お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

User's Manual

PDxxSIM
I/O DLL キット

サンプルマニュアル

ルネサス エレクトロニクス
www.renesas.com

[MEMO]

はじめに

PDxxSIM I/O DLL キットは、シミュレータデバッガ M3T-PDxxSIM の機能を拡張するためのキットです。I/O DLL を作成するには、市販の Windows アプリケーション開発環境である Microsoft Visual C++が必要です。

本サンプルマニュアルには、PDxxSIM I/O DLL キットのサンプルプログラムをご使用いただくための情報を掲載しています。利用する Visual C++の言語仕様、操作方法などについては、Visual C++のマニュアルやオンラインヘルプなどをご参照ください。

対応シミュレータデバッガ

PDxxSIM I/O DLL キットは、全てのシミュレータデバッガ M3T-PDxxSIM で利用できるものではありません。PDxxSIM I/O DLL キットと連携できるシミュレータデバッガ、およびそのバージョンについては、PDxxSIM I/O DLL キットのリリースノートに記述していますので、そちらをご参照ください。

使用権

PDxxSIM I/O DLL キットの使用権は、使用するシミュレータデバッガ M3T-PDxxSIM の「ソフトウェア使用権許諾契約書」に基づきます。また、PDxxSIM I/O DLL キットは、お客様の製品開発の目的でのみ使用できます。その他の目的では使用できませんのでご注意ください。

技術サポート

PDxxSIM I/O DLL キットに関する技術サポートは、ホームページ(URL: http://www.renesas.com/jp/tools/)に最新情報を掲載する事によってのみ対応させていただきますので、あらかじめご了承願います。

[MEMO]

Excel と連携する I/O DLL サンプルプログラム

1. 概要

Microsoft 社の Excel がサポートしているオートメーションサーバー機能(以下 ActiveX サーバー機能)を利用して、PDxxSIM と Excel を連携して動作させる方法について説明します。

本サンプルプログラムは、I/O DLL を使用して Excel から I/O の入出力データの読み込み/書き込みを行います。

本サンプルプログラムは、I/O DLL をインストールしたディレクトリ（以下、C:¥MTOOL¥Iodll とする）の”Samples¥PDxxSIM¥IodllForExcel”ディレクトリに格納されています。

2. IodllForExcel DLL の仕様

以下に IodllForExcel DLL の仕様を示します。

- PDxxSIM 起動時に、Excel を起動しファイル D:¥tmp¥Sample.xls をオープンします。
  Sample.xls はデータ出力用のワークシート(OutputData)とデータ入力用のワークシート(InputData)を持っています。
  Sample.xls は、”Samples¥PDxxSIM¥IodllForExcel”ディレクトリに格納していますので、コピーしてお使いください。
- ターゲットプログラムが 0~ffh 番地にデータを書き込んだとき、ワークシート(OutputData)にデータを書き込んだアドレスとデータ値を出力します。
- ターゲットプログラムが 100h～1ffh 番地のデータを読み込んだとき、ワークシート(InputData)の A1～A10 のデータを繰り返し読み込みます。
- PDxxSIM リセット時、ワークシート(OutputData)に書き込んだデータを消去します。

3. IodllForExcel DLL の使用方法

IodllForExcel DLL を使用するには、PDxxSIM をインストールしたディレクトリ（以下、C:¥MTOOL¥PDxxSIM とする）に Release¥IodllForExcel.dll ファイルをコピーしてください。次に、IodllForExcel DLL を simxx.exe へ登録してください。登録するには、simxx.exe の環境設定ファイル simxx.ini ファイルに以下の記述を追加してください。

```
[DLLNAME]
IODLL=IodllForExcel
```

追加後、PDxxSIM を起動すると IodllForExcel DLL がロードされます。

詳細は、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の使用方法」を参照ください。

4. IodllForExecl DLL の作成方法

Microsoft 社の Visual C++ V.6.0（以下 VC++）を用いた IodllForExcel DLL の作成方法を説明します。

IodllForExcel DLL は、入力ポートからの Read データを Excel のファイルで与え、出力ポートへの Write データを Excel のファイルに保存する I/O DLL です。

最初に、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の作成方法」に記載されている方法で I/O DLL を作成します。その後、Excel と連携するための処理を追加します。

4.1. Excel のタイプライブラリのインポート

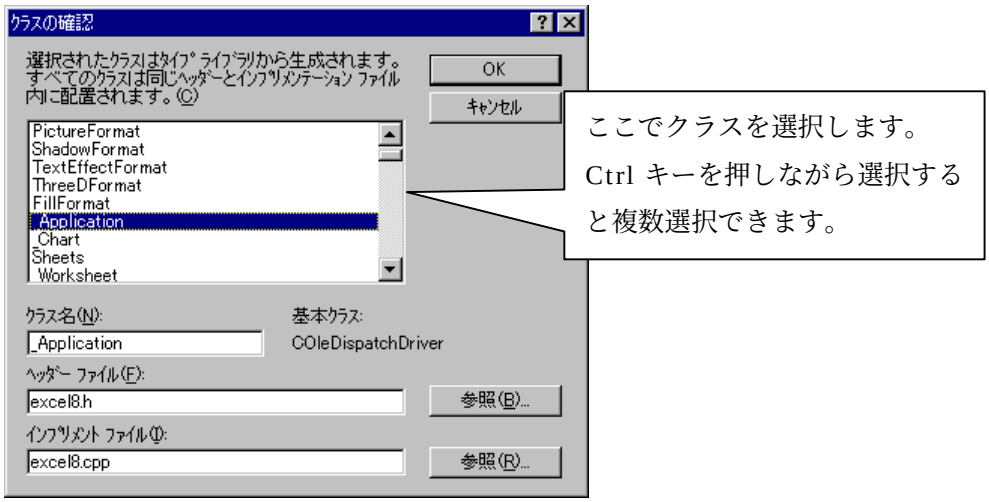
Excel の ActiveX サーバー機能を利用するために、Excel のタイプライブラリ(Excel97 の場合 excel8.olb)をインポートします。手順を以下に示します。

1. VC++のメニュー[表示]→[ClassWizard]を選択すると「MFC ClassWizard」ダイアログがオープンします。
2. ”クラスの追加...”→”タイプライブラリから...”を選択すると、「タイプライブラリからインポート」ダイアログがオープンするので、Excel のタイプライブラリファイル excel8.olb を選択します。
3. 下図に示す「クラスの確認」ダイアログがオープンするので、クラス選択欄（一番上のリスト）から下記のクラスを選択し OK ボタンを押します。

   _Application、_Worksheet、Workbooks、Sheets、Range

   ファイル excel8.cpp と excel8.h が自動的にプロジェクトに追加されます。

   後からクラスを追加する場合も同様の手順で行います。



4.2. ActiveX オートメーションの初期化

ActiveX サーバー機能を持つアプリケーションを利用するためには I/O DLL 側で ActiveX オートメーションの初期化処理を実行する必要があります。初期化処理を行うには、I/O DLL に以下の記述を追加してください。PDxxSIM が I/O DLL を呼び出すと同時に初期化処理が実行されます。

```
struct StartOle {
    StartOle() { CoInitialize(NULL); }
    ~StartOle() { CoUninitialize(); }
} _inst_StartOle;
```

4.3. Excel との連携処理

Excel との連携の処理を iofunc¥iofunc.cpp に記述します。

前述の「タイプライブラリのインポート」で生成されたヘッダーファイル excel8.h を iofunc.cpp にインクルードします。以下に記述例を示します。

```
#include "..¥excel8.h"        // excel8.olb から生成されたヘッダーファイル
```

iofunc.cpp に以下の変数宣言を記述します。_Application などは、前述の「タイプライブラリのインポート」で追加したクラスです。この変数を用いて Excel を操作します。

```
_Application    m_appExcel;
_Worksheet      m_wkInputSheet;     // データ入力用ワークシート(シート名：InputData)
_Worksheet      m_wkOutputSheet;    // データ出力用ワークシート(シート名：OutputData)
Workbooks       m_wkBooks;
Sheets          m_wkSheets;
BOOL            m_bUseExcel = FALSE;          // Excel が使用可能かを示すフラグ
```

Excel を起動するには、以下のように記述します。

```
if ( m_appExcel.CreateDispatch("Excel.Application") == TRUE ){    // Excel を起動
    m_appExcel.m_bAutoRelease = TRUE;
    m_appExcel.SetVisible(TRUE);                                  // Excel を CRT 画面上に表示
    m_bUseExcel = TRUE;

    //Excel のファイルをオープン
    LPDISPATCH pWkBooks = m_appExcel.GetWorkbooks();        // ワークブックの作成
    m_wkBooks.AttachDispatch( pWkBooks,TRUE );
    CString strFile = "D:¥¥temp¥¥sample.xls";
    m_wkBooks.Open((LPCTSTR)strFile,
                    COleVariant((long)0, VT_I4),              // UpdateLinks
                    COleVariant((long)FALSE,VT_BOOL),         // ReadOnly
                    COleVariant((long)1, VT_I4),              // Format
                    COleVariant(""),                          // Password
                    COleVariant(""),                          // WriteResPassword
                    COleVariant((long)FALSE, VT_BOOL),        // IgnoreReadOnlyRecommended
                    COleVariant((long)2, VT_I4),              // Origin
                    COleVariant(),                            // Delimiter
                    COleVariant((long)FALSE, VT_BOOL),        // Editable
                    COleVariant((long)FALSE, VT_BOOL),        // Notify
                    COleVariant(),                            // Converter
                    COleVariant((long)FALSE, VT_BOOL));       // AddToMru

    // ワークシートの割り当て
    m_wkSheets = m_appExcel.GetSheets();
    m_wkInputSheet = m_wkSheets.GetItem( COleVariant( "InputData" ) );
    m_wkOutputSheet = m_wkSheets.GetItem( COleVariant( "OutputData" ) );
}
```

PDxxSIM 起動時に Excel を起動させるには、I/ODLL の関数 NotifyStart()中に Excel の起動処理を記述します。NotifyStart 関数は、シミュレータエンジン SIMxx の起動時に呼び出される関数です。

ターゲットプログラムのメモリリード時に、Excel のファイルからデータを取得するには、以下のように記述します。m_wkInputSheet.GetRange()関数で読み込むセルの位置を指定して、oRng.GetValue()関数でセルからデータを読み込みます。

```
void NotifyPreReadMemory(unsigned long address, int length)
{
    unsigned long data;
    char pos[10];
    COleVariant vOpt(DISP_E_PARAMNOTFOUND, VT_ERROR);
    VARIANT v;

    // 0x100～0x1ff 番地までのアクセスデータは、Excel から取得
    if ( m_bUseExcel == TRUE   && address > 0x100 && address < 0x1ff ){
        // Excel からデータを読み込む
        wsprintf( pos, "A%d", read_pos +1 );
        Range oRng = m_wkInputSheet.GetRange( COleVariant( pos ), vOpt );
        // 指定したセルからデータを読み込む
        v = oRng.GetValue();
        data = (unsigned long)v.dblVal;

        // PDxxSIM にデータを書き込む
        RequestPutMemory( address, length, data );

        // セルの読み込み位置の更新
        read_pos++;
        read_pos %= 10;    // read_pos の値を 0～9 までに制限
    }
}
```

ターゲットプログラムのメモリライト時に、Excel のファイルにデータを書き込むには、以下のように記述します。m_wkOutputSheet.GetRange()関数で書き込むセルの位置を指定して、oRng.SetValue()関数でセルにデータを書き込みます。

```
void NotifyPostWriteMemory(unsigned long address, int length, unsigned long data)
{
    char szPos[10], szData[20];
    COleVariant vOpt(DISP_E_PARAMNOTFOUND, VT_ERROR);

    // 0x00～0xffh 番地に書いたデータをエクセルに表示
    if ( m_bUseExcel == TRUE && address < 0x100 ){
        // Excel にアドレスを書き込む
        wsprintf( szPos, "A%d", write_pos + 2 );
        Range oRng = m_wkOutputSheet.GetRange( COleVariant( szPos ), vOpt );
        wsprintf( szData, "'%06X", address );
        oRng.SetValue( COleVariant( szData ) );
        // Excel にデータを書き込む
        wsprintf( szPos, "B%d", write_pos + 2 );
        oRng = m_wkOutputSheet.GetRange( COleVariant( szPos ), vOpt );
        oRng.SetValue( COleVariant( (double)data ) );

        // セルの書き込み位置の更新
        write_pos++;
    }
}
```

4.4. I/O DLL 終了時の処理

PDxxSIM を終了すると I/O DLL も終了します。その際に、Excel との連携を切断する必要があります。Excel との連携を切断しないと、Excel を終了することができなくなります（Excel を終了しても CRT 画面上で非表示になるだけで、実際に終了していません）。

I/ODLL の終了処理として、必ず下記の処理を実行してください。

```
void DisConnectExcel()
{
        if ( m_bUseExcel == TRUE )
                m_appExcel.DetachDispatch();
}
```

PDxxSIM 終了時に Excel との連携を切断するには、I/O DLL の関数 NotifyEnd 関数中で上記関数を呼び出します。NotifyEnd 関数は、シミュレータエンジン SIMxx の終了時に呼び出される関数です。

以上

[MEMO]

ファイルからデータを Read/Write する I/O DLL サンプルプログラム

1. 概要

本サンプルプログラムは、I/O DLL を使用してファイルから I/O の入出力データの読み込み/書き込みを行います。

IodllForFile DLL は、入力ポートからの Read データをファイルで与え、出力ポートへの Write データをファイルに保存する I/O DLL です。

本サンプルプログラムは、I/O DLL をインストールしたディレクトリ（以下、C:¥MTOOL¥Iodll とする）の”Samples¥PDxxSIM¥IodllForFile”ディレクトリに格納されています。

2. IodllForFile DLL の仕様

以下に IodllForFile DLL の仕様を説明します。

- PDxxSIM 起動時に、入力ファイル iodll_in.txt をオープンします。
  iodll_in.txt は、入力ポートに入力するデータを記述します。入力データは、16 進数をカンマ(',')で区切った書式で記述します。
  例）10,20,30,40,50,60,70,80,90
  ・半角スペース、改行コード（'¥n'）は読み飛ばします。
  ・途中で EOF になった場合はエラーとし、0 を読み込んだと見なします。
  iodll_in.txt は、”Samples¥PDxxSIM¥IodllForFile”ディレクトリに格納していますので、コピーしてお使いください。
- ターゲットプログラムが 0x100 番地のデータを読み込んだとき、入力ファイル(iodll_in.txt)に記述されているデータを繰り返し読み込みます。
- ターゲットプログラムが 0x101 番地にデータを書き込んだとき､出力ファイル(iodll_out.txt)にデータを出力します。
- PDxxSIM でターゲットプログラムをリセットした時、入力ファイル(iodll_in.txt)を一旦クローズ/オープンして、次の実行で再度先頭のデータから入力します。
- エラーが発生した場合は、エラーメッセージをエラーメッセージ出力ファイル（iodll_err.txt）に出力します。

3. IodllForFile DLL の使用方法

IodllForFile DLL を使用するには、PDxxSIM をインストールしたディレクトリ（以下、C:¥MTOOL¥PDxxSIM とする）に Release¥IodllForFile.dll ファイルをコピーしてください。次に、IodllForFile DLL を simxx.exe へ登録してください。登録するには、simxx.exe の環境設定ファイル simxx.ini ファイルに以下の記述を追加してください。

```
[DLLNAME]
IODLL=IodllForFile
```

追加後、PDxxSIM を起動すると IodllForFile DLL がロードされます。

詳細は、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の使用方法」を参照ください。

4. IodllForFile DLL の作成方法

Microsoft 社の Visual C++ V.6.0（以下 VC++）を用いた IodllForFile DLL の作成方法を説明します。

最初に、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の作成方法」に記載されている方法で I/O DLL を作成します。その後、ファイルへ読み書きするための処理を追加します。

4.1. ファイルの入出力処理

ファイルの入出力処理を iofunc¥iofunc.cpp に記述します。

iofunc.cpp に以下のマクロ定義を記述します。マクロ定義では、入出力ファイル名、エラーメッセージを出力するファイル名、入出力ポートを指定します。ファイル名や入出力ポートを変更する場合は、この定義を変更してください。

```
// マクロ定義
#define   INPUT_FILE     "iodll_in.txt"      // 入力ファイル名
#define   OUTPUT_FILE    "iodll_out.txt"     // 出力ファイル名
#define   ERROR_FILE     "iodll_err.txt"     // エラーメッセージ出力ファイル名
#define   PORT_IN        0x100               // 入力ポートのアドレス
#define   PORT_OUT       0x101               // 出力ポートのアドレス
```

以下の関数で入力ファイルをオープンします。

```
static void openInputFile( void )
{
        if ( fp_in ) {          // リセット時は、ファイルを再オープンするためクローズする
                fclose( fp_in );
        }
        fp_in = fopen( INPUT_FILE, "r" );
        if ( fp_in == NULL ) {
                outErrMsg( "openInputFile: open error!¥n" );
        }
}
```

ターゲットプログラムがリセットされた時に入力ファイルをオープンさせるため、I/O DLL の関数 NotifyReset 関数中で上記関数を呼び出します。NotifyReset 関数は、PDxxSIM のリセットコマンド実行時に呼び出される関数ですが、PDxxSIM 起動時にもリセットコマンドが実行されますので、PDxxSIM の起動と同時に入力ファイルがオープンできます。

以下の関数で出力ファイルをオープンします。

```
static void fileOpen( void )
{
	fp_out = fopen( OUTPUT_FILE, "w" );
	if ( fp_out == NULL ) {
		outErrMsg( "fileOpen: open output file error!¥n" );
	}
}
```

PDxxSIM 起動時に出力ファイルをオープンさせるため、I/O DLL の関数 NotifyStart 関数中で上記関数を呼び出します。NotifyStart 関数は、シミュレータエンジン SIMxx の起動時に呼び出される関数です。

ターゲットプログラムのメモリリード時に、入力ファイルからデータを取得するには、以下のように記述します。

```
void NotifyPreReadMemory(unsigned long address, int length)
{
	unsigned long dt;

	if ( address == PORT_IN && length == 1 ) {
		dt = readInputFile();
		if ( RequestPutMemory( PORT_IN, 1, dt ) == FALSE ) {
                                                    // 入力ポートに 1 バイトデータをセット
			outErrMsg( "NotifyPreReadMemory: RequestPutMemory error!¥n" );
		}
	}
	return;
}
```

ターゲットプログラムのメモリライト時に、出力ファイルにデータを書き込むには、以下のように記述します。

```
void NotifyPostWriteMemory(unsigned long address, int length)
{
	static int outDataCount = 0;

	if ( fp_out == NULL ) {
		return;
	} else if ( address == PORT_OUT && length == 1 ) {
		if ( fprintf( fp_out, "%02x:%08x, ", data & 0xffL, totalCycle ) < 0 ) {
                                                                    // ファイルに出力
			outErrMsg( "NotifyPostWriteMemory: output error!¥n" );
			fclose( fp_out );
			fp_out = NULL;
		}
		outDataCount++;
		if ( outDataCount == 16 ) {
			outDataCount = 0;
			if ( fputc( '¥n', fp_out ) == EOF ) {
				outErrMsg( "NotifyPostWriteMemory: output error!¥n" );
				fclose( fp_out );
				fp_out = NULL;
			}
		}
	}
	return;
}
```

4.2. I/O DLL 終了時の処理

PDxxSIM を終了すると I/O DLL も終了します。その際に、入出力ファイルをクローズする必要があります。以下の関数で入出力ファイルをクローズします

```
static void fileClose( void )
{
        if ( fp_in ) {
                fclose( fp_in );
        }
        if ( fp_out ) {
                if ( fclose( fp_out ) == EOF ) {
                        outErrMsg( "fileClose: close output file error!¥n" );
                }
        }
}
```

PDxxSIM 終了時に入出力ファイルをクローズさせるには、I/O DLL の関数 NotifyEnd 関数中で上記関数を呼び出します。NotifyEnd 関数は、シミュレータエンジン SIMxx の終了時に呼び出される関数です。

以上

M16C/80 シリーズ周辺機能の I/O DLL サンプルプログラム

1. 概要

M16C/80 シリーズの周辺機能をシミュレーションするための I/O DLL のサンプルプログラムです。I/O DLL サンプルプログラムには、以下のものが含まれています。

1. タイマのサンプル
2. A-D 変換のサンプル
3. シリアル I/O のサンプル
4. CRC 演算回路のサンプル
5. MR308 用のタイマのサンプル

これらの I/O DLL サンプルプログラムの詳細に関しては、「I/O DLL サンプルプログラムの仕様」の章で説明します。

2. I/O DLL サンプルプログラムの使用方法

I/O DLL サンプルプログラムは、I/O DLL をインストールしたディレクトリ（以下、C:¥MTOOL¥Iodll とする）の"Samples¥Pd308sim"ディレクトリに格納されています。

- TimerA0_TMode¥TimerA0_TMode.dll
  タイマ A0（タイマモード）の I/O DLL
- TimerA0_TMod_Gate¥TimerA0_TMod_Gate.dll
  タイマ A0（タイマモード、ゲート機能選択時）の I/O DLL
- TimerA0_TMod_Pulse¥TimerA0_TMod_Pulse.dll
  タイマ A0（タイマモード、パルス出力機能選択時）の I/O DLL
- TimerA0_OneShotTM¥TimerA0_OneShotTM.dll
  タイマ A0（ワンショットタイマモード）の I/O DLL
- AD¥AD.dll
  A-D 変換器（単発モード）の I/O DLL
- SIO¥SIO.dll
  シリアル I/O（受信）の I/O DLL
- CRC¥CRC.dll
  CRC 演算回路の I/O DLL
- MR¥MR.dll
  MR308 用のタイマの I/O DLL

I/O DLL を使用するには、PDxxSIM をインストールしたディレクトリ（以下、C:¥MTOOL¥PDxxSIM とする）に I/O DLL ファイル（".dll"）をコピーしてください。次に、I/O DLL を simxx.exe へ登録してください。登録するには、simxx.exe の環境設定ファイル simxx.ini ファイルに I/O DLL ファイル名の拡張子を取って記述してください。

（例）タイマ A0（タイマモード）の I/O DLL の場合

```
[DLLNAME]
IODLL= TimerA0_TMode
```

追加後、PDxxSIM を起動すると登録した I/O DLL がロードされます。

詳細は、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の使用方法」を参照ください。

3. I/O DLL サンプルプログラムの作成方法

Microsoft 社の Visual C++ V.6.0（以下 VC++）を用いた、I/O DLL の作成方法について説明します。I/O DLL を作成するには、I/O DLL のサンプルプロジェクト IodllTemplate プロジェクトを使用するか、新規に I/O DLL のプロジェクトを作成してください。新規に I/O DLL のプロジェクトを作成するには、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の作成方法」を参照ください。

IodllTemplate プロジェクトを使用する場合、以下の手順で I/O DLL を作成してください。IodllTemplate プロジェクトは、C:¥MTOOL¥Iodll の以下のディレクトリに格納されています。

"Samples¥Pd308sim¥IodllTemplate"

以下に手順を示します。

1. IodllTemplate プロジェクトのプロジェクトファイル IodllTemplate.dsp を VC++でオープンしてください。
2. IodllTemplate¥iofunc ディレクトリに I/O DLL ファイル（タイマ A0 の場合、TimerA0_TMode¥iofunc ディレクトリにあるファイル）をコピーしてください。
   iofunc¥iofunc.c, iofunc.h
3. iofunc ディレクトリにある infunc.cpp に周辺機能の動作をシミュレートするためのコードが記述されています。変更する場合は、このファイルにコードを記述してください。
4. VC++でビルドすると I/O DLL（".dll"ファイル）が作成されます。

詳細は、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の作成方法」を参照ください。

なお、PD308SIM に指定できる I/O DLL は、**1** つです。複数の周辺機能を指定するには、I/O DLL サンプルプログラムの各サンプルのソースプログラム（iofunc.c）中にある同じ関数の内容を組み合わせて新規に I/O DLL を作成してください。

以下に、タイマ A0（タイマモード）と A-D 変換（単発モード）の２種類の周辺機能をシミュレーションするための NotifyPostWriteMemory 関数の作成例を示します。その他の関数についても NotifyPostWriteMemory 関数と同じようにタイマ A0 と A-D 変換の内容を組み合わせて作成してください。

「NotifyPostWriteMemory 関数の作成例」

```
void NotifyPostWriteMemory(unsigned long address, int length, unsigned long data)
{
    // タイマ A0 の設定
    if (address == TABSR) {
        if ((data & 0x01) == 0x01) {            // カウント開始フラグのチェック
            sCountFlag = TRUE;
        }
    } else if (address == TA0) {                // タイマ A0 レジスタへの書き込みを監視
        sTmReloadReg = data;                    // リロードレジスタに値を設定
    }

    // A-D 変換の設定
    unsigned long           adicData;

    if (sAdDataExist == TRUE) {     // A-D データの入力開始を示すフラグをチェック
        if (address == ADCON0) {    // A-D 制御レジスタ 0 の A-D 変換開始フラグの書き込みを監視
            if ((data & 0x40) == 0x40) {        // A-D 変換開始フラグが 1 なら break
                RequestGetMemory(ADIC, 1, &adicData);
                RequestInterrupt(ADINT, adicData & 0x7);
                                                // A-D 割り込みの発生
                                                // 優先順位は割り込み制御レジスタを参照
                RequestPutMemory(ADCON0, 1, data & 0xbf);     // A-D 変換開始フラグをクリア
                RequestPutMemory(ADIC, 1, adicData | 0x08);   // 割り込み要求ビットのセット
            }
        }
    }
}
```

4. I/O DLL サンプルプログラムの仕様

4.1. タイマ A0（タイマモード）のサンプルプログラム（timerA0_TMode.dll）

このサンプルプログラムでは、タイマ A0 のタイマモードの動作をシミュレーションします。

「タイマの機能」

カウントソース： サイクル数をカウント
パルス出力機能： なし
ゲート機能　　：なし

「動作」

1. カウント開始フラグを 1 にすると 16 サイクル毎に、タイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを続けます。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。
3. カウント開始フラグを 0 にすると、カウント値を保持してダウンカウントを停止します。
4. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

4.2. タイマ A0（タイマモード、ゲート機能選択時）のサンプルプログラム（timerA0_TMod_Gate.dll）

このサンプルプログラムでは、タイマ A0 のタイマモード（ゲート選択時）の動作をシミュレーションします。

「タイマの機能」

カウントソース： サイクル数をカウント

パルス出力機能： なし

ゲート機能　　：TA0IN 端子が 1（"H"レベル）の期間だけカウントを行う。

「動作」

1. カウント開始フラグが 1 で TA0IN 端子の入力が 1（"H"レベル）の時、16 サイクル毎にタイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. TA0IN 端子の入力が 0（"L"レベル）の時、カウント値を保持してダウンカウントを停止します。
3. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを続けます。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。
4. カウント開始フラグを 0 にすると、カウント値を保持してダウンカウントを停止します。
5. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「TA0IN 端子の入力信号の作成方法」

iofunc.cpp ファイルの下記の部分を変更して入力信号を作成します。この例では、10000 サイクル毎に TA0IN 端子の入力信号の H/L を切り替えています。

```
#define HL_SWITCH_CYCLE        10000
                                    // 10000 サイクル毎に TA0IN 端子の入力信号の H/L を切り替え
```

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

4.3. タイマ A0（タイマモード、パルス出力機能選択時）のサンプルプログラム（timerA0_TMod_Pulse.dll）

このサンプルプログラムでは、タイマ A0 のタイマモード（パルス出力機能選択時）の動作をシミュレーションします。

「タイマの機能」

カウントソース： サイクル数をカウント
パルス出力機能： あり
ゲート機能　　：なし

「動作」

1. カウント開始フラグを 1 にすると 16 サイクル毎に、タイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを続けます。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。また、TA0OUT 端子の出力極性が反転します。
3. カウント開始フラグを 0 にすると、カウント値を保持してダウンカウントを停止します。また、TA0OUT 端子は 0（"L"レベル）を出力します。
4. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

「注意」

TA0OUT 端子の出力の変化を I/O ウィンドウの仮想ポート出力機能を利用して参照することはできません。

4.4. タイマ A0（ワンショットタイマモード）のサンプルプログラム（timerA0_OneShotTM.dll）

このサンプルプログラムでは、タイマ A0 のワンショットタイマモードの動作をシミュレーションします。

「タイマの機能」

カウントソース　：サイクル数をカウント
パルス出力機能　：なし
カウント開始条件：ワンショット開始フラグへの 1 の書き込み

「動作」

1. カウント開始フラグが 1 の状態でワンショット開始フラグを 1 にすると 16 サイクル毎に、タイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを停止します。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。
3. カウント中にワンショット開始フラグに 1 が書き込まれた場合、再度リロードレジスタの値をリロードしてカウントを続けます。
4. カウント開始フラグを 0 にすると、ダウンカウントを停止し、リロードレジスタの内容をリロードします。
5. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

4.5. A-D 変換器（単発モード）のサンプルプログラム（AD.dll）

このサンプルプログラムでは、A-D 変換器の単発モードの動作をシミュレーションします。なお、サンプルプログラムでは A-D レジスタ 0 への入力をシミュレーションします。

「PD308SIM における A-D 変換器の考え方（実チップとの動作の違い）」

PD308SIM では、アナログ入力のシミュレーションをサポートしていません。そのため、A-D 変換後の入力値を A-D レジスタに入力することで、A-D 変換器の機能を擬似的に実現します。

「動作」

1. A-D 変換開始フラグを 1 にすると、A-D 変換割り込みが発生します。同時に A-D レジスタ 0 にデータを入力します。
2. その時、A-D 変換割り込み要求ビットが 1 になります。また、A-D 変換開始フラグが 0 になり、A-D 変換器の動作を停止します。
3. 割り込みが受け付けられると、A-D 変換割り込み要求ビットが 0 になります。
4. その後、再び A-D 変換開始フラグを 1 にすると、入力データがなくなるまで上記の 1～3 の動作を行います。

「A-D 入力データの作成方法」

infunc.cpp ファイルの下記の部分を変更して A-D 入力データを作成します。

```
#define AD_DATA_NUM       32          // 入力する A-D データ数

void NotifyInterrupt(unsigned long vec)
{
    unsigned long   adicData;
    // A-D データの設定。ここに入力データを定義します。
    // A-D 変換割り込みが発生する毎に、0x0、0x1、0x2 の順にデータを入力します。
    static const char adData[AD_DATA_NUM] =
        0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7,
        0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
    };      // 入力する A-D データ
    :
    :
}
```

「注意」

プログラムで A-D 変換割り込み要求ビットを 0 にしても、A-D 変換割り込みの要求をキャンセルできません。

4.6. シリアル I/O（受信）のサンプルプログラム（SIO.dll）

このサンプルプログラムでは、シリアル I/O（受信）の動作をシミュレーションします。なお、サンプルプログラムでは UART0 への入力をシミュレーションします。

「サンプルプログラムにおけるシリアル I/O の考え方（実チップとの動作の違い）」

サンプルプログラムでは、シリアル I/O の入力として RxD から入力信号を取り込むのではなく、UART 受信バッファレジスタに直接データを入力することで簡易的にシリアル I/O の動作をシミュレーションしています。

なお、サンプルプログラムは、クロック同期型シリアルモード、クロック非同期型シリアルモード共通で使用できます。但し、クロック非同期型シリアルモードでの受信データ長は 8 ビットです。

「動作」

1. 受信許可ビットを 1 にすると、UART0 受信割り込みが発生します。同時に UART0 受信バッファにデータを入力します。
2. その時、受信完了フラグと UART0 割り込み要求ビットが 1 になります。
3. 受信完了フラグは、UART0 受信バッファレジスタの下位バイトを読み出したとき 0 になります。
4. 割り込みが受け付けられると、UART0 割り込み要求ビットが 0 になります。
5. その後、10000 サイクル毎に入力データがなくなるまで上記の 1～4 の動作を行います。

「受信データの作成方法」

iofunc.cpp ファイルの下記の部分を変更して受信データを作成します。

```
#define SIO_DATA_NUM      32          // 入力する受信データ数

void NotifyInterrupt(unsigned long vec)
{
    unsigned long   u0c1Data;
    unsigned long   s0ricData;
    // 受信データの設定。ここに受信データを定義します。
    // UART0 割り込みが発生する毎に、0x0、0x1、0x2 の順にデータを入力します。
    static const char sioData[SIO_DATA_NUM] =
        0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7,
        0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
    };      // 入力する受信データ
    :
    :
}
```

「受信データ入力タイミングの変更方法」

iofunc.cpp ファイルの下記の部分を変更して受信データ入力タイミングを変更します。サンプル DLL では、10000 サイクル毎にデータを入力します。

```
// ここを変更して、任意サイクル毎にデータを入力するように変更できます。
#define SIO_INPUT_CYCLE   10000      // 10000 サイクル毎に受信データを入力
```

「注意」

1. プログラムで UART0 割り込み要求ビットを 0 にしても、UART0 割り込みの要求をキャンセルできません。
2. エラー検知は、オーバランエラーだけを検知できます。

4.7. CRC 演算回路のサンプルプログラム（CRC.dll）

このサンプルプログラムでは、CRC 演算回路の動作をシミュレーションします。

「動作」

1. CRC データレジスタに初期値 0 を設定します。
2. CRC インプットレジスタに 1 バイトのデータを書き込むと、書き込んだデータと CRC レジスタの内容に基づいて、CRC コードが CRC データレジスタに生成されます。
3. 連続数バイト CRC 演算を行う場合には、続けて次のデータを CRC インプットレジスタに書き込んで下さい。
4. 全データを書き終えた後の CRC データレジスタの内容が CRC 符号となります。

「実チップの CRC 演算回路との違い」

1. 実チップでは、CRC コードの生成に 2 マシンサイクル要しますが、サンプルプログラムではサイクルを要しません（加算されません）。

4.8. MR308 用のタイマのサンプルプログラム（MR.dll）

このサンプルプログラムでは、MR308 でタイマ A0 を使用する場合の動作をシミュレーションします。

PD308SIM で MR308 を使用したアプリケーションプログラムを実行する場合、以下の設定を行う必要があります。

MR308 では、周期ハンドラやアラームハンドラ、get_tim システムコール、dly_tsk システムコール等が参照するシステムクロックを設定するために、以下の例のようにコンフィグレーションファイル中で MR30 が使用するタイマとタイマ割り込みが発生する時間間隔を指定します。

```
clock{
        mpu_clock           = 10MHz;
        timer               = A0;
        IPL                 = 4;
        unit_time           = 100ms;            // ms
        initial_time        = 0:0:0;
};
```

この例の場合、システムのタイマ割り込み用にタイマ A0 を使用し、タイマ A0 の割り込み発生間隔を 100ms に設定しています。

PD308SIM で MR308 を用いたアプリケーションプログラムを実行する場合､MR308 で使用するタイマの設定（この場合タイマ A0）を行う必要があります。

MR.dll は、MR308 で使用するタイマの動作をシミュレートします。MR.dll を PD308SIM で読み込むことにより MR308 のアプリケーションのシミュレートが行えます。

「タイマ A0 以外のタイマを使用する場合の変更方法」

タイマ A3 に変更する場合を例にとって説明します。

●タイマ A0 のサンプルプログラム

```
#define   TA0IC    0x6c      // タイマ A0 割り込み制御レジスタ
#define   TABSR    0x340     // カウント開始フラグ
#define   TA0      0x346     // タイマ A0 レジスタ
#define   TA0INT   12        // タイマ A0 割り込み番号

void NotifyStepCycle(int cycle)
{
        :
        :
    RequestGetMemory(TABSR, 1, &tabsrData);
    if ((tabsrData & 0x01) == 0x01) {     // カウント開始フラグのチェック
        sCountCycle += cycle;
        RequestGetMemory(TA0, 2, &ta0Data);
        if (sCountCycle >= ta0Data + 1) {          // 分周比分カウントダウン
            RequestGetMemory(TA0IC, 1, &ta0icData);
            RequestInterrupt(TA0INT, ta0icData & 0x7);
                    // タイマ A0 割り込みの発生、優先順位は割り込み制御レジスタを参照
            sCountCycle = 0;
        }
    }
}
```

```
void NotifyPostWriteMemory(unsigned long address, int length, unsigned long data)
{
    if (address == TABSR) {
        if ((data & 0x01) == 0x01) {       // カウント開始フラグのチェック
            sCountFlag = TRUE;
        }
    }
}
```

↓　タイマ A3 を使用するように以下の箇所を変更します。

●タイマ A3 のサンプルプログラム

```
#define   TA3IC    0x8e       // タイマ A3 割り込み制御レジスタ
#define   TABSR    0x340      // カウント開始フラグ
#define   TA3      0x34c      // タイマ A3 レジスタ
#define   TA3INT   15         // タイマ A3 割り込み番号

void NotifyStepCycle(int cycle)
{
        :
        :
    RequestGetMemory(TABSR, 1, &tabsrData);
    if ((tabsrData & 0x08) == 0x08) {      // カウント開始フラグのチェック
        sCountCycle += cycle;
        RequestGetMemory(TA3, 2, &ta0Data);
        if (sCountCycle >= ta0Data + 1) {            // 分周比分カウントダウン
            RequestGetMemory(TA3IC, 1, &ta0icData);
            RequestInterrupt(TA3INT, ta0icData & 0x7);
                    // タイマ A3 割り込みの発生、優先順位は割り込み制御レジスタを参照
            sCountCycle = 0;
        }
    }
}

void NotifyPostWriteMemory(unsigned long address, int length, unsigned long data)
{
    if (address == TABSR) {
        if ((data & 0x08) == 0x08) {       // カウント開始フラグのチェック
            sCountFlag = TRUE;
        }
    }
}
```

以上

[MEMO]

M16C/60 シリーズ周辺機能の I/O DLL サンプルプログラム

1．概要

M16C/60 シリーズの周辺機能をシミュレーションするための I/O DLL のサンプルプログラムです。I/O DLL サンプルプログラムには、以下のものが含まれています。

1. タイマのサンプル
2. A-D 変換のサンプル
3. シリアル I/O のサンプル
4. CRC 演算回路のサンプル
5. DMAC のサンプル
6. MR30 用のタイマのサンプル

これらの I/O DLL サンプルプログラムの詳細に関しては、「I/O DLL サンプルプログラムの仕様」の章で説明します。

2．I/O DLL サンプルプログラムの使用方法

I/O DLL サンプルプログラムは、I/O DLL をインストールしたディレクトリ（以下、C:¥MTOOL¥Iodll とする）の"Samples¥Pd30sim"ディレクトリに格納されています。

- TimerA0_TMode¥TimerA0_TMode.dll
  タイマ A0（タイマモード）の I/O DLL
- TimerA0_TMod_Gate¥TimerA0_TMod_Gate.dll
  タイマ A0（タイマモード、ゲート機能選択時）の I/O DLL
- TimerA0_TMod_Pulse¥TimerA0_TMod_Pulse.dll
  タイマ A0（タイマモード、パルス出力機能選択時）の I/O DLL
- TimerA0_OneShotTM¥TimerA0_OneShotTM.dll
  タイマ A0（ワンショットタイマモード）の I/O DLL
- AD¥AD.dll
  A-D 変換器（単発モード）の I/O DLL
- SIO¥SIO.dll
  シリアル I/O（受信）の I/O DLL
- CRC¥CRC.dll
  CRC 演算回路の I/O DLL
- DMAC¥DMAC.dll
  DMAC の I/O DLL
- MR¥MR.dll
  MR30 用のタイマの I/O DLL

I/O DLL を使用するには、PDxxSIM をインストールしたディレクトリ（以下、C:¥MTOOL¥PDxxSIM とする）に I/O DLL ファイル（".dll"）をコピーしてください。次に、I/O DLL を simxx.exe へ登録してください。登録するには、simxx.exeの環境設定ファイル simxx.ini ファイルに I/O DLL ファイル名の拡張子を取って記述してください。

（例）タイマ A0（タイマモード）の I/O DLL の場合

```
[DLLNAME]
IODLL= TimerA0_TMode
```

追加後、PDxxSIM を起動すると登録した I/O DLL がロードされます。

詳細は、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の使用方法」を参照ください。

3. I/O DLL サンプルプログラムの作成方法

Microsoft 社の Visual C++ V.6.0（以下 VC++）を用いた、I/O DLL の作成方法について説明します。I/O DLL を作成するには、I/O DLL のサンプルプロジェクト IodllTemplate プロジェクトを使用するか、新規に I/O DLL のプロジェクトを作成してください。新規に I/O DLL のプロジェクトを作成するには、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の作成方法」を参照ください。

IodllTemplate プロジェクトを使用する場合、以下の手順で I/O DLL を作成してください。IodllTemplate プロジェクトは、C:¥MTOOL¥Iodll の以下のディレクトリに格納されています。

"Samples¥Pd30sim¥IodllTemplate"

以下に手順を示します。

1. IodllTemplate プロジェクトのプロジェクトファイル IodllTemplate.dsp を VC++でオープンしてください。
2. IodllTemplate¥iofunc ディレクトリに I/O DLL ファイル（タイマ A0 の場合、TimerA0_TMode¥iofunc ディレクトリにあるファイル）をコピーしてください。
   iofunc¥iofunc.c, iofunc.h
3. iofunc ディレクトリにある infunc.cpp に周辺機能の動作をシミュレートするためのコードが記述されています。変更する場合は、このファイルにコードを記述してください。
4. VC++でビルドすると I/O DLL（".dll"ファイル）が作成されます。

詳細は、「PDxxSIM I/O DLL キット ユーザーズマニュアル」の「I/O DLL の作成方法」を参照ください。

なお、PD30SIM に指定できる I/O DLL は、**1つです。**複数の周辺機能を指定するには、I/O DLL サンプルプログラムの各サンプルのソースプログラム（iofunc.c）中にある同じ関数の内容を組み合わせて新規に I/O DLL を作成してください。

以下に、タイマ A0（タイマモード）と A-D 変換（単発モード）の２種類の周辺機能をシミュレーションするための NotifyPostWriteMemory 関数の作成例を示します。その他の関数についても NotifyPostWriteMemory 関数と同じようにタイマ A0 と A-D 変換の内容を組み合わせて作成してください。

「NotifyPostWriteMemory 関数の作成例」

```
void NotifyPostWriteMemory(unsigned long address, int length, unsigned long data)
{
    // タイマ A0 の設定
    if (address == TABSR) {
        if ((data & 0x01) == 0x01) {            // カウント開始フラグのチェック
            sCountFlag = TRUE;
        }
    } else if (address == TA0) {                // タイマ A0 レジスタへの書き込みを監視
        sTmReloadReg = data;                    // リロードレジスタに値を設定
    }

    // A-D 変換の設定
    unsigned long           adicData;

    if (sAdDataExist == TRUE) {     // A-D データの入力開始を示すフラグをチェック
        if (address == ADCON0) {    // A-D 制御レジスタ 0 の A-D 変換開始フラグの書き込みを監視
            if ((data & 0x40) == 0x40) {        // A-D 変換開始フラグが 1 なら break
                RequestGetMemory(ADIC, 1, &adicData);
                RequestInterrupt(ADINT, adicData & 0x7);
                                                // A-D 割り込みの発生
                                                // 優先順位は割り込み制御レジスタを参照
                RequestPutMemory(ADCON0, 1, data & 0xbf);     // A-D 変換開始フラグをクリア
                RequestPutMemory(ADIC, 1, adicData | 0x08);   // 割り込み要求ビットのセット
            }
        }
    }
}
```

4. I/O DLL サンプルプログラムの仕様

4.1.タイマ A0（タイマモード）のサンプルプログラム（timerA0_TMode.dll）

このサンプルプログラムでは、タイマ A0 のタイマモードの動作をシミュレーションします。

「タイマの機能」

カウントソース： サイクル数をカウント

パルス出力機能： なし

ゲート機能　　：なし

「動作」

1. カウント開始フラグを 1 にすると 16 サイクル毎に、タイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを続けます。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。
3. カウント開始フラグを 0 にすると、カウント値を保持してダウンカウントを停止します。
4. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

4.2. タイマ A0（タイマモード、ゲート機能選択時）のサンプルプログラム（timerA0_TMod_Gate.dll）

このサンプルプログラムでは、タイマ A0 のタイマモード（ゲート選択時）の動作をシミュレーションします。

「タイマの機能」

カウントソース： サイクル数をカウント

パルス出力機能： なし

ゲート機能　　：TA0IN 端子が 1（"H"レベル）の期間だけカウントを行う。

「動作」

1. カウント開始フラグが 1 で TA0IN 端子の入力が 1（"H"レベル）の時、16 サイクル毎にタイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. TA0IN 端子の入力が 0（"L"レベル）の時、カウント値を保持してダウンカウントを停止します。
3. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを続けます。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。
4. カウント開始フラグを 0 にすると、カウント値を保持してダウンカウントを停止します。
5. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「TA0IN 端子の入力信号の作成方法」

iofunc.cpp ファイルの下記の部分を変更して入力信号を作成します。この例では、10000 サイクル毎に TA0IN 端子の入力信号の H/L を切り替えています。

```
#define HL_SWITCH_CYCLE        10000
                                    // 10000 サイクル毎に TA0IN 端子の入力信号の H/L を切り替え
```

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

4.3. タイマ A0（タイマモード、パルス出力機能選択時）のサンプルプログラム（timerA0_TMod_Pulse.dll）

このサンプルプログラムでは、タイマ A0 のタイマモード（パルス出力機能選択時）の動作をシミュレーションします。

「タイマの機能」

カウントソース： サイクル数をカウント
パルス出力機能： あり
ゲート機能　　：なし

「動作」

1. カウント開始フラグを 1 にすると 16 サイクル毎に、タイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを続けます。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。また、TA0OUT 端子の出力極性が反転します。
3. カウント開始フラグを 0 にすると、カウント値を保持してダウンカウントを停止します。また、TA0OUT 端子は 0（"L"レベル）を出力します。
4. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

「注意」

TA0OUT 端子の出力の変化は、I/O ウィンドウの仮想ポート出力機能を利用して参照することはできません。

4.4. タイマ A0（ワンショットタイマモード）のサンプルプログラム（timerA0_OneShotTM.dll）

このサンプルプログラムでは、タイマ A0 のワンショットタイマモードの動作をシミュレーションします。

「タイマの機能」

カウントソース　：サイクル数をカウント
パルス出力機能　：なし
カウント開始条件：ワンショット開始フラグへの 1 の書き込み

「動作」

1. カウント開始フラグが 1 の状態でワンショット開始フラグを 1 にすると 16 サイクル毎に、タイマ A0 レジスタの内容を 16 づつダウンカウントします。
2. アンダフローすると、タイマ A0 レジスタにリロードレジスタの内容をリロードしてカウントを停止します。
   同時に、タイマ A0 割り込み要求ビットが 1 になります。
3. カウント中にワンショット開始フラグに 1 が書き込まれた場合、再度リロードレジスタの値をリロードしてカウントを続けます。
4. カウント開始フラグを 0 にすると、ダウンカウントを停止し、リロードレジスタの内容をリロードします。
5. 割り込みが受け付けられると、タイマ A0 割り込み要求ビットが 0 になります。

「実チップのタイマとの違い」

1. プログラムでタイマ A0 レジスタの値を変更すると、直ちにタイマ A0 レジスタとリロードレジスタに書き込まれます。
   →実チップでは、直ちに書き込まれずリロードレジスタに保持されます。
2. プログラムでタイマ A0 割り込み要求ビットを 0 にしても、タイマ割り込みの要求をキャンセルできません。
3. カウントソースは、f1（1 分周）に固定です。

4.5. A-D 変換器（単発モード）のサンプルプログラム（AD.dll）

このサンプルプログラムでは、A-D 変換器の単発モードの動作をシミュレーションします。なお、サンプルプログラムでは A-D レジスタ 0 への入力をシミュレーションします。

「PD30SIM における A-D 変換器の考え方（実チップとの動作の違い）」

PD30SIM では、アナログ入力のシミュレーションをサポートしていません。そのため、A-D 変換後の入力値を A-D レジスタに入力することで、A-D 変換器の機能を擬似的に実現します。

「動作」

1. A-D 変換開始フラグを 1 にすると、A-D 変換割り込みが発生します。同時に A-D レジスタ 0 にデータを入力します。
2. その時、A-D 変換割り込み要求ビットが 1 になります。また、A-D 変換開始フラグが 0 になり、A-D 変換器の動作を停止します。
3. 割り込みが受け付けられると、A-D 変換割り込み要求ビットが 0 になります。
4. その後、再び A-D 変換開始フラグを 1 にすると、入力データがなくなるまで上記の 1～3 の動作を行います。

「A-D 入力データの作成方法」

infunc.cpp ファイルの下記の部分を変更して A-D 入力データを作成します。

```
#define AD_DATA_NUM       32          // 入力する A-D データ数

void NotifyInterrupt(unsigned long vec)
{
    unsigned long   adicData;
    // A-D データの設定。ここに入力データを定義します。
    // A-D 変換割り込みが発生する毎に、0x0、0x1、0x2 の順にデータを入力します。
    static const char adData[AD_DATA_NUM] =
        0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7,
        0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
    };      // 入力する A-D データ
    :
    :
}
```

「注意」

プログラムで A-D 変換割り込み要求ビットを 0 にしても、A-D 変換割り込みの要求をキャンセルできません。

4.6. シリアル I/O（受信）のサンプルプログラム（SIO.dll）

このサンプルプログラムでは、シリアル I/O（受信）の動作をシミュレーションします。なお、サンプルプログラムでは UART0 への入力をシミュレーションします。

「サンプルプログラムにおけるシリアル I/O の考え方（実チップとの動作の違い）」

サンプルプログラムでは、シリアル I/O の入力として RxD から入力信号を取り込むのではなく、UART 受信バッファレジスタに直接データを入力することで簡易的にシリアル I/O の動作をシミュレーションしています。

なお、サンプルプログラムは、クロック同期型シリアルモード、クロック非同期型シリアルモード共通で使用できます。但し、クロック非同期型シリアルモードでの受信データ長は 8 ビットです。

「動作」

1. 受信許可ビットを 1 にすると、UART0 受信割り込みが発生します。同時に UART0 受信バッファにデータを入力します。
2. その時、受信完了フラグと UART0 割り込み要求ビットが 1 になります。
3. 受信完了フラグは、UART0 受信バッファレジスタの下位バイトを読み出したとき 0 になります。
4. 割り込みが受け付けられると、UART0 割り込み要求ビットが 0 になります。
5. その後、10000 サイクル毎に入力データがなくなるまで上記の 1～4 の動作を行います。

「受信データの作成方法」

iofunc.cpp ファイルの下記の部分を変更して受信データを作成します。

```
#define SIO_DATA_NUM      32          // 入力する受信データ数

void NotifyInterrupt(unsigned long vec)
{
    unsigned long   u0c1Data;
    unsigned long   s0ricData;
    // 受信データの設定。ここに受信データを定義します。
    // UART0 割り込みが発生する毎に、0x0、0x1、0x2 の順にデータを入力します。
    static const char sioData[SIO_DATA_NUM] =
        0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7,
        0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
    };      // 入力する受信データ
    :
    :
}
```

「受信データ入力タイミングの変更方法」

iofunc.cpp ファイルの下記の部分を変更して受信データ入力タイミングを変更します。サンプルプログラムでは、10000 サイクル毎にデータを入力します。

```
// ここを変更して、任意サイクル毎にデータを入力するように変更できます。
#define SIO_INPUT_CYCLE   10000      // 10000 サイクル毎に受信データを入力
```

「注意」

1. プログラムで UART0 割り込み要求ビットを 0 にしても、UART0 割り込みの要求をキャンセルできません。
2. エラー検知は、オーバランエラーだけを検知できます。

4.7. CRC 演算回路のサンプルプログラム（CRC.dll）

このサンプルプログラムでは、CRC 演算回路の動作をシミュレーションします。

「動作」

1. CRC データレジスタに初期値 0 を設定します。
2. CRC インプットレジスタに 1 バイトのデータを書き込むと、書き込んだデータと CRC レジスタの内容に基づいて、CRC コードが CRC データレジスタに生成されます。
3. 連続数バイト CRC 演算を行う場合には、続けて次のデータを CRC インプットレジスタに書き込んで下さい。
4. 全データを書き終えた後の CRC データレジスタの内容が CRC 符号となります。

「実チップの CRC 演算回路との違い」

1. 実チップでは、CRC コードの生成に 2 マシンサイクルを要しますが、サンプルプログラムではサイクルを要しません（加算されません）。

4.8. DMAC のサンプルプログラム（DMAC.dll）

このサンプルプログラムでは、DMAC のリピート転送の動作をシミュレーションします。
DMA0 と DMA1 の動作を共にシミュレーションできます。

「DMA0、DMA1 の機能」

転送空間　　：　以下の 3 種類
(1) 1M バイトの任意の空間から固定アドレス
(2) 固定アドレスから 1M バイトの任意空間
(3) 固定アドレスから固定アドレス

DMA 転送要因　：　タイマ A0

チャネルの優先順位：　DMA0 の転送要求と DMA1 の転送要求が同時に発生した場合、DMA0 の転送が優先して行われます。

転送モード　　：　リピート転送のみ

転送単位　　：　8 ビット/16 ビット

「動作」

1. DMA 許可ビットが 1 のときタイマ A0 割り込みが発生すると、DMA 要求が受け付けられます。
2. DMA 要求が受け付けられると、指定された転送空間、転送単位で DMA 転送を行います。
3. DMA0/1 転送カウンタがアンダフローしても DMA 許可ビットは 1 のままです。DMA0/1 転送カウンタがアンダフローしたとき DMA0/1 割り込み要求ビットが 1 になります。
4. DMA0/1 割り込みが受け付けられると、DMA0/1 割り込み要求ビットが 0 になります。
5. DMA0/1 転送カウンタがアンダフローした後、次の DMA 要求が発生すると 1.に戻り、DMA 転送を繰り返します。

「実チップの DMAC との違い」

1. プログラムで DMA0/1 割り込み要求ビットを 0 にしても、DMA0/1 割り込みの要求をキャンセルできません。
2. 転送モードには、リピート転送のみ指定できます。
3. DMA 要求ビットのシミュレーションを行っていないため、DMA 許可ビットが 1 のとき、タイマ A0 割り込みが発生すると DMA 要求が必ず受け付けられます。

「DMA 転送要因として他の割り込みを使用する場合の変更方法」

タイマ A1 に変更する場合を例にとって説明します。iofunc.cpp の以下の箇所を変更します。

```
// タイマ A1 割り込み番号 22 を定義します。
#define TA1INT      22          // タイマ A1 割り込み番号


void NotifyInterrupt(unsigned long vec)
{
    // DMA0 の変更箇所
    if (sDma0StartFlag == TRUE) {      // DMA0 開始
        // ここをタイマ A1 割り込み番号に変更します。
        if (vec == TA1INT) {      // タイマ A1 割り込みの発生を監視
            :
            :
        }
    }
    // DMA1 の変更箇所
    if (sDma1StartFlag == TRUE) {      // DMA1 開始
        // ここをタイマ A1 割り込み番号に変更します。
        if (vec == TA1INT) {   // タイマ A1 割り込みの発生を監視
            :
            :
        }
    }
    :
    :
}
```

上記のように、DMA 転送要因となる割り込みのベクタ番号を指定して下さい。

4.9. MR30 用のタイマのサンプルプログラム（MR.dll）

このサンプルプログラムでは、MR30 でタイマ A0 を使用する場合の動作をシミュレーションします。

PD30SIM で MR30 を使用したアプリケーションプログラムを実行する場合、以下の設定を行う必要があります。

MR30 では、周期ハンドラやアラームハンドラ、get_tim システムコール、dly_tsk システムコール等が参照するシステムクロックを設定するために、以下の例のようにコンフィグレーションファイル中で MR30 が使用するタイマとタイマ割り込みが発生する時間間隔を指定します。

```
clock{
        mpu_clock       = 10MHz;
        timer           = A0;
        IPL             = 4;
        unit_time       = 100ms;        // ms
        initial_time    = 0:0:0;
};
```

この例の場合、システムのタイマ割り込み用にタイマ A0 を使用し、タイマ A0 の割り込み発生間隔を 100ms に設定しています。

PD30SIMでMR30を用いたアプリケーションプログラムを実行する場合､MR30で使用するタイマの設定（この場合タイマ A0）を行う必要があります。

MR.dll は、MR30 で使用するタイマの動作をシミュレートします。MR.dll を PD30SIM で読み込むことにより MR30 のアプリケーションのシミュレートが行えます。

「タイマ A0 以外のタイマを使用する場合の変更方法」

タイマ A3 に変更する場合を例にとって説明します。

●タイマ A0 のサンプルプログラム

```
#define   TA0IC    0x55     // タイマ A0 割り込み制御レジスタ
#define   TABSR    0x380    // カウント開始フラグ
#define   TA0      0x386    // タイマ A0 レジスタ
#define   TA0INT   21       // タイマ A0 割り込み番号

void NotifyStepCycle(int cycle)
{
        :
        :
    RequestGetMemory(TABSR, 1, &tabsrData);
    if ((tabsrData & 0x01) == 0x01) {     // カウント開始フラグのチェック
        sCountCycle += cycle;
        RequestGetMemory(TA0, 2, &ta0Data);
        if (sCountCycle >= ta0Data + 1) {           // 分周比分カウントダウン
            RequestGetMemory(TA0IC, 1, &ta0icData);
            RequestInterrupt(TA0INT, ta0icData & 0x7);
                // タイマ A0 割り込みの発生、優先順位は割り込み制御レジスタを参照
            sCountCycle = 0;
        }
    }
}
```

```
void NotifyPostWriteMemory(unsigned long address, int length, unsigned long data)
{
    if (address == TABSR) {
        if ((data & 0x01) == 0x01) {       // カウント開始フラグのチェック
            sCountFlag = TRUE;
        }
    }
}
```

↓　タイマ A3 を使用するように以下の箇所を変更します。

●タイマ A3 のサンプルプログラム

```
#define   TA3IC    0x58      // タイマ A3 割り込み制御レジスタ
#define   TABSR    0x380     // カウント開始フラグ
#define   TA3      0x38c     // タイマ A3 レジスタ
#define   TA3INT   24        // タイマ A3 割り込み番号

void NotifyStepCycle(int cycle)
{
        :
        :
    RequestGetMemory(TABSR, 1, &tabsrData);
    if ((tabsrData & 0x08) == 0x08) {     // カウント開始フラグのチェック
        sCountCycle += cycle;
        RequestGetMemory(TA3, 2, &ta0Data);
        if (sCountCycle >= ta0Data + 1) {          // 分周比分カウントダウン
            RequestGetMemory(TA3IC, 1, &ta0icData);
            RequestInterrupt(TA3INT, ta0icData & 0x7);
                    // タイマ A3 割り込みの発生、優先順位は割り込み制御レジスタを参照
            sCountCycle = 0;
        }
    }
}

void NotifyPostWriteMemory(unsigned long address, int length, unsigned long data)
{
    if (address == TABSR) {
        if ((data & 0x08) == 0x08) {       // カウント開始フラグのチェック
            sCountFlag = TRUE;
        }
    }
}
```

以上

[MEMO]

PDxxSIM　I/O DLL キット
サンプルマニュアル

ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668