice(void)
Shutting off the controller UART.

6.3 int CANSend(char *tx)
Sends the CAN message. Returning a value different from 0 if error.

6.4 int CANReceive(char *rx)
Extracts the CAN message from the RX queue. Returning 1 if a message was extracted, 0 if empty queue.

6.5 int ISR82527(void)
Interrupt routine. Returning a value different from 0 if you have to wake up the RX thread. The Interrupt routine job is to input the received message(s) into the RX queue.

6.6 void setCANBaseAddress(long address)
May be accessed by the application program. Sets the base address of the CAN UART device (if memory mapped, it is the real-mode address of the memory).

6.7 void setCANInterruptLevel(int intLev)
May be accessed by the application program. Sets the interrupt level at which the CAN controller will work.
CANopen DS401 SLAVE-support Library

1. General Features

Like the CANopen Master Library, the CANopen Slave support Library is an original CJB Software product.

With this package you can implement all the functions of the DS401 CANopen profile onto your slave I/O module. Of course, you will have to “cut” the un-necessary functions (if your customized I/O module, for example, has just 16 Digital Outputs, you will not need all other DI, AI, AO functions).

Since the Slave Library is written in C-Language, and we supply the source format, it’s very easy for you to port it for any single-chip for which a good C-Language cross-compiler is available.

More, with this library you can turn any PC-based platform in a customized remoted CANopen I/O node.

2. Introduction

The Slave CANopen Library from CJB supports the DS401 CANopen profile and also simulates remoted serial lines on CANbus like those of WAGO or BECKHOFF. More, the library manages a memory area which can be seen by the master by means of the SDO’s from 0xa640 to 0xa67f, according to DS405 profile. Also allows messaging management using the 0xa8ff SDO.

The h/w resources limits are:

- 128 digital inputs
- 128 digital outputs
- 16 analog inputs
- 16 analog outputs
- 4 serial comm lines

3. Application program interface

3.1

```
short CANOPENSLAVEOpen(short currentBaudRate, long base, short iLev, short devt, short unitnumber, CANOPENSLAVEHOOKS *descr)
```

Open routine. It opens the CANopen slave interface. Initializes the h/w resources and data areas. After executing this routine the CANopen slave interface is active.

Receives:

- `currentBaudRate`: baud rate code, to be used for the CAN interface (1=1Mbits, 2=500Kbits, 3=250Kbits, 4=125Kbits)
- `base`: base address of the CAN-controller chipset (i82527)
- `iLev`: interrupt level used by the CAN-controller chip
- `devt`: code for the used device type (see CANopen master Library)
• **unitnumber**: unit number, corresponding to the slave to be managed

• **descr**: pointer to the structure which includes the pointers to function for the resources management.

Returns 0 if all OK, or an error code.

➔ **CANOPENSLAVEHOOKS structure description**:

```c
typedef struct {
    short (*CSHWOpen)(void) ; //h/w opening routine. Called at startup.
    // Retruns 0 or error code
    short (*CSHWHowManyDoutBytes)(void) ; //must return the number of bytes of
    //digital outputs, i.e. the Digital
    //Output number divided by 8
    short (*CSHWHowManyDinBytes)(void) ; //must return the number of bytes of
    //digital inputs, i.e. the Digital
    //Input number divided by 8
    short (*CSHWHowManyAouts)(void) ; //must return the number of available
    //analog outputs
    short (*CSHWHowManyAins)(void) ; // must return the number of available
    //analog inputs
    short (*CSHWHowManySerials)(void) ; //must return the number of available
    //serial channels
    short (*CSHDinRead)(char *dst) ; //returns, inside the received buffer,
    //the actual value of all Digital Inputs.
    //returns 0 or error code
    short (*CSHAinRead)(short *dst) ; //returns, inside the short int array, of
    //which it receives the pointer, the actual
    //value of all analog inputs.
    //returns 0 or error code
    short (*CSHDoutWrite)(char *src) ; //sets new values for Digital Outputs.
    // Gets the buffer which inlcudes the values
    //returns 0 or error code
    short (*CSHAoutWrite)(short *src) ; // sets new values for Analog Outputs.
    // Gets the buffer which inlcudes the values
    //returns 0 or error code
    void (*CSHWSerialReset)(short sernum) ; //Resets the serial comm line of which
    //it receives the relative number
    // (starting from 0)
    void (*CSHWSerialPutChar)(short sernum, char val) ; //Outputs a character
    //to the serial comm line of which it receives the
    //identifier (relative number starting from 0)
    short (*CSHWSerialStat)(short sernum) ; //returns the number of characters
    // which are in the rx buffer of the serial comm
    //line of which it receives the relative number
    short (*CSHWSerialGetChar)(short sernum) ; // returns the first character
    // which is in the rx buffer of the serial comm
    //line of which it receives the relative number
    char * (*CSHWMemoryAddress)(void) ; //Returns the pointer to the first byte
    //of the shared memory
    void (*CSHWExecuteCommand)(char *Cmd, short CmdLen, char *Ans,
    short *AnsLen) ; //Remote commend management. Receives: commend,
    //command length, answer buffer, and pointer to
    // the length of the answer
    //It’s run when a message is received
} CANOPENSLAVEHOOKS ;
```

You must correctly initialise this structure before you call the CANOPENSLAVEOpen routine.
The function pointers for the services which are not available may set to 0.
3.2  
void CANOPENSLAVEBackground(void)  
This function must be executed in a timing mode or in free-running mode. Includes all the logic for  
the complete management of the CANopen slave.

3.3  
void CANOPENSLAVEEmergencyNotify(short EmergencyCode)  
The user’s functions which manage the h/w will call this function when they find some errors. The  
library will advise the master for the error. When the error is disappeared, you must call the same  
function passing a code = 0.  

   Receives:  
• EmergencyCode : error code (see DS401 specs)

3.4  
short CANOPENSLAVEClose(void)  
Closes all resources. Returns 0 or error code.

4. Operating system interface  
To use the library you must provide some function calls in the OS which is running on the slave  
h/w. Do not be afraid: you will NOT need any complex OS! You simply must provide timing  
management and IRQ management. Usually you always include these functions in your system, even  
if it is based upon a very small single-chip. The functions you need to provide are:  

   long CANclock(void) ;  
   (must return the time in milliseconds)  

   int installCANInterruptHandler(short IRQ_number) ;  
   prepares the system so that the IRQ (whose level is received as parameter) will call the routine  
   which manages the i82527.  

   void deinstallCANInterruptHandler(void)  
   Stops interrupt management.

5. Support for Usage & Implementation & Porting  
CJB will support you 100% to make your porting very easy. Please call our s/w dept.:
1. Introduction

This package is an extension of our package “CANopen Support Library DS401” (I/O A&D). It supports DS402 layer for Motion Control. It cannot be purchased alone, since you must already have the DS401 Master support library to use the DS402 extension.

This library package is supplied in full source format.

Here below we are going to describe the interface functions of the library.

2. Application program interface

2.1. short DriveMotionStartup(short UnitNumber, long CommunicationCyclePeriod, long SynchronousWindowLength, short CubicEnable)

Startup of the Drive. Must be executed after calling the openCAN or openCANUnits (see DS401 library). Within the list of the units received by the openCANUnits the drives must NOT be included. If only the drives are connected on the CAN bus, you will use the openCANUnits just to program the CANcontroller. The function DriveMotionStartup just programs the drive to get the OPERATIONAL status.

Receives:

• **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

• **CommunicationCyclePeriod**: CANopen base period in micro-seconds (it is the period of the SYNC signal)
  periodo base del CANOPEN in microsecondi(periodo con cui verrà inviato il segnale di SYNC)

(following...)

• **SynchronousWindowLength**: time for synchronous management, in micro-seconds, inside the base period seen above (normally it is 1000 microseconds less than the base period)
  tempo per la gestione sincrona in microsecondi all’interno del periodo base di cui sopra (normalmente 1000 microsecondi meno del periodo base).
• **CubicEnable** : flag which enables acceleration/deceleration with cubic management (enabled if different from 0). If 0, acceleration and deceleration are linear (trapezoidal profile).
  flag di abilitazione dell’accelerazione/decelerazione cubica (valore diverso da zero = abilitato). Se la flag vale zero, l’accelerazione e la decelerazione sono lineari (profilo trapezoidale).

  Returns 0 if all ok, or error code
  Restituisce 0 se tutto ok, oppure un codice di errore.

2.2 **short DriveMotionCloseLoop(short UnitNumber )**

This function tell the drive to close the position loop.
*Questa funzione istruisce l’azionamento di chiudere l’anello di posizione.*

Receives:
*Riceve :*

• **UnitNumber** : Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

  Returns 0 if all ok, or error code
  Restituisce 0 se tutto ok, oppure un codice di errore.

2.3 **void DriveMotionOpenLoop(short UnitNumber)**

Tells the drive to open the position loop.
*Questa funzione istruisce l’azionamento di aprire l’anello di posizione.*

Receives:
*Riceve :*

• **UnitNumber** : Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

  Returns 0 if all ok or error code
  Restituisce 0 se tutto ok, oppure un codice di errore.

2.4 **short DriveMotionResetFault(short UnitNumber)**

Clears/resets drive’s error.
*Questa funzione azzerà l’errore sull’azionamento.*

Receives:
*Riceve :*

• **UnitNumber** : Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

  Returns 0 or error code
  Restituisce 0 oppure un codice di errore
2.5 short DriveMotionSetVelocity(short UnitNumber, long TargetVelocity)

Sets the Mode of Operation to “Profile Velocity Mode” and starts the drive with constant speed according to the received value.

Questa funzione imposta il Mode of operation a Profile Velocity Mode e fa partire l’azionamento a velocità costante pari al valore ricevuto.

Receives:
Riceve:

- **UnitNumber** : Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

- **TargetVelocity** : speed to go (encoder counts per second)
  velocità a cui deve muoversi (conteggi encoder/secondo)

Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.6 short DriveMotionStop(short UnitNumber)

Stops actual movement.
Interrompe il movimento corrente.

Receives:
Riceve:

- **UnitNumber** : Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.7 short DriveGetCurrentVelocity(short UnitNumber, long *DstReal, long *DstTheorical)

Returns true speed and theoretic speed at this time.
Restituisce la velocità reale e teorica corrente.

Receives:
Riceve:

- **UnitNumber** : Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

- **DstReal** : address of the variable where the true value will be returned, in encoder pulses per second.
  indirizzo della variabile in cui viene restituito il valore della velocità reale in inpulsi encoder al secondo

- **DstTheorical** : address of the variable where the theoretical value will be returned, in encoder pulses per second.
  indirizzo della variabile in cui viene restituito il valore della velocità teorica in inpulsi encoder al secondo

Returns 0 or error code
Restituisce 0 oppure un codice di errore
2.8 short DriveMotionGetCurrentPosition(short UnitNumber, long *DstPosition)

Returns actual position value
Restituisce la posizione reale corrente.

Receives:/Riceve :

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azioneamento)

- **DstPosition**: address of the variable where the true value of position will be returned, in encoder pulses
  indirizzo della variabile in cui viene restituito il valore della posizione reale in impulsi encoder.

Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.9 short DriveMotionHomingStart(short UnitNumber, long HomingOffset, short HomingMethod, long HomingSwitchSpeed, long HomingZeroSpeed, long Acceleration)

Sets the drive in Homing Mode and starts the drive. Receives the homing method code (see DS402 specs) and the offset to be set as initial position after homing is done. Finished job is sensed through the proper bit in the status register (see DS402).
Mette l’azioneamento in homing mode e fa partire l’azzeramento. Riceve il codice del metodo di azzeramento da utilizzare (vedi DS402) e l’offset da impostare come posizione al termine dell’azzeramento. L’avvenuto azzeramento viene rilevato tramite opportuno bit del registro di stato (vedi DS402)

Receives:/Riceve :

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azioneamento)

- **HomingOffset**: homing position offset
  offset corrispondente alla posizione di azzeramento.

- **HomingMethod**: homing method (see DS402)
  metodo di azzeramento (vedi DS402)

- **HomingSwitchSpeed**: homing fast speed searching input
  velocità di ricerca fine corsa di zero (vedi DS402)

- **HomingZeroSpeed**: homing slow speed waiting index (see DS402)
  velocità nella ricerca della tacca di zero (vedi DS402)

- **Acceleration**: homing acceleration (see DS402)
  accelerazione durante l’operazione di homing (vedi DS402)

Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.10 short DriveMotionGoTo(short UnitNumber, long TargetPosition)

Sets the drive in profile position mode, and starts the movement. Receives the position to be reached, as encoder pulses.
Mette l’azioneamento in profile position mode e fa partire il movimento. Riceve la posizione da raggiungere in impulsi encoder
Receives:

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

- **TargetPosition**: position to be reached, in encoder-pulses
  posizione da raggiungere in impulsi encoder.

Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.11 **short DriveMotionSetProfileMode(short UnitNumber, long Velocity, long Acceleration, long Deceleration)**

Sets the parameters of the movement in Profile Position Mode
Imposta i parametri del movimento in profile position mode.

Receives:
Riceve :

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

- **Velocity**: speed to be reached and kept during the movement in profile position mode
  velocità di regime per il movimento in profile position mode

- **Acceleration**: acceleration in profile position mode
  accelerazione utilizzata nei movimenti in profile position mode

- **Deceleration**: deceleration in profile position mode
  decelerazione utilizzata nei movimenti in profile position mode

Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.12 **short DriveMotionGetStatus(short UnitNumber, short *DstStatus)**

Returns drive’s status register
Restituisce il registro di stato dell’azionamento.

Receives:
Riceve :

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

- **DstStatus**: address of the variable where the value of the status register has to be placed (to understand each bit meaning see DS402 specs)
  indirizzo della variabile in cui viene restituito il valore del regisrato di stato (per il significato dei bit vedi DS402)

Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.13 **short DriveMotionSetInterpolatedMode(short UnitNumber)**

Sets the drive in Interpolated Mode. In this mode the OS where our library is embedded MUST GRANT an absolute hard-real-time environment with max timing precision, since at each system tick time you must give a new position value and, if cubic mode, also the speed of the following target
point to be reached. Position and speed values must be placed in the following fields of the CANUNIT structure for that drive:

DriveMotionTargetPosition

DriveMotionTargetVelocity

Mette l’azionamento in interpolated mode. In questa modalità il sistema operativo su cui opera la libreria deve garantire una precisione temporale, in quanto ad ogni periodo base del sistema operativo è necessario fornire la quota e, nel caso di cubica, anche la velocità del punto successivo. I valori della quota e delle velocità devono essere messi nei seguenti campi della struttura CANUNIT relativa all’azionamento:

DriveMotionTargetPosition

DriveMotionTargetVelocity

Receives:

Riceve:

• **UnitNumber** : Unit number (drive address)
  *numero dell’unità (indirizzo dell’azionamento)*

Returns 0 or error code

Restituisce 0 oppure un codice di errore

### 2.14 `voidDriveMotionControlSet(short UnitNumber, short Val)`

Sets one or more bits inside the control word (see DS402), then sends to the drive

*Set di uno o più bit della parola di controllo (vedi DS402) e suo invio all’azionamento.*

Receives:

Riceve:

• **UnitNumber** : Unit number (drive address)
  *numero dell’unità (indirizzo dell’azionamento)*

• **Val** : word with bits to be set
  *parola contenente i bit da settare*

Returns 0 or error code

Restituisce 0 oppure un codice di errore

### 2.15 `voidDriveMotionControlReset(short UnitNumber, short Val)`

Resets one or more bits inside the control word (see DS402), then sends to the drive

*Reset di uno o più bit della parola di controllo (vedi DS402) e suo invio all’azionamento.*

Receives:

Riceve:

• **UnitNumber** : Unit number (drive address)
  *numero dell’unità (indirizzo dell’azionamento)*

• **Val** : word with bits to be reset
  *parola contenente i bit da resettare*

Returns 0 or error code

Restituisce 0 oppure un codice di errore
2.16 **short DriveMotionSetSoftwarePositionLimits**(short UnitNumber, long Min, long Max)

Set software position limits in the drive.

*Imposta i limiti di posizionamento.*

 Receives:

*Riceve :*

- **UnitNumber** : Unit number (drive address)
  *numero dell’unità (indirizzo dell’azionamento)*

- **Min** : min position to be reached
  *posizione minima possibile da raggiungere*

- **Max** : max position to be reached
  *posizione massima possibile da raggiungere*

 Returns 0 or error code

*Restituisce 0 oppure un codice di errore*

2.17 **long DriveMotionGetCurrentPosition**(short UnitNumber)

Returns the drive current position.

*restituisce la posizione corrente del drive.*

 Receives:

*Riceve :*

- **UnitNumber** : Unit number (drive address)
  *numero dell’unità (indirizzo dell’azionamento)*
2.18 short DriveMotionType(short UnitNumber)

Returns the drive type.

Restituisce il tipo dell’azionamento.

Receives:

Riceve:

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

Returns:

Restituisce:

- DRIVE_TYPE_DEFAULT
- DRIVE_TYPE_CD1K
- DRIVE_TYPE_COPLEY
- DRIVE_TYPE_BAUMER_MSIA
- DRIVE_TYPE_LENZE
- DRIVE_TYPE_MAXON

2.19 Bool DriveMotionInterpolatedModeAvailable(short UnitNumber)

Returns false if the drive doesn’t support interpolated mode, otherwise it returns true.

Restituisce false se l’azionamento non gestisce l’interpolated mode, altrimenti true.

Receives:

Riceve:

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)

2.20 Bool DriveMotionHomingEnded(short UnitNumber)

Returns true if the drive has terminated the homing phase, otherwise it returns false.

Restituisce true se l’azionamento ha terminato la fase di homing, altrimenti false.

Receives:

Riceve:

- **UnitNumber**: Unit number (drive address)
  numero dell’unità (indirizzo dell’azionamento)
### 2.21 short DriveMotionCurrentModeOfOperation(short UnitNumber)

Returns the drive’s current mode of operation.

*restituisce il modo operativo corrente del drive.*

** Receives:

* Riceve:

- **UnitNumber**: Unit number (drive address)
  
  *numero dell’unità (indirizzo dell’azioneamento)*

** Returns :

* Restituisce:

- DM_PROFILE_NOMODE
- DM_PROFILE_POSITION_MODE
- DM_PROFILE_VELOCITY_MODE
- DM_HOMING_MODE
- DM_INTERPOLATED_POSITION_MODE
CANopen Support Library
Application program interface for generic profile

1. Introduction

This module is useful to develop a CANopen profile other than DS401 and DS401. The names of
the functions start always with GP that means generic profile

(Questo modulo contiene le funzioni da utilizzare per realizzare un profile generico CANopen
diverso da DS401 e DS402. È fornito in formato sorgente. Qui di seguito sono descritte le
routine di interfaccia.)

2. Application program interface

2.1 short GPOpenCanLine(int BaudRateCode, int DeviceType, int DeviceAddress, int
InterruptLevel, int MainCanPeriod)

Programs the CAN controller and creates the necessary threads
Programma il CAN controller e crea i threads necessari.

Receives:
Riceve:

- BaudRateCode : Baud rate code

Codes are described in the defines: CAN1MBIT, CAN500KBIT … in file: can82527.h .In depth:

1 = 1 MB  2 = 500 KB  3 = 250 KB  4 = 125 KB
codice della baud rate

- DeviceType : type of the CAN controller to be used

At present there are six types of hardware ready made, associated to the defines: CANDEV_…
inside the file: can82527.h.
tipo del CAN controller

- DeviceAddress : base address of the CAN controller

Indirizzo base del CAN controller

- InterruptLevel : interrupt level for the CAN controller. This argument must be specified if the
CAN controller board is ISA or PC104.
Livello di interrupt del CAN controller

- MainCanPeriod : period of the main canopen thread (msec).

Periodo principale del thread di gestione del CANopen in millisecondi

Returns 0 or error code
Restituisce 0 oppure un codice di errore
2.2 short GPCloseCanLine()
Destroys all CAN handling threads and resets the CAN controller. It must be called before program exit.
Elimina I threads relative al CANopen e resetta il CAN controller
Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.3 short GPReadServiceDataObject(int NodeAddress, int Index, int SubIndex, int *Len, void *Destination, int MaxLen)
Reads a SDO
Legge il valore di un SDO
Receives:
Riceve:
- **NodeAddress**: Address of the CAN node
  Indirizzo dell’unità CAN
- **Index**: number of the SDO
  Numero SDO
- **SubIndex**: sub-index of the SDO
  Sotto-indice del SDO
- **Len**: reference to the variable that will contain the real SDO length
  Riferimento alla variabile che conterrà la lunghezza in bytes del SDO letto
- **Destination**: reference to the variable that will contain the SDO value
  Riferimento alla variabile che conterrà il valore del SDO letto
- **MaxLen**: maximum length allowed for SDO value
  Lunghezza massima del SDO (dimensione della variabile destinataria)
Returns 0 or error code
Restituisce 0 oppure un codice di errore

2.4 short GPWriteServiceDataObject(int NodeAddress, int Index, int SubIndex, int Len, void *Value)
Writes a SDO
Scrive il valore di un SDO
Receives:
Riceve:
- **NodeAddress**: Address of the CAN node
Indirizzo dell’unità CAN

- **Index**: number of the SDO
  
  **Numero SDO**

- **SubIndex**: sub-index of the SDO
  
  **Sotto-indice del SDO**

- **Len**: length in bytes of the SDO to be written
  
  **Lunghezza in bytes del SDO da scrivere**

- **Value**: reference to the variable that contains the value to be written
  
  **Riferimento alla variabile che contiene il valore da scrivere**

Returns 0 or error code

**Restituisce 0 oppure un codice di errore**

### 2.5 short GPStartUnit(int NodeAddress, int NodeGuardingPeriod)

Sends the start message to the specified unit. If NodeGuardingPeriod is not equal to 0 then this function starts the node guarding protocol with the received period in milliseconds

**Invia messaggio di start e, se NodeGuardingPeriod è diverso da zero, inizia la gestione del protocollo di NodeGuarding con il periodo ricevuto**

Receives:

- **NodeAddress**: Address of the CAN node
  
  **Indirizzo dell’unità CAN**

- **NodeGuardingPeriod**: period of node guarding requests in milliseconds
  
  **Periodo per il node guarding**

Returns 0 or error code

**Restituisce 0 oppure un codice di errore**

### 2.6 short GPStopUnit(int NodeAddress)

Sends the stop message to the specified unit and stops node guarding requests.

**Invia messaggio di stop e termina gli invii di node guarding**

Receives:

- **NodeAddress**: Address of the CAN node
  
  **Indirizzo dell’unità CAN**

Returns 0 or error code

**Restituisce 0 oppure un codice di errore**
2.7 short GPStatistics(int NodeAddress, GPSTATISTICS *Dest)

Returns info about node activity.

Restituisce informazioni relative al nodo nella struttura di cui riceve il puntatore

Receives:
Riceve:

- **NodeAddress**: Address of the CAN node
  *Indirizzo dell’unità CAN*

- **Dest**: Address of the info destination structure
  *Indirizzo della struttura destinataria*

This is the format of the structure:

typedef struct {
  long NodeGuardingReceived;
  long NodeGuardingSent;
  long txpdocounter[MAXPDONUM];
  long EmergencyCounter;
  char NodeStatus;
} GPSTATISTICS;

*NodeGuardingReceived* is the counter of the answers to the node guarding requests

*NodeGuardingSent* is the counter of the node guarding remote transmit requests

*txpdocounter* is the counters of the received PDOs

*EmergencyCounter* is the counter of received emergency messages

*NodeStatus* is the node current status (127 = pre-operational, 5 = operational)

Returns 0 or error code

Restituisce 0 oppure un codice di errore

2.8 short GPRemoteTransmitRequest(int NodeAddress, int pdonum, int DataLength)

Sends a remote transmit request for a TXPDO

*Invia una remote transmit request relativa ad un TXPDO*

Receives:
Riceve:

- **NodeAddress**: Address of the CAN node
  *Indirizzo dell’unità CAN*
- **pdonum**: number of PDO to be requested (ranging from 1 to 8)
  
  *Numero del PDO da richiedere*

- **DataLength**: expected data length in bytes of the PDO
  
  *Lunghezza attesa in bytes del PDO che viene richiesto*

  Returns 0 or error code
  *Restituisce 0 oppure un codice di errore*

### 2.9 short GPGetLastPdo(int NodeAddress, int pdonum, int *DstDataLength, char *Dest)

Returns the contents of the last PDO received

*Restituisce i dati dell’ultimo PDO ricevuto*

**Receives:**

- **NodeAddress**: Address of the CAN node
  *Indirizzo dell’unità CAN*

- **pdonum**: number of PDO (ranging from 1 to 8)
  
  *Numero del PDO di cui si desidera l’ultimo valore*

- **DstDataLength**: reference to the variable that will contain the data length in bytes. If the value is zero then no PDO has been received
  
  *Riferimento della variabile che all’uscita della funzione contiene la lunghezza in bytes dei dati. Se questo valore è zero, nessun PDO è stato ricevuto*

- **Dest**: reference to the variable that will contain the PDO data.
  
  *Riferimento della variabile che all’uscita della funzione conterrà i dati del PDO*

  Returns 0 or error code
  *Restituisce 0 oppure un codice di errore*

### 2.10 short GPPdoHook(int NodeAddress, int pdonum, void (*f)(int NodeAddress, int PdoNum, int DataLen, char *PdoData))

Sets an handling function for received PDO. This function will be called immediately when the PDO has been received

*Installa una funzione di gestione della ricezione di un PDO*

**Receives:**

- **NodeAddress**: Address of the CAN node
**Indirizzo dell’unità CAN**

- **pdonum**: number of PDO to be hooked
  
  **Numero del PDO a cui agganciare la function**

- **f**: function to be called. It will receive the node address, the number of the received PDO, the PDO length in bytes and the contents of the PDO
  
  **Funzione che verrà chiamata alla ricezione del PDO.**

  Returns 0 or error code
  
  **Restituisce 0 oppure un codice di errore**

---

### 2.11 short GPPdoSend(int NodeAddress, int pdonum, int DataLength, char *Value)

Sends a PDO

**Invio di un PDO**

**Receives:**

- **NodeAddress**: Address of the CAN node
  
  **Indirizzo dell’unità CAN**

- **pdonum**: number of PDO to be sent (ranging from 1 to 8)
  
  **Numero del PDO che si desidera inviare**

- **DataLength**: PDO length in bytes
  
  **Lunghezza del PDO da inviare in bytes**

- **Value**: PDO contents.
  
  **Contenuto del PDO da inviare**

  Returns 0 or error code
  
  **Restituisce 0 oppure un codice di errore**

---

### 2.12 short GPPdoProgram(int NodeAddress, int Rx, int pdonum, int Index, int FirstSubIndex, int NumObj, int bitSize, int TxType)

Programs the PDO layout and its mode of transmission

**Programma il contenuto di un PDO e il suo modo di trasmissione**

**Receives:**

- **NodeAddress**: Address of the CAN node
  
  **Indirizzo dell’unità CAN**

- **Rx**: Direction flag. 1 = RXPDO, 2 = TXPDO
**Direzione del PDO, 1 = RXPDO, 2 = TXPDO**

- **pdonum**: number of PDO to be programmes (ranging from 1 to 8)
  
  *Numero del PDO che si desidera programmare*

- **Index**: SDO number to be put in the PDO
  
  *Numero del SDO da mettere nel PDO*

- **FirstSubIndex**: First sub index of the SDO to be put in the PDO
  
  *Primo sottoindice da inserire nel PDO*

- **NumObj**: Number of objects to be put in the PDO
  
  *Numero di oggetti da mettere nel PDO*

- **bitSize**: Size in bits of each object in the PDO
  
  *Dimensione in bits degli oggetti da inserire nel PDO*

- **TxType**: Type of transmission (see CANopen doc)
  
  *Tipo di trasmissione del PDO*


Returns 0 or error code

*Restituisce 0 oppure un codice di errore*

### 2.13 short GPSyncEnable(void (*Fun)(void))

Enables Sync transmission and sets the function to be called after each sync transmission

*Abilita la trasmissione del sync ed imposta la funzione da chiamare dopo la sua trasmissione*

Receives:

*Riceve:*

- **Fun**: Function to be called after sync transmission
  
  *Funzione da chiamare dopo la trasmissione del sync*

Returns 0 or error code

*Restituisce 0 oppure un codice di errore*

### 2.14 short GPSyncDisable(void)

Disables Sync transmission

*Disabilita la trasmissione del sync*

Returns 0 or error code

*Restituisce 0 oppure un codice di errore*
CANopen DS401 DS402 release layout and test programs

The CANopen library is supplied in source code format and the application programmer can port it into his environment (operating system and CAN controller). If the application programmer is working with an operating system which has been already supported (QNX4, QNX6, Windows, RTAI Linux), he can use the compiling environment supplied with source code.

The operating system independent source code can be found in the <serverlib> tree.

Within <serverlib> we have the following:

- Serverlib/include: include files
- Serverlib/canopen: canopen library source code
- Serverlib/canopenslave: source code for DS401 slave library
- Serverlib/canopenslave/test: example of DS401 slave hardware dependent modules
- Serverlib/canopen/test: canopen library test program
- Serverlib/findpci: pci server to access PCI boards
- Serverlib/findpci/intflib: pci server interface library to access PCI boards

To compile the modules we have a similar tree; one for each supported operating system:

- Serverlib.qnx: for QNX4 environment
- Serverlib.neu: for QNX6 environment
- Serverlib.win: for windows environment
- Serverlib.lnx: for RTAI Linux environment
Within the serverlib.xxx tree there are the following directories:

- Serverlib.xxx/bin executable file for pci server
- Serverlib.xxx/common some common files mandatory for compiling
- Serverlib.xxx/lib/sys osi.lib library that includes all operating system API
- Serverlib.xxx/lib/appl canopen library object file
- Serverlib.xxx/findpci compiling environment for pci server
- Serverlib.xxx/findpci/intflib compiling environment for pci server interface library
- Serverlib.xxx/canopen compiling environment for canopen library
- Serverlib.xxx/canopen/test/local compiling environment for canopen test program
- Serverlib.xxx/canopenslave compiling environment for canopenslave library
- Serverlib.xxx/canopen/test compiling environment for canopenslave example

If you are using RTAI Linux you have also the driver directory:

- Serverlib.lnx/candrv

CANopen library can use immediately these CAN controllers:

- DUAL CAN ISAbus or PC104 CAN interface
- CJ370 internal CAN controller of the CJ370 SBC
- CRP CJB CAN PCI board

The CANopen library software includes a test program which you can run immediately to verify the connections to the remoted I/O units or motion drive units.

To run this testing program you have to follow these steps:

1) If you are using RTAI Linux you must install RTAI modules and CAN driver. Enter:
   - Cd /serverlib.lnx/candrv <enter>
   - insertmodules <enter>

2) If you want to use the CRP PCI board you must run also the findpci server:
   - Cd /serverlib.xxx/bin <enter>
   - Findpciserver & <enter>
3) You must also edit the **configuration file**. The configuration file is an ASCII file that you can edit with `vi` or other suitable text editor. This is an example of the can configuration file contents:

```
3 2 210 5 10
1 0
48 0
```

the first line contains five numbers:

1. type of CAN controller (3 = DUAL CAN, 4 = CRP PCI board, 5 = CJ370)
2. baud rate code (1 = 1MB, 2 = 500KB, 3 = 250KB, 4 = 125KB)
3. base address of the can controller in hex
4. interrupt level of the can controller (not important for pci)
5. base period of canopen handler thread in milliseconds

Next lines describe the I/O slave units connected to the bus. If you want to connect motion drives you have not to specify any unit. Each line describes a DS401 slave unit with two numbers:

1. unit ID decimal
2. number of serial lines (brand: WAGO) in the unit (normally 0)

4) To start the CANopen testing program you have to enter these commands:

a. `cd /serverlib.xxx/canopen/test/local <enter>`

b. `testloc <config_file_name> <enter>`

The testing program displays the DS401 unit status:

```
UNIT 02: LINK 0 DO 2 DI 1 AO 0 AI 0 SER 0 ENC 0 CNT 0
```

**LINK** (0 = connected 1 = not connected)
**DO** number of digital output groups (16) in the node
**DI** number of digital input groups (16) in the node
**AO** number of analog outputs in the node
**AI** number of analog inputs in the node
**SER** number of WAGO serial lines in the node
**ENC** number of WAGO encoders in the node
**CNT** number of WAGO counters in the node

(following...)

and this menu:

1) Digital inputs         2) Digital outputs
3) Analog inputs         4) Analog outputs
5) Remote terminal       6) Communication statistics
7) Read service data object   8) Write service data object
9) Enable DI change store   10) View DI change
11) Auto-configuration   12) Read encoder
13) Write encoder        14) Read counter
15) Write counter        16) Read slave memory
17) Write slave memory   18) Send message
19) Drive open           20) Drive status
21) Drive reset          22) Drive go to
23) Drive velocity       24) Drive stop
25) Drive homing         26) Drive close loop
27) Drive open loop      28) Drive interpolated mode
0) Exit

Option:

The meaning is the following:

**Digital inputs:** you can read digital input status for each node. You can see digital inputs in groups of 16

**Digital outputs:** you can set and reset digital outputs. Also DOUTS are in groups of 16

**Analog inputs:** display an AIN value

**Analog outputs:** writes an analog output value

**Remote terminal:** tests a (WAGO) CANopen serial line with RX/TX connected

**Communication statistics:** for each node you can see:

```
U:02<CD1K>    Error  errCd  NGcnt  NGErr  emCt  ST  SYNC  TXPDO1
            0      0      7777   12     1     5   777   99999
```

Where:

- **Error**: error status (0/1) of the node
- **ErrCd**: error code
- **NGcnt**: number of node guard messages sent
- **Ngerr**: number of node guard answers not received
- **EmCt**: number of emergency messages received
- **ST**: unit CANopen status
- **SYNC**: number of SYNC msgs sent
- **TXPDO1**: number of TXPDO1 received

(following...)
Read service data object: reads a SDO
Write service data object: writes a SDO
Auto-configuration: checks for units connected
Read encoder – write encoder: you can read and write WAGO encoders
Read counter – write counter: you can read and write WAGO counters
Drive open: opens a drive connection
Drive status: displays the drive status (status word, command word, current position)
Drive reset: resets fault in the drive
Drive go to: a movement in profile position mode
Drive velocity: start a movement in profile velocity mode
Drive stop: interrupts current movement
Drive homing: start a setpoint movement
Drive close loop: close the regulation loop in the drive
Drive open loop: open the regulation loop in the drive
Drive interpolation mode: start a movement in interpolated mode

If you want to exit the CANopen test program you have to do these steps:

1) Press 0 and <enter> to exit the test

2) If you are using CRP PCI board you must also kill the pci server:
   a. Cd /serverlib.xxx/bin  <enter>
   b. Serverkill FindPciServer <enter>

3) If you are using the RTAI Linux operating system you must also remove RTAI modules and CAN driver:
   a. Cd /serverlib.lnx/candrv <enter>
   b. Removemodules <enter>

The CANopen library software includes also a test program useful to point out the application program interface for the generic profile implementation.

To run this generic profile testing program you have to follow these steps:

1) If you are using RTAI Linux you must install RTAI modules and CAN driver. Enter:
   - Cd /serverlib.lnx/candrv  <enter>
   - insertmodules  <enter>

2) If you want to use the CRP PCI board you must run also the findpci server:
   - Cd /serverlib.xxx/bin  <enter>
   - Findpciserver &  <enter>
6. baud rate code (1 = 1MB, 2=500KB, 3=250KB, 4=125KB)
7. base address of the can controller in hex
8. interrupt level of the can controller (not important for pci)
9. base period of canopen handler thread in milliseconds

Next lines describe the I/O slave units connected to the bus. 
If you want to connect motion drives you have not to specify any unit.
Each line describes a DS401 slave unit with two numbers:

3) To start the generic profile testing program you have to enter these commands:
   c. cd /serverlib.xxx/canopen/test/local <enter>
   d. gpdemo <enter>

   The program requires the infos about the CAN controller you want to use.
type of CAN controller (3 = DUAL CAN, 4=CRP PCI board, 5=CJ370)
the baud rate of your fieldbus :
baud rate code (1 = 1MB, 2=500KB, 3=250KB, 4=125KB)

   the base address and the interrupt level of your CAN controller
base address of the can controller in hex
interrupt level of the can controller (not important for pci)

   and, finally, the base period of the main thread.

The generic profile test program has this menu:

1) Read service data object  2) Write service data object
3) Start node                       4) Node statistics
5) Stop node                       6) Remote transmit request
7) Get last Pdo received       8) Set/Reset hook function
9) Send a PDO                   10) Program a PDO
11) Sync transmission enable 12) Sync transmission disable

Each menu entry points out the usage of a function of the generic profile application program interface. The arguments to be passed to the function will be request to the operator.

© CJB 2012