

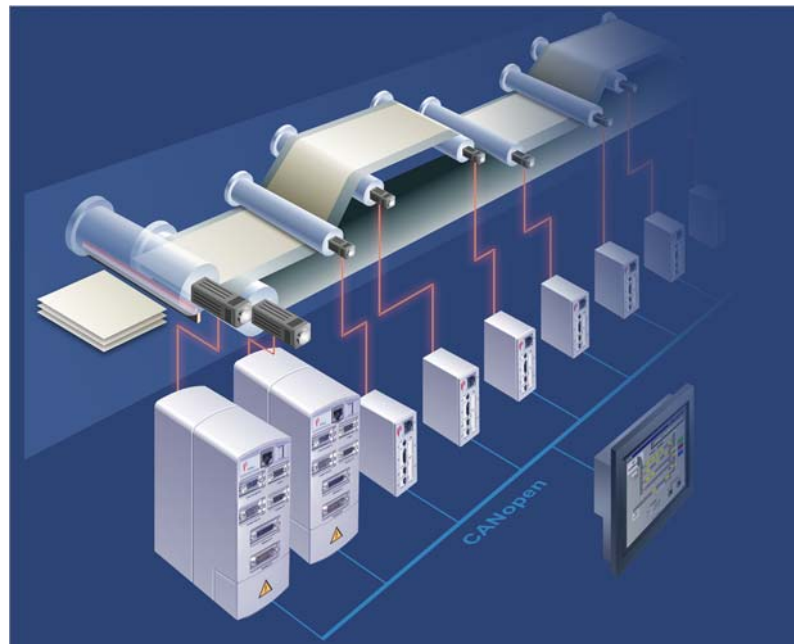
---

# Elmo Interlude

## Communication API

for CANopen and RS-232

### User Guide



Version 1.8: May 2005

---

# ***Important Notice***

This document is delivered subject to the following conditions and restrictions:

- This guide contains proprietary information belonging to Elmo Motion Control Ltd. Such information is supplied solely for the purpose of assisting users of the Elmo Interlude software in writing motion control applications for Elmo digital drives.
- The text and graphics included in this manual are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Information in this document is subject to change without notice. Corporate and individual names and data used in examples herein are fictitious unless otherwise noted.

Doc. no. MAN-INTUG  
Copyright © 2003, 2005  
Elmo Motion Control Ltd.  
All rights reserved.

## **Revision History**

Version 1.8: May 2005 (MAN-INTUG.pdf)  
Functions added: IDriver::DownloadFirmwareEx added

Version 1.7: January 2005 (MAN-INTUG.pdf)  
Functions added: IDriver::UploadProgram and IDriver::DownloadProgram added

Version 1.6: August 2003 (INTUG0803.pdf)  
Interface added: ITabMotion, Appendix added

Version 1.5: May 2003  
Functions added: ICanService::SendCommandGroup ID and ICanService::SendBGSyncOnTime

Version 1.0: February 2003

Microsoft and Visual C++ are registered trademarks of Microsoft Corporation.

---

**Elmo Motion Control Inc.**  
1 Park Drive, Suite 12  
Westford, MA 01886 USA  
Tel: +1 (978) 399-0034  
Fax: +1 (978) 399-0035  
e-mail: info-us@elmomc.com

**Elmo Motion Control GmbH**  
Steinbeisstrasse 41  
D-78056, Villingen-Schwenningen Germany  
Tel: +49 (07720) 8577-60  
Fax: +49 (07720) 8577-70  
e-mail: info-de@elmomc.com

  
Motion Control  
[www.elmomc.com](http://www.elmomc.com)



# Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
<b>2</b>	<b>Installation .....</b>	<b>3</b>
<b>3</b>	<b>IDriver Interface.....</b>	<b>4</b>
<b>3.1</b>	<b>General Communication .....</b>	<b>5</b>
3.1.1	IDriver::DriverConnect.....	5
3.1.2	IDriver::DriverDisconnect .....	8
3.1.3	IDriver::SendCommand .....	9
3.1.4	IDriver::GetComInfo.....	10
3.1.5	DRIVER_CALLBACK Function Type .....	11
3.1.6	SCommunicationInfo Structure .....	12
<b>3.2</b>	<b>Application Handling .....</b>	<b>14</b>
3.2.1	IDriver::IsEqualCommunicationParam.....	14
3.2.2	IDriver::DownloadApplication .....	16
3.2.3	IDriver::UploadApplication.....	18
3.2.4	IDriver::DownloadProgram .....	19
3.2.5	IDriver::UploadProgram.....	20
<b>3.3</b>	<b>Firmware .....</b>	<b>21</b>
3.3.1	IDriver::DownloadFirmware .....	21
3.3.2	IDriver::DownloadFirmwareEx .....	22
<b>3.4</b>	<b>Networking.....</b>	<b>24</b>
3.4.1	IDriver::SaveSComInfoCollection .....	24
3.4.2	IDriver::GetSizeSComInfoCollection .....	25
3.4.3	IDriver::LoadSComInfoFromCollection .....	26
<b>3.5</b>	<b>Error Handling .....</b>	<b>27</b>
3.5.1	IDriver::SetErrorCallback.....	27
<b>3.6</b>	<b>CAN Service Interfaces .....</b>	<b>28</b>
3.6.1	IDriver::GetCanService .....	28
3.6.2	IDriver::GetTabMotion .....	28
<b>4</b>	<b>ICanService Interface (CAN only).....</b>	<b>29</b>
<b>4.1</b>	<b>Fast Motion .....</b>	<b>29</b>
4.1.1	ICanService::InitFastMotionMode (PVT and PT).....	30
4.1.2	ICanService::SetPVTPoint .....	31
4.1.3	ICanService::SetPTPoint .....	32
4.1.4	ICanService::ReadyFastMotion .....	33
4.1.5	ICanService::StopFastMotion .....	34
4.1.6	ICanService:: SendCommandGroupID .....	35
4.1.7	ICanService:: SendBGSyncOnTime .....	36
4.1.8	ICanService::CreateSyncManager.....	37
4.1.9	ICanService::DestroySyncManager .....	38
4.1.10	FAST_MOTION_PARAM Parameters .....	38

4.2	Emergency Handling.....	39
4.2.1	ICanService::SetEmergencyCallback.....	39
4.2.2	EMERGENCY_CALLBACK Parameters .....	40
4.2.3	EMERGENCY_STRUCT Parameters .....	40
5	ITabMotion Interface (CAN only).....	41
5.1	PVT Motion .....	42
5.1.1	ITabMotion::InitPVT .....	42
5.1.2	ITabMotion::AddPVTPoint .....	43
5.1.3	ITabMotion::GetDriveTimer .....	43
5.1.4	ITabMotion:: SendCommandGroupID1.....	43
5.1.5	ITabMotion::SendBGSyncOnTime1 .....	43
5.1.6	ITabMotion::GetQueueSize.....	44
5.1.7	ITabMotion::StopSYNC .....	44
5.2	Emergency Handling.....	44
5.2.1	ITabMotion::SetEmergencyCallback1.....	44
5.3	ITabMotion Examples.....	45
5.3.1	Sending a PVT Trajectory by File .....	45
5.3.2	Sending a PVT Trajectory on-the-fly to a Single Axis .....	46
5.3.3	Sending a PVT Trajectory on-the-fly to Two Axes.....	47
	Appendix: Getting Started Tutorial .....	48
A.1	Creating Your Project File .....	48
A.2	Defining Your Project Parameters.....	49
A.3	Writing the Program Code.....	51
A.4	Compiling and Running the Program .....	52

# 1 Introduction

The Elmo Interlude is a communication software package designed to enable software developers to program applications for motion control systems that integrate Elmo digital servo drives. The software provides an efficient and easy communication channel directly to the connected Elmo drive in order to facilitate integration with the user application. This saves programming time and effort, and ensures compatibility between the user application and the Elmo drive.

The Interlude API is integrated as a DLL directly into the user application, operating as an embedded part of it. It supports the following types of synchronized communication:

- RS-232 serial communication.
- CiA CANopen communication protocol according to DS 301; the exact details are provided in the *Elmo CANopen Implementation Guide*.

The main interfaces provided through the Interlude API include:

- **IDriver:**

This interface contains groups of functions – applicable to both RS-232 and CANopen communication – used to:

- Establish (and disconnect) communication with an Elmo drive and send commands to the drive.
- Upload and download applications from/to the drive.
- Download firmware.
- Create and load a network configuration file.
- Define and activate a callback mechanism for error handling.
- Check for CANopen communication as a first step in operating fast motion mode via CANopen.
- Check for CANopen communication as a first step in operating a direct PVT mode via CANopen.

- **ICanService:**

This interface is used to run the fast motion modes through CANopen communication, performing the following functions:

- Initiate fast motion (PVT and PT) operation.
- Define PVT or PT trajectories.
- Start and stop the fast motion and send the command to one or more Elmo drives.
- Activate (and cancel) a synchronization mechanism.
- Define and activate a defined emergency callback mechanism.

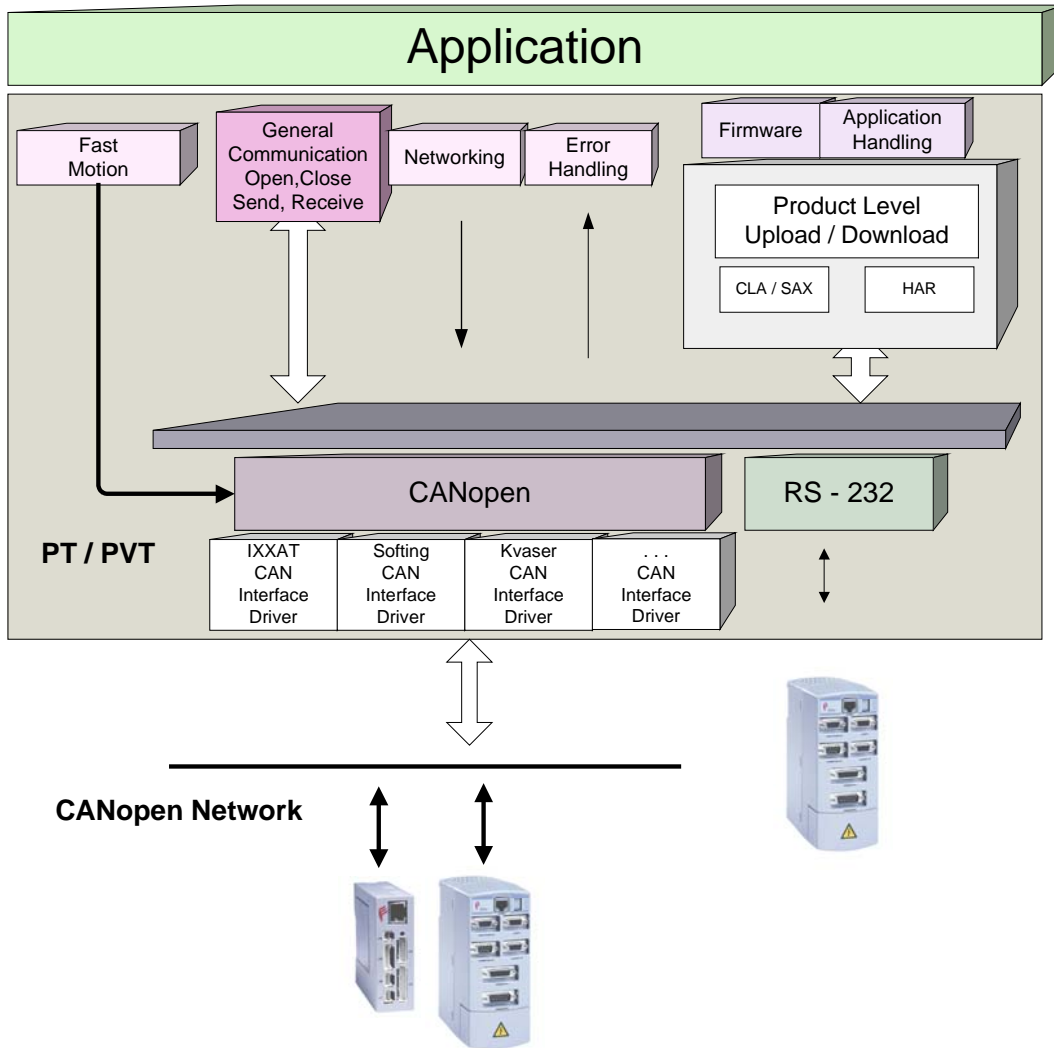
▪ **ITabMotion:**

This interface is used to program and run PVT fast motion modes on-the-fly, through CANopen communication. It includes the following functions:

- Prepare the drive to run PVT motion.
- Send PVT points to the drive buffer.
- Send a Start Motion command to one or more Elmo drives.
- Retrieve the Time Stamp of a designated drive.
- Activate the CANopen emergency object mechanism for error handling.
- Retrieve the queue size to determine how many points are currently in the queue.
- Stop the transmission of SYNC messages at the end of the motion trajectory.



To run the Interlude API, users must be proficient in programming with Microsoft Visual C++.



**Interlude Block Diagram**

## 2 Installation

The Interlude package contains the following files:

- ElmoComApi.dll (release version)
- ElmoComApi\_D.dll (debug version)
- ElmoComApi.lib (release version)
- ElmoComApi\_D.lib (debug version)
- IDriver.h
- {EXE\_Path}\DspCompiler\DamaCompiler.exe

These files are available as a Win-Zip file for copying directly onto the hard drive of the host computer. A setup routine (setup.exe file) can install the files into a destination directory or the user can manually copy them to the required locations for direct access by the user application.

Be sure that the files are located in the same folder as the user application files.

To operate this application with CANopen, a CAN network must already be installed, using one of the following CAN boards:

- IXXAT
- Kvaser
- Softing

Be sure that the CAN board has been installed properly, according to the installation instructions provided in the relevant CAN board manual.



### Notes:

- The ElmoComApi.dll (or ElmoComApi\_D.dll) file must be located in the same folder as the user application \*.exe file.
- The DamaCompiler.exe file must be located in a DspCompiler folder that resides in the user application directory.
- The VCI library must be installed in the host computer in order for the CAN IXXAT card to operate properly.

The [Appendix](#) to this manual provides a step-by-step tutorial for creating your application program.

## 3 *IDriver Interface*

This interface is applicable to both RS-232 and to CANopen. It contains the following groups of functions:

- **General Communication:**
  - DriverConnect Starts communication (by creating an instance of IDriver)
  - DriverDisconnect Stops communication (by destroying an instance of IDriver)
  - SendCommand Sends Elmo commands
  - GetComInfo Retrieves the active communication parameters
- **Application Handling:**
  - UploadApplication Uploads an application (parameters, user program, etc.) from the connected drive to the host computer
  - DownloadApplication Transfers an application (parameters, user program, etc.) from the host computer to the connected drive
  - IsEqualCommunicationParameter Compares communication parameters between the user's application file and those residing in the drive
- **Firmware:**
  - DownloadFirmware Downloads the firmware (\*.abs) file to the connected drive
- **Networking:**
  - SaveSComInfoCollection Saves a set of communication parameters of all currently active drives to the network configuration file
  - GetSizeSComInfoCollection Retrieves the number of communication information blocks defined in the network configuration file
  - LoadSComInfoFromCollection Loads one communication information block from the saved network configuration file
- **Error Handling:**
  - SetErrorCallback Defines the desired error callback function
- **CAN Service:**
  - GetCanService Gets a pointer to the ICanService instance when CAN-based communication is used
  - ITabMotion Gets a pointer to the ITabMotion instance when CAN-based communication is used

The first three functions in the General Communication group – DriverConnect, DriverDisconnect and SendCommand – are mandatory when using the Interlude API. They enable the user to connect directly to a specified Elmo digital drive (through either RS-232 or CANopen communication), send commands/receive responses, and subsequently disconnect the drive from the open communication. Error handling is provided. The target Elmo drive is represented by IDriver.

## 3.1 General Communication

### 3.1.1 IDriver::DriverConnect

This function enables the user to initiate either RS-232 or CAN communication with the connected Elmo drive. The function:

- Initializes the communication hardware.
- Sets the communication parameters, such as baud rate and node ID (in CAN).
- Checks the communication link and, if successful, returns a pointer to the drive specified as IDriver. If a failure occurs, it returns an error (NULL).
- If the drive to be connected is in boot state, initiates a firmware download.

#### Syntax 1:

```
static IDriver* DriverConnect(DRIVER_CALLBACK pfnErrorProc=NULL);
```

#### Syntax 2:

```
static IDriver* DriverConnect(CString sPathToApplicationFile, DRIVER_CALLBACK pfnErrorProc=NULL);
```

#### Syntax 3:

```
static IDriver* DriverConnect(const SCommunicationInfo &SComInfo, DRIVER_CALLBACK pfnErrorProc=NULL);
```

#### Input Parameters:

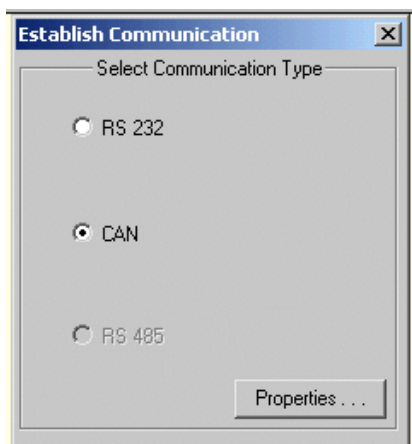
DRIVER_CALLBACK pfnErrorProc	Defines the callback function for error handling (see <a href="#">section 3.1.5</a> ). If the parameter is NULL (default), no error handling is used.
CString sPathToApplicationFile	Full path to application (*.dat) file from which the communication parameters are specified.
SCommunicationInfo &SComInfo	Reference to the SCommunicationInfo structure ( <a href="#">section 3.1.6</a> ), filled in by the user.

#### Returned Values:

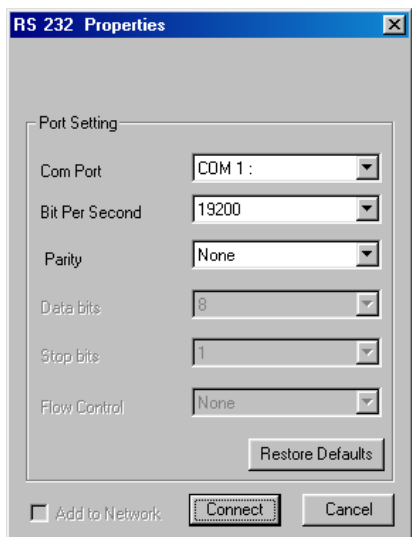
- If successful: pointer to IDriver.
- If error occurs: NULL.

**Remarks:**

- This function must be entered first in order to activate communication to the Elmo drive.
- When using the Interlude graphical interface (Syntax 1 previously), the following dialog boxes are displayed:

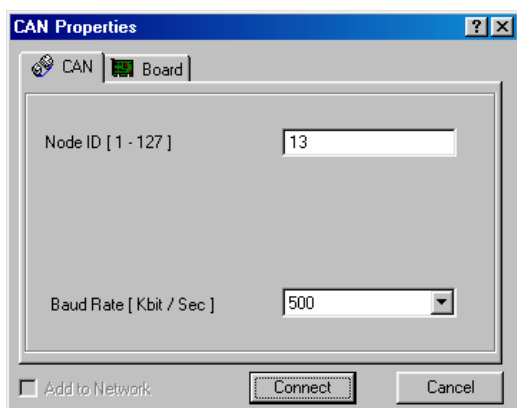


Used to select the communication type. To further define the exact communication parameters, click **Properties**.



Displayed for entering the RS-232 parameters. The **Restore Defaults** button sets the following defaults:

- **Com Port:** COM 1
- **Bits Per Second:** 19,200
- **Parity:** None



If CAN is selected, the CAN tab is used to define the CAN **Node ID** (up to 127) and **Baud Rate**. The Board tab is used to define the board **Manufacturer**, the CAN **Board Type** and the **CAN Number** of the on-board controller .

**Example 1:**

```
    IDriver *m_pDriver=NULL;
...
    m_pDriver = IDriver::DriverConnect();
    if(m_pDriver == NULL)
{
...        // Error processing
}
```

**Example 2:**

```
void CallBackError(IDriver *, const char *sError)
{
...        // Error processing
}
...
IDriver *m_pDriver = NULL;
...
    m_pDriver = IDriver::DriverConnect(CallBackError);
    if(m_pDriver == NULL)
{
...        // Error processing
}
```

**Example 3:**

```
IDriver *m_pDriver = NULL;
...
void CallBackError(IDriver *pDriver, const char *sError)
{
...        // Error processing
    AfxMessageBox(sError);
}

...
SCommunicationInfo SComInfo;
...        // Initialization of SComInfo

    m_pDriver = IDriver::DriverConnect(SComInfo ,CallBackError);
    if(m_pDriver == NULL)
{
...        // Error processing
}
```

### 3.1.2 IDriver::DriverDisconnect

This function shuts down communication with the connected Elmo drive specified by the IDriver pointer and initiated through the DriverConnect function. The function:

- Destroys the IDriver object.

**Syntax:**

```
void DriverDisconnect (IDriver *pIDriver);
```

**Input Parameters:**

IDriver *pIDriver	Specifies the drive for which communication is to be disconnected.
-------------------	--

**Remarks:**

If CAN communication is used, this function resets the communication card if the specified IDriver is the last remaining node in the network.

**Example:**

```
IDriver *m_pDriver = IDriver::DriverConnect();  
...  
IDriver::DriverDisconnect(m_pDriver);
```

### 3.1.3 IDriver::SendCommand

This function enables the user to send commands to and receive responses from the connected Elmo drive. The function:

- Uses a synchronized mechanism so that each send command is always followed by a response before another command is initiated.
- Sends the following types of commands: Set, Query, Configuration, Motion and Status (according to the *Elmo Command Reference Manual*).
- Distinguishes between RS-232 and CANopen communication.

#### Syntax 1:

```
BOOL SendCommand (CString sInput, CString& sOutput, CString& sError,
DWORD dwTimeout)
```

Does not use callback error handling.

#### Syntax 2:

```
BOOL SendCommand(CString sInput, CString& sOutput, DWORD dwTimeOut);
```

Uses callback error handling to get error string.

#### Input Parameters:

CString \*sInput

Input string to be sent to drive. This is an Elmo command or expression, as defined in the *Elmo Command Reference Manual*.

DWORD dwTimeOut

The maximum wait time for receiving a response from the drive, in milliseconds.

#### Output Parameters:

CString& sOutput

The “receive string” sent by the drive to acknowledge receipt of the input command. The command responses are provided in the *Elmo Command Reference Manual*.

CString& sError

Error string if an error occurs.

#### Returned Values:

- If command is sent successfully: TRUE
- If error occurs: FALSE

#### Remarks:

- Error handling (formatted error string or calling the error callback function) is defined by the user through the SetErrorCallback function ([section 3.5](#)).
- A timeout alert is given if communication is lost.

**Example:**

```

CString sOutput, sInput sError;
sInput.Format("PX=%d",1234567);
if(!m_pDriver->SendCommand(sInput, sOutput, sError))
{
    AfxMessageBox(sError); // Error processing      ;

    ...      }
else
{
    . . . //Response processing
}

```

**3.1.4 IDriver::GetComInfo**

This function reads the communication parameters for the presently-active communication. The function:

- Returns the communication type of the currently-activate communication.
- Outputs a pointer to a string that gives the retrieved parameters.

**Syntax:**

```
COMMUNICATON_TYPE GetComInfo (CString *pstrCommunicationInfo = Null);
```

**Output Parameters:**

\*pstrCommunicationInfo                      Pointer to string of communication parameters.

**Returned Values:**

COMMUNICATION\_TYPE

Defined as:

COMMUNICATION\_TYPE::CT\_R232

COMMUNICATION\_TYPE::CT\_CAN

COMMUNICATION\_TYPE::CT\_NONE

**Example:**

```

...
CString strComParams;

m_pDriver->GetComInfo(&strComParams);

GetDocument()->SetTitle(strComParams);

...

```

**Result of this code:**


RS\_232; COM1; 19200[bit/sec]; Parity None

### 3.1.5 DRIVER\_CALLBACK Function Type

This function type defines the error callback handler to be used with the Interlude API.

**Syntax:**

```
typedef void (__cdecl *DRIVER_CALLBACK)(IDriver *pDriver, const char *lpszFormatError);
```

**Callback Parameters:**

IDriver *pDriver	Pointer to the IDriver for which the emergency occurred.
const char *lpszFormatError	Formatted error string, with an explanation of the current error.

**Example:**

```
void CallBackError(IDriver *, const char *sError)
{
...           // Error processing
}
...
IDriver *m_pDriver=NULL;
...
m_pDriver = IDriver::DriverConnect(CallBackError);
if(m_pDriver == NULL)
{
...           // Error processing
}
```

### 3.1.6 SCommunicationInfo Structure

The following SCommunicationInfo structure includes the communication parameters used to fully define communication for a given IDriver.

```

struct SCommunicationInfo
{
    char sLineName [_MAX_PATH];           User description
    COMMUNICATION_TYPE CommType;         Either RS-232 or CAN
    struct RS_232                          RS-232 communication parameters
    {
        BYTE btPort;                       COM port number
        DWORD dwBaudRate;                   Current baud rate
        BYTE btParity;                       One of the following:
        0: No parity
        1: Odd parity
        2: Even parity
        3: Mark
        4: Space
    } rs;
    struct CAN                              CAN communication parameters
    {
        BYTE btNodeID                       CAN node ID
        char sManufacturer [MANUFACTURE_NAME_SIZE];
                                                Manufacturer name
        char sBoardType [BOARDTYPE_NAME_SIZE];
                                                Board name
        BYTE btCanNr                         CAN board number: 0 or 1
        CAN_BOUD_RATE BaudRateIndex;        Baud rate index
        BYTE bt_irq1;                         Interrupt number
        WORD dw_address1;                     Board segment address
    } can;
};
Members: sLineName                          Custom string.
CommType                                     Communication type. One of the following values:
CT_NONE
CT_RS232
CT_CAN
rs.btPort;                                  COM port number, one of the following:
1, 2, 3, 4, 5, 6, 7, 8.

```

<code>rs.dwBaudRate;</code>	Baud rate at which the communications device operates. This member can be an actual baud rate value or one of the following baud rate indices: CBR_4800 CBR_9600 CBR_19200 CBR_38400 CBR_57600 CBR_115,200
<code>rs.btParity</code>	Parity scheme to be used. This member can be one of the following values: 0: No parity 1: Odd parity 2: Even parity
<code>can.btNodeID</code>	CAN node ID: [1...127]
<code>can.sManufacturer</code>	Manufacturer name: either IXXAT, Kvaser or Softing.
<code>can.sBoardType</code>	Board name. One of the following: "tinCAN" "iPC-I 165" "iPC-I 320" "iPC-I 386" "iPC-I 165 PCI" "iPC-I 320 PCI"
<code>can.btCanNr</code>	CAN board number: 0 or 1
<code>can.BaudRateIndex</code>	One of following values: BR_10, BR_20, BR_50, BR_100, BR_125, BR_250, BR_500, BR_1000
<code>can.bt_irq1;</code>	Interrupt number
<code>can.dw_address1;</code>	Board segment address

## 3.2 Application Handling

The \*.dat user application file contains the following parts:

- Drive version
- Host communication parameters
- Full set of drive parameters
- User program

The interface group includes application upload and download functions that facilitate the transfer of user applications to and from the connected drive. The functions have been optimized to match the Elmo drive capabilities, the data type and the communication type.

### 3.2.1 IDriver::IsEqualCommunicationParam

This function serves as a preventive measure for checking the match between the communication parameters – such as communication type, baud rate and node ID – currently stored in the drive against the communication parameters of the specified application file. The function:

- Gets the communication parameters currently active in the drive.
- Gets the host communication parameters from the user application file.
- Compares the two sets of communication parameters and indicates if they match or not.
- Retrieves a formatted string with communication parameters from the specified application file.

#### Syntax:

```
BOOL IsEqualCommunicationParam (CString sPathToApplicationFile,  
CString &pstrCommunicationInfoFromFile);
```

#### Input Parameters:

CString sPathToApplicationFile                      Full path to application to be downloaded.

#### Output Parameters:

CString &pstrCommunicationInfoFromFile      Formatted string with host communication parameters.

#### Returned Values:

- If parameters are equal: TRUE
- If parameters differ from each other: FALSE

#### Remarks:

This function enables the user to check and determine whether or not to proceed with the IDriver::DownloadApplication function ([section 3.2.2](#)). It should always be performed in order to avoid unwanted overwriting of existing drive communication parameters.

**Example:**

```
• • •
IDriver *g_pDriver;

. . .
void DownloadApplication()
{
    CFileDialog fileDlg(TRUE, "dat", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT,
    "Application Files (*.dat)|*.dat||");
    if(fileDlg.DoModal() == IDCANCEL)
    {
        cerr << "Canceled..." << endl;
        return;
    }
    CString sError;
    CString sCommunicationInfo;
    if(!g_pDriver->IsEqualCommunicationParam(fileDlg.GetPathName(),
sCommunicationInfo))
    {
        if(AfxMessageBox("Current Communication Parameters is not Equal
Communication Parameters in this File.\n"
    " Click <Yes> for continue by anyway or <No> to
Cancel",MB_YESNO|MB_ICONEXCLAMATION)==IDNO)
        {
            cerr << _TEXT("Canceled...") << endl;
            return;
        }
    }
    if (!g_pDriver->DownloadApplication(fileDlg.GetPathName(),sError))
    {
        cout << (LPCSTR)sError << endl;
    }
    else
    {
        cerr << _TEXT("Ok") << endl;
    }
}
```

### 3.2.2 IDriver::DownloadApplication

This function downloads a user application to the connected Elmo servo drive. The function:

- Downloads the set of drive parameters to the connected Elmo drive.
- Downloads the user program part of the application.
- With Elmo *SimplIQ* drives, compiles the user program prior to download and then performs the download.
- With Elmo Clarinet, Saxophone and Mini-Saxophone drives, sends the Compile command to the drive after download is complete. If the download fails for some reason and error handling has been defined, a formatted error string will be returned.

#### Syntax:

```
BOOL DownloadApplication (CString sPathToFile, CString &sError,  
HWND hProgressWindow=NULL);
```

#### Input Parameters:

CString sPathToFile	Full path to application to be downloaded.
HWND hProgressWindow=NULL	The window handle, which receives the following messages: <ul style="list-style-type: none"><li>▪ WM_BEGIN: Download begin</li><li>▪ WM_STEP: Download in progress</li><li>▪ WM_FINISH: Download finish</li></ul>

#### Output Parameters:

CString &sError	Formatted error string if download is not successful.
-----------------	---

#### Returned Values:

- If download is successful: TRUE
- If download fails (see error string): FALSE

**Example:**

```
...
IDriver *g_pDriver;
...
void DownloadApplication()
{
    CFileDialog fileDlg(TRUE, "dat", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT,
        "Application Files (*.dat)|*.dat||");
    if(fileDlg.DoModal() == IDCANCEL)
    {
        cerr << "Canceled..." << endl;
        return;
    }
    CString sError;
    CString sCommunicationInfo;
    if(!g_pDriver->IsEqualCommunicationParam(fileDlg.GetPathName(),
sCommunicationInfo))
    {
        if(AfxMessageBox("Current Communication Parameters is not Equal
Communication Parameters in this File.\n"
            " Click <Yes> for continue by anyway or <No> to
Cancel",MB_YESNO|MB_ICONEXCLAMATION)==IDNO)
        {
            cerr << _TEXT("Canceled...") << endl;
            return;
        }
    }
    if (!g_pDriver->DownloadApplication(fileDlg.GetPathName(),sError))
    {
        cout << (LPCSTR)sError << endl;
    }
    else
    {
        cerr << _TEXT("Ok") << endl;
    }
}
}
```

### 3.2.3 IDriver::UploadApplication

This function uploads a user application from the connected Elmo servo drive to the file on the host computer.

**Syntax:**

```
BOOL UploadApplication (CString sPathToFile, CString &sError);
```

**Input Parameters:**

CString sPathToFile	Full path to location at which uploaded file should be saved.
---------------------	---

**Output Parameters:**

CString &sError	Formatted error string if upload is not successful.
-----------------	---

**Returned Values:**

- If upload is successful: TRUE
- If upload fails (see error string): FALSE

**Remarks:**

The file format of the uploaded user application is identical to that of user application files created through the Elmo Composer IDE, thereby guaranteeing full compatibility between user application file formats.

**Example:**

```
...
IDriver *g_pDriver;
...
void UploadApplication()
{
    CFileDialog fileDlg(FALSE, "dat", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT,
        "Application Files (*.dat)|*.dat||");
    if(fileDlg.DoModal() == IDCANCEL)
    {
        cerr << _TEXT("Canceled...") << endl;
        return;
    }

    CString sError;
    CWaitCursor tmpObject;
    if(!g_pDriver->UploadApplication(fileDlg.GetPathName(), sError))
    {
        cout << (LPCSTR)sError << endl;
    }
    else
    {
        cerr << _TEXT("Ok") << endl;
    }
}
```

### 3.2.4 IDriver::DownloadProgram

This function downloads a user program to the connected Elmo servo drive from the file on the host computer.

#### Syntax:

```
BOOL DownloadProgram (CString sPathToFile, CString &sError,
                     HWND hProgressWindow=NULL);
```

#### Input Parameters:

CString sPathToFile	Full path to location of program file.
HWND hProgressWindow=NULL	The window handle, which receives the following messages: <ul style="list-style-type: none"> <li>▪ WM_BEGIN: Download begin</li> <li>▪ WM_STEP: Download in progress</li> <li>▪ WM_FINISH: Download finish</li> </ul>

#### Output Parameters:

CString &sError	Formatted error string if download is not successful.
-----------------	---

#### Returned Values:

- If download is successful: TRUE
- If download fails (see error string): FALSE

#### Remarks:

The program file format: "Elmo High Language" (ehl).

#### Example:

```
...
IDriver *g_pDriver;
...
void DownloadProgram( HWND hOwner )
{
    CFileDialog fileDlg(TRUE, "ehl", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT,
        "Program Files (*.ehl)|*.ehl||");
    if(fileDlg.DoModal() == IDCANCEL)
    {
        cerr << _TEXT("Canceled...") << endl;
        return;
    }

    CString sError;
    CWaitCursor tmpObject;
    if(!g_pDriver->DownloadProgram(fileDlg.GetPathName(), sError,
hOwner))
    {
        cout << (LPCSTR)sError << endl;
    }
    else
    {
        cerr << _TEXT("Ok") << endl;
    }
}
```

### 3.2.5 IDriver::UploadProgram

This function uploads a user program from the connected Elmo servo drive to the file on the host computer.

**Syntax:**

```
BOOL UploadProgram (CString sPathToFile, CString &sError);
```

**Input Parameters:**

CString sPathToFile	Full path to location at which uploaded program file should be saved.
---------------------	---

**Output Parameters:**

CString &sError	Formatted error string if upload is not successful.
-----------------	---

**Returned Values:**

- If upload is successful: TRUE
- If upload fails (see error string): FALSE

**Remarks:**

The file format of the uploaded program: "Elmo High Language" (ehl).

**Example:**

```
...
IDriver *g_pDriver;
...
void UploadProgram()
{
    CFileDialog fileDlg(FALSE, "ehl", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT,
        "Program Files (*.ehl)|*.ehl||");
    if(fileDlg.DoModal() == IDCANCEL)
    {
        cerr << _TEXT("Canceled...") << endl;
        return;
    }

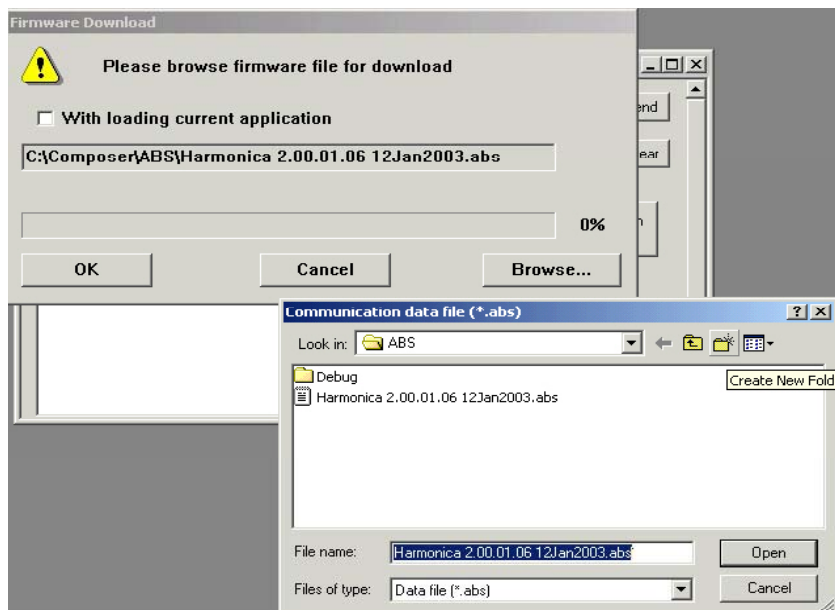
    CString sError;
    CWaitCursor tmpObject;
    if(!g_pDriver->UploadProgram(fileDlg.GetPathName(), sError))
    {
        cout << (LPCSTR)sError << endl;
    }
    else
    {
        cerr << _TEXT("Ok") << endl;
    }
}
```

## 3.3 Firmware

### 3.3.1 IDriver::DownloadFirmware

This function displays a graphical interface dialog box to facilitate the download of a firmware program. The function:

- Downloads the firmware from the host computer to the connected drive.
- Is the same as the **Tools - Firmware Download** function in the Elmo Composer IDE.



Used to browse to the location of the file to be downloaded and to start the download process (by clicking **OK**).

#### Syntax:

```
int DownloadFirmware ();
```

#### Returned Values:

- If download is successful: ID\_OK
- If user presses **Cancel** in displayed dialog box: ID\_CANCEL
- If error occurs during download or if user cancels process after it has begun: ID\_ABORT

#### Remarks:

- During the actual download, status messages are displayed to indicate the progress of the process. Upon completion of the download, the user needs to reboot the drive.
- If error occurs (ID\_ABORT is returned), the next download can be performed only after drive is restarted in boot mode.

**Example:**

```
...
IDriver *g_pDriver;
...

int res = g_pDriver->DownloadFirmware();
switch(res)
{
case IDCANCEL:
    cerr << _TEXT("Canceled...") << endl;
    break;
case IDABORT:
    cout << _TEXT("Abort firmware load!!!") << endl;
    break;
case IDOK:
    cerr << _TEXT("Ok") << endl;
    break;
}
```

### 3.3.2 IDriver::DownloadFirmwareEx

This function downloads the firmware from the host computer to the connected drive.

**Syntax:**

```
int DownloadFirmwareEx(CString sPathToFile, BOOL bBoot, HWND hProgressWindow=NULL);
```

**Input Parameters:**

CString sPathToFile	Full path to firmware file.
BOOL bBoot	Device boot mode indication flag.
HWND hProgressWindow=NULL	The window handle, which receives the following messages: <ul style="list-style-type: none"><li>▪ WM_FIRMDOWNLOAD_ERROR: Download error</li><li>▪ WM_FIRMDOWNLOAD_STEP: Download in progress</li><li>▪ WM_FIRMDOWNLOAD_OK: Download finish</li></ul>

**Returned Values:**

- If download is successful: 0
- If error occurs during download or if user cancels process after it has begun:
  - 1 - communication error;
  - 2 - send MO command failed;
  - 3 - motor must be off;
  - 4 - send PS command failed;
  - 5 - program is running;
  - 6 - invalid product dependent methods;
  - 7 - invalid firmware version;

- 8 - firmware is not appropriate for downloading;
- 9 - can't get product information;
- 10 - firmware download failed;
- 11 - firmware download failed due to timeout;
- 12 - firmware password is empty;
- 13 - firmware data is empty;
- 14 - send portion of data failed;
- 15 - send WS command failed;
- 16 - send TP command failed;
- 17 - invalid firmware download thread;
- 18 - flash error;
- 19 - firmware download canceled;
- 20 - firmware download wasn't completed successfully;
- 21 - unable to create event;

**Remarks:**

- During the actual download, status messages are displayed to indicate the progress of the process. Upon completion of the download, the user needs to reboot the drive.
- If an error occurs, the next download can be performed only after drive is restarted in boot mode.

**Example:**

...

```
IDriver *g_pDriver;
```

...

```
g_pDriver->DownloadFirmwareEx( sPath, FALSE, m_hWnd );
```

## 3.4 Networking

The Interlude network is a collection of blocks of drive communication parameters. Each block is described by an `SCommunicationInfo` structure (section 3.1.6). The Networking group of functions enables the user to first create and store a network file, and subsequently upload the file to establish communication for the entire configuration.

### 3.4.1 `IDriver::SaveSComInfoCollection`

This function creates and saves a file that includes all the network setup parameters, for subsequent fast configuration of the network. The function:

- Creates a \*.net collection file that includes data for all operating `IDrivers`.
- Saves the network to the specified file.

#### Syntax:

```
BOOL SaveSComInfoCollection (CString sPathToNetworkFile, CString &sError);
```

#### Input Parameters:

`CString sPathToNetworkFile` Full path to location at which the network file.

#### Output Parameters:

`CString &sError` Formatted error string if an error occurs.

#### Returned Values:

- If file creation/save is successful: `TRUE`
- If error occurs: `FALSE`

#### Remarks:

The file format of the collection file is the same format as the \*.net file used by the Elmo Composer IDE.

#### Example:

```
...  
IDriver *pDriver  
...  
CString sError;  
if(!pDriver->SaveSComInfoCollection ("Network.net", sError))  
{  
    AfxMessageBox(sError); //Error processing  
}
```



### 3.4.3 IDriver::LoadSComInfoFromCollection

This function loads one block of communication parameters (an SCommunicationInfo structure) from the network collection file.

#### Syntax:

```
BOOL LoadSComInfoFromCollection (SCommunicationInfo&SComInfo,
CString sPathToNetworkFile, CString &sError, int index);
```

#### Input Parameters:

CString sPathToNetworkFile	Full path to saved network file.
int index	Sequential number of IDriver block in the network collection file, used to establish communication with the drive.

#### Output Parameters:

SCommunicationInfo & SComInfo	Structure used in IDriver::DriveConnect to define communication parameters for the drive to be connected ( <a href="#">section 3.1.1</a> ).
CString &sError	Formatted error string if an error occurs.

#### Returned Values:

- If upload is successful: TRUE
- If error occurs: FALSE

#### Remarks:

This function facilitates network restoration.

#### Example:

```
...
CArray<IDriver*, IDriver*> m_arrpIDriver;
...
CString fileName;
const char szFilter[] = "Network Files (*.net)|*.net||";
CFileDialog dlg(TRUE, "net", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT|OFN_FILEMUSTEXIST, szFilter);
dlg.m_ofn.lpstrTitle = _T("Load Network");
if(dlg.DoModal() != IDCANCEL) return ;
fileName = dlg.GetPathName();
SCommunicationInfo m_SCommInfo;
CString sError;
m_arrpIDriver.RemoveAll();
IDriver *pDriver;

for(UINT i=0; i<IDriver::GetSizeSComInfoCollection(fileName); i++)
{

if(!IDriver::LoadSComInfoFromCollection(m_SCommInfo, fileName, sError, i))
{
AfxMessageBox(sError);
}
```



## 3.6 CAN Service Interfaces

### 3.6.1 IDriver::GetCanService

This function checks if it is possible to access the ICanService interface for a selected IDriver. It requires CAN communication defined for the IDriver. The function:

- Checks if the IDriver object is based on CAN communication.
- Returns a pointer to the ICanService interface if the object is CAN-based; otherwise, it returns NULL.

**Syntax:**

```
ICanService * GetCanService ();
```

**Returned Values:**

- If IDriver is CAN-based: pointer to ICanService interface
- If IDriver is RS-232-based: NULL

**Example:**

See [section 4.1.1](#).

### 3.6.2 IDriver::GetTabMotion

This function checks if it is possible to access the ITabMotion interface for a selected IDriver. It requires CAN communication defined for the IDriver. The function:

- Checks if the IDriver object is based on CAN communication.
- Returns a pointer to the ITabMotion interface if the object is CAN-based; otherwise, it returns NULL.

**Syntax:**

```
ITabMotion * GetTabMotion ();
```

**Returned Values:**

- If IDriver is CAN-based: pointer to ITabMotion interface
- If IDriver is RS-232-based: NULL

**Example:**

See [section 5.3](#).

## 4 ICanService Interface (CAN only)

This interface is applicable to CANopen only. It is used to initiate, run and stop fast motion modes, and to define a CANopen emergency handling object. The interface contains the following groups of functions:

- **Fast Motion:**
  - InitFastMotionMode                      Initiate the fast motion operation
  - SetPVTPoint and SetPTPoint            Define the fast motion trajectory
  - ReadyFastMotion                        Prepare the drive for fast motion mode
  - StopFastMotion                         Stop the fast motion
  - CreateSyncManager                     Send synchronization messages
  - DestroySyncManager                    Cancel the Sync Manager
- **Emergency Handling:**
  - SetEmergencyCallback                    Executed through the use of the CANopen  
Emergency object

After communication via CANopen has been established with the Elmo drive, the user may define a PT or PVT fast motion mode. The Interlude ICanService interface uses TPDO1 to transfer the data according to the selected mode, performing the following:

- Mapping the fast motion object to TPDO1.
- Handling the fast mode parameters (MP command).
- Feeding the drive buffer according to the selected motion protocol.
- Handling emergencies related to the fast motion mode.

### 4.1 Fast Motion

In order to activate a fast motion mode, the user must perform these steps:

1. Define the fast motion source data.
2. Initialize the data parameters.
3. Call up the data points from their source table.
4. Activate the fast motion mode to run the trajectory.

This group of functions enable the user to easily and efficiently perform this procedure.

### 4.1.1 ICanService::InitFastMotionMode (PVT and PT)

This function is a collection of encapsulated send commands needed to initialize the selected fast motion mode.

#### Syntax:

```
BOOL InitFastMotionMode (const FAST_MOTION_PARAM* pParam, CString& sError)
```

#### Input Parameters:

const FAST\_MOTION\_PARAM\* pParam                      Pointer to parameter structure defined in [section 4.1.10](#).

#### Output Parameters

CString& sError    Error string if an error occurs.

#### Returned values:

- If function is sent successfully: TRUE
- If an error occurs: FALSE

#### Example:

```
IDriver* pDriver;
...
ICanService * pICanService;
pICanService = pDriver->GetCanService();
if (pICanService==NULL) return;
...
if (!pDriver ->SendCommand("MO=0", str)) return;
if (!pDriver ->SendCommand("UM=5", str)) return;
CString sError;
FAST_MOTION_PARAM Param;
ZeroMemory(&Param, sizeof(FAST_MOTION_PARAM));
Param.StructSize = sizeof(FAST_MOTION_PARAM);
Param.m_bRepetitive = FALSE;
Param.m_hControlWnd = GetSafeHwnd();
Param.MotionType = PVT_TYPE;
Param.StaticTrajLength = NUM_OF_POINTS;
if (!pICanService ->InitFastMotionMode(&Param, sError))
{
    AfxMessageBox(sError);
    return;
}
```

### 4.1.2 ICanService::SetPVTPoint

This function pulls the user's PVT data from its source table into the DLL buffer. The function:

- Gets the position, velocity, time and row number for each data point.
- Formats the data according to the required CAN format (refer to the Elmo *CANopen Implementation Guide*).
- Prepares the data for running the fast motion trajectory through the CANopen network.

#### Syntax:

```
BOOL SetPVTPoint (_int32 Pos, _int32 Vel, BYTE time, unsigned long counter,
                  CString& sError);
```

#### Input Parameters:

_int32 Pos	Position value.
_int32 Vel	Velocity value.
BYTE time	Time instance.
unsigned long counter	Row (point) number.

#### Output Parameters:

CString& sError	Error string returned if an error occurs.
-----------------	---

#### Returned Values:

- If operation is successful: True
- If an error occurs: False

#### Example:

```
//Source table
int arrPVTData[][3] =
{ //position, velocity, time
  {0, 0, 100},
  {21, 427, 100},
  {85, 847, 100},
  {190, 1252, 100},
  {335, 1636, 100},
  {517, 1992, 100},
  { . . . },
  {0, 0, 100}
};
//
IDriver* pDriver;
...
ICanService * pICanService;
pICanService = pDriver->GetCanService()
if(pICanService==NULL)return;
```

```

UINT SizeOfPVTData=...; //size of table
for(int i=0; i< SizeOfPVTData; i++)
{
    if(!pICanService ->SetPVTPoint(arrPVTData[0], arrPVTData[1], (BYTE)
arrPVTData[2], i, sError))
    {
        AfxMessageBox(sError);
        return;
    }
}

```

### 4.1.3 ICanService::SetPTPoint

This function pulls the user's PT data from its source table into the DLL buffer. The function:

- Gets the set of PT position points.
- Formats the data according to the required CAN format (refer to the *Elmo CANopen Implementation Guide*).
- Sends two consecutive points in a single packet.

#### Syntax:

```
BOOL SetPTPoint (_int32 Pos1, _int32 Pos2, unsigned long counter, CString& sError)
```

#### Input Parameters:

_int32 Pos1	First point position.
_int32 Pos2	Second point position.
unsigned long counter	Point-pair number.

#### Output Parameters:

CString& sError	Error string returned if an error occurs.
-----------------	---

#### Returned Values:

- If operation is successful: TRUE
- If an error occurs: FALSE

#### Example:

```

//Source table
int arrPT_PairPoints_Data[][2] =
{ //position 1, position 2
  {0, 21},
  {85, 190},
  {335, 517},
  { . . . },
  {21, 0}
};
//
IDriver* pDriver;
...
ICanService * pICanService;

```





### 4.1.6 ICanService:: SendCommandGroupID

This function enables the user to send commands to the connected Elmo drives that have the same group ID number. The function:

- Sends commands without waiting for any response (in contrast to the Send Command; see [section 3.1.3](#)).
- Sends all types of commands according to the Elmo *Command Reference Manual*.



#### Notes:

- When a drive belonging to a group responds to a command, it uses its own drive ID rather than the group ID.
- A group of drives is defined using the PP[15] command, explained in the *SimplIQ Command Reference Manual*.

#### Syntax:

```
BOOL SendCommandGroupID(BYTE GroupID, const CString & sInput, CString& sError);
```

#### Input Parameters:

BYTE GroupID

Group ID number of the CAN node (Elmo drives) to which the command will be sent.

CString \*sInput

Input string to be sent to drive. This is an Elmo command or expression, as defined in the *Elmo Command Reference Manual*.

#### Output Parameters:

CString& sError

Error string if an error occurs.

#### Returned Values:

- If operation is successful: TRUE
- If an error occurs: FALSE

#### Example:

```
IDriver* pDriver;
...
ICanService * pICanService;
pICanService = pDriver->GetCanService();
if(pICanService==NULL)return;
...
BYTE GroupID =100;
CString sInput = ST;
CString sError;

//Send command ST to all nodes for which group ID equal 100

if(!pICanService -> SendCommandGroupID(GroupID, sInput, sError))
{
    AfxMessageBox(sError);
}
```

### 4.1.7 ICanService:: SendBGSyncOnTime

This function enables the user to synchronize the start of motion to connected Elmo drives that have the same group ID number. The start will occur after the delay defined in the function (in microseconds). The function:

- Gets current time stamp
- Adds the delay time, in microseconds, to the time stamp.
- Sends a BT (Begin on Time) command.

#### Syntax:

```
BOOL SendBGSyncOnTime(DWORD Delay, BYTE NodeID, CString &sError);
```

#### Input Parameters:

**DWORD Delay**

The delay time, in microseconds, after which the start will occur.

**BYTE GroupID**

Group ID number of the CAN node (Elmo drives) to which the command will be sent.

#### Output Parameters:

**CString& sError**

Error string if an error occurs.

#### Returned Values:

- If operation is successful: TRUE
- If an error occurs: FALSE

#### Example:

```
IDriver* pDriver;
...
ICanService * pICanService;
pICanService = pDriver->GetCanService();
if(pICanService==NULL)return;
...
BYTE GroupID =100;
DWORD Delay =1000000; //in microsecond
CString sError;

//Starts all nodes whose group ID equals 100 and after a delay of
1,000,000 microseconds
if(!pCCanService->SendBGSyncOnTime(Delay, GroupID, sError))
{
AfxMessageBox(sError);
}
```



3. From the **Main Node** drop-down list, select the node that is to serve as the Sync Master. This time stamp will be used to synchronize all other connected nodes.
4. From the **Delay** drop-down list, select the time interval after which each sync message should be sent.
5. Click **Apply**. The system will begin to send sync and time stamp messages.
6. In order to stop the message transfer, select **(None)** from the **Main Node** list and click **Apply**.

### 4.1.9 ICanService::DestroySyncManager

This function cancels the instance of the Sync Manager created through the CreateSyncManager function. This function removes the entire Sync Manager mechanism from the host memory.

#### Syntax:

```
static void DestroySyncManager ( )
```

#### Example:

```
...
IDriver *pDriver;
...

    pDriver->DestroySyncManager( );
```

### 4.1.10 FAST\_MOTION\_PARAM Parameters

The following FAST\_MOTION\_PARAM structure includes the parameters used to initialize fast motion mode; they are entered in the InitFastMotionMode function.

```
struct FAST_MOTION_PARAM
{
    unsigned int StructSize;
    Fast_Motion_Type MotionType;
    unsigned int StaticTrajLength;
    unsigned int PTsegmentTime;

    BOOL m_bRepetitive;

    HWND m_hControWnd;
};
```

PVT\_TYPE for PVT; PT\_TYPE for PT  
 Length of data in points, number  
 MP[4]: The number of controller sampling times for each PT motion segment.  
 Repeat trajectory: True for repeat; False for one-time only.  
 Window for sending Fast Motion status messages:  
 PVT\_STOP: Motion has terminated.  
 PVT\_INIT\_DONE: Fast motion mode has been initiated successfully.  
 WM\_PVT\_DONE: Fast motion mode has stopped (trajectory is complete).

## 4.2 Emergency Handling

The Interlude API enables the user of CAN networking to implement an optional unsynchronized emergency message mechanism for error handling in the user application. The emergency object is defined according to the structure of the manufacturer-specific emergency message. During PT/PVT motion, the servo drive may issue emergency objects in order to indicate an error or to signal that it is in immediate need of additional data to prevent data queue underflow.

### 4.2.1 ICanService::SetEmergencyCallback

The callback handler function must be prepared by the user and defined for use with the DLL by a function pointer in the SetEmergencyCallback function. It is up to the user to select this option or to use a NULL pointer to turn off the callback handling emergency mechanism. This function:

- Activates the drive emergency callback function if the input parameter is not NULL.
- Turns off the emergency callback function if the input parameter is equal to NULL.

#### Syntax:

```
BOOL SetEmergencyCallback (EMERGENCY_CALLBACK pfnCANEmergencyCallback,
                          CString& sError);
```

#### Input Parameters:

EMERGENCY\_CALLBACK pfnCANEmergencyCallback

Defines the use of the emergency callback handler, described in [section 4.2.2](#).

#### Output Parameters

CString& sError

Error string if error occurs.

#### Returned Values:

- If operation is successful: TRUE
- If an error occurs: FALSE

#### Remarks:

The emergency is sent asynchronously, in parallel with the response.

#### Example:

```
static void __cdecl EmergencyProc (IDriver* pDriver ,
                                   const EMERGENCY_STRUCT *pEMCY,
                                   const char *lpszFormatError
                                   );

if (!m_pDriver->SetEmergencyCallback(EmergencyProc, sError))
{
    AfxMessageBox(sError);
}
```

## 4.2.2 EMERGENCY\_CALLBACK Parameters

This function defines the emergency callback error handler to be used with the Interlude API.

### Syntax:

```
typedef void (_cdecl *EMERGENCY_CALLBACK) (IDriver *pDriver,
const EMERGENCY_STRUCT *pEMCY, const char *lpszFormatError);
```

### Parameters:

IDriver *pDriver	Pointer to IDriver for which emergency occurred.
Pointer to EMERGENCY_STRUCT	Structure of emergency object, described in <a href="#">section 4.2.3</a> .
const char *lpszFormatError	Formatted error string, with explanation of emergency.



This function must be in progress as soon as possible.

## 4.2.3 EMERGENCY\_STRUCT Parameters

The emergency message data is formatted according to an Elmo-specific structure.

```
typedef struct EMERGENCYSTRUCT{
    unsigned short ErrorCode;           Standard error codes, as defined in "Emergency"
                                        chapter of the Elmo CANopen Implementation
                                        Manual.

    BYTE;           ErrorRegister;     Error register.

    BYTE           ElmoErrorCode;     Elmo error code, for PT/PVT modes only.

    unsigned short DataField1;        Error code data field 1.

    unsigned short DataField2;        Error code data field 2.

}EMERGENCY_STRUCT;
```

## 5 ITabMotion Interface (CAN only)

This interface is applicable to CANopen only. Its functions are used to initiate, run and stop PVT (Position-Velocity-Time) tabulated fast motions, defining the trajectory on-the-fly as necessary. The interface contains the following groups of functions:

- **PVT Motion:**
  - **InitPVT** Prepare the drive and initiate the PVT motion.
  - **AddPVTPoint** Send the first and subsequent PVT point(s) to the drive buffer.
  - **GetDriveTimer** Retrieve the Time Stamp of the designated drive.
  - **SendCommandGroupID1** Send a command to Elmo drive nodes that have the same group ID number.
  - **SendBGSyncOnTime1** Start the PVT motion using a time-based command.
- **Emergency Handling:**
  - **SetEmergencyCallback1** Function is executed through the reception of the CANopen Emergency object.
- **Auxiliary Functions:**
  - **GetQueueSize** Retrieve the queue size to determine the number of points remaining to be sent to the drive.
  - **StopSYNC** Cancel the Sync Manager at the end of the PVT motion.

After communication via CANopen has been established with the Elmo drive (using the `IDriver::GetTabMotion` function), the user may define a PVT fast motion mode on-the-fly. The Interlude ITabMotion interface uses the transmit and receive PDO1 to send the commands and points to the drive.

- The PVT motion object is mapped to RPDO1.
- The PVT mode parameters are sent through the MP commands.
- At each SYNC (synchronization) message, a read and write pointer is sent from the drive PVT buffer, along with a Time Stamp. The pointer indicates the amount of free space available in the drive buffer queue for receiving PVT points.
- The PVT points are sent on-the-fly to the drive.
- The CANopen EMCY (emergency) object is used to handle emergencies, with an optional callback mechanism.

## 5.1 PVT Motion

In order to run a PVT motion, the user may perform these steps:

1. Prepare the drive for PVT motion and initiate the PVT mode.
2. Send the initial PVT points to the buffer.
3. Start the motion.
4. Send subsequent PVT points to the buffer.
5. Terminate the PVT mode either by sending no more points or stopping the motor.

### 5.1.1 ITabMotion::InitPVT

This function prepares the drive to operate in PVT mode. It performs the following operations:

- Prepares the drive buffer for receiving PVT trajectory points.
- Initializes the Interlude queue to receive the point data.
- Initiates the SYNC - Time Stamp procedure.
- **Sends an MO=1 command at the end, to start the motor.**

#### Syntax:

```
BOOL InitPVT(CString& sError, HWND hWnd=NULL);
```

#### Input Parameters:

HWND hWnd

The optional window handle, used to receive messages.

#### Output Parameters

CString& sError

Error string if an error occurs.

#### Returned values:

- If function is sent successfully: TRUE
- If an error occurs: FALSE

### 5.1.2 ITabMotion::AddPVTPoint

This function sends the next point of a PVT trajectory on-the-fly to a drive or drives.



#### Notes:

- The drive buffer depth must be at least 60 milliseconds.
- The drive buffer must have at least two PVT points in it in order to interpolate the motion.

#### Syntax:

```
BOOL AddPVTPoint(__int32 Pos, __int32 Vel, BYTE time, CString& sError);
```

#### Input Parameters

<code>_int32 Pos</code>	Absolute position value of next point.
<code>_int32 Vel</code>	Absolute velocity of next point.
<code>BYTE time</code>	Time interval after which the motor must reach the point position, at the point velocity.

#### Output Parameters

<code>CString&amp; sError</code>	Error string if an error occurs.
----------------------------------	----------------------------------

#### Remarks:

Sending the initial PVT points serves to implement a time delay required for running a smooth PVT trajectory. When the queue runs out of points to process, the PVT motion terminates.

#### Returned values:

- If function is sent successfully: TRUE
- If an error occurs: FALSE

### 5.1.3 ITabMotion::GetDriveTimer

Retrieves the Time Stamp of the drive designated as the Sync Master.

#### Syntax:

```
UINT GetDriveTimer ( );
```

#### Returned Values:

Real-time value of the drive timer, in microseconds.

### 5.1.4 ITabMotion:: SendCommandGroupID1

This function is identical to the ICanService::SendCommandGroupID function. Refer to [section 4.1.6](#).

### 5.1.5 ITabMotion::SendBGSyncOnTime1

This function is identical to the ICanService::SendBGSyncOnTime function. Refer to [section 4.1.7](#).

### 5.1.6 ITabMotion::GetQueueSize

This function retrieves the size of the queue in order to determine the number of PVT points waiting to be processed.

**Syntax:**

```
UINT GetQueueSize ( );
```

**Returned Values:**

Size of queue, in bytes

### 5.1.7 ITabMotion::StopSYNC

This function terminates the transmission of SYNC synchronization messages at the completion of a PVT motion.

**Syntax:**

```
BOOL StopSYNC ( )=0
```

**Returned Values:**

- If operation is successful: TRUE
- If an error occurs: FALSE

## 5.2 Emergency Handling

As with the ICanService interface, ITabMotion uses the same CANopen-based emergency message mechanism for error handling (see [section 4.2](#)).

### 5.2.1 ITabMotion::SetEmergencyCallback1

This function is identical to the ICanService::SetEmergencyCallback function. Refer to [section 4.2.1](#).

## 5.3 ITabMotion Examples

For this section, use the three sample project files that have been included with all of the Interlude setup and system files. They can be found in the Examples folder.

### 5.3.1 Sending a PVT Trajectory by File

The TestPVTfromFile.cpp file is an example of a console application sends a PVT trajectory from a text file to a single drive. It therefore assumes that the trajectory has been fully planned before the motion begins.

The application copies the data from the selected .txt file into an array of structures that include iPos, iVel and iTime variables, using the following code:

```
CArray<PVT_DATA, const PVT_DATA&> g_ArrayPVT;
...

CStdioFile file;
if(!file.Open(sPathName, CFile::modeRead))
{
    AfxMessageBox(CString(_T("Can't open file : ") + sPathName));
    return 56;
}

int iId=0;
CString str;
file.ReadString(str); // Skip title line
PVT_DATA data;
while(!feof(file.m_pStream))
{
    if(!file.ReadString(str)) continue;
    if(str.IsEmpty()) continue;
    sscanf(str, "%d %d %d %d", &iId, &data.iPos, &data.iVel, &data.iTime);
    g_ArrayPVT.Add(data);
}

file.Close();
```

The class CDrive encapsulates the initialization of the drive, including:

- Connecting the drive
- Disconnecting the drive
- Using an emergency callback function in cases of CANopen communication

The following code is used:

```
class CDriver
{
    IDriver *m_pDriver;

public:
    CDriver()
    {
        m_pDriver = IDriver::DriverConnect(Error_Callback);
        if(m_pDriver)
        {
            ICanService * pCan = m_pDriver->GetCanService();
            if(pCan)
            {
                pCan->SetEmergencyCallback(EMCY_Callback);
            }
        }
    }
}
```

```

    }
    ~CDriver()
    {
        if(m_pDriver) IDriver::DriverDisconnect(m_pDriver);
    }

    IDriver *operator->()
    {
        ASSERT(m_pDriver);
        return m_pDriver;
    }
    BOOL IsOK()
    {
        return !!m_pDriver;
    }
    IDriver * GetIDriver() { return m_pDriver;}
};

```

The object `m_spDriver` belongs to the `CDrive` class, with the `TabMotion` interface. It activates the PVT motion using the `InitiPVT()` function. The `TrajectoryThreadProc()` function uses the `AddPVTPoint()` function to add points from the array of structures to the Interlude queue. It uses the `GetQueueSize()` function to ensure that the queue size does not exceed its limits. The following code is used:

```

if(g_pTabMotion->GetQueueSize() > QUEUE_LOW_RANGE) continue;
while(g_pTabMotion->GetQueueSize() < QUEUE_HIGH_RANGE)
{
    const PVT_DATA &data = g_ArrayPVT[iCurrentRec];
    g_pTabMotion->AddPVTPoint(data.iPos, data.iVel, data.iTime,
sError);
    iCurrentRec++;
    if(iCurrentRec >= iSizeArr)
    {
        iCurrentRec = 0;
    }
}

```

### 5.3.2 Sending a PVT Trajectory on-the-fly to a Single Axis

The `ThTestPVTStream01.cpp` file is an example is a console application in which PVT update points are sent on-the-fly from a manipulator drive (functioning as master) to a follower drive (functioning as a slave). The application reads the speed (VX) and position (PX) of the master drive (motor), calculates a time using a SYNC - Time Stamp mechanism and then sends the result as a PVT object to the slave drive, using the following code.

```

g_pManipulator->SendCommand(_T("PX"), Out);
iPX = atoi(Out);
//iPX = 0;
g_pManipulator->SendCommand(_T("VX"), Out);
iVX = atoi(Out);
//iVX = 0;
iTime = g_pTabMotion01->GetDriveTimer() - AbsTime;
iTimeMs = iTime / 1000 ;
AbsTime = AbsTime + iTimeMs * 1000 ;
if(iTimeMs!=0)
{
    g_pTabMotion01->AddPVTPoint(iPX, iVX, (BYTE)iTimeMs, Out);
}

```

This application initiates communication in a manner similar to that of the previous example (section 5.3.1) using the CDrive wrapper class. The application maintains a buffer of 10 points (defined in `START_DELAY_POINTS`) and uses a 10-millisecond time segment for the PVT (defined in `TRAJECTORY_DELTA_TIME`). The main element in this application is the use of the `GetDriveTimer()` function to calculate the time of the PVT trajectory through synchronization with the global timer of the drive.

The application uses DS-301 mapping to retrieve the head and tail pointers from the drive buffer, along with the Time Stamp of the drive. It assigns one of the drives to be the Time Stamp master and then uses the internal driver of that drive for synchronizing the entire network.

### 5.3.3 Sending a PVT Trajectory on-the-fly to Two Axes

This example is similar to the previous example (section 5.3.2) except that the trajectory is now sent from the master (manipulator) to **two** slave axes (`g_pTabMotion01` and `g_pTabMotion02`). It uses the following code:

```
g_pManipulator->SendCommand(_T("PX"), Out);
iPX = atoi(Out);
//iPX = 0;
g_pManipulator->SendCommand(_T("VX"), Out);
iVX = atoi(Out);
//iVX = 0;
iTime = g_pTabMotion01->GetDriveTimer() - AbsTime;
iTimeMs = iTime / 1000 ;
AbsTime = AbsTime + iTimeMs * 1000 ;
if(iTimeMs!=0)
{
    g_pTabMotion01->AddPVTPoint(iPX, iVX, (BYTE)iTimeMs, Out);
    g_pTabMotion02->AddPVTPoint(iPX, iVX, (BYTE)iTimeMs, Out);
}
```

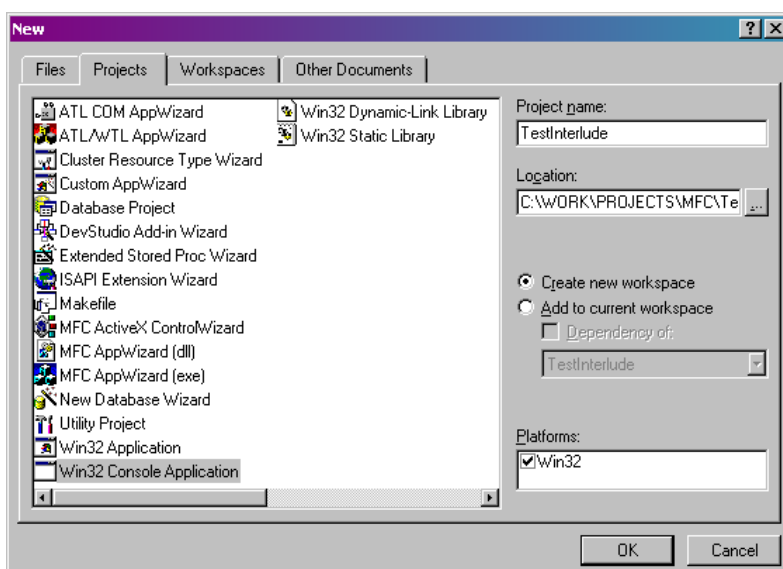
## Appendix: Getting Started Tutorial

The following tutorial is designed to take you step-by-step through the procedure of creating a user program after installing the Interlude software. It uses a very basic application that establishes communication through the Interlude to the drive, sends a VR command to retrieve the drive firmware version and subsequently disconnects the communication.

### A.1 Creating Your Project File

This procedure defines the parameters of your console-application project through the Microsoft Visual C++ software.

1. Access Visual C++ and from the main menu, select **File - New**. The New dialog box will be displayed.
2. Click the Projects tab and from the displayed list, select **Win32 Console Application**, as follows:

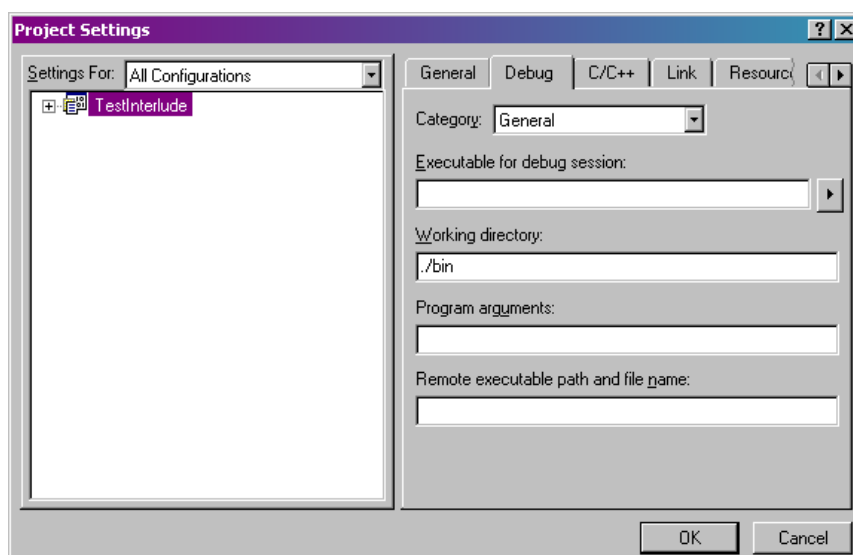


3. In the **Location** text box, browse to the location at which the project file should be saved.
4. In the **Project name** text box, enter a name for the project. This name will be subsequently used by the Win32 Console Application wizard to assign names to related files and file elements.
5. Click **OK**. The Win32 Console Application - Step 1 of 1 dialog box will be displayed with the following question: **What kind of Console Application do you want to create?** From the displayed list of options, select **An application that supports MFC**.
6. Click **Finish**. The New Project Information dialog box will be displayed, summarizing the information about the newly created skeleton project. Click **OK** to continue.

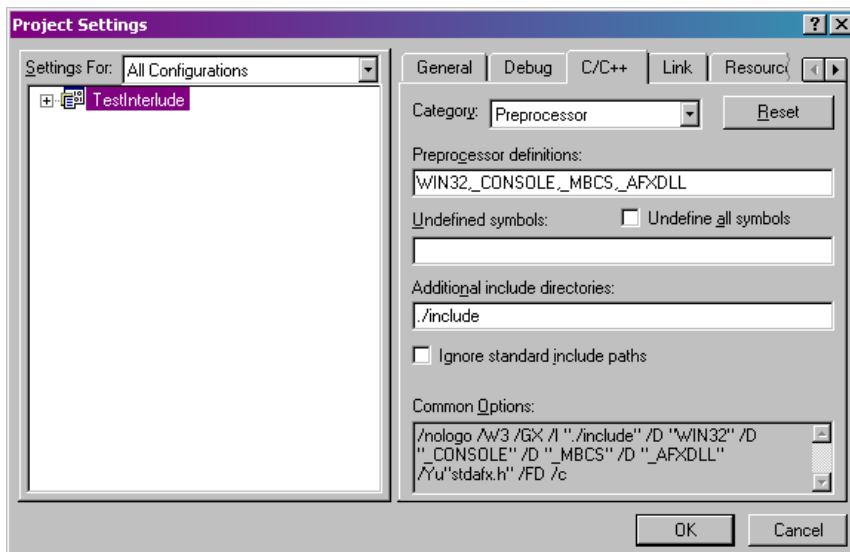
## A.2 Defining Your Project Parameters

Once you have created the skeleton project files, you need to designate the various Interlude files and settings that will be needed in your project.

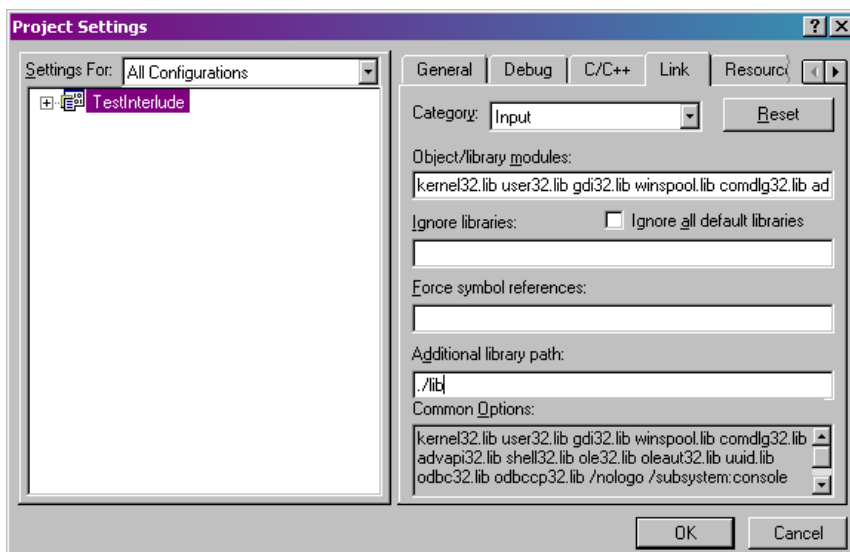
1. In your Windows Explorer, navigate to the Interlude\Interlude folder, copy the Include and Lib folders and paste them into your newly-created project folder.
2. In the same project folder, create an empty BIN folder. Then copy the ElmoComApi.dll and ElmoComApi\_d files from the Interlude\Interlude\DLL folder into the BIN folder in your project folder.
3. Return to the Visual C++ application. From the main menu, select **Project - Settings**. The Project Settings dialog box will be displayed.
4. From the **Settings For** drop-down list, select **All Configurations** and select **TestInterlude** from the displayed project list.
5. Click the Debug tab and in the **Working directory** text box, enter the path to the BIN folder in your project folder, as follows:



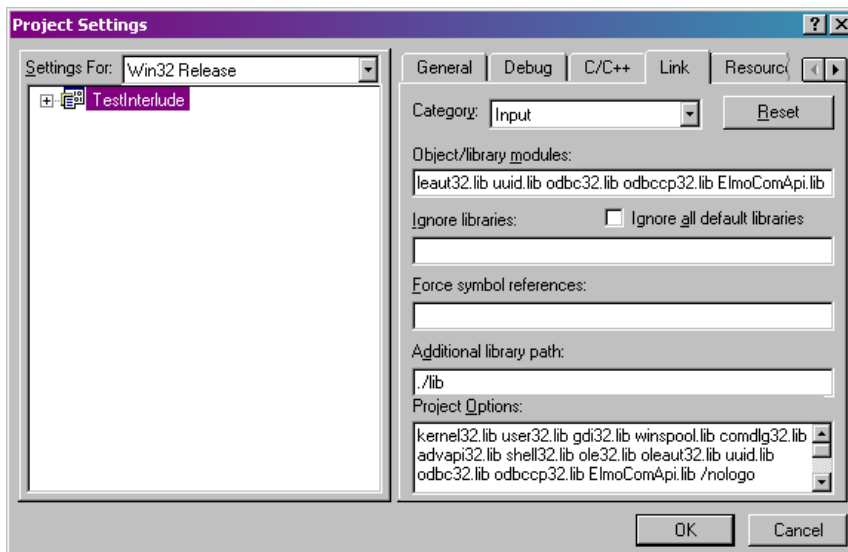
6. Now click the C/C++ tab and do the following:
  - From the **Category** drop-down list, select **Preprocessor**.
  - In the **Additional include directories** text box, enter the path to the Include folder in your project folder, as follows.



7. Click the Link tab and do the following:
  - From the **Category** drop-down list, select **Input**.
  - In the **Additional library path** text box, enter the path to the LIB folder in your project folder.



8. Now return to the **Settings For** drop-down list, select **Win32Release** and in the **Object/library modules** text box, add the ElmoComApi.lib file name to the end of the list, as follows:

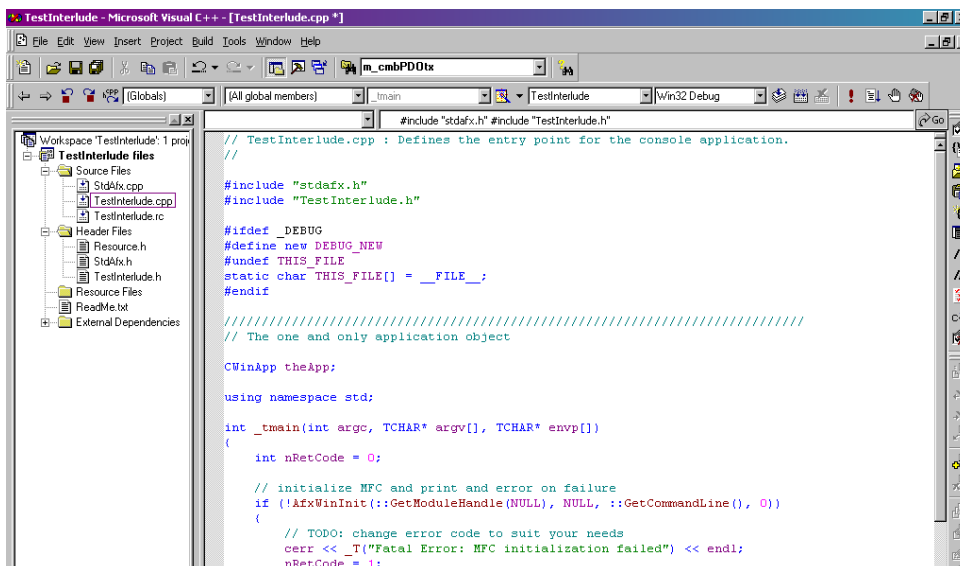


9. Return again to the **Settings For** drop-down list, select **Debug** and in the **Object/library modules** text box, add the `ElmoComApi_d.lib` file name to the end of the list.
10. Click **OK** to finalize your project definition.

### A.3 Writing the Program Code

Now you are ready to open your project file and enter the code that you require, using the Interlude interfaces.

1. Click the **File View** tab at the bottom of the left workspace view pane and from the **Source Files** folder, double-click the `TestInterlude.cpp` file to open it. You will see a small application that has been automatically prepared by the wizard using the MFC library. When run, this program simply displays the word "Hello" in the console window.



2. After the first two lines beginning with `#include`, add the following line:  
`#include "IDriver.h" //Including header file with Interlude definitions to define the Interlude data types.`

3. In the main function, find the line `// initialize MFC and print and error on failure.` This section initializes the MFC drive and error mechanism.
4. Delete the *second* section that begins with `// TO DO: code your application's behavior here.` In its place, enter the following code:

```
else
{
    IDriver *pDriver;
    pDriver = IDriver::DriverConnect();
    if(pDriver == NULL)
    {
        cerr << "Error communication" << endl;
        return 2;
    }
    CString sVersion;
    pDriver->SendCommand("VR", sVersion);
    cout << "Driver version : " << (LPCSTR)sVersion << endl;
    IDriver::DriverDisconnect(pDriver);
}
```

This code uses the Interlude IDriver interface to establish communication between the application and the designated drive (pDriver), and subsequently sends the Elmo command VR in order to retrieve the version number of the drive firmware. The information is printed in the console window. Finally, the Interlude DriverDisconnect command disconnects communication.

## A.4 Compiling and Running the Program

1. From the main menu, select **Build - Build TestInterlude.exe**. Compilation will begin and in the bottom pane of the window, messages will be received. A message of **0 error(s), 0 warning(s)** indicates successful compilation.
2. Run the program by selecting **Build - !Execute TestInterlude.exe**. The Interlude Establish Communication dialog box will be displayed for defining the active communication type. If needed, make your selection and then click **Properties**. Check the information in the CAN and Board tabs and then click **Connect** (full information is provided in [section 3.1.1](#) of this manual). The console window will display the requested drive version.