5.　Windows アプリケーション(2)　コントロールの実装

5.1.　コントロールの種類と役割

Whidows アプリケーションを使いやすくするための機能としてボタンや入力ボックスがあります。このような様々なアプリケーションで共通に使うことのできる Windows が提供する機能をコントロールと呼びます。Windows が提供するコントロールは Windows のバージョンによって若干異なります。表 5.1 に基本的なコントロールを，図 5.1 にそれらの概観を示します。

表 5.1　基本的なコントロールの種類

クラス名 (コントロール)	機能
BUTTON	ボタン，チェックボックス，ラジオボタン，グループボックス
COMBOBOX	コンボボックス
EDIT	エディットボックス
LISTBOX	リストボックス
SCROLLBAR	スクロールバー
STATIC	文字列，ビットマップ画像

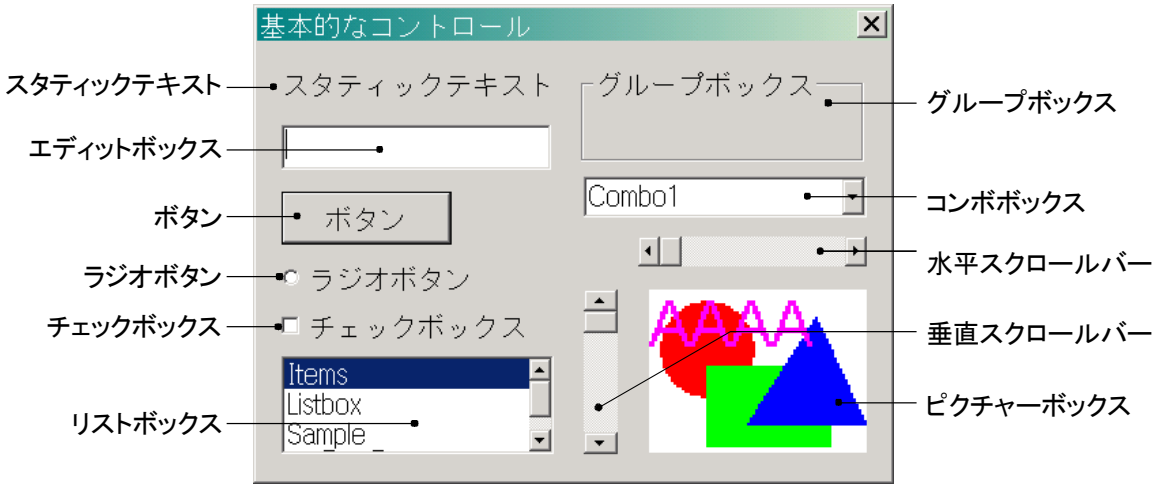

図 5.1　基本的なコントロール

このようなコントロールは基本的にアプリケーションウィンドウやダイアログボックスウィンドウ内の小さなウィンドウ(チャイルドウィンドウ)として動作します。すなわち，これらを作るときはウィンドウクラスを定義(RegisterClass)して，ウィンドウを作成(CreateWindow)するという手順が必要となります。ただし，このように頻繁に使用するクラス全てに対してクラス定義をするというのはとても不便です。そこで，Visual C++ではこのような基本コントロールのクラスをあらかじめ定義してあります(表 5.1 の1列目)。この定義済みクラス名を CreateWindow の引数とすることで簡単にこれらのコントロールを使用できます。また，ダイアログボックスではリソースエディタが使えますので，あらためて CreateWindow を呼ばなくてもリソースエディタでコントロールを貼り付けることができます。それでは，扱いの簡単なダイアログボックスから各コントロールの使い方をマスターしてみましょう。

5.1.1.　スタティックテキスト，ボタン，エディットボックス

図 5.2 のようなダイアログを使ってスタティックテキスト，ボタン，エディットボックスの使い方を見てみましょう。上の白い四角がエディットボックス(IDC_EDIT1)，Static と書いてあるところがスタティックテキスト(IDC_TEXT)，それと下に2つボタンがあります。今回のプログラムではエディットボックスに入力した文字列を，「表示の更新」ボタンを押すと下のスタティックテキストに表示し，「閉じる」を押すとダイアログボックスを閉じる，ということにします。ス

図 5.2　スタティックテキスト，ボタン，エディットボックス

タティックテキストとは単なる文字表示で，ダイアログ内の文字の表示にも使用しています。ID は自動的に IDC_STATIC がつきます。今回はこのテキストにも操作を加えますので，ID を IDC_TEXT に変更します。また，下の2つのボタンはそれぞれ「OK」と「キャンセル」の表示を変更したもので，それぞれの ID は IDOK と IDCANCEL のままにしておきます。

ソースは前回のバージョンの表示に使用したものが利用できますので，基本的な部分は省略して，重要なダイアログプロシージャだけ示します。

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp, LPARAM lp){
    char              lpszMessage[128] = "";

    switch(uMes){
    case WM_INITDIALOG:
        SetDlgItemText(hDlg, IDC_TEXT, lpszMessage);
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            GetDlgItemText(hDlg, IDC_EDIT1, lpszMessage, 128);
            SetDlgItemText(hDlg, IDC_TEXT, lpszMessage);
            break;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return TRUE;
        }
        break;
    }
    return FALSE;
}
```

list5.1　スタティックテキスト，ボタン，エディットボックス

まず，エディットボックスに入力される文字列のための配列 lpszMessage を 128 字分確保しておき，空の文字列で初期化しておきます。次に，ダイアログが表示されたときに WM_INITDIALOG を受け取るので，スタティックテキストに lpszMessage の内容(最初は空ですが)を表示しておきます。表示には SetDlgItemText 関数を使います。

```
BOOL SetDlgItemText(
    HWND hDlg,        // ダイアログボックスハンドル
    int nIDDlgItem,   // コントロールの ID
    LPCTSTR lpString  // 設定する文字列へのポインタ
);
```

「表示の更新」ボタン(IDOK)ボタンが押されるとメッセージ WM_COMMAND の下位ワードに IDOK が来るので，エディットボックスの文字列を lpszMessage に格納します。エディットボックスの文字列を受け取るには GetDlgItemText を使います。

```
UINT GetDlgItemText(
    HWND hDlg,        // ダイアログボックスハンドル
    int nIDDlgItem,   // コントロールの ID
    LPTSTR lpString,  // 文字列を受け取るバッファへのポインタ
    int nMaxCount     // 文字列の最大文字数
);
```

受け取った文字列 lpszMessage はまた SetDlgItemText を用いてスタティックテキストに表示することができます。

【練習問題　5.1】 SetDlgItemInt，GetDlgItemInt

SetDlgItemText，GetDlgItemText と同様に整数を扱う SetDlgItemInt，GetDlgItemInt があるので，これを使って整数を入力して，整数を表示するダイアログ関数を作成せよ。また，同様に小数の場合はどうすと良いか考えよ。

5.1.2.　ラジオボタン，グループボックス

図 5.3 のようなダイアログを使ってラジオボタンとグループボックスの使い方を見てみましょう。4つの丸いボタンがラジオボタン，性別，学年の枠組みがグループボックスです。ラジオボタンはいくつかのグループ分けがあって，そのグループ内では1つしか選択できないボタンのことです。グループボックスはこのようなときにグループ分けを見やすくするための枠組みで，機能として

図 5.3　ラジオボタン，グループボックス

図 5.4　タブオーダー

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp, LPARAM lp){
    UINT                uSex = BST_CHECKED;
    UINT                uGrade = BST_CHECKED;

    switch(uMes){
    case WM_INITDIALOG:
        CheckDlgButton(hDlg, IDC_RADIO_MALE, uSex);
        CheckDlgButton(hDlg, IDC_RADIO_M1, uGrade);
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            uSex = IsDlgButtonChecked(hDlg, IDC_RADIO_MALE);
            uGrade = IsDlgButtonChecked(hDlg, IDC_RADIO_M1);
            switch(uSex * 2 + uGrade){
            case 3:
                MessageBox(hDlg, "男とM1が押されました. ", "確認", MB_OK);
                break;
            case 2:
                MessageBox(hDlg, "男とM2が押されました. ", "確認", MB_OK);
                break;
            case 1:
                MessageBox(hDlg, "女とM1が押されました. ", "確認", MB_OK);
                break;
            case 0:
                MessageBox(hDlg, "女とM2が押されました. ", "確認", MB_OK);
                break;
            }
            EndDialog(hDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return FALSE;
        }
        break;
    }
    return FALSE;
}
```

list5.2　ラジオボタン，グループボックス

は何もありません。さて，それではリソースエディタでダイアログを作成してみましょう。それぞれのラジオボタンにはそれぞれ，IDC_RADIO_MALE，IDC_RADIO_FEMAIL，IDC_RADIO_M1，IDC_RADIO_M2 と割り当てることにします。次にグループ分けの設定をします。ラジオボタンのグループ分けをするためにはまずグループの先頭を決める必要があります。今回は性別と学年の2つのグループがあり，それぞれ「男」と「女」，「M1」と「M2」がありますので，「男」と「M1」をグループの先頭とします。先頭を決めたらそれぞれのコントロールのプロパティで「グループ」のチェックボックスをチェックしておきます。次にタブオーダーの設定をします。タブオーダーとはダイアログ内でタブキーを押したときにアクティブになるコントロールの順番です。リソースエディタで「レイアウト」ー「タブオーダー」を選択すると，図 5.4 のようにそれぞれのコントロールに数字が付けられます。この数字がタブオーダーになります。ラジオボタンのグループ分けは「グループ」のチェックボックスがチェックされたラジオボタンからタブオーダーが連続するラジオボタンがグループであると判断します。今回は図のようにタブオーダーを指定してください。タブオーダーの指定は順番にクリックしていくと変化します。

さて，それではプログラムに移りましょう。それぞれのラジオボタンが押されているかを調べるのは IsDlgButtonChecked 関数です。

```
UINT IsDlgButtonChecked(
    HWND hDlg,        // ダイアログボックスハンドル
    int nIDButton     // ボタンの ID
);
```

ボタンが押されていれば BST_CHECKED，押されていなければ BST_UNCHECKED が戻値として返ってきます。それぞれ1，0と定義されています。また，それぞれのラジオボタンのチェック状態を変更するには CheckDlgButton 関数を使います。

```
BOOL CheckDlgButton(
    HWND hDlg,        // ダイアログハンドル
    int nIDButton,    // ボタンの ID
    UINT uCheck       // ボタンの状態
);
```

uCheck に BST_CHECKED や BST_UNCHECKED を設定することで，ボタンの状態を変更できます。

それでは，「OK」を押すとチェックボックスをチェックしてメッセージボックスを表示するプログラムを作成してみましょう。今回もダイアログプロシージャのみ list5.2 に示しておきました。

まず，性別と学年のための変数 uSex と uGrade を定義し，それぞれを BST_CHECKED に設定しておきます。WM_INITDIALOG を受け取ったら，IDC_RADIO_MALE と IDC_RADIO_M1 のチェックボタンを uSex，uGrade の値にセットしておきます。IDOK が押されたら，IDC_RADIO_MALE と IDC_RADIO_M1 のボタンの状態を取得します。それぞれのグループで1回しかチェックしないのはラジオボタンがそれぞれのグループで2つしかないので，一方がチェックされていたらもう一方はチェックされていないからです。3つ以上ある場合は少し複雑になります。チェックが終わったらそれらの状態によって表示を変えて MessageBox を表示します。今回は BST_CHECKED が1であることを利用して，uSex×2＋uGrade を計算して switch 文で分岐しています。ラジオボタンの数が増えてきたらその判断方法を工夫する必要が出てきます。これはそのうちの一つのテクニックです。

5.1.3.　チェックボックス

図 5.4 のようなダイアログを使ってチェックボックスの使い方を見てみましょう。チェックボックスの使い方はグループ分けのないラジオボタンと考えることができます。使用する関数も IsDlgButtonChecked 関数，CheckDlgButton 関数と全くおなじです。

図 5.4　チェックボックス

【練習問題 5.2】 チェックボックス

図 5.4 に示したダイアログボックスを表示し，OK を押したときにチェックされているチェックボックスの値の合計をメッセージボックスで表示するプログラムを作成せよ。

図 5.5　リストボックス

5.1.4.　リストボックス

図 5.5 のようなダイアログボックスを作ってリストボックスの使い方を見てみましょう。上の四角い部分がリストボックス(IDC_LIST1)です。リストボックスはこのように選択できるリストを表示してその中から項目を選択するのに使用します。

それでは，「OK」を押したら選択された番号を表示してダイアログを閉じるプログラムを作成しましょう。これまでと同じようにテンプレートを基に作成します。今回は sprintf を使いますので，stdio.h も include しておいてください。それでは list5.3 にダイアログプロシージャを示します。今回は SendMessage 関数を使用しています。

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp, LPARAM lp){
    int                 i;
    int                 iListNo = 0;
    LPCTSTR             lpszList[] = {"LIST0", "LIST1", "LIST2", "LIST3"};
    char                lpszMessage[64];

    switch(uMes){
    case WM_INITDIALOG:
        for(i = 0; i <= 4; i++ )
            SendMessage(GetDlgItem(hDlg, IDC_LIST1), LB_INSERTSTRING,
                (WPARAM)i, (LPARAM)lpszList[i]);
        SendMessage(GetDlgItem(hDlg, IDC_LIST1), LB_SETCURSEL,
            (WPARAM)iListNo, 0L);
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            iListNo = (int)(DWORD)SendMessage(GetDlgItem(hDlg, IDC_LIST1),
                LB_GETCURSEL, 0L, 0L);
            sprintf(lpszMessage, "%d番が選択されました。", iListNo);
            MessageBox(hDlg, lpszMessage, "確認", MB_OK);
            EndDialog(hDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return FALSE;
        }
        break;
    }
    return FALSE;
}
```

list5.3　リストボックス

LRESULT SendMessage(
　　HWND hWnd,
　　// ウィンドウハンドル
　　UINT Msg,
　　// 送るメッセージ
　　WPARAM wParam,
　　// メッセージパラメータ

```
    LPARAM lParam
    // メッセージパラメータ
);
```

hWnd にはそのメッセージを送りたいウィンドウのハンドルを指定します。Msg には送るメッセージを指定します。wParam と lParam にはそれぞれメッセージパラメータを指定します。この SendMessage 関数の引数はちょうどウィンドウプロシージャやダイアログプロシージャの引数と同じようになっています。つまり，この関数を使うことでそれぞれのウィンドウハンドルに割り当てられたプロシージャに自由にメッセージを送ることができるということです。ですから，この関数は様々なところで使用されます。今回はこの関数を使ってリストボックスコントロールのウィンドウハンドル（GetDlgItem 関数で取得）に割り当てられたプロシージャ（実際には見えない）にメッセージを送ることでリストボックスの操作を行います。

まず，リストの番号を格納する整数 iListNo とリストを格納する文字列配列 lpszList を定義します。WM_INITDIALOG を捕まえたらリストボックスに文字列を挿入していきます。

SendMessage(リストボックスウィンドウハンドル, LB_INSERTSTRING, (WPARAM)何番目か, (LPARAM)文字列);

次に初期状態でリストボックスのどの文字列が選択された状態にするかを指定します。

SendMessage(リストボックスウィンドウハンドル, LB_SETCURSEL, (WPARAM)何番目か, 0L);

「OK」ボタンが押されると（IDOK），

iListNo = (int)(DWORD)SendMessage(リストボックスウィンドウハンドル, LB_GETCURSEL, 0L, 0L);

で，上から何番目の文字列が選択されたかを取得しています。あとは，sprintf で文字列に格納して MessageBox で表示しています。

5.1.5.　コンボボックス

コンボボックスはリストボックスの表示を1行にまとめたような形をしています。機能もほとんどリストボックスと同じです。また，使用方法，使用する関数も同じです。

【練習問題 5.3】 コンボボックス

リストボックスで作成したダイアログボックスをコンボボックスを使って作成しなさい。

5.2.　エディットコントロール

今までのプログラムはメインウィンドウを使いませんでした。これではメインウィンドウの意味がほとんどありません。そこで，今度はメインウィンドウにコントロールを貼り付けてみます（貼り付けたものを子ウィンドウと言います）。CreateWindow 関数を使うことになりますので，その使い方を確認しておきましょう。

```
HWND CreateWindow(
    LPCTSTR lpClassName,      // クラス名へのポインタ
    LPCTSTR lpWindowName,     // ウィンドウ名へのポインタ
    DWORD dwStyle,            // ウィンドウのスタイル
    int x,                    // ウィンドウの x 座標（水平方向）
    int y,                    // ウィンドウの y 座標（垂直方向）
    int nWidth,               // ウィンドウの幅
    int nHeight,              // ウィンドウの高さ
    HWND hWndParent,          // 親ウィンドウのウィンドウハンドル
    HMENU hMenu,              // メニューハンドル，子ウィンドウの場合はウィンドウ ID
    HANDLE hInstance,         // インスタンスハンドル
    LPVOID lpParam            // WM_CREATE の lParam
);
```

lpClassName:　親ウィンドウを作るときはRegisterClass関数で登録したクラス名でしたが，コントロールを作るときは表 5.1 に示した定義済みコントロール名を指定します。

lpWindowName:　タイトルバーに表示する名前です。子ウィンドウの場合は指定しないことが多いです。

dwStyle:　ウィンドウのスタイルを指定します。コントロールによって指定できる値が異なります。複数のスタイルを指定する場合は，「|」で連結します。Help を参照してください。

x, y, nWidth, nHeight:　親ウィンドウの時はスクリーン座標系ですが，子ウィンドウのときはクライアント座標系（タイトルバーもしくはメニューの左下を原点とした座標系）になります。

hWndParent:　親ウィンドウのときは NULL でしたが，子ウィンドウのときは親ウィンドウハンドルを指定します。

hMenu:　親ウィンドウの時はメニューハンドルを指定しましたが，子ウィンドウの時はそれぞれのウィンドウに固有の ID を指定します。この ID は親ウィンドウイベントを通知するときの識別に使用します。

hInstance:　親ウィンドウでも子ウィンドウでもインスタンスハンドルを指定します。

lpParam:　ここで指定した値がそのウィンドウが作成されたときの WM_CREATE メッセージの lParam になります。

例えば，親ウィンドウ全面にエディットコントロールを貼り付けるためには，

① エディットウィンドウのウィンドウハンドルを宣言する。このとき static で宣言するのを忘れないように。

② クライアントウィンドウの幅と高さを GetClientRect 関数で調べる。

③ CreateWindow 関数でエディットウィンドウを作成する。

という手順で行います。CreateWindow で指定するクラス名は”EDIT”，ウィンドウスタイルは，WS_CHILD | WS_VISIBLE | WS_VSCROLL | ES_MULTILINE | ES_LEFT | ES_WANTRETURN | ES_AUTOVSCROLL あたりが良いでしょう。WS_･･･は共通ウィンドウスタイル，ES_･･･エディットコントロール用のスタイルでそれぞれ，チャイルドウィンドウ，可視，垂直スクロールあり，複数行，左揃えテキスト，改行の許可，自動垂直スクロールあり，という意味です。これをテンプレートのソースに書き加えると，list5.4 のようになります。先頭で IDC_EDIT の定義をしていますが，resource.h がある場合はその中に書いた方が良いでしょう。その際は，したの方にある resource.h の下の方にある _APS_NEXT_CONTROL_VALUE の定義を変えておくのを忘れないでください。

さて，このプログラムをビルドしてみましょう。いままで灰色だった親ウィンドウの色が白くなっていると思います。ここでカーソルをクリックするとメモ帳のように「I」型のカーソルに変わるはずです。何かキーボードから入力をしてみてください。まるでワープロのように文字入力ができるとおもいます。実際に Windows に標準で入っているメモ帳はこのエディットコントロールを使っています。このプログラムにファイルの保存と印刷の機能を追加すると誰でもメモ帳ができてしまうということです。

```
#include    <windows.h>

#define IDC_EDIT    1000

char    szWinName[] = "Template";   // ウィンドウクラスの名前
                        ⋮
    HINSTANCE           hInstance;
    RECT                Rect;
    static HWND         hEdit;

    hInstance = (HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE);
                        ⋮
    case WM_CREATE:                 // ウィンドウの作成
        GetClientRect(hWnd, &Rect);
        hEdit = CreateWindow("EDIT", "",
            WS_CHILD | WS_VISIBLE
            | ES_MULTILINE | ES_LEFT | ES_WANTRETURN
            | ES_AUTOVSCROLL | WS_VSCROLL,
            0, 0, Rect.right, Rect.bottom,
            hWnd, (HMENU)IDC_EDIT, hInstance, NULL);
        break;
                        ⋮
    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            DestroyWindow(hEdit);   // エディットコントロールの破棄
            DestroyWindow(hWnd);    // メインウィンドウの破棄. WM_DESTROY発行
        }
        break;
```

list5.4　エディットコントロールの貼り付け

5.3. タイマー

いろいろと計測を行うためにはタイマーが使えるととても便利です。ここでは， Windows で用意されている2つのタイマーの使い方を紹介します。現在の時刻を一定の周期でエディットコントロールに表示するアプリケーションを作ってみましょう。

5.3.1. SetTimer

API 関数の中に SetTimer という関数があります。まずはこの関数のプロトタイプを見てみます。

```
UINT SetTimer(
    HWND hWnd,              // ウィンドウハンドル
    UINT nIDEvent,          // タイマの ID
    UINT uElapse,           // 設定時間（単位は ms）
    TIMERPROC lpTimerFunc   // マイマープロシージャ
);
```

この関数を実行すると Windows は，設定時間ごとにタイマプロシージャを実行し，WM_TIMER メッセージをアプリケーションに送ります。WndProc で WM_TIMER メッセージを処理するときは最後の引数は NULL にします。使い終わったら KillTimer 関数でタイマを殺します。

```
BOOL KillTimer(
    HWND hWnd,              // タイマを設定したウィンドウハンドル
    UINT uIDEvent           // タイマの ID
);
```

あと必要なのは時間を取得する関数と，エディットコントロールへの表示方法ですね。時間を取得する関数は GetLocalTome 関数でした（4.2.2 参照）。エディットコントロールへの表示には SendMessage を使います。

SendMessage(エディットコントロールハンドル， EM_REPLACESEL，（WPARAM)FALSE,（LPARAM)文字列);

では，これらの関数を使ってプログラムを作ってみましょう。

まず，最初の方でこれから作るタイマーの ID を決めておきます。たくさん作る時は ID を複数作っておけばいいです。WindowFunc では表示する文字列のための lpszMessage と GetLocalTime 関数のための構造体 stSystemTime の定義をしています。あとは WM_CREATE を捕まえたら，SetTimer 関数でタイマーの設定を行います。周期は 1000ms つまり1秒に設定しました。また，今回は WM_CLOSE を捕まえたときにタイマーを殺しています。

タイマーは一定周期ごとに WM_TIMER を送ってきますので，それを捕まえたら GetLocalTime 関数で現在の時刻を取得して lpszMessage に格納します。あとは SendMessage 関数でエディットコントロールに表示するという具合です。複数のタイマーを設定した場合は，

```
#define IDC_EDIT    1000
#define ID_TIMER    0


LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMsg, WPARAM wp, LPARAM lp){
    HINSTANCE           hInstance;
    RECT                Rect;
    char                lpszMessage[128];
    SYSTEMTIME          stSystemTime;
    static HWND         hEdit;


    case WM_CREATE:                 // ウィンドウの作成


        SetTimer(hWnd, ID_TIMER, 1000, NULL);
        break;


    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            KillTimer(hWnd, ID_TIMER);
            DestroyWindow(hEdit);   // エディットコントロールの破棄
            DestroyWindow(hWnd);    // メインウィンドウの破棄. WM_DESTROY発行
        }
        break;
    case WM_TIMER:
        GetLocalTime(&stSystemTime);
        sprintf(lpszMessage, "%02lu:%02lu:%02lu.%03lu¥xd¥xa",
            stSystemTime.wHour,    stSystemTime.wMinute,
            stSystemTime.wSecond, stSystemTime.wMilliseconds);
        SendMessage(hEdit, EM_REPLACESEL, FALSE, (LPARAM)lpszMessage);
        break;
```

list5.5 SetTimer を使ったタイマー

WPARAM にタイマーの ID が格納されていますので，if 文か switch 文でそれぞれの処理を行うということになります。

さて，それでは実際に実行してみましょう。タイマーの周期を変えて実行してみてください。実行してみるといくつか問題点が出てくると思います。一つはあまり周期の精度が良くないということです。単純な時間表示だけでしたらそれほど問題にもなりませんが，これが計測用となると話は別です。もう一つは長時間実行するとエディットコントロールの表示が止まってしまうということです。実はエディットコントロールの最大サイズは 64kbyte（英数半角文字で 64×1024 文字）までという制限があります。これを回避するためには，①リッチエディットコントロールを使う，②制限文字数を越えたら表示をクリアする，の2通りの方法があります。①の方法については各自勉強していただくことにして，ここでは②の方法について説明します。ログの表示だけであればこれで十分です。

エディットコントロールのバッファがいっぱいになると，WM_COMMAND メッセージが親ウィンドウに通知されます。このとき WPARAM の上位ワードに EN_MAXTEXT が，下位ワードにエディットコントロールの ID が格納されます。また，LPARAM にはエディットコントロールのウィンドウハンドルが格納されます。そこで，WindowFunc でこのメッセージを監視し，メッセージが来たらエディットコントロールをクリアする処理を行えばいいことになります。つまり，list5.6 のようにしておけばいいことになります。ここで使っている SetWindowText 関数は，指定したウィンドウハンドルのタイトルバーを変更する関数ですが，ハンドルがコントロールの場合はコントロール内のテキストを変更する関数です。Help で確認してみてください。

```
case WM_COMMAND:
    if(HIWORD(wp) == EN_MAXTEXT)
        SetWindowText((HWND)lp, NULL);
    break;
```

list5.6　エディットコントロールのクリア

5.3.2.　マルチメディアタイマー

Windows には SetTimer 関数を用いたタイマーの他にもタイマーが用意されています。これはマルチメディアタイマーと呼ばれる機能で本来はビデオやサウンドなどのマルチメディアデバイスを操作するために用意されたものです。マルチメディア系では高速で正確な時間管理が必要となるので，これを利用することでかなり正確なタイマー管理が可能となります。まず，使用する関数から見ていきましょう。

```
MMRESULT timeBeginPeriod(
    UINT uPeriod        // 最小のタイマー解像度を ms 単位で指定します。
);
MMRESULT timeEndPeriod(
    UINT uPeriod        // timeBeginPeriod で指定した最小のタイマー解像度を ms 単位で指定します。
};
MMRESULT timeSetEvent(
    UINT uDelay,        // イベントを発生させる時間間隔を ms で指定します。
    UINT uResolution,   // タイマー解像度を指定します。timeBeginPeriod で指定した値以上にします。
    LPTIMECALLBACK lpTimeProc,  // タイマープロシージャを指定します。
    DWORD dwUser,       // タイマープロシージャに与えられるユーザ定義データです。
    UINT fuEvent,       // タイマーイベントのタイプを指定します。
};
MMRESULT timeKillEvent(
    UINT uTimerID       // timeSetEvent の戻値で得たタイマーイベント ID を指定します。
);
```

timeBeginPeriod 関数と timeEndPeriod 関数はタイマーをシステムで動作させるタイマーの解像度を指定します。timeSetEvent 関数が実際にタイマーを開始する関数で，戻値は作成されたタイマーイベントの ID になります。この ID は timeKillEvent 関数で必要になります。uDelay で時間間隔を指定します。uResolution はイベントを発生させるかどうかのタイマーチェックに行くタイマー解像度です。timeBeginPeriod で指定した値以上にする必要があります。

lpTimeProc にはタイマプロシージャを指定します。タイマプロシージャのプロトタイプは，

```
void CALLBACK TimeProc(
    UINT uID,       // タイマイベントの ID
    UINT uMsg,      // 使用できません。
    DWORD dwUser,   // timeSetEvent の dwUser で指定したユーザ定義データです。
    DWORD dw1,      // 使用できません。
    DWORD dw2       // 使用できません。
);
```

となっています。タイマプロシージャの引数は自由に設定できません。そこで，dwUser という引数が用意されています。複数の変数を引数として与えたい場合は，構造体を用意してそのポインタを dwUser として与えるというのが一般的です。fuEvent にはタイマーイベントのタイプを指定します。1度きりの場合は TIME_ONESHOT，繰り返す場合は TIME_PERIODIC を指定します。

前述のタイマーをマルチメディアタイマーに置き換えてみましょう。list5.7 のようになります。マルチメディアタイマを使うためには，mmsystem.h を include することと，winmm.lib をライブラリに追加する必要があります。ライブラリの追加は，「プロジェクト」－「設定」でプロジェクトの設定ダイアログを表示し，リンクタブから「オブジェクト／ライブラリ モジュール」に追加するか，FileView に右ボタンクリックでファイル追加を行います。

それではソースを見ていきましょう。まず，mmsystem.h を include します。続けてタイマープロシージャに渡すユーザ定義データとなる構造体を定義しておきます。今回は hEdit だけなので構造体にする必要はありませんが，後々のためにもこうしておきましょう。次にタイマープロシージャのプロトタイプ宣言をしておきます。これは決まった形なので，自由に決められるのは関数名だけです。WindowFunc の中ではタイマーID の uTimerID とユーザ定義データ構造体 stTimer を定義します。両方とも値を保持する必要があるので static で宣言します。さて，WM_CREATE を捕まえたらエディットコントロールとタイマーの作成を行います。今回は周期を 1000ms つまり1秒にしました。ユーザ定義データには stTimer のアドレス(&stTimer)を与えます。これは dwUser が DWORD (32bit)となっているので，構造体が大きくなると全部を渡すことができなくなるためです。Windows のアドレス(ポインタ)は 32bit となっているため，アドレスを渡すようにすると，どんなデータでも受け渡しすることがかのうとな

```
#include    <mmsystem.h>
                    ⋮
#define IDC_EDIT    1000
typedef struct{
    HWND                hEdit;
}   TIMERSTRUCT;
                    ⋮
void CALLBACK TimerProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1,
                        DWORD dw2);
                    ⋮
    RECT                Rect;
    static UINT         uTimerID;
    static TIMERSTRUCT  stTimer;
                    ⋮
    case WM_CREATE:                 // ウィンドウの作成
        GetClientRect(hWnd, &Rect);
        stTimer.hEdit = CreateWindow("EDIT", "",
            WS_CHILD | WS_VISIBLE
                    ⋮
        timeBeginPeriod(1);
        uTimerID = timeSetEvent(1000, 2, TimerProc,
            (DWORD)&stTimer, TIME_PERIODIC);
        break;
                    ⋮
    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            timeKillEvent(uTimerID);
            timeEndPeriod(1);
            DestroyWindow(stTimer.hEdit);
            DestroyWindow(hWnd);    // メインウィンドウの破棄. WM_DESTROY発行
        }
        break;
                    ⋮
void CALLBACK TimerProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1,
                        DWORD dw2){
    char                lpszMessage[128];
    SYSTEMTIME          stSystemTime;
    TIMERSTRUCT *       lpstTimer = (TIMERSTRUCT *)dwUser;

    GetLocalTime(&stSystemTime);
    sprintf(lpszMessage, "%02lu:%02lu:%02lu.%03lu¥xd¥xa",
        stSystemTime.wHour,   stSystemTime.wMinute,
        stSystemTime.wSecond, stSystemTime.wMilliseconds);
    SendMessage(lpstTimer->hEdit, EM_REPLACESEL, FALSE, (LPARAM)lpszMessage);
}
```

list5.7　マルチメディアタイマー

ります。DWORD にキャストすることも忘れないようにしてください。WM_CLOSE ではタイマーの終了処理を行います。最後にタイマープロシージャを書きます。まず，必要な変数と併せてユーザ定義データ構造体へのポインタを宣言して dwUser を代入しておきます。キャストするのを忘れないでください。これで WindowFunc で作成したエディットコントロールに文字を書きこむことが可能となります。あとは，SetTimer 関数を使ったときとほとんど同じですね。

さて，実行してみましょう。SetTImer 関数を使ったときと比べて時間の精度が上がっていることが確認できると思います。

5.4. ファイル操作，コモンダイアログ

エディットコントロールとタイマーを使ったアプリケーションに A/D 変換ボードや RS232C 用の関数などを組み合わせると様々な実験計測ができるようになります（ボードの扱いはボード毎に異なるのでここでは扱いません。それぞれのボードのマニュアルを参照してください）。このように計測ができるようになると計測したデータをファイルに保存したくなります。ここではファイルを保存する方法とファイル保存のときによく見かける「ファイル保存ダイアログ」の使い方を紹介します。

5.4.1. ファイル操作

Windows でオブジェクトを操作するためにはそのオブジェクトを判別するためのハンドルというものが必要になります（ウィンドウハンドルやインスタンスハンドル，コントロールハンドルもその一つです）。ファイルを操作すためのハンドルはファイルハンドルと言い，CreateFile 関数で取得することができます。閉じるときは CloseHandle 関数を用います。

```
HANDLE CreateFile(
    LPCTSTR lpFileName,             // ファイル名文字列へのポインタを指定します。
    DWORD dwDesiredAccess,          // ファイルへのアクセスモードを指定します。
    DWORD dwShareMode,              // オブジェクトの共有方法を指定します。
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                                    // 子プロセスへの継承方法（構造体）を指定します。
    DWORD dwCreationDisposition,    // ファイルが存在するとき，しないときの動作を指定します。
    DWORD dwFlagsAndAttributes,     // ファイルの属性およびフラグを指定します。
    HANDLE hTemplateFile            // その他のフラグを指定します。
);
```

lpFileName にはファイル名文字列へのポインタを指定します。ファイル名の代わりに”COM1”などを指定すると通信リソースなどにもアクセスできます（詳しくは Help を参照）。dwDesiredAccess は読込みなら GENERIC_READ，書き込みなら GENERIC_WRITE を指定します。dwShareMode はファイルの共有モードで，ここでオープンしたオブジェクトを他のオープン操作に対して許可するかを指定します。読込みを許可するなら FILE_SHARE_READ，書込みを許可するなら FILE_SHARE_WRITE を指定します。lpSecurityAttributes には通常 NULL を指定します。dwCreationDisposition にはファイルが存在するとき，しないときの動作を指定します。表 5.2 を参照してください。dwFlagsAndAttributes にはファイルの属性およびフラグを指定します。属性には表 5.3 のようなものを指定しますが，通常は FILE_ATTRIBUTE_NORMAL を指定します。フラグはほとんど使うことはないと思います。hTemplateFile は NULL を指

表 5.2　dwCreationDisposition

CREATE_NEW	新しいファイルを作成します。すでに存在する場合は失敗します。
CREATE_ALWAYS	新しいファイルを作成します。すでにある場合は上書きします。
OPEN_EXISTING	ファイルをオープンします。ファイルが存在しない場合は失敗します。
OPEN_ALWAYS	ファイルをオープンします。ファイルがない場合は新たに作成します。
TRUNCATE_EXISTING	ファイルをオープンし，サイズを 0 にします。指定ファイルがない場合は失敗します。

表 5.3　dwFlagsAndAttributes

FILE_ATTRIBUTE_ARCHIVE	アーカイブファイル
FILE_ATTRIBUTE_COMPRESSED	圧縮ファイル
FILE_ATTRIBUTE_HIDDEN	隠しファイル
FILE_ATTRIBUTE_READONLY	リードオンリーファイル
FILE_ATTRIBUTE_SYSTEM	システムファイル
FILE_ATTRIBUTE_TEMPORARY	テンポラリファイル

定してください。関数の戻値がオブジェクトへのハンドルとなります。

```
BOOL CloseHandle(
    HANDLE hObject              // 閉じるオブジェクトへのハンドルを指定します。
);
```

hObject には閉じるオブジェクトへのハンドルを指定します。

あと必要になるのはファイルからの読み込みと書き込みですすね。それぞれ ReadFile 関数と WriteFile 関数を使います。

```
BOOL ReadFile(
    HANDLE hFile,                   // ファイルハンドルファイルハンドルを指定します。
    LPVOID lpBuffer,                // データを受け取るバッファへのポインタを指定します。
    DWORD nNumberOfBytesToRead,     // 読み込むバイト数を指定します。
    LPDWORD lpNumberOfBytesRead,    // 読み取ったバイト数を格納する変数のアドレスです。
    LPOVERLAPPED lpOverlapped       // FILE_OVERLAPPED を指定したときに必要です。
);
BOOL WriteFile(
    HANDLE hFile,                   // ファイルハンドルを指定します。
    LPCVOID lpBuffer,               // 書き込むデータバッファへのポインタを指定します。
    DWORD nNumberOfBytesToWrite,    // 書き込むバイト数を指定します。
    LPDWORD lpNumberOfBytesWritten, // 書き込んだバイト数を格納する変数のアドレスです。
    LPOVERLAPPED lpOverlapped       // FILE_OVERLAPPED を指定したときに必要です。
);
```

コメントを読めば大体の意味はわかると思います。

5.4.2.　コモンダイアログ

コモンダイアログとは，Windows があらかじめ用意した良く使うと思われるダイアログのことです。代表的なコモンダイアログには表 5.4 のようなものがあります。多くのコモンダイアログはその設定用の構造体を持っているため，それも併記してあります。今回は保存するファイル名の選択 GetSaveFileName 関数のみを取り上げて，その扱い方を見てみま

表 5.4 コモンダイアログ

機能	関数名	構造体名
色の選択	ChooseColor	CHOOSECOLOR
フォントの指定	ChooseFont	CHOOSEFONT
テキストファイルにおける検索／置換	FindText／ReplaceText	FINDREPLACE
開く／保存するファイル名の選択	GetOpenFileName／GetSaveFileName	OPENFILENAME
印刷	PrintDlg	PRINTDLG
印刷のページ設定	PageSetupDlg	PAGESETUPDLG

す。他の関数についてはヘルプを参照してください。また，コモンダイアログを使うためには，①commdlg.h を include する，②comdlg32.lib を追加する，必要があるので忘れずに行ってください。

では，GetSaveFileName 関数のプロトタイプを見てみましょう。

```
BOOL GetSaveFileName(
    LPOPENFILENAME lpofn     // OPENFILENAME 構造体へのポインタを指定します。
);
```

OPENFILENAME 構造体へのポインタ lpofn のみが引数となっています。つまり，この構造体へ必要な情報を入力してポインタを渡せばいいことになります。では，構造体を見てみましょう。

```
typedef struct tagOFN { // ofn
    DWORD           lStructSize;
    HWND            hwndOwner;
    HINSTANCE       hInstance;
    LPCTSTR         lpstrFilter;
    LPTSTR          lpstrCustomFilter;
    DWORD           nMaxCustFilter;
    DWORD           nFilterIndex;
    LPTSTR          lpstrFile;
    DWORD           nMaxFile;
    LPTSTR          lpstrFileTitle;
    DWORD           nMaxFileTitle;
    LPCTSTR         lpstrInitialDir;
    LPCTSTR         lpstrTitle;
    DWORD           Flags;
    WORD            nFileOffset;
    WORD            nFileExtension;
    LPCTSTR         lpstrDefExt;
    DWORD           lCustData;
    LPOFNHOOKPROC   lpfnHook;
    LPCTSTR         lpTemplateName;
} OPENFILENAME;
```

lStructSize はこの構造体のサイズです。hwnOwner はこのダイアログの親ウィンドウのハンドルです。hInstance は lpTemplateName を指定しない場合は無視します。lpstrFilter はフィルタです。

″CSV ファイル(*.csv)¥0*.csv¥0 すべて(*.*)¥0*.*¥0¥0″

のように指定します。文字列の区切りにはヌル文字（¥0）をいれます。最後はヌル文字2つ（¥0¥0）を入れます。lpstrFile に選択されたファイルのフルパス（F:¥USER¥DATA.CSV など）が入ります。nMaxFile は lpstrFile の大きさを指定します。lpstrFileTitle には選択されたファイル名のみが入ります。nMaxFileTitle は lpstrFileTitle の大きさを指定します。Flags にはダイアログ作成時の細かい設定を指定します。読込みのときは OFN_FILEMUSTEXIST | OFN_EXPLORER（存在しないファイルのときは警告を出す），書き込みのときは OFN_OVERWRITEPROMPT | OFN_EXPLORER（上書きするときは警告を出す）を指定すると良いでしょう。必要なメンバだけセットすれば十分なので，ZeroMemory 関数で初期化しておくと便利です。

では，5.3.2 で作ったプログラムにファイル書き込みの機能を追加しましょう。タイマを開始する前にファイル名を取得してファイルを開きます。画面に表示しているのと同じ内容をファイルに保存し，×を押したらファイルを閉じて終了するプログラムにします。

list5.8 にソースを示します。まず，先頭で commdlg.h の include を行います。次にタイマプロシージャの中でファイルの書き込みを行うことになるので TIMERSTRUCT 構造体に hFile ハンドルを追加しておきます。WM_CREATE を捕まえたら，タイマを作成する前にファイル保存コモンダイアログを表示する準備をします。ファイル名バッファ lpszFileName と構造体 stOfn を宣言して，それぞれ初期化しておきます。ファイル名も初期化しておかないと実行する際にエラーを起します。構造体の設定は一般的なものにして，拡張子のデフォルトを csv（カンマ区切りデータファイル）にしておきました。この形式は Excel 等で標準で読み込むことができます。準備ができたら GetSaveFileName 関数でファイル名を取得します。成功したらこのファイル名を使って CreateFile 関数でファイルを書き込みモードで作成します。ファイル名の読込みとファイルの作成で失敗したらメッセージを表示して終了処理もしています。あとはタイマーを作成しています。WM_CLOSE にはファイルを閉じる操作も加えておきましょう。

タイマープロシージャでは表示用に作成した文字列をそのままファイルに WriteFile 関数を使って書き込んでいます。書き込んだバイト数が間違っていたときはエディットコントロールに「Zero bytes write.」というメッセージを表示しているのもわかると思います。

```
#include    <commdlg.h>
                    ⋮
typedef struct{
    HWND                hEdit;
    HANDLE              hFile;
}   TIMERSTRUCT;
                    ⋮
    case WM_CREATE:                 // ウィンドウの作成
                    ⋮
        char                lpszFileName[MAX_PATH] = "";
        OPENFILENAME        stOfn;

        ZeroMemory(&stOfn, sizeof(OPENFILENAME));
        stOfn.lStructSize   = sizeof(OPENFILENAME);
        stOfn.hwndOwner     = hWnd;
        stOfn.lpstrFilter
            = "CSV File(*.csv)¥0*.csv¥0All Files(*.*)¥0*.*¥0¥0";
        stOfn.lpstrFile     = lpszFileName;
        stOfn.lpstrDefExt   = "csv";
        stOfn.nMaxFile      = MAX_PATH;
        stOfn.Flags         = OFN_OVERWRITEPROMPT;

        if(GetSaveFileName(&stOfn) == TRUE){
            stTimer.hFile = CreateFile((LPCTSTR)stOfn.lpstrFile,
                GENERIC_WRITE,
                FILE_SHARE_WRITE,
                NULL,
                CREATE_ALWAYS,
                FILE_ATTRIBUTE_NORMAL,
                (HANDLE)NULL);
            if(stTimer.hFile == (HANDLE)-1){
                MessageBox(hWnd, "File open faild.", NULL, MB_OK);
                PostMessage(hWnd, WM_CLOSE, 0, 0L);
                break;
            }
        }else{
            MessageBox(hWnd, "File name get faild.", NULL, MB_OK);
            PostMessage(hWnd, WM_CLOSE, 0, 0L);
            break;
        }
        timeBeginPeriod(1);
        uTimerID = timeSetEvent(1000, 2, TimerProc,
            (DWORD)&stTimer, TIME_PERIODIC);
        break;
                    ⋮
    case WM_CLOSE:                  // 各ウィンドウにWM_DESTROYを発行する
        if(QuitMessage(hWnd)){      // 終了の確認
            CloseHandle(stTimer.hFile);
            timeKillEvent(uTimerID);
                    ⋮
void CALLBACK TimerProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1,
                    ⋮
    DWORD               dwBytesWrite;
    WriteFile(lpstTimer->hFile, (LPVOID)lpszMessage, strlen(lpszMessage),
        &dwBytesWrite, NULL);
    if(dwBytesWrite == 0){
        SendMessage(lpstTimer->hEdit, EM_REPLACESEL, FALSE,
            (LPARAM)"Zero bytes write.¥xd¥xa");
    }
}
```

list5.8 ファイル保存コモンダイアログ

5.5. 最後に

Windows プログラムを駆け足でやってきましたが，ここで掲載したのはそのごく一部に過ぎません。ここで紹介しきれなかったコントロールや関数がまだまだ数多くあります。付録に参考資料を載せておきますので，これを参考にいろいろなプログラムに挑戦してみてください。