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。

お客様各位

# 資料中の「三菱電機」、「三菱XX」等名称の株式会社ルネサス テクノロジへの変更について

2003年4月1日を以って株式会社日立製作所及び三菱電機株式会社のマイコン、ロジック、アナログ、ディスクリート半導体、及びDRAMを除くメモリ(フラッシュメモリ・SRAM等)を含む半導体事業は株式会社ルネサス テクノロジに承継されました。
従いまして、本資料中には「三菱電機」、「三菱電機株式会社」、「三菱半導体」、「三菱XX」といった表記が残っておりますが、これらの表記は全て「株式会社ルネサス　テクノロジ」に変更されておりますのでご理解の程お願い致します。尚、会社商標・ロゴ・コーポレートステートメント以外の内容については一切変更しておりませんので資料としての内容更新ではありません。

注:「高周波・光素子事業、パワーデバイス事業については三菱電機にて引き続き事業運営を行います。」

2003年4月1日
株式会社ルネサス テクノロジ
カスタマサポート部

User's Manual

# エミュレータデバッガ
# PD30 用　カスタムビルダ

## CB30 V.1.00　プログラミングマニュアル

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　1998.02

《安全設計に関するお願い》

三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

《本資料ご利用に際しての留意事項》

- 本資料は、お客様が用途に応じた適切な三菱半導体製品をご購入いただくための参考資料であり、本資料中に記載の技術情報について三菱電機株式会社・三菱電機セミコンダクタシステム株式会社が所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は責任を負いません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は、予告なしに、本資料に記載した製品または仕様を変更することがあります。三菱半導体製品のご購入に当たりましては、事前に三菱電機株式会社・三菱電機セミコンダクタシステム株式会社または特約店へ最新の情報をご確認頂きますとともに、三菱電機半導体情報ホームページ（http://www.semicon.melco.co.jp/）および三菱開発ツールホームページ（http://www.tool-spt.mesc.co.jp/）などを通じて公開される情報に常にご注意ください。
- 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社はその責任を負いません。
- 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は、適用可否に対する責任は負いません。
- 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社または特約店へご照会ください。
- 本資料の転載、複製については、文書による三菱電機株式会社・三菱電機セミコンダクタシステム株式会社の事前の承諾が必要です。
- 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたら三菱電機株式会社・三菱電機セミコンダクタシステム株式会社または特約店までご照会ください。

製品の内容及び本書についてのお問い合わせ先

電子メールの場合：インストーラが生成する以下のテキストファイルに必要事項を記入の上、開発ツールサポート窓口support@tool.mesc.co.jpまで送信ください。

Windows 98/95/Windows NT 4.0版：\SUPPORT\製品名\SUPPORT.TXT
EWS版：/support/製品名/toolinfo.txt

FAXの場合：各ユーザーズマニュアル及びガイドブックの最後に添付されている「技術サポート連絡書」に必要事項を記入の上、開発ツールサポート窓口まで送信ください。FAX送信先は「技術サポート連絡書」に記載してあります。

# 1.概要

## 1.1.本マニュアルの概要

本マニュアルでは、CB30を用いてカスタムコマンド/カスタムウィンドウを作成する際の、プログラムの記述方法について解説します。CB30の使用方法については、「CB30 V.1.00 ユーザーズマニュアル」をご参照ください。

## 1.2.CB30とは？

CB30とは、PD30で動作するお客様独自のコマンドやウィンドウを作成して頂くための全く新しい開発環境です。

## 1.3.CB30で何ができる？

CB30を利用して、PD30のコマンド(カスタムコマンド)/ウィンドウ(カスタムウィンドウ)を簡単に作成(プログラミング)できます。

CB30ではカスタムコマンド/カスタムウィンドウのプログラミングからコンパイル、デバッグまで全て統合的にサポートします。

作成したカスタムコマンド/カスタムウィンドウは、PD30上でご使用頂けます。

この機能により、お客様ご自身でPD30を簡単に機能アップ、カスタマイズして頂けます。

CB30を利用して作成するカスタムコマンド/カスタムウィンドウはエミュレータ(PC4701HS/L)を直接制御できますので、

- ターゲットメモリ内容の参照や変更
- ターゲットプログラムの実行や停止、ソース行ステップ実行等の各種制御
- ターゲットシステムの自動テスト環境の構築

等、お客様が希望される様々なデバッグ機能を簡単に実現できます。

## 1.4.CB30の特長とは？

1. PD30と同様のウィンドウデザインを採用し、PD30と操作性の統一を図っています。
2. プログラミング、コンパイル、デバッグを統合した開発環境を提供します。
3. PD30で動作するコマンドとウィンドウをお客様ご自身で作成して頂けます。
4. CB30で記述可能なプログラム記述言語は、C 言語のサブセットをサポートします。
5. CB30では以下のような各種ライブラリをご提供します。
   - 標準関数ライブラリ(stdlib.lib)
   - エミュレータ操作関数ライブラリ(system.lib)
   - ウィンドウ操作関数ライブラリ(winlib.lib)

# 2.カスタムコマンドプログラミング

本章では、カスタムコマンドのプログラミング方法について説明します。

## 2.1.カスタムコマンドの作成手順

カスタムコマンドを作成する場合、CB30を操作して下記の手順で作業を進めます。

1. プロジェクトの作成
   プロジェクトとは、カスタムコマンドを作成する際に必要なソースプログラムの集合のことです。１つのカスタムコマンドを作成する場合、１つのプロジェクトを作成します。プロジェクトの作成方法については、「CB30 V.1.00 ユーザーズマニュアル」の「3.1.1. カスタムコマンドプログラム用新規プロジェクトの作成方法」をご参照ください。

2. ソースプログラムの作成
   ソースファイルにカスタムコマンドの動作を記述します。ソースファイルの作成方法については、「CB30 V.1.00 ユーザーズマニュアル」の「3.1.2. 新規ソースファイルの作成方法」をご参照ください。ソースファイルのプロジェクトへの登録方法については、「CB30 V.1.00 ユーザーズマニュアル」の「3.1.3. ソースファイルのプロジェクトへの登録方法」をご参照ください。

3. ビルド
   ビルドとは、ソースプログラムを翻訳し、カスタムコマンドプログラムを生成することです。ビルドの方法については、「CB30 V.1.00 ユーザーズマニュアル」の「3.1.4. ビルドの方法」をご参照ください。

4. デバッグ
   作成したカスタムコマンドプログラムが仕様通りに動作しない場合には、デバッグを行います。デバッグについては、「CB30 V.1.00 ユーザーズマニュアル」の「2.1. CB30ウィンドウ」をご参照ください。

5. カスタムコマンドのPD30への登録
   完成したカスタムコマンドを使用する場合には、PD30に登録します。登録方法については、「PD30 V.3.00 ユーザーズマニュアル」の「カスタマイズ機能」をご参照ください。

本章で取り上げるのは、2.ソースプログラムの作成におけるプログラミング方法です。その他の作業内容については、「CB30 V.1.00 ユーザーズマニュアル」の該当する章をご参照ください。

## 2.2.最も簡単なカスタムコマンドのソースプログラム例

ここでは、最も簡単なカスタムコマンドを例に挙げて、ソースプログラム例を紹介し、プログラミング方法について解説します。

- カスタムコマンドの例

| | |
|---|---|
| コマンド名 | hello |
| 書式 | hello アドレス<RET> |
| 内容 | ● "Hello CB30 World!"の文字をスクリプトウィンドウに表示する。<br>● その後、スクリプトウィンドウから文字列を入力する。<br>● 文字列入力後、入力された文字列をコマンドの第一パラメータで指定したアドレスに格納する。<br>● 処理中の何らかのエラーが発生した場合、コマンドを終了する。 |

- ソースプログラム例

```
#include <stdlib.h>
#include <system.h>

int main(int argc, char **argv)             /* 1. プログラムは main()関数か */
                                            /* ら実行される */
{
        char    str[128];
        int     val, i, len;

        if(argc != 2){                      /* 2. パラメータが１つでない場 */
                exit(0);                    /* 合、コマンドを終了する */
        }
        printf("Hello CB30 World!\n");      /* 3. スクリプトウィンドウに文 */
                                            /* 字列を出力する */
        if(gets(str) == NULL){              /* 4. スクリプトウィンドウから */
                                            /* 文字列を入力する */
                exit(1);
        }
        if(_exp_eval(argv[1], EXP_DEFAULT, EXP_LABEL, &val) == FALSE){
                                            /* 5. アセンブラ式を解析し、値を */
                                            /* 取得する */
                exit(1);
        }
        len = strlen(str);
        for(i = 0; i < len; i++){
                                            /* 6. メモリ内容を変更する */
                if(_mem_put(val + i, 1, &(str[i])) == FALSE){
                        exit(1);
                }
        }
        exit(0);
}
```

- 解説

1. カスタムコマンドのソースプログラムは、main()関数から始まります。使用するプログラミング言語は、C 言語サブセットのCB30専用の言語です。言語仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「4. プログラム言語仕様」に記載されています。C 言語との大きな違いは、
   - 集合体（構造体・共用体）は未サポート
   - 実数型（float ・ double）は未サポート

   です。argc には引数の個数、argv には引数に指定された文字列が格納されている領域へのポインタが格納されているポインタ配列のアドレスが格納されます。これは、標準の C 言語の main()関数の引数の扱いと同じです。
   なお、この例では、関数は main()関数一つですが、C 言語と同様に複数のユーザ定義関数を記述することができます。

2. コマンドを途中終了する場合は、exit()関数を使用します。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.1.9 exit: 実行の終了」に記載されています。

3. スクリプトウィンドウ上に文字列を表示するには、printf()関数を使用します。使用方法は、C 言語の printf()関数とほぼ同様です。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.1.17 printf: 書式付出力(ScriptWindow への出力)」に記載されています。

4. スクリプトウィンドウから文字列を入力するには、gets()関数を使用します。使用方法は、C 言語の gets()関数とほぼ同様です。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.1.8 gets: 文字列の入力(ScriptWindow からの入力)」に記載されています。

5. アセンブラ式を解析し、値を取得するには、_exp_eval()関数を使用します。_exp_eval()関数で解析する式では、ラベル・シンボル等が使用できます。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.2.59 _exp_eval: アセンブラ式解析」に記載されています。

6. ターゲット CPU のメモリに値を設定するには、_mem_put()関数を使用します。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.2.18 _mem_put: メモリ値設定」に記載されています。

## 2.3.カスタムコマンドのプログラミングで使用できる関数

カスタムコマンドのプログラミングで使用できる関数群は、大きく下記の 2 種類に分類されます。

1. 標準関数
   C 言語の標準関数の中でも比較的使用頻度が高いと思われる関数と同等のものをサポートしています。

2. デバッガ操作関数
   デバッガを操作するための関数をサポートしています。

## 2.4.標準関数の使い方

標準関数を使用する際は、ヘッダファイル stdlib.h をインクルードしてください。

標準関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.1. 標準関数 (stdlib.lib)」をご参照ください。

### 2.4.1.ヒープ領域操作関数の使い方

本節では、以下の関数を使用してヒープ領域操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| malloc | ヒープ領域からのメモリの確保 |

[プログラム例]

```
char *regist_name(char *name)
{
    char    *p;
    int     len;

    if(name != NULL){                   /* name が NULL 以外の場合のみ処理する */
            len = strlen(name);         /* 文字列の長さを取得する */
            p = malloc(len + 1);        /* len + 1 バイトの領域を確保する */
            if(p == NULL){              /* p が NULL の場合には領域確保に失敗 */
                    return NULL;        /* エラー */
            }
            strcpy(p, name);            /* 文字列を複写する */
            return p;           /* 格納した領域を返す */
    }
    return NULL;                        /* エラー */
}
```

上記のプログラム例は、malloc()関数を用いて、引数 name で指定された文字列を、ヒープ領域に格納する、ユーザ定義関数を作成した例です。

### 2.4.2.文字列操作関数の使い方

本節では、以下の関数を使用して文字列操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| strcmp | 文字列の比較 |
| strtoi | 文字列の数値への変換 |
| sprintf | 書式付出力（メモリへの出力） |

[プログラム例]

```
int eval_str(char *str1, char *str2, char *str3)
{
    int     value;

    if(strcmp(str1, "go") == 0){                    /* str1 が "go" の時 */
        if(strtoi(str2, 0, &value) == TRUE){
                                                    /* str2 が数値に変換できた時 */
                                                    /* str3 に書式付で出力 */
            sprintf(str3, "%X(%d)", value, value);
            return TRUE;                            /* 成功 */
        }
    }
    return FALSE;                                   /* エラー */
}
```

上記のプログラム例は、引数 str1 が"go"の時に、引数 str2 で指定された数値を表す文字列を数値に変換し、引数 str3 で指定された領域に書式付きで出力する、ユーザ定義関数を作成した例です。

### 2.4.3.入出力関数の使い方

本節では、以下の関数を使用して入出力関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| gets | 文字列の入力（Script Window からの入力） |
| printf | 書式付出力（Script Window への出力） |

[プログラム例]

```
int echo_str()
{
    char    str[1024];

    if(gets(str) != NULL){                          /* 文字列が取得できた */
        printf("Your input is [%s].\n", str);       /* 書式付で出力 */
        return TRUE;                                /* 成功 */
    }
    return FALSE;                                   /* エラー */
}
```

上記のプログラム例は、Script Window の入力領域に入力された文字列を、Script Window の表示領域に書式付で出力する、ユーザ定義関数を作成した例です。

### 2.4.4.ファイル操作関数の使い方

本節では、以下の関数を使用してファイル操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| fopen | ファイルのオープン |
| fclose | ファイルのクローズ |
| fprintf | 書式付出力（ファイルへの出力） |

[プログラム例]

```
int put_file(char *filename, int data1, int data2, int data3, int data4)
{
    int     fd;

    if((fd = fopen(filename, "w")) == NULL){     /* ファイルをオープンする */
            return FALSE;                        /* エラー */
    }
    fprintf(fd, "Data1 = %d\n", data1);          /* data1を出力 */
    fprintf(fd, "Data2 = %d\n", data2);          /* data2を出力 */
    fprintf(fd, "Data3 = %d\n", data3);          /* data3を出力 */
    fprintf(fd, "Data4 = %d\n", data4);          /* data4を出力 */
    fclose(fd);                                  /* ファイルをクローズ */
    return TRUE;                                 /* 成功 */
}
```

上記のプログラム例は、引数 filename で指定されたファイルを作成し、引数 data1～data4 で指定されたデータを書式付きで出力する、ユーザ定義関数を作成した例です。

## 2.5.デバッガ操作関数の使い方

デバッガ操作関数を使用する際は、ヘッダファイル system.h をインクルードしてください。

デバッガ操作関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.2. デバッガ操作系システムコール関数(system.lib)」をご参照ください。

### 2.5.1.実行操作関数の使い方

本節では、以下の関数を使用して実行操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _cpu_go | フリーラン実行 |
| _cpu_stop | 実行停止 |
| _cpu_reset | ターゲットリセット |

[プログラム例]

```
int go_stop_10()
{
    int     i;

    for(i = 0; i < 10; i++){                    /* 10 回繰り返し */
            if(_cpu_go() == FALSE){             /* フリーラン実行 */
                    return FALSE;               /* 同エラー */
            }
            if(_cpu_stop() == FALSE){           /* 実行停止 */
                    return FALSE;               /* 同エラー */
            }
    }
    if(_cpu_reset() == FALSE){                  /* ターゲットリセット */
            return FALSE;                       /* 同エラー */
    }
    return TRUE;                                /* 成功 */
}
```

上記のプログラム例は、フリーラン実行/実行停止を 10 回繰り返した後に、ターゲットをリセットする、ユーザ定義関数を作成した例です。

### 2.5.2.レジスタ操作関数の使い方

本節では、以下の関数を使用してレジスタ操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _reg_get_pc | プログラムカウンタ値取得 |
| _reg_put_reg | レジスタ値設定 |

[プログラム例]

```
int pc_inc_intb()
{
    int     reg;

    if(_reg_get_pc(&reg) == FALSE){                 /* PC 値取得 */
            return FALSE;                           /* 同エラー */
    }
    reg += 1;                                       /* PC 値 + 1 */
    if(_reg_put_reg(reg, レジスタ番号) == FALSE){    /* レジスタ値設定 */
            return FALSE;                           /* 同エラー */
    }
    return TRUE;                                    /* 成功 */
}
```

上記のプログラム例は、現在のプログラムカウンタ値に＋１した値をレジスタ番号で指定したレジスタに設定する、ユーザ定義関数を作成した例です。

レジスタ番号は、「CB30 V.1.00 ユーザーズマニュアル」の「5.2.13._reg_put_reg: レジスタ値設定」をご参照ください。

### 2.5.3.メモリ操作関数の使い方

本節では、以下の関数を使用してメモリ操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _mem_get | メモリ値取得 |
| _mem_put | メモリ値設定 |

[プログラム例]

```
int inc_1000H()
{
    char    data[128];
    int     i;

    if(_mem_get(0x1000, 128, data) == FALSE){   /* 1000H 番地から 128 バイトを取得 */
            return FALSE;                       /* 同エラー */
    }
    for(i = 0; i < 128; i++){                   /* 128 回繰り返し */
            (data[i])++;                        /* データをインクリメント */
    }
    if(_mem_put(0x1000, 128, data) == FALSE){   /* 1000H 番地から 128 バイトを設定 */
            return FALSE;                       /* 同エラー */
    }
    return TRUE;                                /* 成功 */
}
```

上記のプログラム例は、1000H 番地からの 128 バイトのメモリ値を＋１する、ユーザ定義関数を作成した例です。

### 2.5.4.ソフトウェアブレーク操作関数の使い方

本節では、以下の関数を使用してソフトウェアブレーク操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _break_set | ソフトウェアブレーク設定/有効化 |
| _break_reset | ソフトウェアブレーク解除 |

[プログラム例]

```
int go_F000H()
{
    if(_break_set(0xF000) == FALSE){    /* F000H 番地にソフトウェアブレークを設定 */
            return FALSE;               /* 同エラー */
    }
    if(_cpu_gb() == FALSE){             /* ブレーク付き実行 */
            return FALSE;               /* 同エラー */
    }
    _cpu_wait();                        /* ターゲット実行停止まで待つ */
    if(_break_reset(0xF000) == FALSE){  /* F000H 番地のソフトウェアブレークを解除 */
            return FALSE;               /* 同エラー */
    }
    return TRUE;                        /* 成功 */
}
```

上記のプログラム例は、F000H 番地まで実行する、ユーザ定義関数を作成した例です。

### 2.5.5.デバッグ情報操作関数の使い方

本節では、以下の関数を使用してデバッグ情報操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _line_addr2line | アドレスのソース行取得 |
| _exp_eval | アセンブラ式解析 |
| _c_exp_eval | C 言語式解析 |

[プログラム例]

```
int str_eval(char *str, int *is_c, char *filename, int *line, int *find_line)
{
    int     value, val;
    char    s1[128], s2[128], s3[128];

    if(_exp_eval(str, EXP_DEFAULT, EXP_LABEL, &value) == TRUE){
            *is_c = FALSE;                      /* アセンブラ式 */
    }else if(_c_exp_eval(str, &value, &val, s1, s2, s3) == TRUE){
            *is_c = TRUE;                       /* C 言語式 */
    }else{
            return FALSE;                       /* 式解析エラー */
    }
    if(_line_addr2line(value, line, filename) == TRUE){
            *find_line = TRUE;          /* ソースファイル名、行番号あり */
    }else{
            *find_line = FALSE;                 /* ソースファイル名、行番号なし */
    }
    return TRUE;
}
```

上記のプログラム例は、引数 str で指定された文字列がアセンブラ式か C 言語式かを判定し、解析結果の値をアドレスとして対応するソースファイル名、行番号を取得する、ユーザ定義関数を作成した例です。

### 2.5.6.スクリプトコマンド実行関数の使い方

本節では、以下の関数を使用してスクリプトコマンド実行関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _syscom | PD30のスクリプトコマンドの実行 |

[プログラム例]

```
int DB(int addr)
{
    char    str[128];

    sprintf(str, "DumpByte %X", addr);          /* スクリプトコマンド文字列作成 */
    if(_syscom(str) == FALSE){                  /* スクリプトコマンド実行 */
        return FALSE;                           /* 同エラー */
    }
    return TRUE;                                /* 成功 */
}
```

上記のプログラム例は、引数 addr で指定されたアドレスを第一引数として DumpByte スクリプトコマンドを実行する、ユーザ定義関数を作成した例です。

### 2.5.7.DOS コマンド実行関数の使い方

本節では、以下の関数を使用して DOS コマンド実行関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _doscom | DOS コマンドの実行 |

[プログラム例]

```
int CP(char *src, char *dest)
{
    char    str[256];

    sprintf(str, "copy %s\\*.* %s\\*.*", src, dest);    /* DOS コマンド文字列作成 */
    if(_doscom(str) == FALSE){                          /* DOS コマンド実行 */
        return FALSE;                                   /* 同エラー */
    }
    return TRUE;                                        /* 成功 */
}
```

上記のプログラム例は、引数 src で指定されたディレクトリのファイルを、引数 dest で指定されたディレクトリにコピーする DOS コマンドを実行する、ユーザ定義関数を作成した例です。

# 3.カスタムウィンドウプログラミング

本章では、カスタムウィンドウのプログラミング方法について説明します。

## 3.1.カスタムウィンドウの作成手順

カスタムウィンドウを作成する場合、CB30を操作して下記の手順で作業を進めます。

1.プロジェクトの作成

プロジェクトとは、カスタムウィンドウを作成する際に必要なソースプログラムの集合のことです。１つのカスタムウィンドウを作成する場合、１つのプロジェクトを作成します。プロジェクトの作成方法については、「CB30 V.1.00 ユーザーズマニュアル」の「3.2.1. カスタムウィンドウプログラム用新規プロジェクトの作成方法」をご参照ください。

2.ソースプログラムの作成

プロジェクトを作成する際にCB30によって自動生成されるフレームワークソースファイルに、カスタムウィンドウの動作を記述します。フレームワークソースファイルの編集方法については、「CB30 V.1.00 ユーザーズマニュアル」の「3.2.2. CB30によって自動生成された、フレームワークソースファイルの編集方法」をご参照ください。

3.ビルド

ビルドとは、ソースプログラムを翻訳し、カスタムウィンドウプログラムを生成することです。ビルドの方法については、「CB30 V.1.00 ユーザーズマニュアル」の「3.1.4. ビルドの方法」をご参照ください。

4.デバッグ

作成したカスタムウィンドウプログラムが仕様通りに動作しない場合には、デバッグを行います。デバッグについては、「CB30 V.1.00 ユーザーズマニュアル」の「2.1. CB30ウィンドウ」をご参照ください。

5.カスタムウィンドウのPD30への登録

完成したカスタムウィンドウを使用する場合には、PD30に登録します。登録方法については、「PD30 V.3.00 ユーザーズマニュアル」の「カスタマイズ機能」をご参照ください。

本章で取り上げるのは、2.ソースプログラムの作成におけるプログラミング方法です。その他の作業内容については、「CB30 V.1.00 ユーザーズマニュアル」の該当する章をご参照ください。

## 3.2.最も簡単なカスタムウィンドウのソースプログラム例

ここでは、最も簡単なウィンドウコマンドを例に挙げて、ソースプログラム例を紹介し、プログラミング方法について解説します。

- カスタムウィンドウの例

| ウィンドウ名 | Hello Window |
|---|---|
| 内容 | ● タイトルに”Hello Window”を表示する。<br>● 大きさは 300×200（ピクセル）である。<br>● "Hello World!"の文字を表示し、改行する。 |

- ソースプログラム例（フレームワークソースファイルからの抜粋）

```
OnCreate()          /* 1. フレームワークソースファイルの OnCreate()関数 */
{
        /* Write message handler code here, please. */
                    /* 2. タイトルに”Hello Window”を表示 */
        _win_set_window_title("Hello Window");
                    /* 3. ウィンドウサイズを 300×200(ピクセル)に設定 */
        _win_set_window_size(300, 200);
}

OnDraw()            /* 4. フレームワークソースファイルの OnDraw()関数 */
{
        /* Write message handler code here, please. */
        int         pc;
                    /* 5. 描画開始位置を(0, 0)(カーソル座標)に設定 */
        _win_set_cursor(0, 0);
                    /* 6. ウィンドウに”Hello World!”を表示 */
        _win_printf("Hello World!\n");
        _reg_get_pc(&pc);
}
```

細字は、フレームワークソースファイルに自動生成されたコードを表します。
太字は、ユーザによって追加されたコードを表します。

- 実行例

- 解説

1.カスタムウィンドウ用プロジェクトを作成する際に、CB30によって自動生成されるソースファイル（フレームワークソースファイルと呼びます）には、予め幾つかの関数が記述されています。これらの関数は、カスタムウィンドウに対して何らかの操作を行った場合に、PD30によって自動的に呼び出される特殊な関数です。これらの関数をハンドル関数と呼びます。
OnCreate()ハンドル関数は、ウィンドウが生成される直前に呼び出され、ウィンドウフレームの大きさや、タイトル、必要な変数の初期化などを行います。
OnCreate()ハンドル関数は、カスタムウィンドウプログラムを起動した際、一番最初に実行される関数です。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.4 OnCreate ハンドル関数」に記載されています。

2.カスタムウィンドウのタイトルを設定する場合は、_win_set_window_title()関数を使用します。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.3.23 _win_set_window_title: カスタムウィンドウのタイトルの設定」に記載されています。

3.カスタムウィンドウのサイズを設定するには、_win_set_window_size()関数を使用します。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.3.38 _win_set_window_size: カスタムウィンドウのサイズの設定」に記載されています。

4.OnDraw()ハンドル関数は、他のウィンドウによって隠されているウィンドウの一部（または全部）が表示される場合などに呼び出され、ウィンドウの再描画などを行います。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.6 OnDraw ハンドル関数」に記載されています。

5.カーソル位置を設定するには、_win_set_cursor()関数を使用します。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.3.3 _win_set_cursor: カーソル位置の設定」に記載されています。

6.カスタムウィンドウの現在のカーソル位置に文字列を出力するには、_win_printf()関数を使用します。この時カーソル位置は、出力した最後の文字の次の位置に移動されます。関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.3.1 _win_printf: 書式付出力(カスタムウィンドウへの出力)」に記載されています。

## 3.3.ハンドル関数について

カスタムウィンドウは、PD30の１つのウィンドウとして機能します。したがって、カスタムウィンドウは、PD30および OS と情報交換しながら動作します。

カスタムウィンドウに対する操作や、システムタイマからの時間経過通知などが発生した場合には、PD30が操作や通知に対応するカスタムウィンドウプログラムのハンドル関数を呼び出します。ハンドル関数に記述されている処理は、そのハンドル関数が呼び出されるような操作をカスタムウィンドウに対して行った場合に実行されます。

全てのハンドル関数は、引数を持ちません。また、戻り値はPD30によって評価されません。

マウスを移動させた場合に呼び出される OnMouseMove()ハンドル関数や、ウィンドウの大きさを変更させた場合に呼び出される OnSize()ハンドル関数などでは、現在のマウスの座標や、変更されたウィンドウの大きさ等のデータが、ハンドル関数が呼び出される直前に、ライブラリ内にあるグローバル変数_HandleMsgBlock の示す領域に格納されます。

ハンドル関数内で、_HandleMsgBlock に格納されているデータを参照することにより、操作や通知に付属する情報が取得できます。データを取得する手続きは、フレームワークソースファイルに自動的に記述されています。

ハンドル関数についての詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4. カスタムウィンドウ用ハンドル関数」をご参照ください。

**（注意事項）　ハンドル関数は、PD30から呼び出される特殊な関数ですので、ユーザ定義関数のように自由に呼び出さないでください。PD30からの呼び出しが正しく行われなくなる恐れがあります。**

## 3.4.フレームワークソースファイルについて

CB30では、PD30によって呼び出される全てのハンドル関数と、データを取得する手続きが記述されたソースファイル(フレームワークソースファイル)を、プロジェクト作成時に自動生成します。

フレームワークソースファイルには、カスタムウィンドウに対する操作や通知に付属する情報を_HandleMsgBlock から取り出し、ハンドル関数のローカル変数にコピーするコードが記述されています。

カスタムウィンドウに対する操作に対応した処理を記述する際は、フレームワークソースファイルのハンドル関数の次のコメントの後に記述していください。

```
/* Write message handler code here, please. */
```

**（注意事項）　フレームワークソースファイルに記述されている、ハンドル関数を削除しないでください。正常にビルドできなくなります。また、ハンドル関数内に予め記述されているローカル変数の設定手続きを変更しないでください。カスタムウィンドウプログラムが正常に動作しなくなる恐れがあります。**

## 3.5.ハンドル関数の使い方

本節で、フレームワークソースファイルに記述されているハンドル関数の使用方法について説明します。

**（注意事項）　ハンドル関数は、PD30から呼び出される特殊な関数ですので、ユーザ定義関数のように自由に呼び出さないでください。PD30からの呼び出しが正しく行われなくなる恐れがあります。**

### 3.5.1.ウィンドウ作成開始ハンドル関数の使い方（OnCreate）

カスタムウィンドウプログラムの実行が開始され、カスタムウィンドウが作成される直前に一度だけ **OnCreate()**ハンドル関数が呼び出されます。**OnCreate()**ハンドル関数では、ウィンドウのオープン位置、オープン時の大きさ、ウィンドウのタイトル等の設定、コントロールアイテム（ボタンなど）の生成などを行います。

```
自動生成される OnCreate ハンドル関数を示します。(付属情報はありません。)
OnCreate()
{
        /* Write message handler code here, please. */

}
```

**OnCreate()**ハンドル関数は、カスタムウィンドウプログラムソースファイルにある関数の中で、最初に実行される関数です。

ウィンドウが生成されるときには、OnCreate → OnSize → OnDraw の順でハンドル関数が呼び出されます。

### 3.5.2.ウィンドウ破壊開始ハンドル関数の使い方（OnDestroy）

カスタムウィンドウのシステムメニューのクローズが選択され、カスタムウィンドウが破壊される直前に一度だけ **OnDestroy()**ハンドル関数が呼び出されます。**OnDestroy()**ハンドル関数では、ヒープ領域の解放、システムタイマの解放等を行います。コントロールアイテムの破壊は、本関数が実行された後に自動的に行われます。

```
自動生成される OnDestroy ハンドル関数を示します。(付属情報はありません。)
OnDestroy()
{
        /* Write message handler code here, please. */

}
```

**OnDestroy()**ハンドル関数は、カスタムウィンドウプログラムソースファイルにある関数の中で、最後に実行される関数です。

PD30は、**OnDestroy()**ハンドル関数の処理が終了した後に、使用されたコントロールアイテムを開放し、カスタムウィンドウを破壊し、カスタムウィンドウプログラムの実行を終了します。

### 3.5.3.ウィンドウ再描画要求ハンドル関数の使い方（OnDraw）

**OnDraw()**ハンドル関数は、以下の場合に呼び出されます。**OnDraw()**ハンドル関数では、ウィンドウの描画領域への描画を行います。

- カスタムウィンドウの一部または全部が他のウィンドウ等によって隠されており、その隠されていた部分が露出された場合。
  この場合は、**OnDraw()**ハンドル関数呼び出し直前に、ウィンドウの描画領域がクリアされます。
- ウィンドウ操作関数のウィンドウの再描画関数を呼び出した場合。
  再描画関数には、以下の２つがあります。
  1. _win_redraw_clear()
     この関数を呼び出した場合は、**OnDraw()**ハンドル関数呼び出し直前に、ウィンドウの描画領域がクリアされます。
  2. _win_redraw()
     この関数を呼び出した場合は、**OnDraw()**ハンドル関数呼び出し直前に、ウィンドウの描画領域はクリアされません。

```
自動生成される OnDraw ハンドル関数を示します。(付属情報はありません。)
OnDraw()
{
        /* Write message handler code here, please. */

}
```

**OnDraw()**ハンドル関数は、比較的頻繁に呼び出されますので、描画のみを行うようにし、描画に必要なデータ（メモリ値・レジスタ値など）の取得・加工等の時間がかかる処理は、次節で説明します **OnEvent()**ハンドル関数などで行うことをお勧めします。

### 3.5.4.PD30の状態変化ハンドル関数の使い方（OnEvent）

PD30の状態が変化した場合に、**OnEvent()**ハンドル関数が呼び出されます。**OnEvent()**ハンドル関数では、PD30の状態の変化に対応した処理を行います。

PD30の状態の変化とは、次のような事象を指します。

1.新しいターゲットプログラムをダウンロードした場合。
2.ターゲットプログラムをシングルステップ実行した場合。
3.レジスタ値を変更した場合。
4.その他、PD30が表示すべき情報が変更された場合。

PD30の状態の変化の種別は、ローカル変数 nEventID に渡されます。状態の種別の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.7 OnEvent ハンドル関数」に記載されています。

一つのハンドル関数で、複数の状態の変化に対応した処理を行う必要があるため、通常は関数の先頭で switch 文等でローカル変数 nEventID を判別し、状態の変化に対応した処理に分岐させます。

```
自動生成される OnEvent ハンドル関数を示します。(付属情報は nEventID です。)
OnEvent()
{
        int     nEventID;

        nEventID = ((int *)_HandleMsgBlock)[0];

        /* Write message handler code here, please. */

}
```

PD30のダンプウィンドウ等でターゲットメモリの値を変更した場合には、nEventID == EVENT_PUT_MEM で **OnEvent()**ハンドル関数が呼び出されますので、メモリ値を表示するようなカスタムウィンドウでは、メモリ値を再取得し表示を更新します。

なお、ターゲットプログラムの実行中は、定期的に nEventID == EVENT_TIME_10MS で **OnEvent()**ハンドル関数が呼び出されます。ターゲットプログラム実行中にのみ定期的に行う必要のある処理（サンプリングによる処理など）は、nEventID == EVENT_TIME_10MS の場合に分岐する場所に記述します。

### [ウィンドウの描画処理について]

これまでに説明した、OnDraw()ハンドル関数と OnEvent()ハンドル関数を用いて、ウィンドウの描画処理について説明します。

通常よく行われる **OnDraw()**ハンドル関数の使い方には、以下の 2 種類あります。

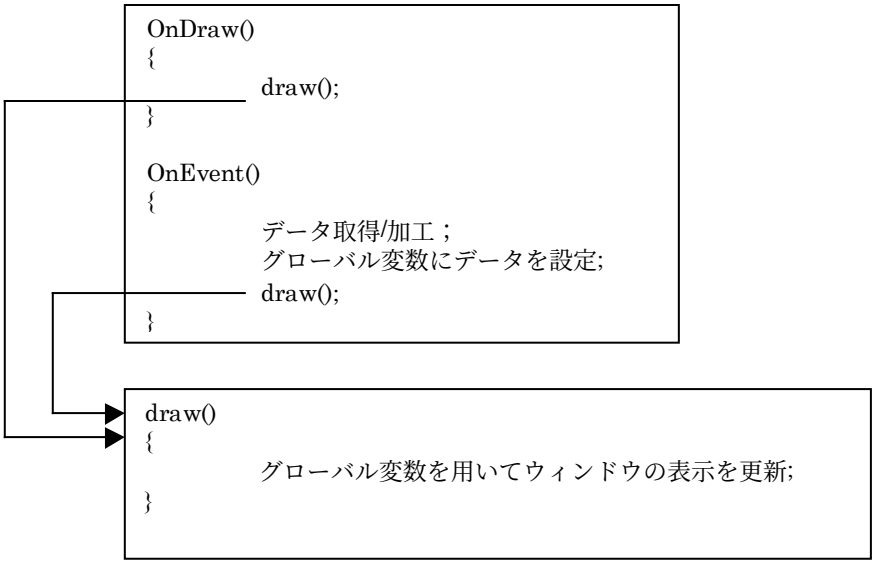
1. 関数内では描画処理を行わずに、全ての描画処理を行う他の関数を呼び出す。
   （**OnDraw()**ハンドル関数は、再描画要求があったことを受け取るだけの関数として機能します。全ての描画処理は、ほかの関数を呼び出すことによって実行します。）
2. 関数内で描画処理を行う。(下請け関数を使用することもあります。)
   （描画処理を行うには、必ず **OnDraw()**関数が実行される必要があります。）

これらの方法の違いと、その使い方を以下で説明します。

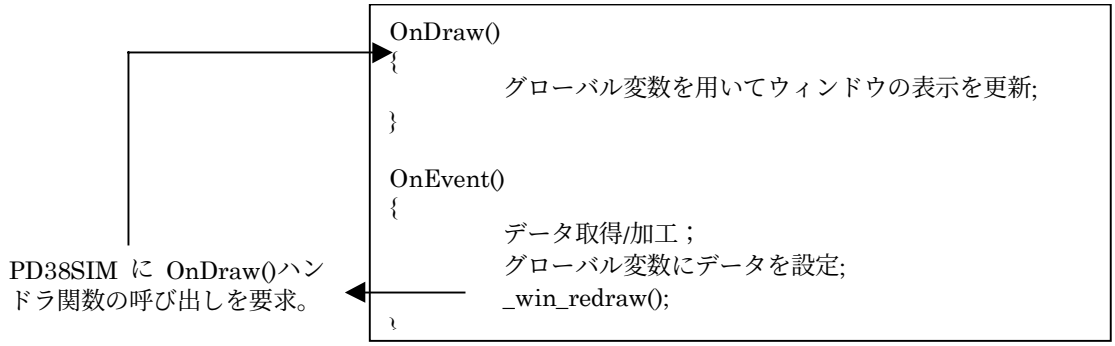
描画処理は場合により、**OnDraw()**関数が呼び出されるタイミング以外でも行う必要があります。例えば、特定のメモリ内容を継続して表示するようなウィンドウ（PD30のメモリウィンドウのようなもの）を考えてみましょう。このようなウィンドウでは、ターゲットメモリ値が変更された場合には、新しいメモリ値でウィンドウの表示を更新する必要があります。

そのような場合には、**OnEvent()**ハンドル関数で、ターゲットメモリ値を取得してグローバル変数に格納し、描画関数でウィンドウに描画します。

ウィンドウへの描画で、1.の方法を用いた場合には、**OnEvent()**ハンドル関数内でメモリ値を取得した後に、描画処理を行う関数を直接呼び出します。このように、1.の方法は、描画が必要な時に、描画に必要なデータを取得しておいて描画処理を行う関数を直接呼び出す方法です。以下に、概念図を示します。




ウィンドウへの描画で、2.の方法を用いた場合には、**OnEvent()**ハンドル関数内でメモリ値を取得した後に、_win_redraw()関数、または_win_redraw_clear()関数を呼び出し、PD30に対して **OnDraw()**ハンドル関数の呼び出しを要求します。このように、2.の方法は、描画が必要なときに、描画処理を行うデータを取得しておいて間接的に **OnDraw()**関数を呼び出すように要求する方法です。以下に、概念図を示します。




OnDraw()ハンドル関数は、他のウィンドウが前を横切った際などにも頻繁に呼び出されます。そのため、OnDraw()ハンドル関数は描画に専念し、データ取得/加工など時間がかかる処理は必要に応じてほかのタイミングで行うようにします。

### 3.5.5.ウィンドウサイズ変更ハンドル関数の使い方（OnSize）

カスタムウィンドウのサイズが変更された場合に、**OnSize()**ハンドル関数が呼び出されます。**OnSize()**ハンドル関数では、カスタムウィンドウの大きさの変更に対応した処理を行います。

サイズ変更の種別（最大化、アイコン化など）、クライアント領域の新しい幅、高さは、それぞれローカル変数 nType、cx、cy に渡されます。サイズ変更の種別の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.18 OnSize ハンドル関数」に記載されています。

自動生成される OnSize ハンドル関数を示します。(付属情報は nType, cx, cy です。)

```
OnSize()
{
        int     nType;
        int     cx;
        int     cy;

        nType = ((int *)_HandleMsgBlock)[0];
        cx = ((int *)_HandleMsgBlock)[1];
        cy = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

ウィンドウのサイズ情報を使用して描画等を変更するようなカスタムウィンドウプログラムでは、**OnSize()**ハンドル関数で取得したウィンドウのサイズ（cx、cy）をグローバル変数に保持しておきます。

**OnSize()**ハンドル関数は、ウィンドウ生成時に **OnCreate()**ハンドル関数に引き続き呼び出されますので、ウィンドウ生成時の大きさも取得できます。

### 3.5.6.コントロールアイテム(ボタン)操作ハンドル関数の使い方（OnCommand）

生成したコントロールアイテム(ボタン)が操作された場合に、**OnCommand()**ハンドル関数が呼び出されます。**OnCommand()**ハンドル関数では、操作されたコントロールアイテムに対応した処理を行います。

コントロールアイテムのコマンド ID、コントロールアイテムの通知コード、コントロールアイテムのハンドルは、それぞれローカル変数 nId、nMsg、nHandle に渡されます。**OnCommand()**ハンドル関数の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.3 OnCommand ハンドル関数」に記載されています。

```
自動生成される OnCommand ハンドル関数を示します。(付属情報は nId, nMsg, nHandle です。)
OnCommand()
{
        int     nId;
        int     nMsg;
        int     nHandle;

        nId = ((int *)_HandleMsgBlock)[0];
        nMsg = ((int *)_HandleMsgBlock)[1];
        nHandle = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

CB30 V.1.00でサポートされるコントロールアイテムはボタンのみです。

ボタンでは、nMsg は使用されません。nMsg は将来バージョンで使用する予定のシステム予約です。

### 3.5.7.スクロールバー操作ハンドル関数の使い方（OnHScroll など）

水平スクロールバーが操作された場合に、**OnHScroll()**ハンドル関数が呼び出されます。

また、垂直スクロールバーが操作された場合に、**OnVScroll()**ハンドル関数が呼び出されます。これらの関数では、スクロールバーに対する操作に対応した処理を行います。

スクロールバーに対する操作コード（ドラッグ、ページスクロールなど）、スクロールサム(スライダ)の位置は、それぞれローカル変数 nSBCode、nPos に渡されます。操作コードおよびスクロールサムの位置の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.8 OnHScroll ハンドル関数」などに記載されています。

```
自動生成される OnHScroll ハンドル関数を示します。(付属情報は nSBCode, nPos です。)
OnHScroll()
{
        int     nSBCode;
        int     nPos;

        nSBCode = ((int *)_HandleMsgBlock)[0];
        nPos = ((int *)_HandleMsgBlock)[1];

        /* Write message handler code here, please. */

}
```

nPos は、nSBCode が SB_THUMBPOSITION または SB_THUMBTRACK の時にのみ使用します。

スクロール操作が終了した際には、nSBCode == SB_ENDSCROLL でこれらの関数が呼び出され、スクロール操作の終了が通知されます。

### 3.5.8.マウス操作ハンドル関数の使い方（OnLButtonDblClk など）

マウスが操作された場合に、以下に示すハンドル関数が呼び出されます。

| ハンドル関数 | 呼び出される場合 |
|---|---|
| OnLButtonDblClk | マウスの左ボタンがダブルクリックされた際に呼び出されます。 |
| OnLButtonDown | マウスの左ボタンが押された際に呼び出されます。 |
| OnLButtonUp | マウスの左ボタンが離された際に呼び出されます。 |
| OnMouseMove | マウスカーソルが移動した際に呼び出されます。 |
| OnRButtonDblClk | マウスの右ボタンがダブルクリックされた際に呼び出されます。 |
| OnRButtonDown | マウスの右ボタンが押された際に呼び出されます。 |
| OnRButtonUp | マウスの右ボタンが離された際に呼び出されます。 |

これらの関数では、マウスに対する操作に対応した処理を行います。

マウスが操作された際に同時に押されているキーのコード、マウスカーソルの x 座標、マウスカーソルの y 座標は、それぞれローカル変数 nFlags、x、y に渡されます。キーコードの詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.11 OnLButtonDblClk ハンドル関数」などに記載されています。

```
自動生成される OnLButtonDblClk ハンドル関数を示します。(付属情報は nFlags, x, y です。)
OnLButtonDblClk()
{
        int     nFlags;
        int     x;
        int     y;

        nFlags = ((int *)_HandleMsgBlock)[0];
        x = ((int *)_HandleMsgBlock)[1];
        y = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

マウスボタンのダブルクリック時には、OnXButtonDown → OnXButtonUp → OnXButtonDblClk → OnXButtonUp の順でハンドル関数が呼び出されます。(X は左ボタンの場合に L、右ボタンの場合に R となります。)

### 3.5.9.キー操作ハンドル関数の使い方（OnChar など）

キーボードのキーが操作された場合に、以下に示すハンドル関数が呼び出されます。

| ハンドル関数 | 呼び出される場合 |
|---|---|
| OnChar | WM_KEYDOWN メッセージが文字コードに変換された際に呼び出されます。キーコードには、変換された ASCII コードが格納されます。 |
| OnKeyDown | システムキー以外のキーが押された際に呼び出されます。キーコードには、押されたキーの仮想キーコードが格納されます。 |
| OnKeyUp | システムキー以外のキーが離された際に呼び出されます。キーコードには、離されたキーの仮想キーコードが格納されます。 |

これらの関数では、キー操作に対応した処理を行います。

押されているキーのコード、リピートカウント値、スキャンコード値は、それぞれローカル変数 nChar、nRepCnt、nFlags に渡されます。キーコード、リピートカウント値、およびスキャンコードの詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.9 OnKeyDown ハンドル関数」などに記載されています。

```
自動生成される OnChar ハンドル関数を示します。(付属情報は nChar, nRepCnt, nFlags です。)
OnChar()
{
        int     nChar;
        int     nRepCnt;
        int     nFlags;

        nChar = ((int *)_HandleMsgBlock)[0];
        nRepCnt = ((int *)_HandleMsgBlock)[1];
        nFlags = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

文字コードに変換可能なキーが入力された場合には、OnKeyDown → OnChar → OnKeyUp の順でハンドル関数が呼び出されます。キーを押しつづけた場合には、OnKeyDown → (OnChar) →OnKeyDown → (OnChar) → ･･･ → OnKeyUp の順でハンドル関数が呼び出されます。（文字コードに変換可能なキーの場合にのみ OnChar ハンドル関数が呼び出されます。）

押されたキーが ASCII 文字に対応している場合は、ASCII コード値が nChar に格納されます。ファンクションキーなど、ASCII 文字に対応していないキーの場合には、nChar に対応する仮想キーコード値が格納されます。仮想キーコードについては、「CB30 V.1.00 ユーザーズマニュアル」の「5.4.9 OnKeyDown ハンドル関数」をご参照ください。

### 3.5.10.タイマハンドル関数の使い方（OnTimer）

Windows が持っているシステムタイマを使用した場合、設定された間隔で **OnTimer()** ハンドル関数が呼び出されます。**OnTimer()**ハンドル関数では、設定された間隔で実行する処理を記述します。

タイマの識別番号は、ローカル変数 nIDEvent に渡されます。タイマの識別番号は、Windows のシステムタイマを使用する際に、ユーザが自由に設定することができる番号です。

一つのハンドル関数で、複数のタイマに対応した処理を行う必要があるため、通常は関数の先頭で switch 文等でローカル変数ｎIDEvent を判別し、タイマに対応した処理に分岐させます。

```
自動生成される OnTimer ハンドル関数を示します。(付属情報は nIDEvent です。)
OnTimer()
{
        int     nIDEvent;

        nIDEvent = ((int *)_HandleMsgBlock)[0];

        /* Write message handler code here, please. */

}
```

なお、ターゲットプログラムの実行中は、定期的に nEventID == EVENT_TIME_10MS で **OnEvent()**ハンドル関数が呼び出されます。ターゲットプログラム実行中にのみ定期的に行う必要のある処理（サンプリングによる処理など）は、**OnEvent()**ハンドル関数で処理することをお勧めします。

**（注意事項）　システムタイマの総数には、OS による制限があります。必要以上にシステムタイマを使用すると、他のアプリケーションに影響を与えることがあります。**

## 3.6.カスタムウィンドウのプログラミングで使用できる関数

カスタムウィンドウのプログラミングで使用できる関数群は、大きく下記の 3 種類に分類されます。

1.標準関数
C 言語の標準関数の中でも比較的使用頻度が高いと思われる関数と同等のものをサポートしています。

2.デバッガ操作関数
デバッガを操作するための関数をサポートしています。

3.ウィンドウ操作関数
ウィンドウを操作するための関数をサポートしています。

## 3.7.ウィンドウ操作関数の使い方

ウィンドウ操作関数を使用する際は、ヘッダファイル winlib.h をインクルードしてください。

ウィンドウ操作関数仕様の詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.3. ウィンドウ操作系システムコール関数(winlib.lib)」をご参照ください。

### 3.7.1.描画関数の使い方

本節では、以下の関数を使用して描画関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _win_printf | 書式付きテキスト出力 |
| _win_set_cursor | カーソル位置の設定 |
| _win_set_color | テキストの色の設定 |
| _win_set_bkcolor | 背景色の設定 |
| _draw_frame_rect | 四角形の描画 |

関数名が_win で始まる関数は、カーソル座標(行、桁で指定する座標系)に対して描画します。カーソル座標の１カラムには、システムフォントの１文字が出力されます。

関数名が_draw で始まる関数は、ピクセル座標(ドット位置で指定する座標系)に対して描画します。

カスタムウィンドウに”Hello world.”と出力する例を以下に示します。

```
OnDraw()
{
    /* Write message handler code here, please. */
    _win_set_cursor(3, 1);              /* カーソル位置を(3, 1)に設定する */
    _win_printf("Hello World.");        /* 文字列出力 */
}
```

細字は、フレームワークソースファイルに自動生成されたコードを表します。
太字は、ユーザによって追加するコードを表します。
（以下の例でも、同様です。）

表示例

“Hello world.”を反転出力する例を以下に示します。

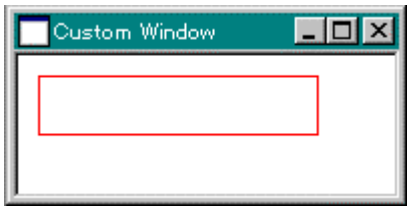
```
OnDraw()
{
    /* Write message handler code here, please. */
    int     old_color;              /* 変更前のテキスト色保存用変数 */
    int     old_bkcolor;            /* 変更前の背景色保存用変数 */

    _win_set_cursor(3, 1);          /* カーソル位置を(3, 1)に設定する */
                                    /* テキスト色を白に設定 */
    old_color = _win_set_color(COLOR_WHITE);
                                    /* 背景色を黒に設定 */
    old_bkcolor = _win_set_bkcolor(COLOR_BLACK);
    _win_printf("Hello World.");    /* 文字列出力 */
    _win_set_color(old_color);  /* テキスト色を元に戻す */
    _win_set_bkcolor(old_bkcolor);  /* 背景色を元に戻す */
}
```

表示例


赤い矩形領域を描画する例を以下に示します。

```
OnDraw()
{
    /* Write message handler code here, please. */
                                    /* 左上座標が(10, 10)、右下座標が */
                                    /* (150, 40)の赤い矩形を描画する */
    _draw_frame_rect(10, 10, 150, 40, COLOR_RED);
}
```

表示例



### 3.7.2.コントロールアイテム（ボタン）操作関数の使い方

カスタムウィンドウに貼り付けることのできるコントロールアイテムとして、ボタンをサポートしています。本節では、以下の関数を使用してコントロールアイテム操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _win_button_create | ボタンの作成 |
| _win_button_set_text | ボタンのテキストの変更 |

生成時に“button”をラベルに持ち、ボタンの入力のたびにラベルを大文字・小文字に変換するボタンの例を以下に示します。

```
#define     IDB_BUTTON      (1000)   /* ボタン ID 番号の定義 */
int         hButton;                 /* ボタンのハンドルを格納する変数 */
int         count;                   /* ボタンを押された回数を格納する変数 */

OnCommand()
{
    int     nId;
    int     nMsg;
    int     nHandle;

    nId = ((int *)_HandleMsgBlock)[0];
    nMsg = ((int *)_HandleMsgBlock)[1];
    nHandle = ((int *)_HandleMsgBlock)[2];

    /* Write message handler code here, please. */
    switch(nId){
    case IDB_BUTTON:                 /* ボタン ID が IDB_BUTTON の場合 */
            if(++count % 2){         /* ボタンを押された回数が奇数の場合 */
                                     /* ラベルを”BUTTON”に変更 */
                _win_button_set_text(hButton, "BUTTON");
            }else{                   /* ボタンを押された回数が偶数の場合 */
                                     /* ラベルを”button”に変更 */
                _win_button_set_text(hButton, "button");
            }
            break;
    }
}

OnCreate()
{
    /* Write message handler code here, please. */
    count = 0;                       /* ボタンが押された回数を 0 に初期化 */
                                     /* 左上座標が(10, 10)、右下座標が */
                                     /* (100, 40)の”button”をラベルに持つ */
                                     /* ID が IDB_BUTTON のボタンを作成し、 */
                                     /* ボタンのハンドルを hButton に保持する。 */
    hButton = _win_button_create(10, 10, 100, 40, "button", IDB_BUTTON);
}
```

表示例（生成時）


表示例（ボタン入力時）

### 3.7.3.ステータスバー操作関数の使い方

本節では、以下の関数を使用してステータスバー操作関数の使用例を示します。

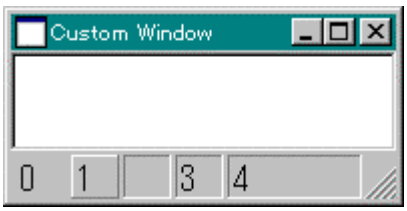
| 関数名 | 説明 |
|---|---|
| _win_statusbar_create | ステータスバーの作成 |
| _win_statusbar_set_pane | ステータスバーの項目の設定 |
| _win_statusbar_set_text | ステータスバーのテキストの設定 |

５つの項目を持つステータスバーの例を以下に示します。

```
OnCreate()
{
    /* Write message handler code here, please. */
    _win_statusbar_create(5);/* ５つの項目を持つステータスバーを生成する */
                             /* 0 番目の項目(一番左の項目)を SBPS_NOBORDERS スタ */
                             /* イルで大きさ 20 ピクセルに設定する */
    _win_statusbar_set_pane(0, SBPS_NOBORDERS, 20);
                             /* 0 番目の項目に"0"を描画する */
    _win_statusbar_set_text(0, "0");
                             /* 1 番目の項目を SBPS_POPOUT スタイルで大きさ 20 */
                             /* ピクセルに設定する */
    _win_statusbar_set_pane(1, SBPS_POPOUT, 20);
                             /* 1 番目の項目に"1"を描画する */
    _win_statusbar_set_text(1, "1");
                             /* 2 番目の項目を SBPS_DISABLED スタイルで大きさ */
                             /* 20 ピクセルに設定する */
    _win_statusbar_set_pane(2, SBPS_DISABLED, 20);
                             /* 3 番目の項目を SBPS_NORMAL スタイルで大きさ 20 */
                             /* ピクセルに設定する */
    _win_statusbar_set_pane(3, SBPS_NORMAL, 20);
                             /* 3 番目の項目に"3"を描画する */
    _win_statusbar_set_text(3, "3");
                             /* 4 番目の項目を SBPS_STRETCH | SBPS_NORMAL */
                             /* スタイルに設定する */
                             /* SBPS_STRETCH スタイルを設定しているので、ウィン */
                             /* ドウの拡大/縮小に応じて 4 番目の項目が伸縮する */
    _win_statusbar_set_pane(4, SBPS_STRETCH | SBPS_NORMAL, 0);
                             /* 4 番目の項目に"4"を描画する */
    _win_statusbar_set_text(4, "4");
}
```

_win_statusbar_set_pane()関数の第二引数には、ステータスバーの項目のスタイルを指定します。スタイルの詳細は、「CB30 V.1.00 ユーザーズマニュアル」の「5.3.32 _win_statusbar_set_pane: ステータスバーの項目の設定」に記載されています。

表示例



### 3.7.4.スクロールバー操作関数の使い方

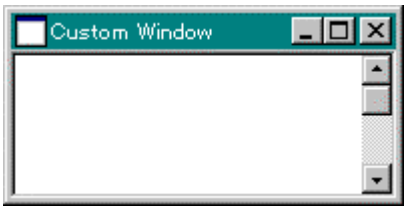
本節では、以下の関数を使用してスクロールバー操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _win_vscroll_range | 垂直スクロールバーのスクロール範囲の設定 |
| _win_vscroll_pos | 垂直スクロールボックスの位置の設定 |

垂直スクロールバーを表示させる例を以下に示します。

```
OnCreate()
{
    /* Write message handler code here, please. */
    _win_vscroll_range(0, 100);/* スクロール範囲を 0～100 に設定 */
}
```

表示例



スクロールバーの操作に対応した処理を記述した OnVScroll()ハンドル関数の例を以下に示します。

```
int  VScrollPageSize;                    /* 1 ページの行数が格納されている */
int  VScrollPos;                         /* サムの位置を格納する */

OnVScroll()
{
    int      nSBCode;
    int      nPos;

    nSBCode = ((int *)_HandleMsgBlock)[0];
    nPos = ((int *)_HandleMsgBlock)[1];

    /* Write message handler code here, please. */
    switch(nSBCode){
    case SB_BOTTOM:                      /* 一番下までスクロール */
            VScrollPos = 100;
            break;
    case SB_ENDSCROLL:           /* スクロール終了 */
            break;
    case SB_LINEDOWN:                    /* １行下へスクロール */
            VScrollPos++;
            break;
    case SB_LINEUP:                      /* １行上へスクロール */
            VScrollPos--;
            break;
    case SB_PAGEDOWN:                    /* １ページ下へスクロール */
            VScrollPos += VScrollPageSize;
            break;
    case SB_PAGEUP:                      /* １ページ上へスクロール */
            VScrollPos -= VScrollPageSize;
            break;
    case SB_THUMBPOSITION:               /* 絶対位置へスクロール */
                                         /* (現在位置は nPos で指定される) */
    case SB_THUMBTRACK:                  /* スクロールボックスを指定位置へドラッ */
                                         /* グする(現在位置は nPos で指定される) */
            VScrollPos = nPos;
            break;
    case SB_TOP:                         /* 一番上までスクロール */
    default:
            VScrollPos = 0;
    }
    if(VScrollPos < 0)           /* スクロール範囲外の場合の処理 */
            VScrollPos = 0;
    if(VScrollPos > 100)
            VScrollPos = 100;
    _win_vscroll_pos(VScrollPos);        /* スクロールサム位置の設定 */
    _win_redraw_clear();                 /* カスタムウィンドウの再描画 */
}
```

### 3.7.5.ダイアログ操作関数の使い方

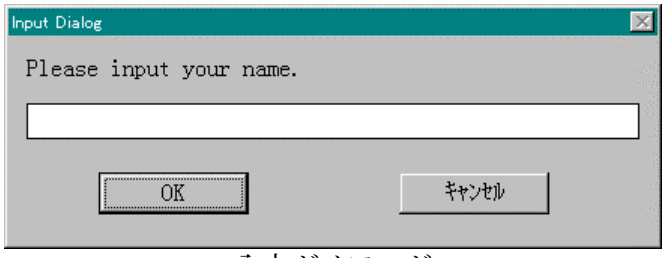
本節では、以下の関数を使用してダイアログ操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _win_dialog | 入力ダイアログの作成 |
| _win_message_box | メッセージボックスの作成 |

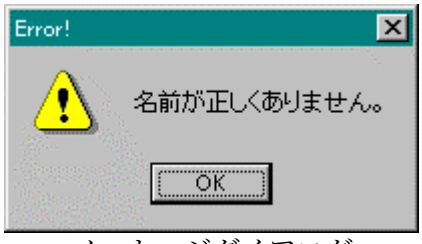
入力ダイアログを用いて値を取得する例を以下に示します。この関数は、入力ダイアログをオープンし、ユーザに入力を求め、”eisuke”が入力された場合に TRUE を返し、それ以外の場合には、メッセージボックスにエラーを表示し、FALSE を返します。

```
int check_name()
{
    char    name[100];

                                /* 入力ダイアログをオープンし、入力を求める */
    if(_win_dialog("Please input your name.", name) == TRUE){
                                                    /* OK ボタンが入力された */
        if(strcmp(name, "eisuke") == 0){
                                                    /* 文字列が”eisuke”である */
            return TRUE;
        }
    }
                                /* メッセージダイアログにエラーを表示する */
    _win_message_box("名前が正しくありません。", "Error!",
                                        MB_ICONEXCLAMATION | MB_OK);
    return FALSE;
}
```

表示例




入力ダイアログ




メッセージダイアログ

### 3.7.6.ウィンドウフレーム操作関数の使い方

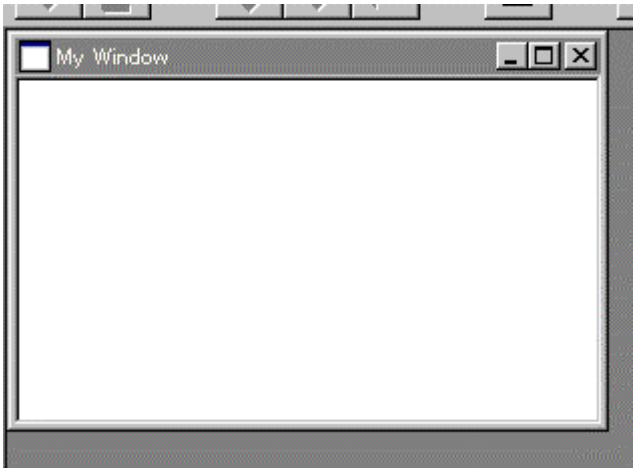
本節では、以下の関数を使用してウィンドウフレーム操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _win_redraw | カスタムウィンドウの再描画 |
| _win_redraw_clear | カスタムウィンドウの再描画(領域のクリア付き) |
| _win_set_window_title | カスタムウィンドウのタイトルの設定 |
| _win_set_window_pos | カスタムウィンドウの位置の設定 |
| _win_set_window_size | カスタムウィンドウのサイズの設定 |

カスタムウィンドウのタイトルを”My Window”とし、オープン時の大きさを(300, 200)ピクセルとし、オープン時の位置を(0, 0)とする例を以下に示します。位置は、PD30ウィンドウのクライアント領域の左上を(0, 0)とし、右方向に x、下方向に y となります。

```
OnCreate()
{
    /* Write message handler code here, please. */
    _win_set_window_title("My Window");          /* ウィンドウタイトルの設定 */
    _win_set_window_size(300, 200);              /* ウィンドウサイズの設定 */
    _win_set_window_pos(0, 0);                   /* ウィンドウ位置の設定 */
}
```

表示例



### 3.7.7.システムタイマ操作関数の使い方

本節では、以下の関数を使用してシステムタイマ操作関数の使用例を示します。

| 関数名 | 説明 |
|---|---|
| _win_timer_set | システムタイマの設定 |
| _win_timer_kill | システムタイマの解除 |

システムタイマを使用し、200 ミリ秒毎にカウンタをインクリメントする例を以下に示します。

```
#define     IDT_1    (10)                    /* タイマ ID 番号の定義 */
int         count;                           /* カウンタ */

OnCreate()
{
    /* Write message handler code here, please. */
    _win_timer_set(IDT_1, 200);          /* システムタイマの設定 */
    count = 0;                           /* カウンタを 0 に初期化 */
}

OnDestroy()
{
    /* Write message handler code here, please. */
    _win_timer_kill(IDT_1);              /* システムタイマの解除 */
}

OnTimer()
{
    int      nIDEvent;

    nIDEvent = ((int *)_HandleMsgBlock)[0];

    /* Write message handler code here, please. */
    if(nIDEvent == IDT_1){               /* システムタイマ IDT_1 の場合 */
        count++;                         /* カウンタをインクリメント */
    }
}
```

なお、ターゲットプログラムの実行中は、定期的に nEventID == EVENT_TIME_10MS で **OnEvent()**ハンドル関数が呼び出されます。ターゲットプログラム実行中にのみ定期的に行う必要のある処理（サンプリングによる処理など）は、**OnEvent()**ハンドル関数で処理することをお勧めします。

**（注意事項）　システムタイマの総数には、OS による制限があります。必要以上にシステムタイマを使用すると、他のアプリケーションに影響を与えることがあります。**

# MEMO

# 技術サポート連絡書

年　月　日（合計　　枚）

三菱電機セミコンダクタシステム株式会社
マイコンソフトツール部
# 開発ツールサポート窓口行

［電子メール］ support@tool.mesc.co.jp
［大阪地区］ FAX： 06-6398-6191
［東京地区］ FAX： 03-5783-7339
［中部地区］ FAX： 052-221-7318
［九州地区］ FAX： 092-452-1427

※インストーラが生成する以下のテキストファイルもサポート連絡書としてご利用できます。
Windows 98/95/Windows NT 4.0版の場合：\SUPPORT\製品名\SUPPORT.TXT
EWS版の場合：/support/製品名/toolinfo.txt

| ご連絡先 | 製品情報 |
|---|---|
| 会社名： | ソフトウェア： |
| 部署名： | バージョン番号：V. |
| 担当者名： | ライセンスID： |
| 電話番号： | – – – – |
| FAX番号： | ホストマシン： |
| 電子メール： | OS： V. |
| 通信欄： | |

# MEMO

**CB30 V.1.00プログラミングマニュアル**
**第1版：1998年2月16日発行**
**資料番号：MSD-CB30-UP-980216**

# エミュレータデバッガ　PD30 用　カスタムビルダ
# CB30 V.1.00　プログラミングマニュアル

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