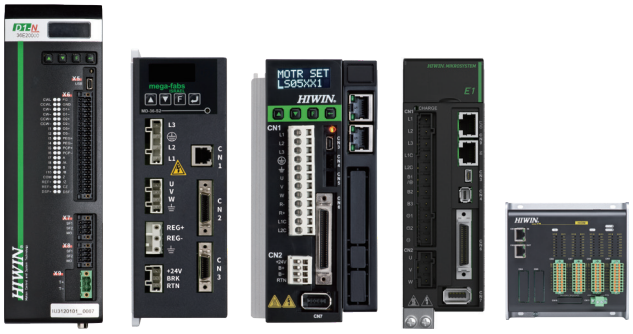
HIWIN® MIKROSYSTEM



MPIライブラリ

リファレンスマニュアル

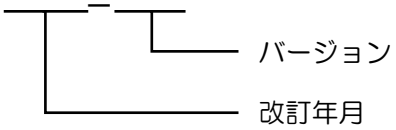
www.hiwinmikro.tw

MD19UJ-2501_V2.4

改訂履歴

マニュアルのバージョンは表紙の下部にも記載されています。

MD19UJ01-2501_V2.4



日付	バージョン	DLL バージョン	改訂内容
2025 年 1 月	2.4	6.0.8.0	(1) 関連ドキュメントを追加します。 (2) 通信機能を最適化： セクション 1.1 このマニュアルについて を更新。 セクション 3.1.1 EtherCAT mega-ulink への接続 を更新。 セクション 3.1.2 USB への接続を更新。
2022 年 3 月 23 日	2.3	6.0.7.0	(1) セクション 3.1.1 EtherCAT mega-ulink への接続 を更新。 (2) セクション 3.11 setNetworkAdapter を追加。 (3) セクション 3.12 getNetworkAdapterCount を追加。 (4) セクション 3.13 getNetworkAdapterInfo を追加。 (5) 10 章エラー コードを更新。
2021 年 4 月 28 日	2.2	6.0.3.0	(1) セクション 1.1「このマニュアルについて」を更新。 (2) セクション 6.18「SetVarUN」を追加。 (3) セクション 6.19「SendRAMtoFlash」を追加。
2020 年 10 月 30 日	2.1	6.0.2.0	(1) セクション 1.1 このマニュアルについて: 使用前に MFC ライブラリがインストールされていることを確認。 (2) セクション 2.1 openDCE: E1 シリーズドライバーの構成ファイルのパスを更新。
2020 年 2 月 10 日	2.0	6.00	(1) 関数 setComPar に「USB に接続」セクションを追加。 (2) 関数 getUsbHubPort を追加。 (3) 関数 setComParEx、ConnectEthernet、getComParEx、openRemote、RunKmi、KillKmi、RunAppl、KillAppl、loadFirmName、HisStart、HisStop、HisSave、HisView、setRecClip16、SetArrayRunFunc、setCallBackPuUpd を削除。 (4) 「周波数アナライザー」の章を削除。

日付	バージョン	DLL バージョン	改訂内容
2014 年 5 月 26 日	1.51	1.13	初版

関連文書

関連ドキュメントを通じて、ユーザーはこのマニュアルの位置付けとマニュアルと製品の相関関係をすぐに理解できます。詳細については、HIWIN MIKROSYSTEM の公式 Web サイト → ダウンロード → マニュアルの概要 (https://www.hiwinmikro.tw/Downloads/ManualOverview_EN.htm) にアクセスしてください。

目次

1. 序文

1.1 このマニュアルについて

このマニュアルは、HIWIN D シリーズドライバーおよび E1 シリーズドライバーに適用されます。DLL ファイル (mpi.dll) によって実装される MPI ライブラリは、Microsoft Visual C++ または Visual Basic .NET によって生成されたすべてのアプリケーションを受け入れます。その機能には、MFC の利用と Windows 95/98/2000/XP/7/10 Professional での実行が含まれます。したがって、使用する前に MFC がインストールされていることを確認し、関数 setComPar のさまざまな接続方法に関する説明に注意してください。MPI ライブラリを介して次のことを実現できます。

- 通信構成を設定します。（ポート番号、ボーレート、RS232/USB/CAN など）
  注意: HIWIN 製の D シリーズドライバーは CAN をサポートしていません。
- 複数の通信ポートで動作します。各ポートは別々のドライバーに接続されますが、2 つ以上の RS232 ポートがある場合は同じドライバーに接続することもできます。
- マルチタスクをサポートします。複数のタスクが DLL インターフェイスを介して最小限の遅延でドライバーにアクセスできます。
- エラーを処理します。たとえば、通信エラーが発生すると、エラーが解消されるまで、または「再試行」パラメーターの制限回数に達するまで、エラー メッセージが送信されます。
- ドライバー内の変数または配列の読み取り/書き込み。64 ビット変数をサポートします。
- PDL 関数を実行します。
- 設定/クリア/トグル/読み取り状態。
- printl PDL コマンドを使用してコントローラーのイベントを監視します。
- データ収集（記録）をサポートします。
- mpi を使用する 2 つ以上のアプリケーション間のインターネット接続を構築します。

MPI ライブラリは、mpi.lib、mpi.dll、canlib32.dll、mpint.h の 4 つのファイルで構成されています。PDL 言語と DCE データベースについては、HIWIN 公式 Web サイトの「D シリーズ ドライバーの PDL リファレンス マニュアル」を参照してください。

1.2 データベースの検証

ドライバーのメインプログラムであるデータベースには、MDP ファイルと PDL ファイルが含まれています。これは、各ドライバーのレジスタと変数間の参照に使用されます。ドライバーのデータベースは、PC のデータベースと同じである必要があります。詳細については、関数 compareFw を参照してください。

ユーザーは、Wizalg プログラム (ソフトウェア インストール パッケージ内のツール) を使用して、ドライバーのデータベースを検証できます。操作手順を以下に示します。(注: E1 シリーズドライバーは、データベースの検証をサポートしていません。)

Step 1. ディレクトリ「C:¥HIWIN¥dce¥toolswin¥winkmi」から Wizalg プログラムを開きます。

Step 2. PC をドライバーに接続します。

Step 3. [**File**] で [**Verify data base…**] を選択して、ファームウェアのウィンドウを開きます。

Step 4. ファームウェア ウィンドウの左側にあるスレーブ ツリーで、すべてのドライバーをクリックします。図 1.2.1 に示すように、2 列 3 行の表が表示されます。1 列目はドライバーのファームウェアに関するもので、2 列目は PC の作業ディレクトリ内のファイルに関するものです。各 Csum サブ行のチェックサムは同じである必要があります。(1 番目の Csum サブ行である Boot Dsp Program は適用されません。)

Step 5. 青いテキストはチェックサムが一致していることを示します (図 1.2.1 を参照)。一方、赤いテキストはチェックサムが一致していないことを示します (図 1.2.2 を参照)。チェックサムが一致しない場合は、セクション 1.3 を参照してデータベースを更新してください。

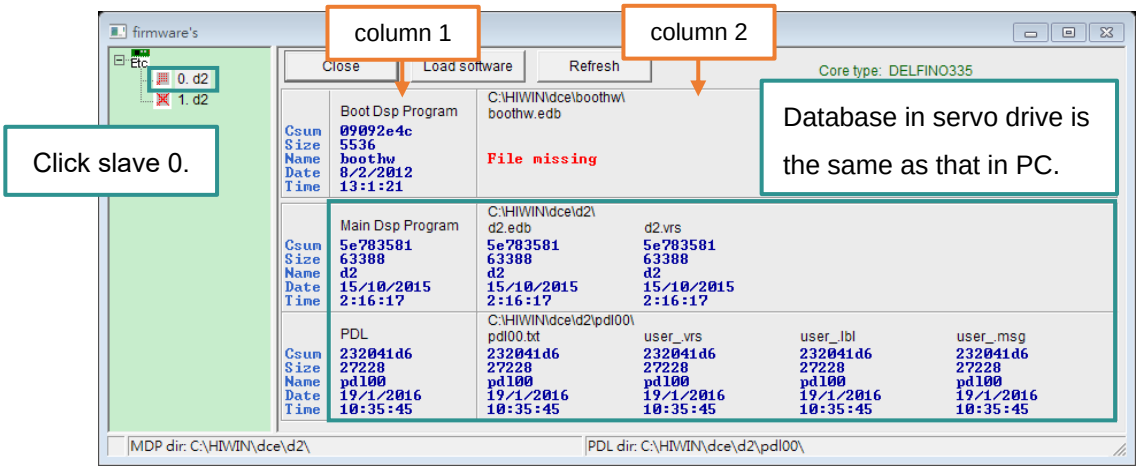


図 1.2.1 チェックサムの一致

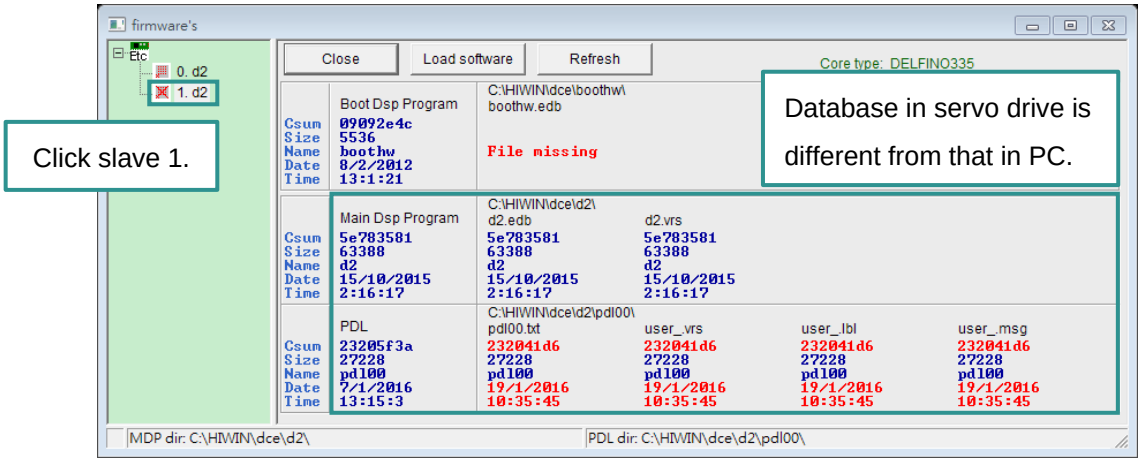


図 1.2.2 チェックサムが一致しない

1.3 データベースの更新

ユーザーは、Wizalg プログラムを利用して MDP ファイルと PDL ファイルを更新できます。操作手順を以下に示します。(注: E1 シリーズドライバーはこの機能をサポートしていません。複数のドライバーが接続されている場合、それらのファームウェア バージョンは同じである必要があります。)

Step 1. ディレクトリ「C:¥HIWIN¥dce¥toolswin¥winkmi」から Wizalg プログラムを開きます。

Step 2. PC をドライバーに接続します。

Step 3. ブートで [**Auto load programs……**] を選択して、[**Auto load programs**] ウィンドウを開きます。

Step 4. 「PDL のコンパイル」のチェックを外し、「Run」ボタンをクリックします。

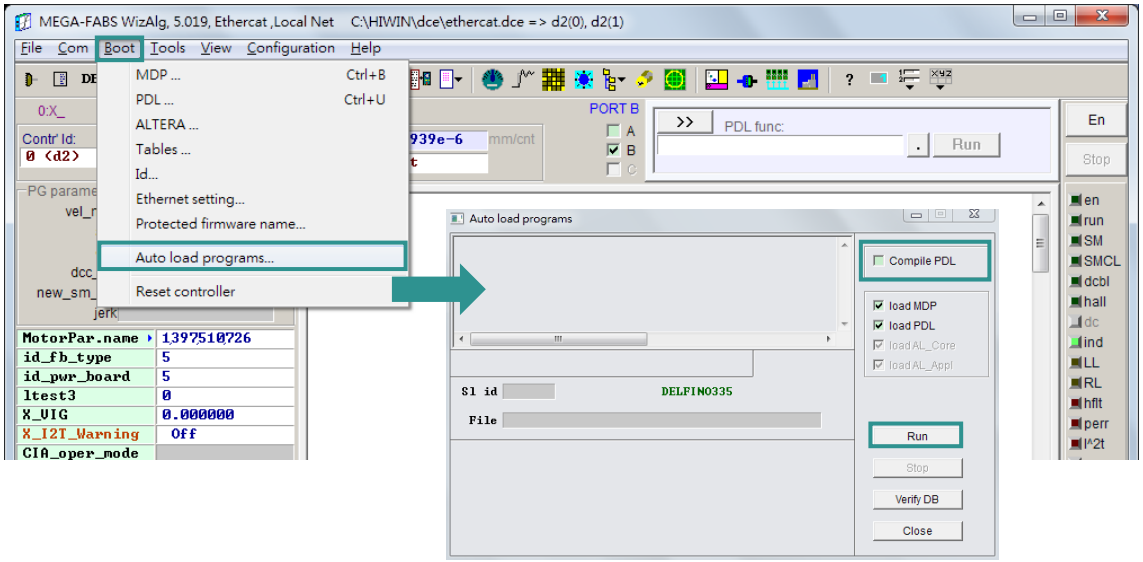


図 1.3.1

2. 初期化

この章では、アプリケーションの起動時または終了時に呼び出す必要があるエクスポートされた関数について説明します。

2.1 openDCE

目的

PC とドライバー間の通信を構築します。

プロトタイプ

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

パラメーター

*pwdin　　Configuration file's path.

*cnfname　　Configuration file's name.

NULL に設定されている場合、デフォルトの構成ファイル「system.dce」が使用されます。

Return

通信オブジェクトへのポインター。

このポインターは他の関数では *pcom* として表示できます。

備考

(1) ユーザーは複数の通信オブジェクトを開くことができます。関数 setComPar を使用すると、各オブジェクトは異なるポートに設定されます。返される各 MDCE ポインターは、他の関数の最後のパラメーターであることが多く、通信オブジェクトの識別子です。

(2) セッションが 1 つしかない場合、ユーザーは返された MDCE ポインターを無視し、他の関数で pcom=NULL を設定できます。

(3) 関数 openDCE を呼び出した後、関数 setComPar を呼び出して通信ポートのパラメーターを定義します。関数 openDCE は、通信オブジェクトごとに 1 回だけ呼び出す必要があります。

(4) 設定ファイルのフルパス名を取得するには、関数 getDceCnfFname を呼び出します。

(5) 通信オブジェクトを閉じるには（おそらくアプリケーションを終了する前に）、関数 deleteDCE を呼び出します。

(6) ドライバーの種類によって異なる設定ファイルを採用する必要があります。表 2.1.1 は D シリーズドライバー用、表 2.1.2 は E1 シリーズドライバー用です。

表 2.1.1

D シリーズドライバー		
ドライバータイプ		path in openDCE
D2T	D2T-□□□□-S	openDCE("C:\\HIWIN\\dce", "d2.dce");
	D2T-□□□□-F	
	D2T-□□□□-E	openDCE("C:\\HIWIN\\dce", "D2COE.dce");
D1	D1-□□-S	openDCE("C:\\HIWIN\\dce", "tamuz.dce");
	D1-□□-F	
	D1-□□-E	openDCE("C:\\HIWIN\\dce", "D1COE.dce");
D1-N	D1-N-□□-S	openDCE("C:\\HIWIN\\dce", "D1N.dce");
	D1-N-□□-F	
	D1-N-□□-M	
	D1-N-□□-E	openDCE("C:\\HIWIN\\dce", "D1NCOE.dce");

表 2.1.2

E1 シリーズドライバー		
Thunder software version	ドライバータイプ	path in openDCE
1.5.10.0 またはそれ以上	ED1S	openDCE("C:\\Thunder\\dce", "D3.dce");
	ED1F	openDCE("C:\\Thunder\\dce", "D3COE.dce");
1.5.10.0 より下	ED1S	openDCE("C:\\HIWIN\\dce", "D3.dce");
	ED1F	openDCE("C:\\HIWIN\\dce", "D3COE.dce");

2.2 deleteDCE

目的

PC とドライバー間の通信を終了します。

プロトタイプ

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

パラメーター

pcom	通信オブジェクトへのポインター
closeMFC	ユーザーがアプリケーションの先頭で関数 setMFCmode を呼び出したときにのみアクティブになる MFC 内部スレッドをクリーンアップするには、これを TRUE に設定します。ほとんどの場合、DLL が閉じられると MFC 内部スレッドはいずれにせよクリーンアップされるため、これを FALSE に設定する必要があります。

Return

N/A

備考

これ以上の通信が必要なくなったとき（おそらくアプリケーションを終了する前）にこの関数を呼び出します。

2.3 ver

目的

DLL バージョンを取得します。

プロトタイプ

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

パラメーター

print DLL のバージョンをメッセージボックスに表示するには、ゼロ以外の値に設定します。

Return

DLL バージョンへのポインター。

備考

DLL は随時更新される可能性があるため、アプリケーションでこの機能を使用して DLL のバージョンを定期的に確認することをお勧めします。

2.4 disErrMsgBox

目的

通信エラーのメッセージ ボックスを無効にします。
(バージョンの競合が検出された場合、この機能はバージョン比較ウィンドウを無効にすることはできません。)

プロトタイプ

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

パラメーター

dis	通信エラーのメッセージ ボックスを無効にするには 1 に設定し、有効にするには 0 に設定します。

Return

N/A

備考

(1) この関数は、メッセージボックスを無効/有効にするためにいつでも呼び出すことができます。
(2) PC とサーボドライブ間で初めてデータベース検証を行う場合は、setComPar 関数が呼び出される前にこの関数を呼び出してメッセージボックスを無効にします。
(3) バージョンの競合が検出された場合、バージョン比較ウィンドウがポップアップ表示されます。この場合、この機能ではウィンドウを無効にすることはできません。
(4) この機能を使用すると、全てのスレーブで利用可能となる。

2.5 disAllErrMsgBox

目的

通信エラーのメッセージ ボックスを無効にします。
(バージョンの競合が検出された場合、この機能によりバージョン比較ウィンドウを無効にすることができます。)

プロトタイプ

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

パラメーター

dis	通信エラーのメッセージ ボックスを無効にするには 1 に設定し、有効にするには 0 に設定します。

Return

N/A

備考

(1) この関数は、メッセージボックスを無効/有効にするためにいつでも呼び出すことができます。
(2) PC とドライバー間で初めてデータベース検証を行う場合は、setComPar 関数が呼び出される前にこの関数を呼び出してメッセージボックスを無効にします。
(3) バージョンの競合が検出されると、バージョン比較ウィンドウがポップアップ表示されます。関数 disErrMsgBox とは異なり、この関数はウィンドウを無効にすることができます。
(4) この機能を使用すると、全てのスレーブで利用可能となる。

2.6 getFwErr

目的

PC とドライバーファームウェアの間にバージョンの競合がないか確認します。

プロトタイプ

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

パラメーター

pcom 通信オブジェクトへのポインター

Return

PC とドライバーファームウェア間のバージョンの競合が検出された場合は、ゼロ以外の値を返します。

備考

バージョンの比較は、通信が最初にセットアップされたとき、または関数 compareFw が呼び出されたときに実行されます。

2.7 getVerErrCodeSI

目的

詳細なバージョンエラーコードを取得します。

プロトタイプ

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

パラメーター

slave スレーブ番号

pcom 通信オブジェクトへのポインター。

備考

詳細なバージョンエラーコード。

各 2 ビットは各ファイル データベースのエラー ステータスを表します。

0 – エラーはありません。

1 – バージョンの比較を実行できませんでした。

2 – バージョンの競合があります。

ビットとファイルの関係は以下のようになります。

ビット	ファイル
0, 1	BOOT ファームウェア ファイル *.edb
4, 5	MDP ファームウェア ファイル <appl name>.edb
6, 7	MDP 変数リスト ファイル <appl name>.vrs
8, 9	PDL ファームウェアファイル<name>.edb
10, 11	PDL MDP 変数リスト ファイル <user_$>.vrs
12, 13	PDL MDP 関数リスト ファイル <user_$>.lbl
14, 15	PDL MDP メッセージ リスト ファイル <user_$>.msg

注: ドライバーの種類によって、異なる構成ファイルを使用する必要があります。表 2.1.1 と表 2.1.2 を参照してください。

ドライバーモデル「D2T-□□□□-S」を例にとると、<appl name> は <d2> になります。

2.8 compareFw

目的

PC とドライバーのファームウェアのバージョン比較を実行します。

プロトタイプ

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

パラメーター

show	1 に設定するとバージョン比較ウィンドウが開きます（すべてのファームウェアが一致している場合でも）
pcom	通信オブジェクトへのポインター。

Return

N/A

備考

(1) バージョンの競合が検出されると、バージョン比較ウィンドウがポップアップ表示され（関数 disErrMsgBox が事前に呼び出されていない場合）、関数 getFwErr を呼び出した後、ゼロ以外の値が返されます。
(2) 通信開始時にバージョン比較が行われます。

2.9 setMFCmode

目的

メイン スレッドからではない関数 showcomstatus と OpenRecord を呼び出します。

プロトタイプ

```
void setMFCmode();
```

パラメーター

N/A

Return

N/A

備考

(1) 他の関数を呼び出す前にこの関数を呼び出します。この関数は一度だけ呼び出す必要があります。
(2) 関数 showcomstatus と OpenRecord がメインスレッドからのものである場合、ユーザーはこの関数を呼び出す必要はありません。

2.10 resetController

目的
ドライバーをリセットします。

プロトタイプ

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

パラメーター

slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return
N/A

備考
この関数を呼び出すことは、ハードウェアをリセットすることと似ています。最初にブート モードになり、その後通常モードに戻ります。

3. コミュニケーション

この章では、通信設定に関連するすべての機能について説明します。

3.1 setComPar

目的

通信パラメーターを設定します。

プロトタイプ

```
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);
```

パラメーター

port　ポート番号

baudrate　USB モード経由の RS232 のボーレートコード

Code	0	1	2	3	4	5	6
Baud rate	1200	2400	4800	9600	19200	38400	56000
Code	7	8	9	11	12	13	
Baud rate	57600	115200	128000	230400	460800	921600	

mode　通信モードのコード

Code	1	4	5
Mode	RS232 via USB	EtherCAT mega-ulink	USB (E1 シリーズドライバー専用)

trid　予約済み。0 に設定

rcid　予約済み。0 に設定

canbaudrate　予約済み。0 に設定

msgStand　予約済み。0 に設定

canpipelevel　予約済み。0 に設定

timeout　ドライバーがメッセージを送り返すのを待つ時間（100 が推奨）
単位: msec

locktime　通信エラーが発生するまでの時間（200 を推奨）
単位: msec

iternum　PC がドライバーにメッセージを繰り返し送信する最大回数
(6 を推奨)

pcom　通信オブジェクトへのポインター

Return

0 – 関数は成功しました

2 – ポートが存在しません

5 – ポートは他のアプリケーションによって占有されています。

備考

この関数は、関数 openDCE が呼び出された後に呼び出す必要があります。その後は、通信設定を変更するためにいつでも呼び出すことができます。

3.1.1 EtherCAT mega-ulink に接続する

EtherCAT mega-ulink に接続するには、mode = 4 で関数 setComPar を呼び出します。

例 1

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

備考

(1) 関数 openDCE の前に関数 setMFCmode を呼び出します。

(2) 関数 setComPar では、関係のないパラメーターはすべて 0 で、3 番目のパラメーターmode は 4 です。

(3) 次の図に示すように、PC で使用するネットワーク デバイスを手動で選択します。SetUp… ボタンを押して wizalg Communication Setup ウィンドウを開き、EtherCat… ボタンを押して目的のネットワークを選択します。その後、Apply ボタンを押して設定を保存します。ユーザーが EtherCAT mega-ulink に接続するたびに設定を復元できます。

(4) EtherCAT mega-ulink は Windows 11 以降のオペレーティングシステムをサポートしていません。

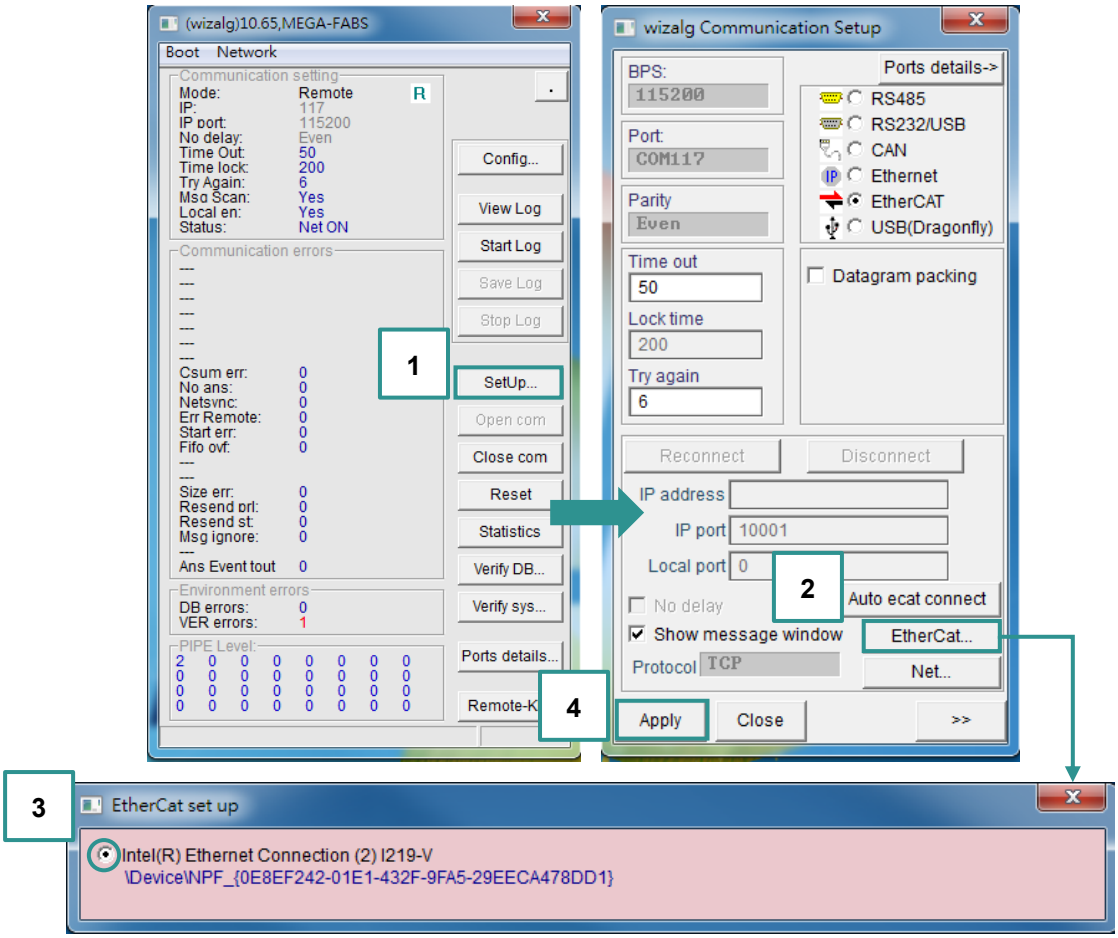

(wizalg)10.65,MEGA-FABS
Boot Network
Communication setting
Mode: Remote
IP: 117
IP port: 115200
No delay: Even
Time Out: 50
Time lock: 200
Try Again: 6
Msg Scan: Yes
Local en: Yes
Status: Net ON
Communication errors
Csum err: 0
No ans: 0
Netsync: 0
Err Remote: 0
Start err: 0
Fifo ovf: 0
Size err: 0
Resend prl: 0
Resend st: 0
Msg ignore: 0
Ans Event tout 0
Environment errors
DB errors: 0
VER errors: 1
PIPE Level:
Config...
View Log
Start Log
Save Log
Stop Log
SetUp...
Open com
Close com
Reset
Statistics
Verify DB...
Verify sys...
Ports details...
Remote-K
1
wizalg Communication Setup
BPS: 115200
Port: COM117
Parity Even
Time out 50
Lock time 200
Try again 6
Ports details->
RS485
RS232/USB
CAN
Ethernet
EtherCAT
USB(Dragonfly)
Datagram packing
Reconnect
Disconnect
IP address
IP port 10001
Local port 0
No delay
Show message window
Protocol TCP
Auto ecat connect
EtherCat...
Net...
Apply
Close
>>
2
4
3
EtherCat set up
Intel(R) Ethernet Connection (2) I219-V
\Device\NPF_{0E8EF242-01E1-432F-9FA5-29EECA478DD1}

図 3.1.1.1

例 2

```
char szNetworkAdapterName[NETWORK_ADAPTER_STRING_SIZE];
char szNetworkAdapterDescription[NETWORK_ADAPTER_STRING_SIZE];
char szNetworkAdapterErrorMessage[NETWORK_ADAPTER_ERROR_STRING_SIZE];

setMFCmode();
MDCE *pcom=openDCE("c:\\HIWIN\dce\\", NULL);
//   Get the name and the description of the first network interface card in PC
getNetworkAdapterInfo(0, szNetworkAdapterName, szNetworkAdapterDescription,
                                    szNetworkAdapterErrorMessage);
//   Set the connection with the name of the first network interface card in PC
setNetworkAdapter(pszNetworkAdapterName, pcom);
setComPar(0, 0, 4, 0, 0, 0, 0, 0, 50, 0, 8, pcom);
```

備考

(1) 関数 openDCE の前に関数 setMFCmode を呼び出します。

(2) 関数 setComPar では、関係のないパラメーターはすべて 0 で、3 番目のパラメーターmode は 4 です。

(3) 関数 setComPar の前に関数 setNetworkAdapter を呼び出すことで、ユーザーは手動で選択することなくネットワークインターフェースカードを割り当てることができます。

3.1.2 USB に接続する

USB に接続するには、mode = 5 で関数 setComPar を呼び出します (E1 シリーズドライバーのみ)。

例: 2 つの USB デバイスに接続する

```
setMFCmode();
int port0 = 0, port1 = 1;
//   Open communication object for the corresponding communication port
MDCE *pcom_0 = openDCE("c:\\HIWIN\dce\\", NULL);
MDCE *pcom_1 = openDCE("c:\\HIWIN\dce\\", NULL);

setComPar(port0, 0, 5, 0, 0, 0, 0, 0, 50, 0, 8, pcom_0);   //   Connect to port0
setComPar(port1, 0, 5, 0, 0, 0, 0, 0, 50, 0, 8, pcom_1);   //   Connect to port1
```

備考

最初に USB ドライバーをインストールする必要があります。USB ドライバーは ARM アーキテクチャをサポートしていないため、ARM アーキテクチャ上の Windows では接続が正常に行われない可能性があります。

3.2 getComPar

目的

関数 setComPar によって設定されたパラメーターを取得します。

プロトタイプ

```
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);
```

パラメーター

これらの引数はポインターであり、関数 setComPar の引数と同じパラメーターです。

Return

0 – 関数は成功しました。

-1 – 通信オブジェクトがありません。

3.3 getDceCnfFname

目的
設定ファイルのフルパス名を取得します。

プロトタイプ

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

パラメーター

*str	設定ファイルのフルパス名
pcom	通信オブジェクトへのポインター

Return
0 – 関数は成功しました
-1 – 通信オブジェクトがありません。

備考
str と *pcom* は、返される文字列の長さに応じて十分なサイズ (200 を推奨) の文字列バッファを指す必要があります。

3.4 getSlaveFname

目的

スレーブ名を取得します。

プロトタイプ

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

パラメーター

*str	スレーブ名
slave	スレーブ番号 ドライバーが 1 つしかない場合は 0 になります。
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

-1 – 通信オブジェクトがありません。

3.5 getUsbHubPort

目的

USB デバイス名を取得します。

プロトタイプ

```
int getUsbHubPort(int port, char *hubport, int strlen, MDCE *pcom=NULL);
```

パラメーター

port	ポート番号 USB デバイスが 1 つしかない場合は、0 に設定します。
*hubport	USB デバイス名
strlen	USB デバイス名の文字列の長さ
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr (*str* を "" に設定) を呼び出します。

備考

(1) この機能は E1 シリーズドライバーでのみ使用できます。

(2) この関数は関数 openDCE が呼び出された後に呼び出されなければなりません。

(3) この機能により、ユーザーは PC に接続されたすべてのドライバーを取得できます。

例

```
#define MaxSlaveNum 32
MDCE *pCom[MaxSlaveNum];    //    Up to 32 connected slaves
int port=0;
char hubport[200];
int errcode;

//    Scan USB device
for (int i=0;i<MaxSlaveNum;i++)
{
        pCom[i]=openDCE("c:\\HIWIN\dce\\", NULL);    //    Open communication object
        if ((errcode=getUsbHubPort(port, hubport, sizeof(hubport), pCom[port]))==0)
        {
                printf("Device %d: %s\n", port, hubport);
                port++;
        }
        else
        {
                //    Print error message
                char errstr[200];
                GetErrorStr(errcode, " ", 0, errstr, pCom[port]);
                printf("Device %d: %s\n", port, errstr);
                break;
        }
}
```

3.6 closePort

目的
ポートを閉じます。

プロトタイプ

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

パラメーター
pcom 通信オブジェクトへのポインター

Return
N/A

備考
この関数は、通信オブジェクトを削除する代わりに (関数 deleteDCE のように)、ポートを閉じるだけです。

3.7 openPort

目的

最後の setComPar で指定されたポートを再度開きます。

プロトタイプ

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

パラメーター

pcom 通信オブジェクトへのポインター

Return

0 – 関数は成功しました。
2 – ポートが存在しません。
5 – ポートは他のアプリケーションによって占有されています。

備考

(1) ユーザーは setComPar 関数を呼び出す代わりにこの関数を呼び出すことができます。この場合、最後の通信設定が適用されます。
(2) ユーザーは通信状態ダイアログボックスを介して通信設定を変更することができます（関数 showcomstatus を参照）。

3.8 showcomstatus

目的

ダイアログ ボックスを開きます。このダイアログ ボックスには、通信設定の情報と関連する統計情報が表示されます。ダイアログ ボックスでは、COM ポートを閉じる/開く、ブート ダイアログ ボックス (PDL および MDP) を開く、ネットワークにアクセスするなど、複数のタスクを実行できます。

プロトタイプ

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

パラメーター

pcom 通信オブジェクトへのポインター。

Return

N/A

備考

アプリケーションの実行中、ダイアログ ボックスはアクティブなままにすることも、最小化することもできます。

3.9 loadDceSw

目的

MDP や PDL など、ドライバーの関連ソフトウェアをすべて更新します。

プロトタイプ

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

パラメーター

pdlComp　PDL コンパイルを実行するには、ゼロ以外の値に設定します。それ以外の場合は、0 に設定します。

pcom　通信オブジェクトへのポインター。

Return

関数が成功した場合は int 値 0 を返します。それ以外の場合は、読み込みプロセスで発生したエラーの数を返します。

3.10 updateDataBase

目的

DLL 内部データベースを更新します。

プロトタイプ

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

パラメーター

pcom 通信オブジェクトへのポインター。

Return

N/A

備考

DLL の内部変数/関数リストを更新するには、アプリケーションを閉じて開く代わりに、この関数を呼び出します。

3.11 setNetworkAdapter

目的

PC のネットワークインターフェースカードの 1 つを設定します。

プロトタイプ

```
int setNetworkAdapter(char *pszNetworkAdapterName, MDCE *pcom=NULL);
```

パラメーター

*pszNetworkAdapterName	ネットワーク インターフェイス カードの名前。文字列の長さは 250 文字です。
pcom	通信オブジェクトへのポインター。

Return

0 –関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

(1) setComPar 関数の前にこの関数を呼び出すことで、ユーザーは手動で選択することなくネットワークインターフェースカードを割り当てることができます。

(2) 関数 getNetworkAdapterInfo を呼び出して、ネットワークインターフェースカードの名前を取得します。

3.12 getNetworkAdapterCount

目的

PC 内のすべてのネットワークインターフェースカードの数を取得します。

プロトタイプ

```
int getNetworkAdapterCount(int *iCount, char *pszErrorMessage);
```

パラメーター

*iCount	PC 内のすべてのネットワークインターフェースカードの数。
*pszErrorMessage	PC 内のすべてのネットワークインターフェースカードの数を取得するプロセス中に返されるエラーメッセージ。文字列の長さは 256 文字です。

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

3.13 getNetworkAdapterInfo

目的

PC 内のネットワークインターフェースカードの 1 つの名前と説明を取得します。

プロトタイプ

```
int getNetworkAdapterInfo(unsigned int iIndex, char *pszNetworkAdapterName,
                          char *pszNetworkAdapterDescription,
                          char *pszErrorMessage);
```

パラメーター

iIndex	ネットワークインターフェースカードのインデックス； その最大値は関数 getNetworkAdapterCount によって取得できます。
*pszNetworkAdapterName	ネットワークインターフェースカードの名前； 文字列の長さは 250 文字です。
*pszNetworkAdapterDescription	ネットワークインターフェースカードの説明； 文字列の長さは 250 文字です。
*pszErrorMessage	ネットワーク インターフェイス カードの情報を取得するプロセス中に返されたエラー メッセージ。文字列の長さは 256 文字です。

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

(このページはブランクになっています)

4. 受信したイベント

4.1 waitOnMsgP

目的

ドライバー上で実行されている PDL プログラムから送信された PDL printl コマンドを受信します。PDL printl コマンドの構文は次のとおりです。

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

この構文では、mode2 はオプション (指定しない場合は 0 になります)、mode1 は文字列の色やその他の属性を指定します。var1…varN はオプションの変数です。詳細については、「D シリーズドライバーの PDL リファレンスマニュアル」を参照してください。

プロトタイプ

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

パラメーター

*pcode　モード 2 の値。

*pmsg　println コマンド内の文字列。サイズは少なくとも 200 文字である必要があります。

*pcolor　mode1 で指定されたカラー コードは、COLORREF の対応する値に変換されます。

*pparam　配列。配列の最初の項目は変数の数を表し、次の項目はそれぞれ各変数の 32 ビット値を示します。float 型の変数の場合は、整数ではなく float に昇格する必要があります。
注意: 文字列 (pmsg) 内の変数タイプは、printl コマンド内の変数タイプと対応している必要があります。

*pnumovrflw　失われた printl メッセージの数。たとえば、関数 waitOnMsgP をリッスンしているスレッドがないのに、大量の printl メッセージが来ると、オーバーフローが発生する可能性があります。関数 waitOnMsgP が再度リッスンされると、オーバーフロー カウンターはリセットされます。
注: オーバーフローが発生した場合、受信メッセージの属性 (pcode、pmsg、pcolor、pparam) は有効です。

timeout　printl メッセージを取得するための時間。timeout = -1 (INFINITE を意味する) の場合、関数は printl メッセージが送信されるまで (または関数 releaseWaitMessage が他のスレッドによって呼び出されるまで) 値を返しません。timeout = 0 の場合、関数は利用可能なメッセージが存在する場合はすぐに 0 を返し、利用可能なメッセージが存在しない場合は -2 を返します。timeout が他の正の値 (timeout > 0) の場合、関数は設定された時間 (単位: ミリ秒) の間待機します。この場合、メッセージがない場合、または関数 releaseWaitMessage が呼び出されると、関数は 5 を返します。

pcom　通信オブジェクトへのポインター。

Return

0 – ok.

-1 – 通信オブジェクトがありません。

-2 – 利用可能なメッセージがありません (関数 releaseWaitMessage が呼び出されるか、タイムアウト = 0)、または複数のスレッドがこの関数を呼び出します。

5 – タイムアウト。

備考

(1) ポインターの内容を無視するには、NULL に設定します。例えば、mode2 の値を気にしない場合は、*pcode = NULL に設定します。

(2) この関数は一度に 1 つのスレッドからのみ呼び出すことができます。それ以外の場合、-2 が返され、メッセージが失われる可能性があります。

(3) スレッドがドライバーから送信されたメッセージを迅速に処理できない場合、オーバーフローが発生する可能性があります。mpi.dll の FIFO サイズは 200 です。

例 1

PDL プログラムは次の printl コマンドを実行します。(x_enc_pos = 2000、x_p_p_g = 0.01 と仮定)

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

waitOnMsgP 関数をリッスンしているスレッドは 0 (ok) を返し、パラメーターは以下のように入力されます。

```
*pcode=0x44;
pmsg="x axis: enc_pos=2000, pos loop gain=0.01";
*pcolor=0x02ff0000;   //   103 in mode1 will be converted to 0x02ff000 in COLORREF:
                      //   color blue
pparm[0]=2; pparm[1]=2000; *((float *)&pparam[2])=0.01;
*pnumovrflw=0   //   Assume no overflow
```

例 2

関数 waitOnMsgP はループ内のスレッドによって実行されます (変数 endMsgTread が設定されるまで)。

```
int endMsgTread=0;
Uint msgThrd(LPVOID pPar)
{
    MDCE *pcom=(MDCE*)pPar;   //   pointer to communication object
    int st, code, c, numOvrFlow, param[10];
    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 further return value
            sprintf(str, "Message return error %ld", st);
            break;
        }
    }
return(0);
}
```

4.2 releaseWaitMessage

目的

waitOnMsgP 関数をリッスンしているスレッドを解放します。

プロトタイプ

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

パラメーター

pcom 通信オブジェクトへのポインター。

Return

N/A

備考

(1) アプリケーションを終了する前、または deleteDCE 関数を呼び出す前に、この関数を呼び出します。
(2) waitOnMsgP 関数で timeout = 0 の場合、ユーザーはこの関数を呼び出す必要はありません。

例

この関数を適用するには、セクション 4.1 の例 2 の Uint msgThrd に次のステートメントを追加します。

```
endMsgTread = 1;   //   Exit while loop
releaseWaitMessage(pcom);   //   Release the thread listening to function waitOnMsgP
……
```

4.3 insertEvent

目的

イベント ID を使用して、多数の PDL printl コマンドから監視する特定のメッセージをふるいにかけます。

プロトタイプ

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

パラメーター

id	イベント ID; 範囲: 0~199 これは PDL printl コマンドの var1 の値と同じである必要があります (説明については関数 waitOnMsgP を参照)。そのため、イベント ID と var1 の値の両方をスレーブ番号として設定することをお勧めします。
h	Windows (32 ビット) の関数 CreateEvent によって作成されたイベントハンドル。 NULL に設定された場合、イベントはクリアされます。
pcom	通信オブジェクトへのポインター。

Return

0 –関数は成功しました。
-1 – 関数 openDCE はまだ呼び出されていません。
-2 – イベント ID が範囲外です。

備考

この関数を呼び出す前に、まず関数 setEventCode を呼び出します。

4.4 closeEvent

目的

イベントハンドルを閉じます。

プロトタイプ

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

パラメーター

id	イベント ID; 範囲: 0~199 これは PDL printl コマンドの var1 の値と同じである必要があります (説明については関数 waitOnMsgP を参照)。そのため、イベント ID と var1 の値の両方をスレーブ番号として設定することをお勧めします。
pcom	通信オブジェクトへのポインター。

Return

0 – 関数は成功しました。

-1 – 関数 openDCE はまだ呼び出されていません。

-2 – イベント ID が範囲外です。

-3 – 関数が失敗しました。

4.5 getEvent

目的

イベントハンドルを取得します。

プロトタイプ

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

パラメーター

id	イベント ID; 範囲: 0~199 これは PDL printl コマンドの var1 の値と同じである必要があります (説明については関数 waitOnMsgP を参照)。そのため、イベント ID と var1 の値の両方をスレーブ番号として設定することをお勧めします。
pcom	通信オブジェクトへのポインター。

Return

イベントハンドル

-1 – 関数 openDCE はまだ呼び出されていません。

-2 – イベント ID が範囲外です。

4.6 setEventCode

目的

イベントコードを設定します。フィルターとして表示できるイベントコードは、多数の PDL printl コマンドから監視する特定のメッセージをふるいにかけます。

プロトタイプ

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

パラメーター

code	PDL printl コマンドの mode2 の値 (説明については関数 waitOnMsgP を参照)。デフォルトは 1 です。
pcom	通信オブジェクトへのポインター。

Return

0 – 関数は成功しました。

-1 – 関数 openDCE はまだ呼び出されていません。

備考

PDF printl コマンドのモード 2 の値は定数であり、PDL コンパイル時に定義されるため、動的に変更することはできません。

4.7 getLastEventData

目的

最後に呼び出されたイベントコードの文字列と変数を取得します。

プロトタイプ

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

パラメーター

id	イベント ID; 範囲: 0~199 これは PDL printl コマンドの var1 の値と同じである必要があります (説明については関数 waitOnMsgP を参照)。そのため、イベント ID と var1 の値の両方をスレーブ番号として設定することをお勧めします。
*pmsg	printl コマンド内の文字列。サイズは少なくとも 200 文字である必要があります。
*pparam	配列。配列の最初の項目は変数の数を表し、次の項目はそれぞれ各変数の 32 ビット値を示します。float 型の変数の場合は、整数ではなく float に昇格する必要があります。 注意: 文字列 (pmsg) 内の変数タイプは、printl コマンド内の変数タイプと対応している必要があります。
pcom	通信オブジェクトへのポインター。

Return

0 – 関数は成功しました。
-1 – 関数 openDCE はまだ呼び出されていません。
-2 – イベント ID が範囲外です。

備考

ポインター (*pmsg、*pparam) の内容を無視するには、NULL に設定します。

例

```
char reportStr[200];
int code=0x00000055;
int id=1;
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 PDL printl 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, pcom);    //    Close and clear the event
```

この例では、PDL プログラムは printl コマンドを実行して、mode2 の値が 0x55 であり、スレーブ番号 (イベント ID) が 1 であるイベントをトリガーする必要があります。

```
#long Z_id, X_id, Y_id;
Z_id=0; X_id=1; Y_id=2;
……
printl/103/00000055("axis id=%ld; x axis: enc_pos=%ld, velocity=%g",
X_id, x_enc_pos, x_vel_max);
```

5. データ収集（記録）

5.1 StartRecordData

目的

記録を開始します。

プロトタイプ

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

パラメーター

*p[]	1 つ以上の変数名を含む配列； そのサイズは少なくとも numVar の値である必要があります。
numVar	記録する変数の数。
rate	サンプルレートを 32000/rate Hz と定義します (ドライバーのサンプルレートが 32000 であると想定)。したがって、各変数は「rate*0.00003125」秒ごとにサンプリングされます。
numSamples	各変数に記録される最大サンプル数。
*pdata	変数のデータを格納する配列。変数が複数ある場合、変数 1 のサンプルがすべて記録された後に、変数 2 のサンプルが記録されます。したがって、pdata[0] は変数 1 の最初のデータ、pdata[numSamples] は変数 2 の最初のデータ、pdata[n*numSamples] は変数 n (n = 0 … numVar-1) の最初のデータです。したがって、そのサイズは少なくとも numVar * numSamples である必要があります。
*errstr	エラーの説明。
pcom	通信オブジェクトへのポインター。

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明は *errstr から取得します。

備考

(1) この関数は、すぐに戻り値を返します。戻り値が 0 (関数が成功) の場合、記録プロセスが正常に開始されたことのみを示します。記録プロセスが終了したかどうかは不明です。関数 GetRecdordStatus を呼び出すと、記録プロセスが終了したかどうか、および収集されたサンプルの数を知ることができます。

(2) 最終的に記録されるサンプル数は、namSamples の定義より少なくなる場合があります。これは、レートが小さすぎる場合に発生する可能性があります。（通信速度と比較して、実際の記録頻度が速すぎる場合）。ただし、正常に収集されたデータは有効です。

(3) 連続的に記録するには（numSamples まで）、rate >= 8*numVar に設定することをお勧めします。

(4) 記録を停止するには、関数 StopRecord を呼び出します。

(5) 記録ウィンドウを開くには、OpenRecord 関数を呼び出します。ユーザーは、StartRecordData 関数で設定されたすべてのパラメーターとその他のリアルタイム記録情報を確認できます。

5.2 StartRecordDataN

目的

記録を開始します。

プロトタイプ

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

パラメーター

*varnames	1 つ以上の変数名を含む識別子。 これらの変数名はスペースまたはカンマで区切る必要があります。
rate	サンプル レートを 32000/rate Hz と定義します (ドライバーのサンプル レートが 32000 であると想定)。したがって、各変数は「rate*0.00003125」秒ごとにサンプリングされます。
numSamples	各変数に記録される最大サンプル数。
*pdata	変数のデータを格納する配列。変数が複数ある場合、変数 1 のサンプルがすべて記録された後に、変数 2 のサンプルが記録されます。したがって、pdata[0] は変数 1 の最初のデータ、pdata[numSamples] は変数 2 の最初のデータ、pdata[n*numSamples] は変数 n (n = 0 … numVar-1) の最初のデータです。したがって、そのサイズは少なくとも numVar * numSamples である必要があります。
*errstr	エラーの説明
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明は *errstr から取得します。

備考

これは関数 StartRecordData に似ていますが、違いは numVar がキャンセルされ、*p[] が *varnames に置き換えられることです。

例

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

5.3 StartRecordFileN

目的

記録を開始し、結果を特定のファイルに保存します。

プロトタイプ

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

パラメーター

*varnames	1 つ以上の変数名を含む識別子。 これらの変数名はスペースまたはカンマで区切る必要があります。
rate	サンプル レートを 32000/rate Hz で定義します (ドライバーのサンプル レートが 32000 であると想定します)。したがって、各変数は「rate*0.00003125」秒ごとにサンプリングされます。
numSamples	各変数に記録される最大サンプル数。
*filename	ファイル名
*errstr	エラーの説明。
append	結果を既存のファイルに保存するには、1 に設定します。 結果を新しいファイルに保存するには、0 に設定します。
pcom	通信オブジェクトへのポインター。

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明は *errstr から取得します。

備考

(1) StartRecordDataN 関数に似ていますが、結果が配列ではなく特定のファイルに保存される点が異なります。

(2) 結果は gpp ファイルにも保存されます。したがって、以下の例では、結果は「C:¥¥myresult.txt」と「C:¥¥myresult.gpp」に保存されます。
ユーザーは wingraph ツールを使用して gpp ファイルを表示できます。コマンドラインから操作するには、次のように入力します: >wingraph myresult.gpp

例

```
StartRecordFileN("X_ref_pos, X_pcmd_err", 16, 5000, "C:\\myresult.txt", pdata,
                 errstr, 0);
```

5.4 GetRecdordStatus

目的

記録プロセスが終了したかどうか、および収集されたサンプルの数を確認します。

プロトタイプ

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

パラメーター

*pnumSampColect　すでに収集されたサンプルの数。
pcom　通信オブジェクトへのポインター。

Return

0 – 記録処理が終了します。
2 – まだ進行中です。
3 – ファイルの保存に失敗しました。
-1 – 関数 StartRecordData はまだ呼び出されていません。

備考

ユーザーはループ内でこの関数を呼び出し、*pnumSampColect を印刷してデータ収集プロセスを表示できます。

5.5 StopRecord

目的

記録を停止します。

プロトタイプ

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

パラメーター

pcom　　通信オブジェクトへのポインター。

Return

N/A

備考

(1) 記録処理がすでに終了している場合は、この関数を呼び出しても何も起こりません。

(2) この関数を呼び出した後に新しいレコード処理を開始するには、まず GetRecdordStatus 関数を呼び出して、前のレコード処理が完全に完了したことを確認します。

例

```
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) {
        //    Fail to save file
}
else if(cnt>=2000) {
```

```
        //    Time out
}
s=StartRecordFileN(str, rate, numsampreq, "C:\\file2.txt", errstr, 0, pcom);
```

5.6 OpenRecord

目的

記録ウィンドウを開きます。

プロトタイプ

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

パラメーター

show	記録ウィンドウを開くには 1 に設定します。
pcom	通信オブジェクトへのポインター。

Return

N/A

備考

(1) このウィンドウでは、記録する 1 つ以上の変数、サンプル数、レートを定義できます。設定が完了したら、開始 (F5) ボタンをクリックして記録を開始し、グラフ ボタンをクリックして結果を表示します (次の図を参照)。

(2) 関数 StartRecordData が呼び出されると、このウィンドウ内の変数は関数によって指定された変数に従って更新されます。

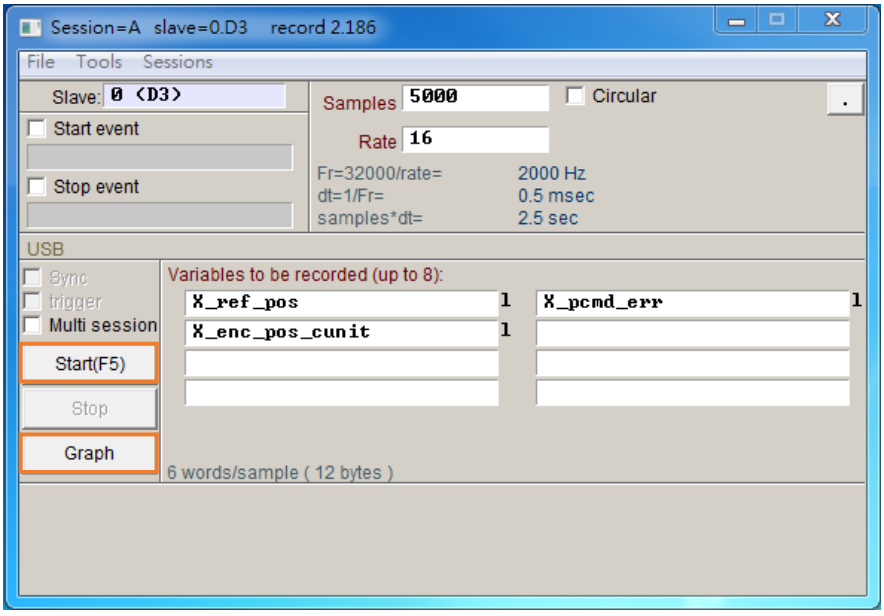


図 5.6.1

(このページはブランクになっています)

6. 変数/配列にアクセスする

この章のほとんどの関数は、関数が成功すると int 値 0 を返します。ゼロ以外の値が返された場合は、関数 GetErrorStr を呼び出してエラーの説明を取得します。

すべての関数は変数名をパラメーターとして受け取ることができ、変数は定数でインデックス付けできます (おそらく配列内の 1 つの項目にアクセスするために使用されます)。例: SetVarN("rec_buf[300]", 15, 0, NULL);

6.1 GetVarAddr

目的
変数の存在とサイズを確認します。

プロトタイプ

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

パラメーター

*varName	変数名
slave	スレーブ番号
*psize	変数サイズ 配列以外の変数の場合は 1 を返します。配列変数の場合は配列のサイズを返します。 変数が見つからない場合は -1 を返します。
pcom	通信オブジェクトへのポインター。

Return
変数のアドレス
変数が存在しない場合は、ADDERR (-100) を返します。

備考
変数の型を取得するには、関数 GetVarAddrType を呼び出します。

6.2 GetVarAddrType

目的

変数の存在、サイズ、タイプを確認します。

プロトタイプ

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

パラメーター

*varName	変数名	
slave	スレーブ番号	
*psize	変数サイズ.	
	配列以外の変数の場合は 1 を返します。配列変数の場合は配列のサイズを返します。	
	変数が見つからない場合は-1 を返します.	
*ptype	変数タイプ.	
	1: short (16 ビット整数)	4: pointer (32-bit)
	2: f long (32 ビット整数)	8: 64-bit variable
	3: float (32-bit float)	0: Variable not found.
pcom	通信オブジェクトへのポインター	

Return

変数のアドレス

変数が存在しない場合は、ADDERR (-100) を返します。

6.3 SetVarN

目的

変数値を設定します (32 ビット)。

プロトタイプ

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

パラメーター

*varName	変数名
data	設定する変数値 ユーザーが float/long/short 型で値を設定すると、正しい double 型に自動的に変換されます。
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

6.4 GetVarN

目的

変数値を取得します (32 ビット)

プロトタイプ

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

パラメーター

*varName	変数名
*pdata	変数値 ユーザーが float/long/short 型で値を設定すると、正しい double 型に自動的に変換されます。
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

6.5 SetVarN64

目的

変数値を設定します (64 ビット)

プロトタイプ

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

パラメーター

*varName	変数名
data	設定する変数値 (64 ビット)
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

変数の型は 64 ビットである必要があります。そうでない場合、関数は失敗します。

6.6 GetVarN64

目的

変数値を取得します (64 ビット)

プロトタイプ

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

パラメーター

*varName	変数名
*pdata	変数値 (64 ビット)
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

変数の型は 64 ビットである必要があります。そうでない場合、関数は失敗します。

6.7 setArrayN

目的

配列を設定します（32 ビット）

プロトタイプ

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

パラメーター

*varName	配列名
*pdata	ドライバーにコピーされるアプリケーション内の配列
num	コピーする変数の数
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

6.8 getArrayN

目的

配列を取得します (32 ビット)

プロトタイプ

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

パラメーター

*varName	配列名
*pdata	アプリケーションにコピーされるドライバー内の配列
num	コピーする変数の数
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

6.9 setArrayDN

目的

配列を設定します（64 ビット）

プロトタイプ

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

パラメーター

*varName	配列名
*pdata	ドライバーにコピーされるアプリケーション内の配列
num	コピーする変数の数
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

これは関数 setArrayN に似ていますが、*pdata が指す配列が 64 ビットであるという点が異なります。

6.10 getArrayDN

目的
配列を取得します (64 ビット)

Prototype

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

パラメーター

*varName	配列名
*pdata	アプリケーションにコピーされるドライバー内の配列。
num	コピーする変数の数
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return
0 – 関数は成功しました。
ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考
これは関数 getArrayN に似ていますが、*pdata が指す配列が 64 ビットであるという点が異なります。

6.11 setArraySN

目的

配列を設定します（16 ビット）

プロトタイプ

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

パラメーター

*varName	配列名
*pdata	ドライバーにコピーされるアプリケーション内の配列
num	コピーする変数の数
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

これは関数 setArrayN に似ていますが、*pdata が指す配列が 16 ビットであるという点が異なります。

6.12 getArraySN

目的

配列を取得します（16 ビット）

プロトタイプ

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

パラメーター

*varName	配列名
*pdata	アプリケーションにコピーされるドライバー内の配列。
num	コピーする変数の数
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

これは関数 getArrayN に似ていますが、*pdata が指す配列が 16 ビットであるという点が異なります。

6.13 getPac

目的

一度に複数の変数値を取得します。

プロトタイプ

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

パラメーター

*varnames	1 つ以上の変数名 (最大 20 個) を含む識別子 これらの変数名はスペースまたはカンマで区切る必要があります。
*I	変数名に対応する変数の型を順番に格納する char 配列 : サイズは少なくとも 20 である必要があります。 1: short (16 ビット整数) 3: 浮動小数点数（32 ビット浮動小数点数） 2: long (32 ビット整数) 8: 64 ビット変数
*data	変数名に対応する変数値を順番に格納する配列 : そのサイズは変数の数によって異なります。
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

この関数内の変数を個別に読み取るには、関数 GetVarN または GetVarN64 を呼び出します。

例

```
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);
```

6.14 getAndSetArrayN

目的

1 つの関数で配列 (32 ビット) の読み取りと書き込みを行います。
getArrayN 関数と setArrayN 関数を個別に呼び出すよりも実行が高速です。

プロトタイプ

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

パラメーター

*varrep	読み取る配列名
numrep	読み取る変数の数；範囲は 0 から 124 のショートまたは 62 のロングです。
*parrrep	ドライバーにコピーされるアプリケーション内の配列
*varset	書き込む配列名
numset	書き込む変数の数。範囲は 0 から 124 のショートまたは 62 のロングです。
*parrset	アプリケーションにコピーされるドライバー内の配列。
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。
ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

parrrep と *parrset* は、ドライバーに適した同じ型 (float/long/short) の配列を指す必要があります。この関数は 64 バイトをサポートしていません。

6.15 getAndSetArrayDN

目的

1 つの関数で配列 (64 ビット) の読み取りと書き込みを行います。
getArrayDN 関数と setArrayDN 関数を個別に呼び出すよりも実行が高速です。

プロトタイプ

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

パラメーター

*varrep	読み取る配列名
numrep	読み取る変数の数；範囲は 0 から 124 のショートまたは 62 のロングです。
*parrrep	ドライバーにコピーされるアプリケーション内の配列
*varset	書き込む配列名
numset	書き込む変数の数；範囲は 0 から 124 のショートまたは 62 のロングです。
*parrset	アプリケーションにコピーされるドライバー内の配列
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。
ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

これは関数 getAndSetArrayN に似ていますが、違いは次のとおりです：
(a) 配列ポインターは double 型でなければなりません。
(b) ドライバー内の変数(*varrep および *varset) は float 型である必要があります。
(c) DLL は内部的にキャストを実行します。

6.16 GetScopeData

目的

同じ DSP フェーズで変数のグループと時間カウンターを読み取ります。

プロトタイプ

```
int GetScopeData(char *varnames, double *pdata, short *pfclk, int slave=0,
                 MDCE *pcom=NULL);
```

パラメーター

*varnames	記録される 1 つ以上の変数名を含む文字列 これらの変数名はスペースまたはカンマで区切る必要があります。
*pdata	変数のデータを含む配列：そのサイズは変数の数によって異なります
*pfclk	DSP サンプル レートをカウントし、データがサンプリングされた時間を表示するカウンター；ユーザーは、スコープ内の変数値を視覚的に描画できます。
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

(1) この機能はスコープに適用できます。

(2) この関数を 200 ミリ秒以内の遅延で継続的に呼び出します。そうでない場合は、エラーコード 300 を返します。したがって、この関数が最初に呼び出されたときは常にエラーコード 300 を返します（他のエラーがない場合）。

6.17 GetErrorStr

目的

エラーの説明を取得します。

プロトタイプ

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

パラメーター

errcode	変数/状態アクセス関数 (SetVarN、GetVarN、setArrayN など) からの戻り値
*str	変数/状態名
slave	スレーブ番号
*errstr	エラーの説明。サイズは少なくとも 200 バイトである必要があります。
pcom	通信オブジェクトへのポインター。

Return

N/A

備考

変数/状態アクセス関数の 1 つからゼロ以外の値が返された場合は、この関数を呼び出します。

6.18 SetVarUN

目的

変数値を設定します (符号なし、32 ビット)。

プロトタイプ

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

パラメーター

*varName	変数名
data	設定する変数値 ユーザーが double 型で値を設定すると、自動的に unsigned double 型に変換されます。
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

6.19 SendRAMtoFlash

目的

変数データをドライバーのメモリに保存します。

プロトタイプ

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

パラメーター

slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

5 – タイムアウト

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

(このページはブランクになっています)

7. アクセス状態

状態は 1 ビット変数 (ブール変数など) として指定され、デジタル入力/出力とドライバーの内部状態 (軸の実行、軸の位置エラー フラグなど) を表します。状態はステータス配列のビットです。

7.1 setStateN

目的
状態を設定します。

プロトタイプ

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

パラメーター

slave	スレーブ番号
*stateName	状態名
mode	設定する状態。「1」はオン、「0」はオフです。
pcom	通信オブジェクトへのポインター

Return
0 –関数は成功しました。
ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

7.2 getStateN

目的

状態を取得します。

プロトタイプ

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

パラメーター

slave	スレーブ番号
*stateName	状態名
*state	状態。「1」はオン、「0」はオフです。
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr を呼び出します。

備考

(1) 状態は DLL の内部テーブルから取得されるため、状態の読み取りは変数の読み取りよりもはるかに高速です。情報は通信リンクによって渡される必要はありません。

(2) 状態が変化すると、内部テーブルが自動的に更新されます。

(このページはブランクになっています)

8. PDL 関数を実行する

8.1 RunFuncPdlN

目的

ドライバーに新しいタスクを作成し、特定の PDL 関数を実行します。

プロトタイプ

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

パラメーター

*pdlName	PDL 関数名 PDL プログラム内の「_」（アンダースコア）で始まるラベルはすべてホストによって認識されます。
*errstr	エラーの説明 (GetErrorStr 関数を呼び出す必要はありません。)
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

この関数を実行するタスク ID（0～39）

-1 – 関数は失敗しました。*errstr からエラーの説明を取得します。

備考

特定の PDL 関数を持つタスクがすでに実行されている場合、新しいタスクは作成されず、既存のタスク ID が返されます。

8.2 KillTask

目的

タスクを強制終了/停止/続行します。

プロトタイプ

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

パラメーター

st	開始タスク ID
en	終了タスク ID
what	0 → タスクを強制終了する; 1 → タスクを停止する; 2 → タスクを続行する
slave	スレーブ番号
pcom	通信オブジェクトへのポインター

Return

0 – 関数は成功しました。

ゼロ以外の値 – 関数は失敗します。エラーの説明を取得するには、関数 GetErrorStr (str を "" に設定) を呼び出します。

備考

1 つのタスク n のみを終了するには、st = n および en = n+1 を設定します。ここで、n は関数 RunFuncPdlN から返されるタスク ID です。

(このページはブランクになっています)

9. コールバック関数

ユーザーは、この章の関数を呼び出すことによって、いつでも変更または無効化（パラメータ func を NULL に設定）できるコールバック関数を定義できます。

9.1 setCallBackStEvnt

目的

状態が変化すると、ユーザーに積極的に通知され、対応するプロセスが実行されます。

プロトタイプ

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

パラメーター

type	状態の種類
id	状態の ID
pcom	通信オブジェクトへのポインター

Return

N/A

備考

type と id は、ファイル <appl name.stt> 内の各状態に対して定義される状態属性です。状態が変更されると、関数 PutEventSt が呼び出され、type の最初の 2 ビットの少なくとも 1 つが 1 になります(これは、送信イベントの変更方向を定義します)。詳細については、「D シリーズドライバーの PDL リファレンスマニュアル」を参照してください。

例

```
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 DLL with pointer of callback function
```

10. エラーコード

エラーコードのリストは以下の通りです。

表 11.1

コード番号	説明
1	読み取りに失敗しました
2	書き込みに失敗しました
3	フレームエラー
4	返答なし
5	チェックサムエラー
6	サイズが範囲外です
7	パリティエラー
8	オーバーランエラー
9	バッファオーバーフロー
10	フレームエラー
11	アボート
12	ネット同期エラー
13	Can エラー
14	回答サイズエラー
15	開始バイトエラー
16	マスターブート
17	EtherCAT mega-ulink エラー
18	テストモードの EtherCAT mega-ulink
19	EtherCAT mega-ulink 通信なし
20	EtherCAT mega-ulink タイムアウト
21	USB エラー
31	スレーブ番号が範囲外です (0~31)
101	読み取りに失敗しました
102	書き込みに失敗しました
103	フレームエラー
104	返信なし
105	チェックサムエラー
106	サイズが範囲外です
107	パリティエラー
108	オーバーランエラー
109	バッファオーバーフロー
110	フレームエラー
111	アボート
112	ネット同期エラー

コード番号	説明
113	Can エラー
114	返信サイズエラー
115	開始バイトエラー
116	マスターブート
117	EtherCAT mega-ulink エラー
118	テストモードの EtherCAT mega-ulink
119	EtherCAT mega-ulink の通信なし
120	EtherCAT mega-ulink タイムアウト
121	USB エラー
131	USB デバイスが存在しません
151	通信 FIFO がいっぱいです
152	サイズ送信エラー
153	通信が利用できません
154	リモート側に接続されていません
155	ローカル通信がブロックされました
156	通信に失敗しました
157	イーサネット経由でリモート側に接続できません
158	EtherCAT mega-ulink スレーブが応答しない
159	USB 通信なし
160	スレーブがタイムアウトに応答するまで待機
161	無効な引数
162	ネットワークインターフェースカードエラーを取得する
201	認識されない変数
202	スレーブ名が定義されていません
203	リストには、salve に認識されない変数が 1 つ以上含まれています
204	変数はショート型ではありません
220	変数は 64 ビット型ではありません
221	変数は 64 ビット型です
251	スレーブで認識されない機能
252	スレーブ名が定義されていません
260	PDL 機能がスレーブで実行されていません
261	スレーブはブートモードです
262	タスク番号が範囲外です
263	スレーブはブートモードではありません
270	書き込みバッファが拒否されました

コード番号	説明
271	設定するバッファサイズが範囲外です
272	変数の読み取りが拒否されました
273	変数の数が範囲外です
274	予想される回答の長さが長すぎます
275	配列アクセスタイプ 16/32 ビットのみをサポート
276	無効なメモリアクセス
300	スコープの再起動
301	スコープは 2 つの変数のみサポートします
500	リンクがビジーです
501	サポートなし
502	マスターブート
503	スレーブなし
504	リンクエラー
505	スレーブエラー
506	無効なアドレス
507	サポートされていないターゲット
525	サーバーがビジー状態です
600	状態番号が範囲外です
601	スレーブで認識されない状態
602	スレーブ名が定義されていません
651	ID 通信コマンドをサポートしていません
801	追加パラメーター
802	パラメーターが不足しています
803	認識されないコマンド
1000	通信ドライバーが NULL です

11. コード例

11.1 MPI ライブラリの初期化

MPI ライブラリ初期化のフローを次の図に示します。関数 showcomstatus はオプションです。

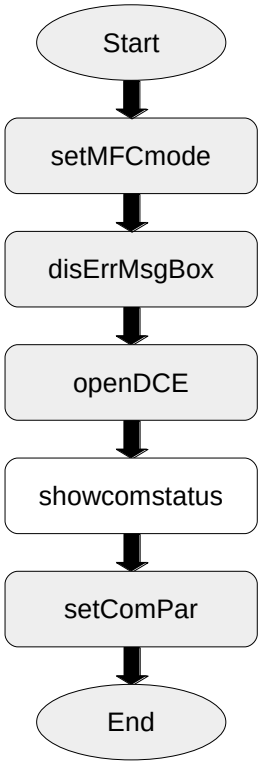


図 12.1.1

例

この例では、AC サーボモーターを搭載したドライバーモデル「D2T-□□□□-S」の MPI ライブラリの初期化を行う方法を示します。

```
setMFCmode();
disErrMsgBox(1);
pCom=openDCE("C:\\HIWIN\\dce\\", "d2.dce");
showcomstatus(pCom);
Status.AC_Connect=setComPar(
    m_AC_COM,       //  Set it as the number of servo drive's communication port.
                        For example, 2 for COM2.
    nBaudrate1,     //  For 115200 bps, set it as 8.
    nMode,          //  For RS-232 via USB, set it as 1;
                        for EtherCAT mega-ulink, set it as 4.
    nTrid,          //  Not in use; set it as 0.
```

```
    nRcid,          //  Not in use; set it as 0.
    nCanbaudrate,   //  Not in use; set it as 0.
    nMsgStand,      //  Not in use; set it as 0.
    nCanpipelevel,  //  Not in use; set it as 0.
    nTimeout,
    nLocktime,
    nIternum,
    pCom);
```

11.2 エンコーダーフィードバック

この例では、コンソール経由でモーターのエンコーダーフィードバックを取得する方法を示します。

備考

(1) ドライバーに接続する前に、関数 setMFCmode を呼び出して DLL を初期化します。
(2) DLL を初期化した後、関数 openDCE を呼び出して通信オブジェクトを構築します。
(3) 通信オブジェクトを構築した後、関数 setComPar を呼び出してドライバーの通信パラメーターを設定します。
(4) PC とドライバー間のリンクが構築されたら、関数 GetVarN を呼び出してドライバーの変数を取得します。
(5) X_enc_pos は、ドライバーのモーターフィードバック位置変数であり、この例では、モーターのエンコーダーフィードバックを読み取るのに役立てられています。

MPI 機能要件

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

例

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

void main()
{
    MDCE *pCom=NULL;    //    Declare
    //    MPI Function : DLL initialization
    setMFCmode();
    //    MPI Function : Open communication object
    pCom=openDCE("C:\\HIWIN\\dce\\", "d2.dce");
    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 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);
        //   bConnected = 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\n");

    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 in the 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);
    pCom=NULL;
    system("PAUSE");
```

MPI ライブラリリファレンスマニュアル
バージョン：V2.4　　2025 年 1 月改訂

1．HIWIN は HIWIN Mikrosystem Corp., HIWIN Technologies Corp., ハイウィン株式会社の登録商標です。ご自身の権利を保護するため、模倣品を購入することは避けてください。
2．実際の製品は、製品改良等に対応するため、このカタログの仕様や写真と異なる場合があります。
3．HIWIN は「貿易法」および関連規制の下で制限された技術や製品を販売・輸出しません。制限された HIWIN 製品を輸出する際には、関連する法律に従って、所管当局によって承認を受けます。また、核・生物・化学兵器やミサイルの製造または開発に使用することは禁じます。
