MPI Library Reference Manual for D-series Drives

MPI Library

Windows 95/98/2000/XP/7

Version 1.51
May 2014

HIWIN Mikrosystem Corp.

Table of Contents

Revision History:

Version	Date	DLL Version	Remarks
1.51	2014-05-26	1.13	Frist Release.

(This page is intentionally left blank.)

1. Preface

1.1. About the manual

This document is applicable for mega-fabs and HIWIN D-series servo drives. It describes the MPI library(s) which is (are) implemented as DLL file (mpi.dll). These libraries enable the user to access in its application on the mega-fabs controller(s). The libraries can include any application generated by Microsoft Visual C++, VB, LabView and run on Windows 95/98/2000/XP/7. The libraries make the use of MFC. The libraries enable the user to make the following:

- Set communication configuration (port number, baud rate, RS232/USB/ CAN…).
- Work with multiple communication ports. Each is connected to separate controller (or even to the same controller if it has two RS232 ports for example).
- Support multitasking. Several tasks may access the controller via the DLL interface with minimum latency.
- Error handling. When a communication error happens, they try again to send the message till success, or arrive 'try again' parameter limit.
- Read/write to any variable/array in the controller. Support 64-bit variables.
- Run PDL functions.
- Set/clear/toggle/read states.
- Monitor events from the controller that are sent by the `printl` PDL command.
- Support data collection (record).
- Internet connections between two or more applications that use mpi.

The mpi library comprises of four files (**mpi.lib, mpi.dll canlib32.dll, mpint.h**).

The “PDL.coc” describes PDL language and DCE database.

Here, all exported functions of mpi.dll are described.

2. Initialization

This chapter describes the exported functions that should be called when an application is started and terminated.

2.1. openDCE

Prototype:

```
MDCE *openDCE(char *pwdin, char *cnfname);
```

Parameters:

pwdin – a string of path that contains the configuration file.

cnfname – a string of the name of configuration file.

Return:

- A pointer points to a communication session object. This pointer may be supplied to all other functions as the parameter **pcom**.

Notes:

- You can open several communication session objects (each one will be set to different port using `setComPar` function). Need to hold for each one the return **MDCE** pointer, which will be communication object identifier. All other functions require this parameter as the last parameter.
- If you work with only one communication session, you may ignore the return **MDCE** pointer, and supply to all other function NULL for **pcom** parameter. Set **pcom** = NULL in all other function commands, then access the default communication object which is the last opened by `openDCE`.
- **Pwdin** and **cnfname** may be set to NULL. In this case, the directory will be defined by the environment variable **DCE_DB**, which is defined in the registry (usually equal to: \HIWIN\dce\) and the configuration file will be "system.dce". When working with multiple communication objects, which are connected to different controllers, you should give at list different configuration files to define working directories for different controllers. For more detail about configuration file, see "PDL.coc".
- After calling `openDCE`, you must call `setComPar` to define communication port parameters. `openDCE` should be called only once for each communication object.
- To retrieve the configuration file of full path name, call `getDceCnfFname`.
- To close the communication object (probably when before an application terminated), call `deteteDCE`.

2.2. deleteDCE

Prototype:

```
void deleteDCE(MDCE *pocm, int closeMFC=FALSE);
```

Parameters:

pcom – pointer to a communication object.

closeMFC – if TRUE, it cleans up the MFC internal thread. The internal MFC threads are active only if you called `setMFCmode` in begin of application. In most cases, it should be FALSE, because the internal thread is cleaned up anyway when DLL is closed. However, in some application, setting it to TRUE prevents an application error message. This parameter may be TRUE only for the last session call of `deleteDCE`, because the MFC internal thread is common to all sessions. If you didn't call `setMFCmode`, this parameter is not relevant.

Call this function when no more communication session is needed (probably before application terminate).

2.3. ver

Prototype:

```
char *ver(int print);
```

Parameter:

print – if nonzero, mpi.dll version is printed on a message box.

Return:

- Return pointer to a string that contains DLL version information.
- It is recommended to allow your application to get DLL version information as this DLL may be updated from time to time.

2.4. disErrMsgBox

Prototype:

```
void disErrMsgBox(int dis);
```

Parameter:

dis – if 1, disable message box for communication error; if 0, enable message box.

Call this function with **dis** = 1 to disable any message box due to communication error. It can be called any time to disable/enable message boxes. It may be called before `setComPar` to disable message box when doing first time database verification between controller and PC. It does not disable the version comparison window in case of version confliction found. This function affects all opened sessions.

2.5. disAllErrMsgBox

Prototype:

```
void disAllErrMsgBox(int dis);
```

Parameter:

dis – if 1, disable message box for communication error; if 0, enable message box.

Call this function with **dis** = 1 to disable any message box due to communication error. Also, it disables the version comparison window to popup when the version confliction is detected**.** It can be called any time to disable/enable message boxes. It may be called before `setComPar` to disable message box when doing first time database verification between controller and PC. This function affects all opened sessions.

The different of `disAllErrMsgBox` from `disErrMsgBox` is that this function also affects the compare version window.

2.6. getFwErr

Prototype:

```
int getFwErr(MDCE *pcom=NULL);
```

Parameter:

pcom – pointer to communication object.

Return:

- Return nonzero value if version confliction between PC and controller firmware programs is detected. Version verification is done when communication first time setup, and after each call to `compareFw`.

2.7. getVerErrCodeSl

Prototype:

```
unsigned int gerVerErrCodeSl(int slave=0, MDCE *pcom=NULL);
```

Parameters:

slave – slave number.

pcom – pointer to communication object.

Returns:

- Return detailed version error code per slave.
- Each bit represents an error of one file database:
  0 **–** no error.
  1 **–** fail to compare version.
  2 **–** mismatch version file-controller.
- The 2 bits interposition is as follow:
  Bit 0, 1 – BOOT firmware file *.edb.
  Bit 2, 3 – BOOT altera firmware file *.etf.
  Bit 4, 5 – MDP firmware file <appl name>.edb.
  Bit 6, 7 – MDP variable list file <appl name>.vrs.
  Bit 8, 9 – PDL firmware file < name>.edb.
  Bit 10, 11 – PDL MDP variable list file <user_$>.vrs.
  Bit 12, 13 – PDL MDP function list file <user_$>.lbl.
  Bit 14, 15 – PDL MDP message list file <user_$>.msg.
  Bit 16, 17 – Core altera firmware file *.etf.
  Bit 17, 19 – Application altera firmware file *.etf.

2.8. compareFw

Prototype:

```
int compareFw(int show=0, MDCE *pcom=NULL);
```

Parameters:

show – if 1, open the message version window even if all firmwares are matched.

pcom – pointer to communication object.

Perform version comparison between PC and controller firmware programs. If any version conflicts, the message window version will pop up (unless call before `disAllErrMsgBox`(1)), and calling `getFwrr` will return nonzero. Version comparison is always done after communication first setup.

2.9. setMFCmode

Prototype:

```
void setMFCmode();
```

If you call this function, you can call the following functions from threads that are not the main application threads:

RunKmi, KillKmi, showcomstatus, OpenRecord.

Call `setMFCmode` before calling any other functions. This function should be call only once.

If you access these functions always from the main thread, you do not need to call `setMFCmode`.

2.10. resetController

Prototype:

```
void resetController(int slave=0, MDCE *pcom=NULL);
```

Parameters:

slave – slave number.

pcom – pointer to communication object.

Reset the controller. Calling this function is similar to make hardware reset to controller. It will go to boot mode first, and then back to normal mode.

(This page is intentionally left blank.)

3. Communication

This chapter describes all functions related to communication setting on open and close, view communication setting, database and so on.

3.1. setComPar

Set communication parameters.

Prototype:

```
int setComPar(int port, int baudrate, int mode,
              int trid, int rcid, int canbaudrate,
              int msgStand, int canpipelevel,
              int timeout, int locktime,
              int iternum, MDCE *pcom=NULL);
```

Parameters:

port – port number 1, 2, 3, 4,…..

baudrate – code of baud rate in RS485/RS232/USB mode (0,1).

0 => 1200; 1 => 2400; 2 => 4800; 3 => 9600; 4 => 19200; 5 => 38400;

6 => 56000; 7 => 57600; 8 => 115200; 9 => 128000; 11 => 230400;

12 => 460800; 13 => 921600.

mode – 0 => RS485; 1 => RS232/USB; 2 => CAN; 3 => Ethernet;

4 => EtherCAT mega-ulink (see below about EtherCAT mega-ulink).

trid – relevant in CAN mode (**mode** = 2) to set CAN transmit ID.

rcid – relevant in CAN mode (**mode** = 2) to set CAN receive ID.

canbaudrate – relevant in CAN mode (**mode** = 2) to set CAN baud rate.

0 => 50K; 1 => 62K; 2 => 83K; 3 => 100K; 4 => 125K; 5 => 250K;

6 => 500K; 7 => 1M.

msgStand – relevant in CAN mode (**mode** = 2).

0 => message extended; 1 => message standard.

canpipelevel – relevant in CAN mode (**mode** = 2) to set pipe level (1…10).

timeout – time out in msec to wait for answer message sent to controller (recommended 100).

locktime – time in msec, in which communication is locked due to error (recommended 200).

iternum – if communication error happens, try again to send message till **iternum** time. After it fails to get the acknowledgement of message sent **iternum** times, it returns error to calling process (recommended 6).

pcom – pointer to communication object.

Return:

- If success to open port, return 0.
- If fail, return operating system error code.
- 2 – port not exist.

- 5 – port is busy by other application.

Note:

- This function must be called after `openDCE`. Then, it can be called any time to change communication setting.

3.1.1. Connect to EtherCAT mega-ulink

To connect to EtherCAT mega-ulink, call `SetComPar` with **mode** = 4. Also, must call `setMFCmode`.

Example:

```
setMFCmode();
MDCE *pcom=openDCE("c:\\HIWIN\\dce\\", NULL);
setComPar(0,0,4, 0,0,0,0,0,50,0,8,pcom);
```

Notes:

- Must call `setMFCmode` before `openDCE`.
- In `openDCE` function, only the first argument is important. It defines the root directory of the firmware's applications used by slaves. The second one, which defines the *.dce configuration file, can be NULL as in EtherCAT mega-ulink mode. It always uses "ethercat.dce" file, which is updated dynamically according to slave detected on the network.
- In `setComPar`, in this example, all the non-relevant parameters are 0 and the 3th parameter should be 4 (EtherCAT mega-ulink mode).
- Currently, the network devices used in PC need to be selected manually. Call `showcomststus` function, and press the "**SetUp**" button on the com status window. Then, click "**EtherCat…**" button and select the desired network device (see picture below). After press "**Apply**", this setting is saved (to commpi.sav in the application working directory) and restored each time when connecting to EtherCAT mega-ulink.

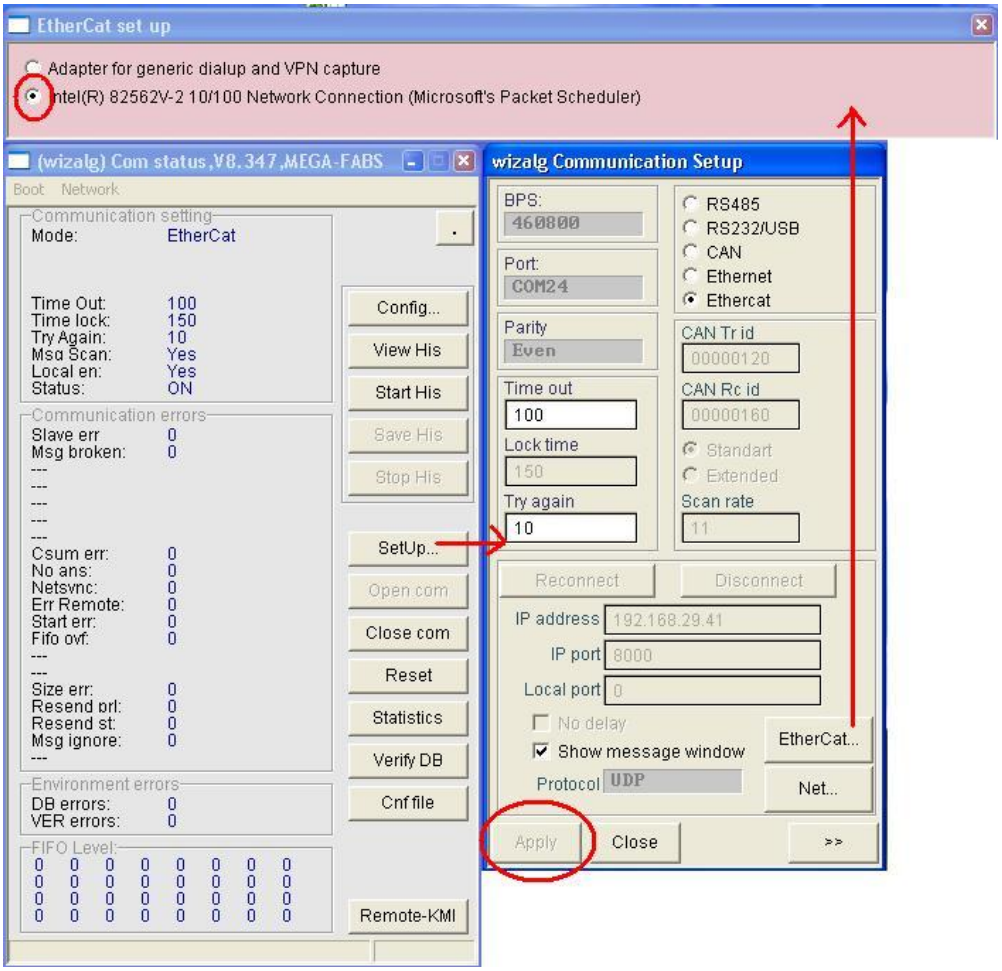
EtherCat set up
Adapter for generic dialup and VPN capture
ntel(R) 82562V-2 10/100 Network Connection (Microsoft's Packet Scheduler)
(wizalg) Com status,V8.347,MEGA-FABS
Boot Network
Communication setting
Mode: EtherCat
Time Out: 100
Time lock: 150
Try Again: 10
Msg Scan: Yes
Local en: Yes
Status: ON
Communication errors
Slave err 0
Msg broken: 0
Csum err: 0
No ans: 0
Netsync: 0
Err Remote: 0
Start err: 0
Fifo ovf: 0
Size err: 0
Resend orl: 0
Resend st: 0
Msg ignore: 0
Environment errors
DB errors: 0
VER errors: 0
FIFO Level:
Config...
View His
Start His
Save His
Stop His
SetUp...
Open com
Close com
Reset
Statistics
Verify DB
Cnf file
Remote-KMI
wizalg Communication Setup
BPS: 460800
Port: COM24
Parity Even
Time out 100
Lock time 150
Try again 10
RS485
RS232/USB
CAN
Ethernet
Ethercat
CAN Tr id 00000120
CAN Rc id 00000160
Standart
Extended
Scan rate 11
Reconnect
Disconnect
IP address 192.168.29.41
IP port 8000
Local port 0
No delay
Show message window
Protocol UDP
EtherCat...
Net...
Apply
Close
>>

Fig. 3-1

3.2. setComParEx

Set communication parameters.

Prototype:

```
int setComParEx(int port, int baudrate, int mode,
                int trid, int rcid, int canbaudrate,
                int msgStand, int canpipelevel,
                int timeout, int locktime,
                int iternum, char *ipstr, int ipport,
                int nodelay, MDCE *pcom=NULL);
```

Parameters:

Similar to `setComPar` but have 3 other parameters to support Ethernet setting.

ipstr – string that contains IP address.

ipport – port number.

nodelay – if TRUE => disables the Nagle algorithm for sending coalescing.

Return:

- If success to open port, return 0.
- If fail, return operating system error code.
- 2 – port not exist,
- 5 – port is busy by other application (as in `seComPar`).
- In case that fail to connect in Ethernet, it returns **157.**

To set Ethernet, you need to set **mode** = **3**; otherwise the Ethernet parameters are not relevant.

Example:

```
setComParEx(0,0,3,0,0,0,0,0,150,200,10, "192.168.100.180", 10001,
0, NULL);
```

If the communication setting is for Ethernet connection, use `ConnectEthernet` function.

3.3. ConnectEthernet

Set communication for Ethernet connection. Use this function instead of `setComParEx`/ `setComPar`, in case that it needs to connect to controller via Ethernet.

Prototype:

```
int ConnectEthernet(char *ipstr, int ipport,
                    int flag, int showMsgWin,
                    int timeout, int locktime,
                    int iternum, MDCE *pcom=NULL);
```

Parameters:

ipstr – string that contains IP address.

ipport – port number.

flag – relevant in controllers when using HIWIN Ethernet drive.

0 => connection will fail, if the controller socket (as defined by the **ipport**) is already connected to other application.

1 => the previous connection of this port (if any) is disconnected. After that, try to connect this application.

showMsgWin –

1 => open the message window. It is useful for debugging purpose to view all messages through the DLL print while connection executing.

timeout – time out in msec to wait for answer the message sent to controller (recommended 100).

locktime – time in msec, in which communication is locked due to error (recommended 200).

iternum – if communication error happens, try again to send the message till **iternum** time. After it fails to get the acknowledgement of message sent **iternum** times, it returns error to the calling process (recommended 6).

pcom – pointer to communication object.

Return:

- If connect successfully, return 0.
- If fail, return 157.

When the application is terminated before closing the socket connection to controller, it may fail to reconnect again to controller socket. This is because the controller socket didn't close its previous connection.

In this case, you may set **flag = 1**. First, it will send a disconnection message to the controller to close any previous connection and free the socket (in the controller) for new connection. After that, it will do the new connection.

Note:

- Setting flag = 1 will disconnect any connection of the specified socket, if any which may be to other application and/or other PC.

3.4. getComPar

Retrieve communication parameters.

Prototype:

```
int getComPar(int *pport, int *pbaudrate, int *pmode,
              int *ptrid, int *prcid, int *pcanbaudrate,
              int *pmsgStand, int *pcanpipelevel,
              int *ptimeout, int *plocktime,
              int *piternum, MDCE *pcom=NULL);
```

Parameters:

These arguments are pointers, which are the same parameters as those in function `setComPar`. This function retrieves the parameters set by `setComPar`.

Return:

- 0 – ok.
- -1 – no communication object.

3.5. getComParEx

Retrieve communication parameters.

Prototype:

```
int getComPar(int *pport, int *pbaudrate, int *pmode,
              int *ptrid, int *prcid, int *pcanbaudrate,
              int *pmsgStand, int *pcanpipelevel,
              int *ptimeout, int *plocktime,
              int *piternum, char *ipstr, int *pipport,
              int *pnodelay, MDCE *pcom=NULL);
```

Parameters:

These arguments are pointers, which are the same parameters as those in function `setComParEx`. This function retrieves the parameters set by `setComParEx`.

Return:

- 0 – ok.
- -1 – no communication object.

3.6. getDceCnfFname

Prototype:

```
int getDceCnfFname(char *str, MDCE *pcom=NULL);
```

Parameters:

str – return the full-path file name of the configuration file.

pcom – pointer to communication object.

Str and **pcom** should point to string buffer with enough size according to the length of the strings returned (probably 200 is good).

Return:

- 0 – ok.
- -1 – no communication object.

3.7. getSlaveFname

Prototype:

```
int getSlaveFname(char *str, int slave=0, MDCE *pcom=NULL);
```

Parameters:

str – return slave name, which is the working directory containing all database of the controller.

slave – slave number (in case of single controller it will be 0).

pcom – pointer to communication object.

Return:

- 0 – ok.
- -1 – no communication object.

3.8. colsePort

Prototype:

```
void closePort(MDCE *pcom=NULL);
```

Parameter:

pcom – pointer to communication object.

This function closes the port. It does not delete the communication object (as do `deleteDCE`**).**

3.9. openPort

Prototype:

```
int openPort (MDCE *pcom=NULL);
```

Parameter:

pcom – pointer to communication object.

This function reopens the port as specified in the last `setComPar`.

Return:

- If open port successfully, return 0.
- If fail, return operating system error code.
- 2 – port does not exist.
- 5 – port is occupied by other application.
- 157 –fail to connect via Ethernet.

Notes:

- You may call `openPort` instead of calling `setComPar/setComParEx`. In this case the last communication setting will be applied.
- You can change the communication setting via the communication status dialog (see `showcomstatus`). The last communication setting is saved in file “com.sav” at the working directory.

3.10. openRemote

Prototype:

```
int openRemote (MDCE *pcom=NULL);
```

Parameter:

pcom – pointer to communication object.

This function connects via TCP/IP to other application which should be in listen mode.

Return:

- Return -1 if `openDce` was not called yet; otherwise, return 0.

The network parameters (port, URL ...) remain the same as the last setup. Setting is done from the network menu item (in communication status window and main KMI window).

3.11. showcomstatus

Prototype:

```
void showcomstatus(MDCE *pcom=NULL);
```

Parameter:

pcom – pointer to communication object.

This function opens a dialog box, which shows information on the communication settings and statistics. Also from this dialog box, the user may perform several functions, e.g. close/open com port, run KMI tool, open boot dialog boxes (PDL, MDP, and ALTERA), access the internet/network, etc. This window can remain active or be minimized while application running.

This function should be called from the main application thread; otherwise it may not respond to user events on it. You may create on your GUI a button/menu dedicated for operating this function. But if you call `setMFCmode` before, you may not call `showcomstatus` from the main application thread.

3.12. RunKmi

Prototype:

```
void RunKmi(MDCE *pcom=NULL);
```

Parameter:

pcom – pointer to communication object.

Run KMI tool in remote mode.

KMI works in remote mode (client), while the application works in local mode (server). The network setting of the KMI should be URL: `localhost`. And the socket port of KMI and application should be set to the same number. Network setting is done from the network menu item (in communication status window and main KMI window). In the application, chose local; while in the KMI, choose remote. On the terminal window that will be opened, right-click on mouse and choose 'Set network'. If the KMI is already opened, calling this function puts the main KMI window on top.

The KMI tool may be opened for debugging. So, you can watch/modify any variable/state or run PDL function via this tool while your application running and accessing the controller's variables/states/PDL functions.

See also `RunAppl`.

Note:

- If you have several sessions, you should run KMI only for one session. If you want to run KMI for other session, first call `KillKmi` for the current session to run KMI with **closenet** = 1. After that, call `RunKmi` for the other session.

Example:

```
MDCE *pdce1=openDCE("c:\HIWIN\dce\","contr1.dce"); //session 1.
pdce1->setComPar……
MDCE *pdce2=openDCE("c:\HIWIN\dce\","contr2.dce"); //session 2.
pdce2->setComPar(……
……
RunKmi(pdce1); //run KMI for session1.
……
KillKmi(pdce1,1); //close KMI (if still running) and disable session 1 to be a server.
RunKmi(pdce2); //run KMI for session2.
```

3.13. KillKmi

Prototype:

```
void KillKmi(MDCE *pcom=NULL, int closenet);
```

Parameters:

pcom – pointer to communication object.

closenet – if 1, it also closes the local net server.

Kill KMI opened from `RunKmi`. KMI application may be closed also by simply closing its main window. However, calling `KillKmi` with **closenet = 1** ensures to close the server listen mode.

See also example in `RunKmi` function.

3.14. RunAppl

Prototype:

```
int RunAppl(char *applname, MDCE *pcom=NULL);
```

Parameters:

applname – application name to run.

pcom – pointer to communication object.

This function is an extension of `RunKmi`. However, instead of running only KMI, it runs any special application in remote mode, such as wizalg, franal, intrptest axistest, tune and also KMI.

Return:

- Application id. This value may use in `KillAppl` to terminate the application running in remote.

Note:

- If you have several sessions, you should run application in remote only for one session. If you want to run application for other session, first call `KillAppl` for the current session with **closenet** = 1. After that, call `RunAppl` for the other session.

Exmple:

```
MDCE *pdce1=openDCE("c:\HIWIN\dce\","contr1.dce"); //session 1.
pdce1->setComPar(……
MDCE *pdce2=openDCE("c:\HIWIN\dce\","contr2.dce"); //session 2.
pdce2->setComPar(……
……
int id1=RunAppl("wizalg", pdce1); //run wizalg for session1.
……
KillAppl(id1, pdce1, 1); //close wizalg (if still running) and disable session 1 to be a server.
int id2= RunAppl("intrptest", pdce2); //run intrptest for session2.
```

3.15. KillAppl

Prototype:

```
void KillAppl(int applid, MDCE *pcom=NULL, int closenet);
```

Parameters:

applid – return from the `RunAppl`.

pcom – pointer to communication object.

closenet – if 1, it also closes the local net server.

Kill application opened from `RunAppl`. Applications may be closed also by simply closing its main window. However, calling `KillAppl` with **closenet = 1** ensures to close the server listen mode.

See also example in `RunAppl` function.

3.16. loadDceSw

Prototype:

```
int loadDceSw(int pdlComp, MDCE *pcom=NULL);
```

Parameters:

pdlComp – if not 0, it will do PDL compilation if needed.

pcom – pointer to communication object.

Return:

- If success, return 0; otherwise, return the number of errors happened in the loading process.
- Call this function to update controller with all relevant software (MDP, PDL ALtera …).

3.17. loadFirmName

Prototype:

```
int loadFirmName(char *name, int slave, MDCE *pcom);
```

Parameters:

name – firmware name to be saved. If NULL, take current application name.

slave – slave number.

pcom – pointer to communication object.

Return:

- If success, returns 0; otherwise, return nonzero.

3.18. updateDataBase

Prototype:

```
void updateDataBase(MDCE *pcom);
```

Parameter:

pcom – pointer to communication object.

Update the DLL internal database. For example, if change the firmware MDP/PDL and the variable list has been changed, call this function to refresh internal DLL variable/ functions list instead of closing and opening the application.

3.19. History logger

These functions support the operation of internal logger that stores every message transferred between PC and controller. The data cab is viewed with the dcehis.exe tool. The data is stored in circular buffer (if not file mode) with maximum size defined in the `HisStart` function. In file mode, the data is saved to disk. This allows more data to be saved.

3.19.1. HisStart

Prototype:

```
int HisStart(char *logFileName, int size, int stepToSave,
             int fileMode, MDCE *pcom);
```

Parameters:

logFileName – file name to save log data.

size – maximum number of communication messages to be stored.

stepToSave – relevant in **fileMode** = 1. Store every **stepToSave** event to disk.

fileMode – if 0, data is saved to memory cyclically (that is the log file will show the last up to size events). Only when stop the data is saved to file.

If `fileMode` is nonzero, data is saved to file on the fly each **stepToSave** event. When the number of events goes over size, it stops to save to log file.

pcom – pointer to communication object.

Return:

- 0 – success.
- 1 – already active.
- -1 – fail to allocate memory.

Call this function to start the logger process. Call `HisStop` or `HisSave` to save the result. Call `HisView` to see the result.

3.19.2. HisStop

Prototype:

```
void HisStop(MDCE *pcom);
```

Parameter:

pcom – pointer to communication object.

Call this function to stop the logger process. It also saves the data collected until now to

file “dce.log”.

3.19.3. HisSave

Prototype:

```
void HisSave(MDCE *pcom);
```

Parameter:

pcom – pointer to communication object.

Save the data collected until now to file “dce.log”, and continue the logger process.

3.19.4. HisView

Prototype:

```
void HisView(MDCE *pcom);
```

Parameter:

pcom – pointer to communication object.

Run the application “dcehis.exe” that opens the last saved file “dce.log” and shows its content.

4. Recived Events

4.1. waitOnMsgP

This function receives PDL `printl` commands sent from the PDL program running on the controller. The PDL `printl` command has the following syntax

printl/mode1/mode2 ("…String…", var1, …varN);

where `mode2` is optional (if not specified, it is 0), `mode1` specifies the color of the string and other attributes, and **var1**…**varN** are optional controller variables. For more details, see the **PDL.coc** document.

Prototype:

```
int waitOnMsgP(int *pcode, char *pmsg=NULL, int *pcolor=NULL,
               int *pparam=NULL, int *pnumovrflw=NULL,
               int timeout=INFINITE, MDCE *pcom=NULL);
```

Parameters:

pcode – will be stored with the `mode2` value.

pmsg – store the string after processing. Should point to string with at least 200 chars.

pcolor – the color code as specified in `mode1` is converted to value **COLORREF** type.

pparam – pointer to array. The first item will be written with number of parameters, and the next items will be stored with the 32 bit value of parameters. If parameter is a float type, it should be interrupted as float and not as integer.

Note that, the parameters are printed to the string (**pmsg**) according to the `printl` pressings codes (%g %ld …).

pnumovrflw – store the number of `printl` message lost. If, for example, no thread is listening to `waitOnMsgP` and a lot of `printl` message comes, overflow may happen. When return from `waitOnMsgP`, the overflow counter is reset.

Note that, if overflow happens, the attributes of the received message (in **pcode**, **pmsg**, **pcolor**, **pparam**) are valid.

timeout – if infinite (**timeout** = -1), the function will not return till any PDL message is sent (or `ReleaseWaitMessage` is called by other thread). If **timeout** = 0, the function returns immediately with return code 0 when message available, and -2 when no message available. If it is specified to other positive value, `waitOnMsgP` will be released after this time in msec (if no message will be sent meanwhile, or `ReleaseWaitMessage` is called) with returning code 5 (time out).

pcom – pointer to the communication object.

Return:

- 0 – ok.
- -1 – communication object not valid.
- -2 – no message available (this may happen when call `ReleaseWaitMessage`, or **timeout** = 0**),** or more than one tread call this function.
- 5 – time out.

If, for example, the PDL program executes `printl` like this: (assume *x_enc_pos* = 2000, *x_p_p_g* = 0.01).

```
printl/103/00000044("x axis: enc_pos=%ld , pos loop gain=%g",
x_enc_pos, x_p_p_g);
```

Then, the thread, which listens on `waitOnMsgP`, will return with code 0 (ok) and the parameters will be filled as follow:

`*pcode=0x44.`

`pmsg="x axis: enc_pos=2000, pos loop gain =0.01"`

`*pcolor= 0x02ff0000;` (code 3 in `mode1` converted to blue color)

`pparm[0]=2; pparm[1]=2000; *((float *)&pparam[2])= 0.01;`

`*pnumovrflw=0` (assume no overflow)

Notes:

- If one of pointers is NULL, it is ignored. For example, if you don't interest with the `mode2` value, you may set **pcode** = NULL.
- The `waitOnMsgP` should not be called by the main thread application with **timeout** > 0 or infinite, because it will block the GUI. In this case, you may use a polling sequence, in which you call it each specified time or in loop. When use polling sequence, set **timeout** = 0 and test the return code. If it is -2, no message is valuable. If 0, message is available and you may process it. Polling mode is useful for application with only one thread. If multi-thread is available, it is recommended to create special thread to listen on this function with **timeout** = INFINITE.
- `waitOnMsgP` should be called only by one thread at a time; otherwise, it may return -2 and messages may be lost.
- The `mode2` (= ***pcode**) is a general purpose parameter, which may use by application to identify the specific `printl` message. The application may also use the **ppram** values to decode the message. Note that, **pparam** contains variable values form the controller, which can be dynamically changed (so the same `printl` command may send different values in **pparm** when executed in different times). However, the `mode2` (= ***pcode**) and `mode1` (=> color) parameters are constants and defined once when PDL program is compiled.
- Overflow situation may happen when thread handles messages too slowly with compared to the rate that they are sent from the controller. The FIFO size in the mpi.dll is 200.

Example of calling waitOnMsgP

This function runs by thread in loop (till variable **endMsgTread** is set) which listens to `waitOnMsgP`.

```
int endMsgTread=0;
Uint msgThrd(LPVOID pPar)
```

```
{
    void *pcom=pPar;  //pointer to communication object
    int st, code, c, numOvrFlow,  param[10], numOvrFlow;
    char msg[200], str[200];
    while(!endMsgTread) {
        st = waitOnMsgP(&code, msg, &c, param, &numOvrFlow, INFINITE,
pcom);
        switch(st){
        case 0: //ok
            if(numOvrFlow!=0){
                sprintf(str,"########## %ld Message lost",
numOvrFlow);
            }
            strcpy(str,msg);
            break;
        case -1:
            sprintf(str,"####### No communication object");
            break;
        case -2:
            break;  //do nothing
        case 5:
            sprintf(str,"####### Time out");
            break;
        default: //for furthers returns code
            sprintf(str,"Message return error %ld",st);
            break;
        }
// Here should be coded that send str to any visual object (console, terminal
window, edit box….)
// also may send/process other parameters return from waitOnMsgP
    }
return(0);
}
```

4.2. releaseWaitMessage

This function releases the thread waiting on `waitOnMsgP`. It will probably call before application terminated and before calling `deleteDCE`.

Prototype:

```
void releaseWaitMessage(MDCE *pcom);
```

Example:

To release the thread in the previous example that run `msgThrd` function do:

```
endMsgTread = 1;   // to exit the while loop
releaseWaitMessage(pcom);  // to release waitOnMsgP
……
```

When using polling sequence in which the parameter timeout in `waitOnMsgP` is zero, it is no need to call this function.

4.3. insertEvent

Instead of monitoring all message of `printl` PDL command (using `waitOnMsgP`); you can wait for only part of them specified by event code (`mode2` value of the `printl` command) and ID (the first parameter value of the `printl` command). You supply handle of event that will be signaled when a `printl` message with the specific code and ID event received. The code is specified by the function `setEventCode` (see below).

Prototype:

```
int insertEvent (int id, HANDLE h, MDCE *pcom);
```

Parameters:

id – event ID. Only values in the range 0…199 are acceptable. This is the table index that stores all events in the DLL.

h – handle to event created by `CreateEvent` win32 function.

pcom – pointer to the communication object.

Return:

- 0 – ok.
- -1 – not called `openDce` yet.
- -2 – code out of range.

Notes:

- The event will be signaled if PDL will run `printl` with that:

  (1) code (`mode2` field) equals to the initialization by the function `setEventCode` (see below).

  (2) mumber of parameters is one or more.

  (3) the value of the first parameter is equal to the **id**.
- You may call `closeEvent` to close the event handle (by `CloseHandle`), which also puts NULL in the internal table that stores all event handles. You may call `insertEvent` with **h**=NULL to clear event from the table entry without close it.
- When calling `deleteDce`, all events in the table are closed.
- Up to 200 events can be inserted to the DLL internal table per communication session. And the application may wait on up to 200 simultaneous events.
- No need to insert the same event handle each time you wait for the same event ID.
- After a thread waiting to this event is released, it may call `getLastEventData` function to retrieve the `printl` information, such as the string and the parameters.

4.4. closeEvent

Prototype:

```
int closeEvent (int id, MDCE *pcom);
```

Close the event handle in the table entry specified by **id** and put there NULL.

Return:

- 0 – ok.
- -1 – not called `openDce` yet.
- -2 – code out of range.
- -3 – fail to close handle.

4.5. getEvent

Prototype:

```
int getEvent (int id, MDCE *pcom);
```

Return:

- -1 – not called `openDce` yet,
- -2 – code out of range.
- Otherwise, the event handle is returned (as int type).

4.6. setEventCode

Specified the code which is common to all `printl` PDL commands that are selected to trigger events to the application. This function may be called only once. If not called, the default is **code** = 1.

Actually, this code may be seemed as a 'filter" of all PDL `printl` commands that allow triggering event to application. For example, the PDL program may contain a lot of `printl` commands that are used for debugging by issue messages with parameters to KMI/wizalg and even for the application using this DLL (which also listens on the `WaitOnMsgP`) to catch all `printl` commands. So this code detects `printl` commands that trigger an event.

Note:

- The code specified in the PDL command cannot be changed dynamically (it is constant and defined in PDL compilation time), while the **id** value, which is the first parameter value in the PDL `printl` command, can be changed in run time.
- For example, the **id** may be specified an axis ID and PDL code including the `printl` implementation can be implemented in one procedure for all axes in the controller, where axis ID variable per axis will be the first parameter in the `printl`.

Prototype:

```
int setEventCode(int code MDCE *pcom)
```

Parameters:

code – code (correspond to `mode2` field in the PDL command).

pcom – pointer to the communication object

Return value:

- 0 – ok.
- -1 – not called `openDce` yet.

4.7. getLastEventData

This function retrieves the string and parameters of a specific event code. It may be called after thread is released by the event previously registered to dll with `insertEvent`.

Prototype:

```
int getLastEventData(int id, char *pmsg,  int *pparam, MDCE *pcom)
```

Parameters:

id – this number should be corresponding to the value of the first parameter. Only values in the range 0…199 are acceptable.

pmsg – store the string after processing. Should point to string with at list 200 characters.

pparam – pointer to array. The first item will be written with number of parameters, and the next items will be stored with the 32 bit value of the parameters. If the parameter is a float type, it should be interrupted as float and not as integer.

Note that, the parameters are printed to the string (**pmsg**) according to the `printl` pressings codes (%g %ld …). The first parameter value (**pparam**[1]) will be the **id** value.

pcom – pointer to the communication object.

Return value:

- 0 – ok.
- -1 – not called `openDce` yet.
- -2 – code out of range.

If the pointers **pmsg and pparam** are NULL, they are ignored. So, if no need, the information of one of them puts the pointer NULL.

Example:

```
char reportStr[200];
int code=0x00000055;
int id=2;
HANDLE hevent=CreateEvent(NULL,FALSE,FALSE,NULL);
int timeOut=10000;
setEventCode(code,pcom);

int error=insertEvent(id,hevent,pcom);
if(error){
    sprintf(reportStr,"insertEvent error = %ld",error);
```

```
        printout(reportStr);
    }
    else{
      error=WaitForSingleObject(hevent,timeOut);
      if(error==WAIT_TIMEOUT){
           sprintf(reportStr,"Time out");
           printout(reportStr);
      }
      else{
        int param[10];
        char message[200];
     error=getLastEventData(id, message,  param, pcom);
        if(error){
          sprintf(reportStr,"get Data error = %ld",error);
          printout(reportStr);
        }
        else{
          sprintf(reportStr,"arrived message of id=%ld ",id);
          printout(reportStr);
          printout(message); //print message of printl PDL command
          int n;
          int numPar=min(param[0],8);
          for(n=0;n<numPar;n++){ //print parameters
             sprintf(reportStr,"%ld %ld",n+1,param[n+1]);
             printout(reportStr);
          }
        }
      }
    }
    closeEvent(id,owner.pcom); //close and clear the event
```

For this example, the PDL program should execute printl to trigger the event with `mode2` value=0x55, and the axis ID should be equal to 2 (event ID):

```
#long Z_id, X_id,Y_id;
Z_id=1; X_id=2; Y_id=3;
.....
printl/103/00000055("axis id=%ld x axis: enc_pos=%ld ,
   velocity=%g", X_id,, x_enc_pos, x_vel_max);
```

(This page is intentionally left blank.)

5. Data Collection (Record)

This chapter describes the functions to support data collection from controller.

The data collection algorithm is the same as that implemented in the KMI tool, in which you open a record window, define one or more variables to be recorded, define the number of samples and rate.

5.1. StartRecordData

Prototype:

```
int StartRecordData(char *p[], int numVar, int rate, int numSamples,
double *pdata, char *errstr, MDCE *pcom=NULL);
```

Parameters:

***p[]** – array of strings to define the variable names. The array size should be at list **numVar**.

numVar – number of variables to be recorded.

rate – define sample rate by 20000/rate Hz (assume the sample rate of controller = 20000). So, each variable will be sampled by every **rate***0.00005 sec.

numSamples – maximum samples to be recorded.

pdata – pointer to array to retrieve the data. The first variable will be stored staring from **pdata**[0], the second from **pdata**[**numSamples**], the n-th variables (n = 0 … **numVar**-1) will be stored from **pdata**[n***numSamples**]. So the size of the array **pdata** should be at list **numVar*****numSamples**.

errstr – string to fill with error description.

pcom – pointer to communication object.

Return:

- If 0, it is ok; otherwise, it is error.
- The **errstr** contains error description.

Notes:

- The function starts the record process and returns immediately. You poll the record status by calling `GetRecdordStatus` to test if ended and how many samples it already collected.
- The number of samples finally recorded may be less than definition (in the parameter '**numSamples**'). This may happen if for example the **'rate'** is too small (record frequency=**20000/rate** Hz). So, the actual record frequency is too fast comparing to the communication speed. However, even if the record is stopped before all data is collected, the data that it success to collect is valid.
- To get continuous record (up to '**numSamples**'), it is recommended to set rate >= 8***numVar**.
- You may stop recording while in process by calling `StopRecord`.
- The function `OpenRecord` opens a window (similar to that of the KMI tool), in which you can see all parameters that you set in `StartRecordData` and other record information in real time.

StartRecordDataN

Prototype:

```
int StartRecordDataN(char *varnames,  int rate,
                     int numSamples, double *pdata,
                     char *errstr, MDCE *pcom=NULL);
```

It is similar to the `StartRecordData`, but the **numVar** parameter is canceled and the first parameter ***p[]** is replaced by simple one string ***varnames.** The variable names should be written to **varnames** string parameter separated by spaces or comma.

Example:

```
StartRecordDataN("X_ref_pos,X_perr,X_vel_ff", 16, 5000, pdata,
   errstr);
```

5.2. StartRecordFileN

Prototype:

```
int StartRecordFileN(char *varnames,  int rate, int numSamples, char
*filename, char *errstr, int append, MDCE *pcom=NULL);
```

It is similar to the `StartRecordDataN`, but the result is saved to the file specified (instead to array). The append parameter if 1, add result to existing file if any.

Example:

```
StartRecordFileN("X_ref_pos, X_perr, X_vel_ff", 16, 5000,
   "c:\\myresult.txt", errstr, 0);
```

The result will be also saved to gpp file. So, in this example, the result will be saved to: “**c:\myresult.txt**” and “**c:\myresult.gpp**”.

The gpp file can be viewed by the wingraph tool. To operate from command line, do:

>**wingraph** myresult.gpp

5.3. GetRecdordStatus

Prototype:

```
int GetRecdordStatus(int *pnumSampColect, MDCE *pcom);
```

Parameters:

pnumSampColect – retrieve number of samples already collected.

pcom – pointer to communication object.

Return:

- 0 – record process ended
- 2 – still in process
- -1 – `StartRecordData` not called yet.

Call this function after calling `StartRecordData` to test when the record is ended. You may call it in a loop and print the `pnumSampColect` to view the data collection process.

5.4. StopRecord

Prototype:

```
void StopRecord(MDCE *pcom=NULL);
```

Call it to stop record process. If called when process already ended, nothing happens.

When you stop recording and want to start new record, call `GetRecdordStatus` before starting new one to verify that previous record process is done completely. This is because that `stopRecord` may return before all record tasks are done (for example, if need to save to file when terminate record starts with `StartRecordFileN`).

Example:

```
s=StartRecordFileN(str, rate, numsampreq,"c:\\file1.txt",
errstr,0,pcom);


Sleep(1000);
StopRecord(pcom);
int numSamp,cnt=0;
int busy=GetRecdordStatus(&numSamp,pcom);
while(busy==2 && cnt++<2000){
        busy=GetRecdordStatus(&numSamp,pcom);
        Sleep(10);
}
if(busy==3){
  // … file save fail
}
else if(cnt>=2000)
{
  // … time out
}
s=StartRecordFileN(str, rate, numsampreq,"c:\\file2.txt",
errstr,0,pcom);
```

5.5. OpenRecord

Prototype:

```
void OpenRecord(int show=1, MDCE *pcom=NULL);
```

The **show** parameter should be set to 1.

This function opens the record window. This window is much the same as the record window integrated in the KMI/wizalg. You can also operate the record directly from this window by setting the parameters and variable names. Press ‘start’ button (as you do in the KMI/wizalg) and view the results (press ‘graph’ button). When calling `StartRecordData`, the parameters on the window are updated according to the parameters sent via the function.

Note:

- It is not necessary to open the record window and it may be used for debugging.

5.6. setRecClip16

Prototype:

```
void setRecClip16(int mode, MDCE *pcom);
```

Parameters:

mode - each bit in **mode** corresponds to each variable. It is defined if to be clipped to 16 bit value. A variable that is float or 32 bit integer will be clipped in this mode to 16 bit. If the original value of the variable is greater out of the range -32767....32767, it will be clipped to these limits.

If, for example, **mode** = 0x5, the 1st and 3rd variables to be recorded in the list will be clipped to 16 bit value.

pcom – pointer to communication object.

5.7. setTrigger

Prototype:

```
void setTrigger(int mode, MDCE *pcom);
```

Parameters:

mode – if 1, configure controller to start record when satisfy internal conditions (for example, when detect external hardware input active). When press start in the record window or call any of the functions `StartRecord`, record buffer in controller will start to fill only after detecting the internal condition(s). In case that controller not support trigger mode, calling this function has no effect.

If implementing trigger mode in the controller has matched fast response (about only 1 phase), use the start event option in the record window (in which event is detected by PC via the communication link).

pcom – pointer to communication object.

(This page is intentionally left blank.)

6. Access Variables/Arrays

Most of the functions here return 0 when it is ok. If nonzero value is returned, you can call `GetErrorStr` to retrieve error description.

For all functions that get controller variable names as parameter and variable can be indexed by constant (probably used to access one item in array). For example:

```
SetVarN("rec_buf[300]", 15, 0, NULL);
```

6.1. GetVarAddr

Prototype:

```
int GetVarAddr(char *varName,   int slave,
               int *psize, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable name string.

slave – slave number.

***psize –** if not NULL, store the variable size. For non-array variable, return 1; for array variables, return the array size (non-variable array may be considered as array with size 1). If variable is found, return -1.

pcom – pointer to communication object.

Return:

- If the variable does not exist, it returns **ADDERR**, the address of the variable.

Use this function to detect if variable exists and/or the size of variables.

To also detect variable type, use `GetVarAddrType` instead of this function.

6.2. GetVarAddrType

Prototype:

```
int GetVarAddrType(char *varName, int slave, int *psize,
                   int *ptype, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable name string.

slave – slave number.

***psize –** if not NULL, store the variable size. For non-array variable, return 1. For array variables, return the array size (non-variable array may be considered as array with size 1). If variable is not found, return -1.

***ptype** - if not NULL, store the variable type according to following:

1 – short (16 bit integer).

2 – long (32 bit integer).

3 – float (32 bit float).

4 – pointer (32 bit).

8 – 64 bit integer.

0 – variable not found.

pcom – pointer to communication object.

Return:

- If the variable does not exist, it returns **ADDERR**, the address of the variable.

Use this function to detect if variable exists and/or the size and/or type of variables.

6.3. SetVarN

Prototype:

```
int SetVarN(char *varName, double data,
            int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable name string.

data – data to write to variable. It automatically converts the value to the correct controller variable format (float/long/short).

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

6.4. GetVarN

Prototype:

```
int GetVarN(char *varName, double *pdata,
            int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable name string.

***pdata** – store data variable. It automatically converts the value to the correct controller variable according to format (float/long/short).

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

6.5. SetVarN64

Prototype:

```
int SetVarN64(char *varName, _int64 data, int slave=0,
              MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable name string.

data – 64 bit data to write to variable.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

Note:

- Variable type must be 64 bit; otherwise, the function returns error.

6.6. GetVarN64

Prototype:

```
int GetVarN64(char *varName, _int64 *pdata,
              int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable name string.

***pdata** – store data 64 bit variable.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

Note:

- Variable type must be 64 bit; otherwise, the function returns error.

6.7. setArrayDN

Prototype:

```
int setArrayDN(char *varName, double *pdata, int num,
               int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable/array name string.

***pdata** – pointer to array in the application which will copy to array in controller.

num – number of variables to copy.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

Description:

- Copy the array of doable in the application to array in the controller.
- If needed, casting is done to controller variable type.

6.8. getArrayDN

Prototype:

```
int getArrayDN(char *varName, double *pdata,
               int num, int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable/array name string.

***pdata** – pointer to array in the application which will copy from array in controller.

num – number of variables to copy.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; otherwise is error.
- Call `GetErrorStr` to get the description of error.

Description:

- Copy array in the controller to array of doable in the application.
- If needed, casting is done according to the controller variable type.

6.9. setArrayN

Prototype:

```
int setArrayN(char *varName, int  *pdata, int num,
              int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable/array name string.

***pdata** – pointer to array in the application which will copy to array in controller.

num – number of variables to copy.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; otherwise is error.
- Call `GetErrorStr` to get the description of error.

Description:

- It is similar to `setArrayN`, but the array pointed by **pdata** is 32 bit.

Note:

- If the controller variable is float, **pdata** should point to array of float with 32 bit (even that pointer type is int).
- This function is required for now variable type, but does not do internal casting and only allows you to save memory as **pdata** pointed to 32 bit array.

6.10. getArrayN

Prototype:

```
int getArrayN(char *varName, int  *pdata, int num,
              int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller variable/array name string.

***pdata** – pointer to array in the application which will copy from array in controller.

num – number of variables to copy.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; otherwise is error.
- Call `GetErrorStr` to get the description of error.

Description:

- It is similar to `getArrayN`, but the array pointed by **pdata** is 32 bit.

Note:

- If the controller variable is float, **pdata** should point to array of float 32 bit (even that pointer type is int).
- This function is required for now variable type, but does not do internal casting and allows you to save memory as **pdata** pointed to 32 bit array and not 64 bit array.

6.11. setArraySN

Prototype:

```
int setArraySN(char *varName, short *pdata, int num,
               int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller short variable/array name string.

***pdata** – pointer to short array in the application, which will copy to array in controller.

num – number of variables to copy.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; otherwise is error.
- Call `GetErrorStr` to get the description of error.

Description:

- It is similar to `setArrayN`, but the array pointed by **pdata** is 16 bit.
- The controller variable must be short type (otherwise function returns error). It is useful to read short array in the controller into short array *n* PC.
- No internal casting is done.

6.12. getArraySN

Prototype:

```
int getArraySN(char *varName, short *pdata, int num,
               int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varName** – controller short variable/array name string.

***pdata** – pointer to array in the application which will copy from array in the controller.

num – number of variables to copy.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

Description:

- It is similar to `getArrayN`, but the array pointed by **pdata** is 16 bit.
- The controller variable must be short type (otherwise function return error). It is useful to set short array in the controller from short array *n* PC.
- No internal casting is done.

6.13. getPac

Prototype:

```
int getPac(char *varnames, char *l, void *data,
           int slave=0, MDCE *pcom=NULL);
```

Parameters:

***varnames** – string contains one or more variable names up to 20.

***l** – pointer to char array that will store with the variable types. Array size should be at list 20.

***data** – array data to store value answered for all variables. Its size depends on the number of variables. It has three types: long/float, short, and 64 bit.

slave – slave number.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

Descriptions:

- This function reads several variables in one shoot (one message transmitted and one message received via the communication link).
- It is much efficiency to read several variables with `getPac`. Read them separately by using `GetVarN` or `GetVarN64`. Each variable may be different type and/or byte size.
- The data should treat according to variable types.
- You may use the **l** array returned to identify each variable type, where the value is coding to

  1 **–** 2 bytes short.

  2 **–** 4 bytes long.

  3 **–** 4 bytes float.

  8 **–** 8 bytes (64 bit) variable.

  These are the same codes as return from `GetVarAddrType` function.

Example:

```
struct vargroup{
    long time;
    __int64 position;
    float velocity;
    short analoginput;
```

```
    long spare[10];
};
char vartypes[20];
char varlist[200]="fclk X_enc_pos X_vel_ff a2d[2]";
err=getPac(varlist vartypes, &vargroup, 0, NULL);
```

Note:

- Variable list types should be much than actual types in the struct '**vargroup**'.

6.14. getArrayAndSetN

Prototype:

```
int getAndSetArrayN(char *varrep,  int numrep,
                    void *parrrep, char *varset,
                    int numset, void *parrset,
                    int slave, MDCE *pcom);
```

Read and write to array in one function (and one communication message). It is faster than using `getArray` and `setArray` separately.

Parameters:

***varrep** – controller variable/array name string to read from.

numrep – number of items to read. Range: 0 up to maximum 124 shorts or 62 longs.

***parrrep** – pointer to array in the application which will copy from array in the controller.

***varset** – controller variable/array name string to write to.

numset – number of items to write. Range: 0 up to maximum 120 shorts or 60 longs.

***parrset** – pointer to array in the application which will copy to array in the controller.

slave – slave number.

pcom – pointer to communication object.

The array pointers `parrrep` and `parrset` should be pointed to array of the same type as the appropriate variable type in the controller (long/short/float). Here, 64-byte is not supported.

6.15. getArrayAndSetDN

Prototype:

```
int getAndSetArrayN(char *varrep,  int numrep,
                    double *parrrep, char *varset,
                    int numset, double *parrset,
                    int slave, MDCE *pcom);
```

It is similar to `getArrayAndSetN`, but:

- the array pointer must be double float type.
- the variables in the controller **varrep** and **varset** must be float type.
- the DLL does the casting internally.

6.16. GetScopeData

Prototype:

```
long GetScopeData(LPCTSTR varnames, double FAR* pdata,
                  short *pfclk, long slave);
```

Parameters:

varnames - a string that defines variables name to be recorded. The names should be separated by spaces.

pdata – array type double that will receive the variable data. Its size should be as the number of variables.

pfclk – a short (16 bit) that returns the fclk counter variable. This value is a counter that counts in the DSP sample rate and shows the “time” when the data is sampled. It may use to draw the variables value visually in Scope.

slave – slave ID.

.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

This function reads a group of variables and a time counter in the same DSP phase. It may use in the Scope application. It should be called continues with delay no more than 200 msec between calls; otherwise, it returns an error code 300. So the first call to this function will always return error 300 (if no other error), but the following calls will return ok.

6.17. GetErrorStr

Prototype:

```
void GetErrorStr(int errcode, char *str, int slave,
                 char *errstr, MDCE *pcom=NULL );
```

Parameters:

errcode – the return value from the access functions (`SetVarN`, `GetVarN`, `setArrayDN` …).

***str** – variable/state name.

slave – slave number.

***errstr –** pointer to a string of 200 bytes length at least, to be printed with the error description.

pcom – pointer to communication object.

Call this function when nonzero returned from one of the variables/state access functions to get error description.

(This page is intentionally left blank.)

7. Access States

States are specified as one bit variables in the controller (like Boolean variable) which represent digital inputs/outputs and controller internal states (axis run, axis position error flag …). States are the bits of status array.

7.1. setStateN

Prototype:

```
int setStateN(int slave ,char *stateName,
              int mode, MDCE *pcom=NULL);
```

Parameters:

slave – slave number.

***stateName** – controller state name string.

mode – 1 sets state on, while 0 sets state off.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

7.2. getStateN

Prototype:

```
int getStateN(int slave ,char *stateName,
              int *state, MDCE *pcom=NULL);
```

Parameters:

slave – slave number.

***stateName** – controller state name string.

***state** – return state value. 1 => state is on. 0 => state is off.

pcom – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` to get the description of error.

Note:

- Reading states via `getStateN` is much faster than read variable (`GetVarN`, `getArrayN`…), because states are taken from internal table inside the DLL and does not need to be passed via the communication link from controller.
- This internal table is updated automatically from controller each time that any state is changed.

(This page is intentionally left blank.)

8. Run PDL Functions

8.1. RunFuncPdlN

Prototype:

```
int RunFuncPdlN(char *pdlName, char *errstr,
                int slave, MDCE *pcom=NULL);
```

Parameters:

***pdlName** – string of PDL `func` name. Any label in the PDL program that starts with _ (underscore) is recognized by host. This function creates new task that starts to run from this label in the controller.

***errstr** – return error description (no need to call `GetErrorStr`).

slave – slave number.

***pcom** – pointer to communication object.

Return:

- If fail, return -1; otherwise, return task number that runs this function (0..39).
- If the return value is -1, you can get error description from **errstr**.

Descriptions:

- This function creates new task in the controller that starts to run from the label specified.
- If any task is already executed from this label, no new task will be created and the existing task ID will be returned.

8.2. SetArrayRunFuncN

Prototype:

```
int SetArrayRunFuncN(char *varName , void *arr,
                     int num, char *pdlName, char *errstr,
                     int slave, MDCE *pcom=NULL);
```

Parameters:

***varname** – variable/array name to copy data into.

***arr** – pointer to array to copy into variable. The **arr** type should point to array type that much than the variable type (long/float/short). However, if the variable in the controller is the array of mixed float/long (32 bit), the **arr** can point to array of mixed float/long corresponding to each item array in the controller.

num – number of variables to copy. If type is long/float, the maximum is 40. If type is short, the maximum is 80.

***pdlName** – string of PDL `func` name. Any label in the PDL program that starts with _ (underscore) is recognized by host. This function creates new task in the controller that starts to run from this label.

***errstr** – return error description (no need to call `GetErrorStr`).

slave – slave number.

***pcom** – pointer to communication object.

Return:

- If fail, return -1; otherwise, return task number that runs this function (0…39).
- If the return value is -1, you can get error description from **errstr** (same as in `RunFuncPdlN`).

Description:

- Combine setting array with running PDL function in one function.
- Running this function is faster than setting array (by `setArray`) and then running PDL function (by `runFuncPdl`).
- The <**varName**> array in the controller data is probably processed by the PDL function that runs after this array is filled.

Notes:

- That <**vaName**> may be only one variable. In this case, **num** = 1.
- If you need to fill array larger the limit of 13/26, call `setArrayN` to fill the start of array and then call `SetArrayRunFuncN`.

Example:

Assume that the array in the controller is called **'indata**' with size 100 types long, and the PDL function is called '**procdata**'.

```
int myarr[100];
char errstr[200];
……
int err=setArrayN("indata", myarr, 87, 0,  NULL); //Write first 87
   items
if(err){
 GetErrorStr(err, "indata", 0,  errstr );
 MessageBox(errstr, "CONTROLLER ERORR", MB_OK)
 return;
}
//Write residual 13 items and run PDL function "procdata"
err= SetArrayRunFuncN("indata[87]", &myarr [87], 13, ."procdata",
   errstr, 0, NULL);
if(err){
 MessageBox(errstr, "CONTROLLER ERORR", MB_OK)
 return;
}
```

8.3. Kill/stop/Cont task

Prototype:

```
int killTask(int st, int en, int what,
             int slave, MDCE *pcom=NULL);
```

Parameters:

st – start task ID.

en – end task ID.

what – 0 => kill task(s). 1 => stop task(s). 2 => continue tasks(s).

slave – slave number.

***pcom** – pointer to communication object.

Return:

- 0 is OK; else is error.
- Call `GetErrorStr` (set **str** parameter to " ") to get the description of error.

Description:

- Kill/stop/continue tasks from **st** to **en**. To kill only one task *n*, set **st** = *n* and **en** = *n*+1, where *n* can be the task number returned from `RunFuncPdlN/ SetArrayRunFuncN.`

(This page is intentionally left blank.)

9. Frequency Analyzer

This chapter describes the methods for frequency response measurement between any 2 points (variables) in the DSP application.

9.1. FreqAnalStart

Prototype:

```
long FreqAnalStart(double startfr, double stopfr,
                   double stepfr, double amplitude,
                   LPCTSTR varsignal, LPCTSTR varinput,
                   LPCTSTR varoutput, long slave,
                   long mincyc, double idletime,
                   double datatime, LPCTSTR filename,
                   char *comment=NULL, MDCE *pcom=NULL);
```

Parameters:

startfr – frequency scan start in Hz.

stopfr – frequency scan end in Hz.

stepfr – frequency logarithm step start in Hz. For example, the value of 0.1 will give you 10 points per decade (decade is the range between f1 (Hz) to 10*f1 (Hz)). The total number of frequency points that will be tested is log10 (**stopfr**/**startfr**)/**stepfr**.

amplitude – sinusoidal wave swap signal value which will enter to **varsignal**. This value depends on the signal variable. If it is an encoder position type variable (`ref_pos_ext`), for example, the value represents the sinusoidal amplitude in encoder counts units. If it is a command variable (`disturb`), the value is a command unit, where the value 1000 probably corresponds to maximum command.

varsignal – variable name in which the sinusoidal wave will be injected.

varinput – input variable name.

varoutput – output variable name. The result of bode value will be the gain and phase between the sinusoidal in the output signal to that measured on the input signal in each frequency.

slave – slave number. In single slave system, it will be 0.

mincyc – the minimum sinusoidal cycles used in each frequency to measure gain and phase response. The minimum number will be 1.

idletime – how much time (seconds) the sinusoidal wave is active before DSP starts to calculate the frequency response. This time may be needed to overcome some initial conditions on the implemented algorithm. The actual **idletime** will not be less than **mincyc****T*, when *T*=1/*f* is the current frequency measured. So, for example, when frequency response of 1Hz is measured, the **idletime** will be at least 1 second (even if you write to this parameter value less the 1).

datatime – how much time (seconds) the sinusoidal wave is active when DSP starts to calculate the frequency response. The greater the number is, the more accurate the measurement is. However, the process will take more time. The actual **datatime** will not be less than **mincyc****T*, when *T*=1/*f* is the current frequency measured. So, for example, when frequency response of 1 Hz is measured, the **datatime** will be at least 1 second (even if you write to this parameter value less the 1).

Filename – a file name to write the data result. The data will be written to 2 files. The first file is in text file format and its name is the parameter 'filename'. The second file

is in **fgr** format and the name is the same but the extension will be .fgr. The fgr file can be loaded to the franal tool, and the text file can be inputted to other applications (Matlab, Excel …). If you do not specify full path of file name, it will be saved to current working directory. If the process stops due to error (or user stops), it will not save the result to file.

Comment – optional comment string that is added to bode plot attribute.

***pcom** – pointer to communication object.

Return:

- 0 is ok; otherwise, the frequency analyzing is already in process.

Note:

- This function returns immediately and does not wait till the process is ended.
- You should call `FreqAnalStatusGet` and `FreqAnalErrorStrGet` to monitor the process and detect if any error occurs.

Example:

- Call this function for measured open-loop response of X axis from 1 to 1000 Hz with step 0.05. Set **mincyc** = 1, **idletime** = 0.1, and **datatime** =0.1. Results are saved to files “c:\res1.txt” and “c:\res1.fgr”. Sinusoidal amplitude to command is 150.

```
Err=FreqAnalStart(1,1000, 0.05, 150, "X_disturb",
"X_command", "X_vel_fb" 1, 0.1, 0.1,"c:\\res1.txt", NULL);
```

9.2. FreqAnalStartP

Prototype:

```
long FreqAnalStartP(char *frp_filename, double amplitude,
                    LPCTSTR varsignal, LPCTSTR varinput,
                    LPCTSTR varoutput, long slave,
                    long mincyc, double idletime,
                    double datatime, LPCTSTR filename,
                    char *comment=NULL, MDCE *pcom=NULL);
```

This function is similar to `FreqAnalStart` except that the frequency points to test are taken from file <frp_filename> instead of the parameters **startfr**, **stopfr**, and s**tepfr**. This is useful to define not unique logarithm (basis 10) frequency points. For example, assume that you want to test in the range of 10 to 100 Hz with step size of 0.01 Hz, and 100 to 1000Hz with step size of 0.001 Hz, create a file named <myFreqPoints.txt> like this:

```
10 100 0.01
100 1000 0.001
```

And then call `FreqAnalStartP` like this:

```
Err=FreqAnalStartP("c:\\myFreqPoints.txt", 150, "X_disturb",
   "X_command", "X_vel_fb" 1, 0.1, 0.1,"c:\\res1.txt", null);
```

Notes:

- You may write points in the file in any order, not necessary from small to big. It will sort the points and delete duplicate points.
- When write, for example, 10 100 0.1, the 10 and 100 Hz will be included in the list. The step value is in log, which is 0.01, with the range of 10 to 100; means 101 points (include 10 and 100).
- Adding single frequency is also available. Assume you want to add 885Hz to the list, the file <myFreqPoints.txt> will contain:

  ```
  10 100 0.01
  100 1000 0.001
  885
  ```

Return:

- 0 is ok; nonzero is error.
- Call `FreqAnalStatusGet` to get error description in **errstr** parameter.

9.3. FreqAnalStatusGet

Prototype:

```
long FreqAnalStatusGet(double *pfreq, double *process,
                       long *plasterror, char *errstr,
                       MDCE *pcom=NULL)
```

Parameters:

pfreq – return current frequency (Hz) in process.

process – return a number in the range of 0 to 100 for how much % of process already done. User may use this value to draw a progress bar view to show progress status.

plasterror – return last error code.

errstr – last error description string.

Return:

- 0 – process done.
- -1 – `FreqAnalStart` not called yet.
- 1, 2 – process active.

You may use the return value to detect if process ended.

Note:

- It may end before done all frequency with required points due to error or user stop (by calling `FreqAnalStop`). So, it is also recommended to watch the return error (in plasterer).

9.4. FreqAnalStop

Prototype:

```
long FreqAnalStop( MDCE *pcom=NULL);
```

Return:

- -1 => `FreqAnalStart` is not called yet; else is 0.

If the process is not active, this function does nothing; otherwise, it stops the process.

9.5. FreqAnalOpen

Prototype:

```
void FreqAnalOpen(long show, MDCE *pcom=NULL);
```

Call it with **show** =1 to open and show the frequency analyzer window object. If **show** = 0, it hides the window; otherwise, it only opens the window (if it is not opened yet) without showing.

It may be used for debugging purpose when it shows the frequency response measured process in real time by drawing the bode graph. You may watch this window to verify parameters that you set from `FreqAnalStart` when all parameters are updated in the edit boxes when call `FreqAnalStart`.

This window includes many options, and may use as a frequency analyzer tool. You can change all parameters, start /stop/single operation, view bode in all modes (gain+phase, Nyquist, Nichols ...), and so on. All methods operated from DLL are done via this object.

9.6. fftrun

Prototype:

```
int fftrun(double *pin, int num, double *pabs,
           double *parg, int showWnd);
```

Parameters:

pin – array of double data.

num – size of array input data.

pabs – point to array with size **num** that will fill with the absolute values of FFT result.

parg – point to array with size **num** that will fill with the angles of FFT result in radians.

showWnd – if 1, it will show a progress window, in which the user can stop the FFT process while running.

Return:

- 0 – ok.
- -1 – memory fault.
- -2 – in process.
- 1 – user stop.

This function performs an FFT on input data and puts the result to **pabs** and **parg** buffer. All buffers should be allocated by user to minimum size of **num**.

You can calculate the frequency resolution and frequency range from the sample rate (of the input data) *dt*, and the size of the data.

freq. resolution = *df* = 1/ (**num** **dt*).

freq. range = 0.....1/*dt* – *df*.

The FFT process time depends on the buffer size **num**. The large **num** can be divided into small integers to take less time. The minimum time is achieved if **num** = 2^*n*, and the maximum time is achieved when **num** is a prime number.

10. Call-back Functions

You can define some call-back functions that will be called from the DLL as responding to internal event. The DLL should get a pointer of the function by setting call-back functions. You can call this function any time to change the call-back function, or disabled it (by setting the '**func**' parameter with NULL).

10.1. setCallBackStEvnt

Prototype:

```
void setCallBackStEvnt(void (*func)(BYTE ,int), MDCE *pcom);
```

Parameters:

type – type of state.

id – ID of state.

The **type** and the **id** are state attributes defined for each state in <appl name.stt> file. The function (`PutEventSt`) is called when a state is changed, and at least one of the first 2 bits in the **type** is 1 (which defines the direction of change to send event). For more detail see the< pdl.doc> file.

Example:

```
void PutEventSt(BYTE type, int id)
{
    char str[200];
    sprintf(str,"state change, type=%02x, id=%ld",type,id);
    MessageBox(str);
}
……
setCallBackStEvnt(PutEventSt, NULL); //Set the DLL with pointer of the call-back function.
```

10.2. setCallBackPuUpd

Prototype:

```
void setCallBackPuUpd(void (*func)(BYTE*), MDCE *pcom );
```

Parameters:

BYTE* – pointer to the full message arrived from controller.

The function `PuUpd`, call-back function, will be called in the controller that supports '`puupd`' algorithm, in which a special data message is sent from controller (usually start byte =0x33).

Example:

```
void PuUpd(BYTE *rcbuf)
{
    char str[200];
    sprintf(str,"puupd: %02x %02x…",rcbuf[0],rcbuf[1]);
    MessageBox(str);
}……
setCallBackPuUpd(PuUpd, NULL); //Set the DLL with pointer of the call-back function
```

(This page is intentionally left blank.)

11. Error Code

11.1. List of error code

The list of error codes is below:

Error code	Description
31	Slave number out of range (0~31)
101	Read failed
102	Write failed
103	Frame error
104	No answer
105	Check sum error
106	Size out of range
107	Parity error
108	Over run error
109	Buffer over flow
110	Frame error
111	Abort
112	Net sync error
113	Can error
114	Answer size error
115	Start byte error
116	Master boot
117	EtherCAT mega-ulink error
118	EtherCAT mega-ulink in test mode
151	Communication FIFO full
152	Size transmit error
153	Communication is not available
154	No connection to remote side
155	Local communication blocked
156	Communication failed
157	No connection to remote side via Ehernet
158	EtherCAT mega-ulink slave not respond
201	Unrecognized variable
202	Slave name not defined
203	The list contain one or more unrecognized variables in slave
204	Variable is not short type

220	Variable is not a 64 bit type
221	Variable is a 64 bit type
251	Unrecognized function in slave
252	Slave name not defined
260	PDL function is not running in slave
261	Slave is in boot mode
262	Task number out of range
263	Slave in not in boot mode
270	Write buffer denied
271	Buffer size to set out of range
272	Read variable denied
273	Number of variables out of range
274	Expected answer length too big
275	Support only array access type 16/32 bits
300	Scope restart
301	Scope support only 2 variables
500	Link busy
501	No support
502	Master boot
503	No slave
504	Link error
505	Slave error
506	Invalid address
525	Server busy
600	State number out of range
601	Unrecognized state in slave
602	Slave name not defined
651	No support id communication command
801	Extra parameters
802	Missing parameters
803	Unrecognized command
1000	Communication driver is NULL

(This page is intentionally left blank.)

A. Code Example

A.1. MPI library initialization

The flow of MPI library initialization is shown in the following figure. Here, `showcomstatus` is optional.

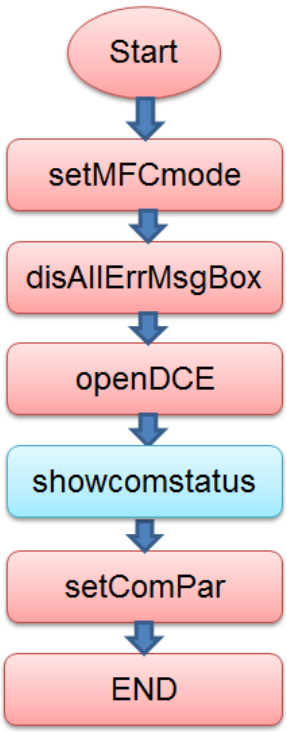

Fig. A-1

Example:

This example shows how to do MPI library initialization for D1 drive with DD motor.

```
setMFCmode();
disAllErrMsgBox(1);
pCom_DD= openDCE("C:\\HIWIN\\dce\\", "DD.dce");
//showcomstatus(pCom_DD) ;
Status.DD_Connet = setComPar(
    m_DD_COM,      // Please set to the number of communication port of Drive.
                   For example, 2 for COM2.
    nBaudrate1,    // Please set to 8 for 115200 bps.
    nMode,         // Please set to 1 for RS-232/USB or 4 for mega-ulink.
    nTrid,         // No use, please set to 0.
    nRcid,         // No use, please set to 0.
    nCanbaudrate, // No use, please set to 0.
```

```
    nMsgStand,       // No use, please set to 0.
    nCanpipelevel, // No use, please set to 0.
    nTimeout,
    nLocktime,
    nIternum,
    pCom_DD);
```

A.2. Encoder feedback

This example shows how to get the encoder feedback of motor at the console.

Notes:

- Before connecting to drive, call `setMFCmode` to initialize DLL.
- After initializing DLL, call `openDCE` to build communication object.
- After building communication object, call `setComPar` to set communication parameters for drive.
- When the link between PC and drive is built, call `GetVarN` to get variables of the drive.
- The drive's variable for motor feedback position is *X_enc_pos*. Hence, this example uses this variable to read the encoder feedback of motor.

Used MPI functions:

```
void setMFCmode();
MDCE *openDCE();
int setComPar();
int GetVarN();
void deletDCE();
```

Code Example:

```
#include "stdafx.h"
#include <Windows.h>
#include <conio.h>
#include <stdio.h>
#include "mpint.h"      //Include MPI header

void main()
{
  MDCE *pCom = NULL  //Declare
  //MPI Function: DLL Initilization
  setMFCmode();
  //MPI Function: Open Communication Object.
  int nPort,nBaudrate,bConnected;
  do
  {
     printf("\n Input number to select Communication port (1:COM1,
2:COM2, …) or -1 quit: ");
```

```
    scanf("%d", &nPort);
    if(nPort == -1)  goto End; //Get -1 to go to end the application.
    //MPI Function: Start to communicate with amplifier.
    bConnected = setComPar(nPort,nBaudrate=8,1,0,0,0,0,0,100,
    200,6,pCom);
    //*****************************************************************************//
  }while (bConnected!=0); //nConnected = 0 denotes that communication is
successful; otherwise, please re-select communication port.


    int key;
    printf("Command List:\n");
    printf("Input 'r' : Read Encoder\n");
    printf("Input 'q' : Quit The Application");
  Loop:
    printf(">>Command Input: ");
    key=getche();printf("\n");
    switch(key)
      {
       //+++++++++ Read Motor Feedback Position ++++++++++++//
       case 'r' :
         {
           char *var = "X_enc_pos"; //Variable of motor feedback position
inside of amplifier.
           double fpos;
           //MPI Function: Retrieve Motor Feedback Position.
           if(GetVarN(var,&fpos,0) == 0)
             printf("Motor feedback position =%.0f counts.\n",fpos);
           else
             printf("Failed to read feedback position!\n")
         }
         break;
          //++++++++++ Quit the Application ++++++++++++//
       case 'q':
          goto End;
          break;
      }
      goto Loop;
      End:
```

```
    //MPI Function: Close Communication Object.
    deleteDCE(pCom);
    system("PAUSE");
}
```

MPI Library Reference Manual for D-series Drives

© HIWIN Mikrosystem Corp.