講　座

OS/2 における並行プログラムの作り方 (IV完)†

鷹　野　　澄††

4. PMにおける並行プログラミング

最終回の今回は，PM (presentation manager) 上での並行プログラムの作り方について述べる．PM 応用プログラムは，従来型のプログラムとは異なり，メッセージ駆動型 (message-driven) のプログラムであるという特徴をもつ．メッセージ駆動型のプログラムでは，次々と発生するメッセージ (message) を迅速に処理するために，マルチスレッドによりメッセージを並行処理する問題や，メッセージ転送によりスレッド間通信やプロセス間通信を実現する問題などが興味深いテーマとなる．

4.1 PM 応用プログラムの概要

PM 応用プログラムでは，ユーザとの対話のためにウィンドウ (window) を生成し，ウィンドウに対してメッセージを発信すると，各ウィンドウに用意されたウィンドウ関数* (window procedure) が呼び出されてメッセージが処理されるというメッセージ駆動型のプログラミング・モデルが採用されている (図-1)．これは，ウィンドウをオブジェクト (object)，ウィンドウ関数をメソッド (method) と考えれば，Smalltalk におけるオブジェクト指向 (object-oriented) プログラミング・モデルと同様のものである．

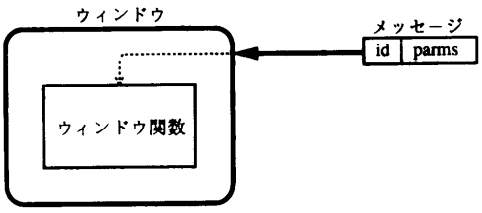

図-1　メッセージ駆動型プログラミング・モデル

(1)　PM 応用プログラムの main 関数の処理

たいていの PM 応用プログラムでは，メッセージ駆動型の機構を実現するために，その main 関数すなわちメイン・スレッドにおいてまずメッセージ・キュー (message queue) を生成し，一つ以上のウィンドウを生成した後に，メッセージ・キューからメッセージを取り出して宛先のウィンドウに発信するというメッセージ・ループ (message loop) 処理を行っている．これを具体的に示したのが図-2 のプログラム例である．

最初の WinInitialize は，スレッドが PM の機能を使用できるようにするための初期化処理で，これによりそのスレッドが使用する PM 環境領域 (これをアンカーブロック (anchor block) と呼ぶ) のハンドル (handle) が得られる．さきに述べたように各ウィンドウにはウィンドウ関数が割り当てられるが，ウィンドウを生成するときは，直接そのウィンドウ関数を指定するのではなく，ウィンドウ・スタイルとウィンドウ関数を定めたウィンドウ・クラス (window class) を指定するという多少まわりくどい方法がとられる．Smalltalk においても，メソッドはクラス (class) において定義され，オブジェクトはクラスのインスタンス (instance：実例) として生成されるので，ウィンドウ・クラスは smalltalk のクラスに対応して設けられたものであると考えられる．図-2 では，標準ウィンドウを1つ生成し，そのタイトルバーや標準アイコンの設定などの前処理を行った後に，メッセージ・ループが開始される．メッセージ・ループは，WM_QUIT メッセージがメッセージ・キューから取り出されたときに終了する (WinGetMsg が FALSE となる)．通常の場合は，メッセージ・ループが終了したら，ウィンドウやメッセージ・キューを破壊してすみやかにプログラムを終了するようになっているので，PM 応用プログラムを終了する場合は，WM_

† Concurrent Programming in OS/2 (IV・finish) by Kiyoshi TAKANO (Earthquake Research Institute, University of Tokyo).

†† 東京大学地震研究所

* 英語では procedure であるが，常に処理結果が返されるため，ここでは「関数」と訳した．

```
/*-------------------------------------------------------------------*/
/* pmmain.sub: PM応用プログラムのmain関数の例                        */
/*-------------------------------------------------------------------*/
#include "sample4.h"    /* 例題用ヘッダファイル(図-9) */
                        /*-- 今回の例題で共通の変数 ------*/
HAB     hab; /* アンカーブロック(PM環境領域)のハンドル */
HWND  hwndFrame,hwndClient; /* 主ウインドウのハンドル*/
ULONG  FCFlags =              /* フレーム生成フラグ  */
                    FCF_TITLEBAR      | FCF_SYSMENU |
                    FCF_SIZEBORDER    | FCF_MINMAX  |
                    FCF_SHELLPOSITION | FCF_TASKLIST ;
int main(void) {
   static CHAR ClassNm[ ]="MainWindow"; /* クラス名 */
   HMQ  hmq;          /* メッセージキューハンドル    */
   QMSG qmsg;         /* メッセージ構造体            */
   hab = WinInitialize(NULL); /* PM環境領域の初期化 */
        /*-- メッセージキューの生成 ----------------------*/
   hmq = WinCreateMsgQueue(hab, 0);
        /*-- ウインドウクラスとウインドウ関数の登録 ----*/
   WinRegisterClass( hab, ClassNm, MainWndProc,
              CS_SIZEREDRAW, 0);
        /*-- 標準フレームウインドウの生成 ----------------*/
   hwndFrame = WinCreateStdWindow(
            HWND_DESKTOP,  /* 親はデスクトップ */
            WS_VISIBLE, &FCFlags, ClassNm, NULL,
            0L, NULL, 0, &hwndClient);
   if( hwndFrame == NULL )  /* エラーの表示と確認 */
     msgout("Window Creation Failed.", ClassNm);
   else {   /* 前処理:タイトルバーと標準アイコンの設定 */
     setTitleTxt(hwndFrame,ClassNm);
     setStdIcon(hwndFrame);
          /*-- メッセージループの実行 ----------------------*/
     while( WinGetMsg(hab, &qmsg, NULL, 0, 0) )
          WinDispatchMsg(hab, &qmsg);
     WinDestroyWindow(hwndFrame); /* ウインドウ破壊 */
   }
   WinDestroyMsgQueue(hmq); /* メッセージキュー破壊 */
   WinTerminate(hab);       /* PMの使用終了        */
   return(0);
}
```

図-2 PM 応用プログラムにおける main 関数の例

QUIT メッセージをメッセージ・キューに入れるという方法が採られる.

PM 応用プログラムの main 関数は, 以上に述べたように, 作成するウィンドウの数や形状などが同じであれば, どのプログラムでもほとんど同じである. このためプログラムの本質的な処理の記述は, すべてウィンドウ関数において行われることになる. そこで今回の例題ではあえて, main 関数はすべて図-2 を使用することにした.

(2) ウィンドウ関数の処理

ウィンドウ関数は, メッセージとそのパラメータを受け取って各メッセージに応じた処理を行い, その結果 (通常は 0) を返すという形の記述からなる. たとえばキーボード入力処理の場合, システムがキーボード入力を常に検知しており, キーの押し下げ/押し上げのたびに WM_CHAR メッセージを生成して, ウィンドウに送信してくれるので, ウィンドウ関数では,

```
/* ウインドウ関数WndProc */
MRESULT EXPENTRY WndProc( HWND hwnd,
        USHORT msg, MPARAM mp1, MPARAM mp2 )
{
 switch(msg) { /* メッセージを判定 */
   case WM_CHAR: /* キー入力処理 */ return(0);
     :
 }
 return(WinDefWindowProc(hwnd,msg,mp1,mp2));
}
```

のようにして, WM_CHAR メッセージが渡されたときにキー入力処理を行うように記述しておけばよい. 入力文字やキーの上げ下げ, シフトキーの状態などは, *mp1* と *mp2* の二つの長さ 4 バイトのメッセージ・パラメータにより渡される.

ウィンドウに送られるメッセージは, キー入力やマウス入力, メニュー入力などのユーザ入力事象のメッセージ, ウィンドウの生成や破壊, 描画などの内部的な事象のメッセージ, タイマからのメッセージなど, システムで定められているものだけでも 150 種類以上ある. さらに応用プログラムの中で定義して, 自分自身やほかのウィンドウとの通信に使用するユーザ定義のメッセージもある. しかしウィンドウ関数では, これらのメッセージのうち, 本当に処理したいメッセージについてのみ記述し, それ以外のメッセージは, 上の例に示したように, システムで用意したデフォルトのウィンドウ関数 WinDefWindowProc にそのまま渡せばよいので, プログラミングはむしろ簡単である. また, こうすることにより, OS のバージョンアップなどでデフォルトのウィンドウ関数が強化された場合, プログラムを書き換えることなしに新機能が利用できるようになる. PM では, デフォルトのウィンドウ関数として WinDefWindowProc のほかに, ダイアログボックス用の WinDefDlgProc, AVIO ウィンドウのサイズ変更などのときのための WinDefAVioWindowProc などが用意されている.

4.2 マルチウィンドウとウィンドウ間通信

ここでは, 応用プログラムの中で複数の標準ウィンドウを生成し, ウィンドウ間でメッセージを転送する方法について述べる. **図-3** は, 一つのスレッドで主ウィンドウと副ウィンドウの二つのウィンドウを生成し, 主ウィンドウのマウス座標を副ウィンドウに転送し, 両方のウィンドウで同

```
/*--- dupdraw1.c ----------------------------------------------------*/
/* 2つのウインドウ間のメッセージ転送プログラムの例                    */
/* [MS-Cによる作成例]                                                */
/*     cl -c -G2sw -W3 dupdraw1.c                                    */
/*     link dupdraw1,/align:16,NUL,os2,dupdraw1                      */
/* [モジュール定義ファイル dupdraw1.defの例]                         */
/*     NAME        DUPDRAW1     WINDOWAPI                            */
/*     PROTMODE                                                      */
/*     HEAPSIZE    10240                                             */
/*     STACKSIZE   10240    ; 注：PMでは8KB以上必要                  */
/*     EXPORTS     MainWndProc                                       */
/*                 Sub_WndProc                                       */
/*-------------------------------------------------------------------*/
#include "pmmain.sub"          /* main関数(図-2参照) */
HWND   hSubFrame,hSubClient; /* 副ウインドウハンドル*/
       /*-- ユーザ定義のメッセージ ----------------------------*/
#define WM_DO_MOVE     WM_USER     /* 新座標まで移動 */
#define WM_DO_LINE     WM_USER+1 /* 新座標まで描画 */
       /*-- ウインドウへのメッセージ送信用マクロ -------*/
#define postSub(msg,mp1,mp2) /* 副ウインドウへポスト */ \
       WinPostMsg( hSubClient, msg, mp1, mp2 )
#define postMain(msg,mp1,mp2)/* 主ウインドウへポスト */ \
       WinPostMsg( hwndClient, msg, mp1, mp2 )
/*-- 主ウインドウ（送信側）のウインドウ関数 ------*/
MRESULT EXPENTRY MainWndProc( HWND hwnd, USHORT msg,
                              MPARAM mp1, MPARAM mp2) {
  static CHAR ClassNm[ ]="SubWindow"; /* クラス名   */
  static POINTL ptl;     /* マウスポインタの(x,y)座標   */
  static BOOL B1down=FALSE; /*マウスボタン1の状態 */
  static BOOL SubOpn=FALSE; /* 副ウインドウの状態  */
  HPS  hps;       /* プレゼンテーション空間のハンドル  */
  switch (msg) {
   case WM_CREATE:  /* ウインドウ生成メッセージ：  */
        /* ここで副ウインドウを生成(sample4.h参照) */
      if( ! mkwnd( hab, ClassNm, Sub_WndProc,
           &hSubFrame, &hSubClient, &FCFlags ) )
        /* 生成失敗:終了メッセージをポスト */
        postMain( WM_QUIT, 0L, 0L);
      else SubOpn=TRUE; /* 副ウインドウ生成完了 */
      return(0);
   case WM_CLOSE:     /* ウインドウ終了メッセージ：  */
      if(SubOpn){  /* 副ウインドウを先に破壊する */
        WinDestroyWindow(hSubFrame); SubOpn=FALSE;
      } break;        /* デフォルト関数へ(終了処理) */
   case WM_BUTTON1DOWN: /* マウスボタン1 DOWN：*/
      WinSetCapture(HWND_DESKTOP, hwnd);
      set_new(ptl,x); set_new(ptl,y);  /* 新座標 */
                /* 副ウインドウへ新座標を送信 */
      if(SubOpn) postSub(WM_DO_MOVE,mp1,mp2);
      B1down= TRUE; break; /* デフォルト関数へ */
   case WM_BUTTON1UP:      /* マウスボタン1 UP：  */
      WinSetCapture(HWND_DESKTOP, NULL);
      B1down= FALSE; return(0);
   case WM_MOUSEMOVE:     /* マウス移動：          */
      if(B1down) {  /* ボタン1 DOWNなら描画 */
        hps=WinGetPS(hwnd); GpiMove(hps, &ptl);
        set_new(ptl,x); set_new(ptl,y); /* 新座標 */
        GpiLine(hps, &ptl);  WinReleasePS(hps);
                /* 副ウインドウへ新座標を送信 */
        if(SubOpn) postSub(WM_DO_LINE,mp1,mp2);
      } break; /* デフォルト関数へ */
   case WM_PAINT:     /* ウインドウ再描画：          */
      hps=WinBeginPaint(hwnd,NULL,NULL);
      GpiErase(hps); WinEndPaint(hps); return(0);
  }
  /*  その他のメッセージの場合:デフォルト関数を呼ぶ  */
  return( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
}
/*-- 副ウインドウ（受信側）のウインドウ関数 ------*/
MRESULT EXPENTRY Sub_WndProc( HWND hwnd, USHORT msg,
                              MPARAM mp1, MPARAM mp2) {
  static POINTL ptl;     /* マウスポインタの(x,y)座標   */
  HPS  hps;       /* プレゼンテーション空間のハンドル  */
  switch (msg) {
   case WM_CLOSE:  /* 主ウインドウのほうに転送      */
      postMain( WM_CLOSE, 0L, 0L ); return(0);
   case WM_PAINT: hps=WinBeginPaint(hwnd,NULL,NULL);
      GpiErase(hps); WinEndPaint(hps); return(0);
            /*-- 以下はユーザ定義メッセージ -------*/
   case WM_DO_MOVE:   /* 新座標へ移動：            */
      set_new(ptl,x); set_new(ptl,y); return(0);
   case WM_DO_LINE:    /* 旧座標から新座標へ描画：  */
      hps=WinGetPS(hwnd); GpiMove(hps, &ptl);
      set_new(ptl,x); set_new(ptl,y); /* 新座標 */
      GpiBox(hps, DRO_OUTLINE, &ptl, 0L,0L);
      WinReleasePS(hps);  return(0);
  }; return( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
}
```

図-3 二つのウィンドウ間のメッセージ転送プログラムの例（シングルスレッド版）

じ図を描くというプログラムの例である．図-4に，実際にこれを実行してマウスで IPSJ（情報処理学会の英文名）と書いたときの例を示す．

図-3は，主ウィンドウのウィンドウ関数 MainWndProc と副ウィンドウのウィンドウ関数 Sub_WndProc のみの簡単なものとなっている．主ウィンドウは main 関数で生成されるが，副ウィンドウのほうは，主ウィンドウの WM_CREATE メッセージの処理の中で生成され，WM_CLOSE メッセージの処理の中で破壊されている．

主ウィンドウから副ウィンドウへのメッセージの転送は，副ウィンドウのハンドル hSubClinent を宛先にした WinPostMsg (hSubClient, msg, *mp1*, *mp2*) で行っている（postSub マクロ参照）．図-3 の場合は，この代わりに WinSendMsg を使ってもよい．WinPostMsg の場合は，メッセージがメッセージ・キューに入れられるが，WinSendMsg の場合は，直接相手のウィンドウ関数が呼び出されるという違いがある．

図-3 の例で，主ウィンドウから副ウィンドウに送られるメッセージは，WM_DO_MOVE と WM_DO_LINE の 2 種類のユーザ定義メッセージである．WM_DO_MOVE は，マウスのボタン 1（左端）を押したときに送られ，WM_DO_LINE は，ボタン 1 を押しながらマウスを動かしたときに送られる．副ウィンドウでは，前者のメッセージがくると新しい座標に移動し，後者のメッセージがくると前の座標から新しい座標まで GpiBox で四角形を描く．主ウィンドウでは同じ二つの座標を直線で結んでいるので，図-4 に見られるように，両ウィンドウの描画は若干異なっている．

なお図-3 では簡単なために，描いた図を保存

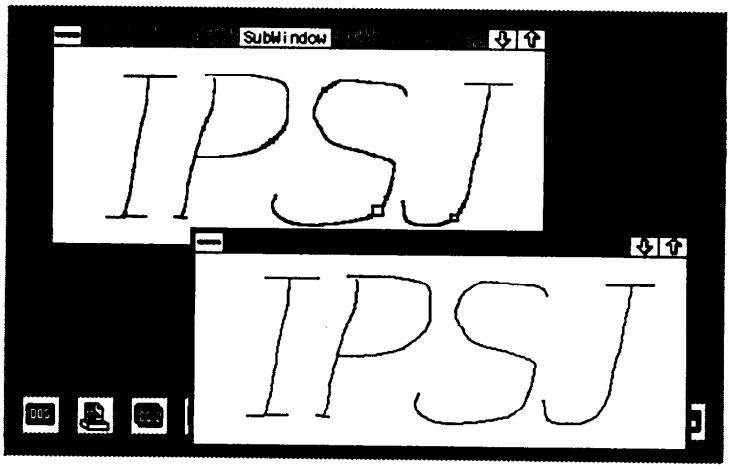

図-4 図-3 のプログラムの実行例

していないので，一度消された部分は二度と出すことができず，サイズ変更時には画面がクリアされるなど不十分な点がある．この問題については，本講座のテーマから外れるので省略した．

4.3 マルチウィンドウの並行プログラミング

前の図-3 では，二つのウィンドウを一つのスレッドで処理していたが，ここでは二つのスレッドを使って，各ウィンドウを並行処理する方法について述べよう．PM では，メッセージ・キューを生成しメッセージ・ループを実行するスレッドをメッセージ・スレッド (message thread) という．このメッセージ・スレッドを複数生成し，各メッセージ・スレッドがそれぞれウィンドウを生成すれば，マルチウィンドウの並行処理ができる．各ウィンドウに対するメッセージは，それを生成したスレッドのメッセージ・キューのほうに入れられる．その結果ウィンドウ間のメッセージ転送により，メッセージ・スレッド間の同期やデータ転送などの通信が可能になる．ここでメッセージ転送には WinPostMsg を用い，WinSendMsg のほうは使わないように注意する必要がある．

図-5 に，主ウィンドウと副ウィンドウをメイン・スレッドと子スレッドの二つのメッセージ・スレッドで並行処理する例を示す．子スレッドは，主ウィンドウの WM_CREATE メッセージ処理の中で生成され，副ウィンドウは子スレッドにより生成されている．子スレッドにおける処理は，メイン・スレッドの処理と基本的に同じであるが，副ウィンドウが生成されたことを WM_SUB_OPEND メッセージで通知し，それが終了したことを WM_SUB_CLOSED メッセージで通知する処理が追加されている．メッセージ・スレッド間でも，OS/2 カーネルで提供されているセマフォなどのツールを使うことは可能であるが，メッセージ・スレッド間の通信には，ここの例で示したように，ウィンドウ間のメッセージ転送を使うほうが自然であろう．

4.4 メッセージの並行処理

図-5 のようにすれば，マルチスレッドによるマルチウィンドウの並行処理が可能になるが，これでもまだ問題が残っている．それは，キーボードやマウスの入力メッセージに対する処理の間は，新たなキーボードやマウス入力がロックされてしまうという問題である．たとえば図-5 の主ウィンドウに対する WM_MOUSEMOVE メッセージの処理の中に，DosSleep (500 L) を挿入すると，その間マウス入力がロックされるので応答が非常に悪くなるということがおこる．ロックされている間マウスの形状を砂時計に変えて，入力を待つようユーザに通知するのも一つの方法であるが，メッセージを並行処理してロックされないようにするほうが優れた方法であろう．PM では別のメッセージ・スレッドや非メッセージ・スレッド (non-message thread) に時間のかかる処理を下請けさせて並行処理し，入力をロックしないようにすることができる．

メッセージ・スレッドに下請けさせる例としては，さきほどの主ウィンドウの WM_MOUSEMOVE の処理に挿入した DosSleep を，副ウィンドウの WM_DO_LINE の処理に移動すること

```
/*--- dupdraw2.c ---------------------------------------------------------*/
/* 2つのメッセージ・スレッド間のメッセージ転送の例                        */
/* [MS-Cによる作成例]                                                     */
/*     cl -c -G2sw -W3 dupdraw2.c                                         */
/*     link dupdraw2,/align:16,NUL,os2,dupdraw2                           */
/* [モジュール定義ファイル dupdraw2.defの例]                              */
/*     NAME        DUPDRAW2   WINDOWAPI                                   */
/*     PROTMODE                                                           */
/*     HEAPSIZE    10240                                                  */
/*     STACKSIZE   10240                                                  */
/*     EXPORTS     MainWndProc                                            */
/*                 Sub_WndProc                                            */
/*-------------------------------------------------------------------------*/
#include "pmmain.sub"           /* main関数(図-2参照) */
HWND   hSubFrame,hSubClient; /* 副ウインドウハンドル */
       /*-- ユーザ定義のメッセージ ----------------------------------*/
#define WM_DO_MOVE     WM_USER    /* 新座標まで移動 */
#define WM_DO_LINE     WM_USER+1  /* 新座標まで描画 */
#define WM_SUB_OPENED  WM_USER+2  /* 副ウインドウ生成*/
#define WM_SUB_CLOSED  WM_USER+3  /* 副ウインドウ消滅*/
       /*-- ウインドウへのメッセージ送信用マクロ ------ */
#define postSub(msg,mp1,mp2) /* 副ウインドウへポスト */ ¥
       WinPostMsg( hSubClient, msg, mp1, mp2 )
#define postMain(msg,mp1,mp2)/* 主ウインドウへポスト */ ¥
       WinPostMsg( hwndClient, msg, mp1, mp2 )
void far childth(void);          /* 子スレッドルーチン */
/*--- 主ウインドウ（送信側）のウインドウ関数 ------ */
MRESULT EXPENTRY MainWndProc( HWND hwnd, USHORT msg,
                              MPARAM mp1, MPARAM mp2) {
  static POINTL ptl;    /* マウスポインタの(x,y)座標   */
  static BOOL B1down=FALSE; /* マウスボタン1の状態 */
  static BOOL SubOpn=FALSE; /* 副ウインドウの状態  */
  HPS  hps;       /* プレゼンテーション空間のハンドル */
  switch (msg) {
    case WM_CREATE:  /* ウインドウ生成メッセージ:  */
          /* ここで副ウインドウ用のスレッドを生成 */
      if( pmmaketh(childth)==0 )
          /* 生成失敗:終了メッセージをポスト */
        postMain( WM_QUIT, 0L, 0L);
      return(0);
    case WM_CLOSE:    /* ウインドウ終了メッセージ:  */
      if(SubOpn){ /* 子スレッドを先に破壊する */
        postSub( WM_QUIT, 0L, 0L); return(0);
      } else break; /* デフォルト関数へ(終了処理) */
    case WM_BUTTON1DOWN: /* マウスボタン1 DOWN: */
      WinSetCapture(HWND_DESKTOP, hwnd);
      set_new(ptl,x); set_new(ptl,y);  /* 新座標 */
                /* 副ウインドウへ新座標を送信 */
      if(SubOpn) postSub(WM_DO_MOVE,mp1,mp2);
      B1down= TRUE; break; /* デフォルト関数へ */
    case WM_BUTTON1UP:     /* マウスボタン1 UP:  */
      WinSetCapture(HWND_DESKTOP, NULL);
      B1down= FALSE; return(0);
    case WM_MOUSEMOVE:     /* マウス移動:        */
      if(B1down) { /* ボタン1 DOWNなら描画 */
        hps=WinGetPS(hwnd); GpiMove(hps, &ptl);
        set_new(ptl,x); set_new(ptl,y); /* 新座標 */
        GpiLine(hps, &ptl); WinReleasePS(hps);
                /* 副ウインドウへ新座標を送信 */
        if(SubOpn) postSub(WM_DO_LINE,mp1,mp2);
      } break; /* デフォルト関数へ */
    case WM_PAINT:     /* ウインドウ再描画:          */
      hps=WinBeginPaint(hwnd,NULL,NULL);
      GpiErase(hps); WinEndPaint(hps); return(0);
                /*-- 以下はユーザ定義メッセージ -------*/
    case WM_SUB_OPENED: /* 主ウインドウをトップに */
      WinSetActiveWindow(HWND_DESKTOP,hwnd);
      SubOpn = TRUE ; return(0);
    case WM_SUB_CLOSED: SubOpn = FALSE;
      postMain(WM_QUIT,0L,0L); /* 主ウインドウ終了 */
      return(0);
  }
  /* その他のメッセージの場合:デフォルト関数を呼ぶ  */
  return( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
}
/*=== childth:子スレッド ==========================*/
void far childth(void) {
  static CHAR ClassNm[ ]="SubWindow";  /* クラス名 */
  HAB  hab; /* アンカーブロック(PM環境領域)ハンドル*/
  HMQ  hmq;          /* メッセージキューハンドル   */
  QMSG qmsg;         /* メッセージ構造体           */
  hab = WinInitialize(NULL);  /* PM環境領域の初期化 */
     /*-- メッセージキューの生成 ---------------------------*/
  hmq = WinCreateMsgQueue(hab, 0);
     /*-- ここで副ウインドウを生成(sample4.h参照)------- */
  if( mkwnd( hab, ClassNm, Sub_WndProc,
         &hSubFrame, &hSubClient, &FCFlags ) ) {
    postMain(WM_SUB_OPENED,0L,0L); /* 生成完了通知*/
  } else {     /* 強制終了:終了メッセージをポスト */
    postMain( WM_QUIT, 0L, 0L );
    postSub ( WM_QUIT, 0L, 0L);
  }
       /*-- メッセージループの実行 ---------------------------*/
  while( WinGetMsg(hab, &qmsg, NULL, 0, 0) )
       WinDispatchMsg(hab, &qmsg);
  WinDestroyWindow(hSubFrame);  /* ウインドウ破壊 */
  postMain(WM_SUB_CLOSED,0L,0L); /* 終了通知 */
  WinDestroyMsgQueue(hmq); /* メッセージキュー破壊 */
  WinTerminate(hab);          /* PMの使用終了      */
  DosExit(EXIT_THREAD, 0);
}
/*-- 副ウインドウ（受信側）のウインドウ関数 ------ */
MRESULT EXPENTRY Sub_WndProc( HWND hwnd, USHORT msg,
                              MPARAM mp1, MPARAM mp2) {
  static POINTL ptl;   /* マウスポインタの(x,y)座標   */
  HPS  hps;     /* プレゼンテーション空間のハンドル */
  switch (msg) {
    case WM_PAINT:     /* ウインドウ再描画:          */
      hps=WinBeginPaint(hwnd,NULL,NULL);
      GpiErase(hps); WinEndPaint(hps); return(0);
                /*-- 以下はユーザ定義メッセージ -------*/
    case WM_DO_MOVE:   /* 新座標へ移動:              */
      set_new(ptl,x); set_new(ptl,y); return(0);
    case WM_DO_LINE:   /* 旧座標から新座標へ描画:    */
      hps=WinGetPS(hwnd); GpiMove(hps, &ptl);
      set_new(ptl,x); set_new(ptl,y); /* 新座標 */
      GpiBox(hps, DRO_OUTLINE, &ptl, 0L,0L);
      WinReleasePS(hps);
      return(0);
  }
  return( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
}
```

図-5 二つのウィンドウ間のメッセージ転送プログラムの例（マルチスレッド版）

で試すことができる．これにより，副ウィンドウの描画は遅くなり，メッセージ・キューのオーバフローがおこると表示も乱れてしまうが，主ウィンドウのほうのマウス入力はスムーズになる．したがって，メッセージ・キューのオーバフローの問題は残るが，メッセージ・スレッドに下請けさせるのは簡単で有効な解決方法である．下請けのメッセージ・スレッドのウィンドウを表示する必要がない場合は，オブジェクト・ウィンドウ（object window）と呼ばれる描画されないウィン

```
/*--- pmterm.c ------------------------------------------------------*/
/* PM版端末プログラム(非メッセージスレッドの使用例) */
/* [MS-Cによる作成例]                                */
/*    cl -c -G2sw -W3 pmterm.c                       */
/*    link pmterm,/align:16,NUL,os2,pmterm           */
/* [モジュール定義ファイル pmterm.defの例]           */
/*    NAME        PMTERM    WINDOWAPI                */
/*    PROTMODE                                       */
/*    HEAPSIZE    10240                              */
/*    STACKSIZE   10240                              */
/*    EXPORTS     MainWndProc                        */
/*--------------------------------------------------------------------*/
#define   WM_THREAD_OK   WM_USER+1
#define   WM_THREAD_END  WM_USER+2
#define   WM_ADJUST_ORG  WM_SEM4 /*最低優先メッセージ*/
#include "pmmain.sub"     /* main関数 (図-2参照)   */
#include "pmrs232c.sub"   /* rs232c関数(図-7参照)  */
    /*-- ウインドウへのメッセージ送信用マクロ ----------*/
#define postMain(msg,mp1,mp2) /*主ウインドウへポスト*/ \
        WinPostMsg( hwndClient, msg, mp1, mp2 )
HVPS hvps;  /* AVIOプレゼンテーション空間のハンドル */
HDC  hdc;   /* デバイスコンテクストのハンドル       */
BOOL   RsOpn = FALSE; /* rs232c オープン状態        */
SHORT  celly, cellx;  /* デバイスセルサイズ         */
SHORT  win_h, win_w;  /* ウインドウの高さと幅       */
SHORT  win_Y, win_X;  /* ウインドウの行／桁         */
SHORT  org_Y, org_X;  /* 原点の行／桁位置           */
USHORT cur_Y, cur_X;  /* カーソル行／桁位置         */
#define  MAX_Y  24    /* AVIOスクリーンサイズ:24行*/
#define  MAX_X  80    /*              80桁 */
ULONG  pause = 0L;    /* スレッド実行待ちセマフォ*/
void far comreader(void);  /* RS232C入力用スレッド  */
/*== 主ウインドウ（キー入力側）のウインドウ関数 ===*/
MRESULT EXPENTRY MainWndProc( HWND hwnd, USHORT msg,
                              MPARAM mp1, MPARAM mp2) {
  HPS  hps;    /* プレゼンテーション空間のハンドル */
  USHORT fsKey; CHAR ch;  /* キーフラグと入力文字  */
  SHORT old_Y, old_X;
  switch (msg) {
    case WM_CREATE:   /* ウインドウ生成メッセージ : */
        /* AVIOプレゼンテーション空間を生成        */
      VioCreatePS(&hvps, MAX_Y, MAX_X, 0, 1, 0);
        /* ウインドウのデバイスコンテクストと結合 */
      hdc = WinOpenWindowDC(hwnd);
      VioAssociate(hdc, hvps);
        /* 原点座標とデバイスのセルサイズを獲得    */
      VioGetOrg(&org_Y,&org_X,hvps);
      VioGetDeviceCellSize(&celly,&cellx,hvps);
        /* RS232Cの初期化とRS232Cスレッドの生成 */
      if( ! rsinit() ) {        /* RS232Cの初期化失敗  */
        postMain( WM_QUIT, 0L, 0L); return(0); }
      RsOpn=TRUE;               /* RS232Cオープン完了  */
      DosSemSet(&pause);  /* スレッド実行待ちセット*/
      if( pmmaketh(comreader)==0 ) {  /* 生成失敗  */
        postMain( WM_QUIT, 0L, 0L); return(0); }
      DosSemClear(&pause); /* スレッド実行可         */
      return(0);
    case WM_CLOSE:    /* ウインドウ終了メッセージ :  */
      if(RsOpn) { RsOpn=FALSE;  /* フラグをオフ     */
        rsterm();       /* RS232Cをクローズ          */
        return(0);      /* RS232Cスレッドの終了を待つ */
      } break;          /* デフォルト関数へ(終了処理) */
    case WM_DESTROY: /* ウインドウ破壊メッセージ : */
        /* デバイスコンテクストとの結合の切り離し */
      VioAssociate(NULL, hvps);
        /* AVIOプレゼンテーション空間の破壊        */
      VioDestroyPS(hvps); return(0);
    case WM_SIZE:     /* ウインドウサイズの変更 :  */
      win_w=SHORT1FROMMP(mp2); win_X=win_w/cellx;
      win_h=SHORT2FROMMP(mp2); win_Y=win_h/celly;
      postMain(WM_ADJUST_ORG,0L,0L); /* 原点調整 */
        /*AVIOではWM_SIZEに対して次の関数を呼ぶ*/
      WinDefAVioWindowProc(hwnd,msg,mp1,mp2);
      return(0);
    case WM_PAINT:    /* ウインドウ再描画 :        */
      hps=WinBeginPaint(hwnd,NULL,NULL);
      GpiErase(hps); VioShowPS(MAX_Y,MAX_X,0,hvps);
      WinEndPaint(hps); return(0);
    case WM_CHAR:     /* キーの押下または解放 :    */
      fsKey=SHORT1FROMMP(mp1);  /* キーフラグ */
      if(fsKey&KC_CHAR) {  /* 通常の文字が入力   */
        ch=(CHAR)SHORT1FROMMP(mp2); /* 入力文字 */
        rsputs(&ch,1);          /* RS232Cに出力   */
      } else
      if(fsKey&(KC_CHAR|KC_VIRTUALKEY|KC_KEYUP) &&
        fsKey&KC_CTRL) {  /* Ctrl文字の入力       */
        ch=(CHAR)SHORT1FROMMP(mp2) & 0x1f;
        if(ch) rsputs(&ch,1);  /* RS232Cに出力    */
      }; return(0);
                /*-- 以下はユーザ定義メッセージ -------*/
    case WM_ADJUST_ORG: /* ここで原点を調整する  */
      old_Y=org_Y; old_X=org_X; org_Y=org_X=0;
      VioGetCurPos(&cur_Y,&cur_Y,hvps);
      if(cur_Y>=win_Y) org_Y=cur_Y-win_Y;
      if(cur_X>=win_X) org_X=cur_X-win_X;
      if(org_Y!=old_Y||org_X!=old_X)
        VioSetOrg(org_Y,org_X,hvps);
      return(0);
    case WM_THREAD_OK: /* ここでタイトルを変更する */
      setTitleTxt(hwndFrame,"PMTERM"); return(0);
    case WM_THREAD_END: /* ここでプログラムを終了 */
      msgout("RS232C thread terminated","PMTERM");
      postMain(WM_QUIT,0L,0L); return(0);
  }; return( WinDefWindowProc( hwnd, msg, mp1, mp2 ) );
}
/*== RS232C入力スレッド ================== */
void far comreader(void) {  int n,nb; CHAR b[1024];
  DosSemWait(&pause,-1L);  /*  実行待ち          */
  postMain(WM_THREAD_OK,0L,0L);  /* 実行開始通知 */
  while(RsOpn){
    n = rshit();          /* n=入力文字数 */
    if( n==0 ) { n=1;   /* 1文字入力  */
      postMain(WM_ADJUST_ORG,0L,0L); /*原点調整*/ }
    nb= rsgets(b,n);          /* RS232Cからn文字入力 */
    if( !RsOpn ) break;       /* RS232Cがクローズ    */
    VioWrtTTY(b,nb,hvps);     /* VIO出力             */
  }
  postMain(WM_THREAD_END,0L,0L); /* 実行終了通知 */
  DosExit(EXIT_THREAD, 0);
}
```

図-6 PM 版端末プログラムの例

ドウを利用することができる．

一方，非メッセージ・スレッドに下請けさせる例としては，次のようなものが考えられる．

ここで，ウィンドウ関数中で下請け処理の完了待ちを行うと，その間入力がロックされてしまうことに注意されたい．このため非メッセージ・スレッドからメッセージ・スレッドへの通知にはWinPostMsgによるメッセージ転送を使うというのが一般的である．

4.5 非メッセージ・スレッドの活用法

メッセージ・スレッドはメッセージ・キューを介さない入力を受け取ることができない．このた

```
#define WM_DONE WM_USER+10
ULONG reqest=0; /* RAMセマフォの初期化*/
  :
/*--- ウインドウ関数 ---------- */
case WM_MOUSEMOVE:
   DosSemClear(&request); /* 下請けの要求 */
   :
case WM_DONE: /* 完了メッセージの処理 */
   :
/*--非メッセージ・スレッド ---------- */
DosSemSetWait(&resquest,-1L); /* 要求待ち */
  : /* (下請け処理) */
    /* ウインドウに完了メッセージを報告 */
WinPostMsg(hwndClient,WM_DONE,mp1,mp2);
```

め通信回線の入力のようにシステムがメッセージを発生してくれないデバイスの入力に対しては，非メッセージ・スレッドの利用が不可欠のものとなる．

図-6 は，非メッセージ・スレッドを利用した非同期通信回線用の端末プログラムの例である．これは，本講座で唯一の“実用的”プログラムであるので，多少長くなってしまったがご勘弁願いたい．参考のために，ここで使用している非同期通信関連のサブルーチンを**図-7** に示したが，その中身については本講座のテーマから外れるので説明を省略する．**図-8** は，この端末プログラムの実行例で，ウィンドウをクローズする直前のものである．

以下では，図-6 の最後の RS 232 C 入力スレッドと，主ウィンドウとの間の通信のみに着目しよう．RS 232 C 入力スレッドの先頭では，RAM セマフォ pause により，主ウィンドウが生成されるのを待ち，その後，主ウィンドウに対して WM_THREAD_OK というユーザ定義メッセージにより実行開始を通知している．

RS232 C スレッドは，回線からの入力がないときは，1 文字の DosRead でデータが受信されるまで待つが，その直前に WM_ADJUST_ORG というメッセージを主ウィンドウに送り，カーソル位置とウィンドウ表示域の調整を要求している．このメッセージは，場合によっては非常に頻繁に送られるので，前に述べたようなメッセージ・キューのオーバフローの問題がおこりうることに注意されたい．PM には，このメッセージ・キューのオーバフローを避けることなどを目的とした，セマフォ・メッセージと呼ばれる事象通知

```
/*--- pmrs232c.sub ---------- */
/* RS232Cサブルーチン(注：エラー表示にmsgoutを使用) */
/*---------- */
#define COM      "com1" /* RS232Cポートのファイル名 */
HFILE  hcom;            /* そのファイルハンドル     */
#define ATTR     0x0000 /* FILE_NORMAL (read/write) */
#define FLAG     0x0001 /* FILE_OPEN  (open only )  */
#define MODE     0x0012 /* OPEN_ACCESS_READWRITE |
                           OPEN_SHARE_DENYREADWRITE*/
#define SETBAUD  0x41   /* 通信速度の設定           */
#define B9600    9600   /* 通信速度=9600bps         */
#define SETLCTL  0x42   /* 回線制御の設定           */
#define DATA7    7      /* データビット=7           */
#define EVENP    2      /* パリティ=EVEN            */
#define STOP1    0      /* ストップビット=1         */
#define GETDCB   0x73   /* デバイス制御の取得       */
#define SETDCB   0x53   /* デバイス制御の設定       */
#define XONOFF   0x03
#define XON      TRUE   /* xon/xoffを有効           */
/*-- rsinit():RS232C初期化 ---------- */
BOOL rsinit(void) { USHORT r, bps;
   struct _lc { UCHAR b,p,s; } lc;
   struct _dcb { USHORT WTOut,RTOut; BYTE HndShk,
        FlowRep,TOutS,ErrRepCh,BrkRepCh,XonCh,XoffCh;
        } dcb;
   if(DosOpen(COM,&hcom,&r,0L,ATTR,FLAG,MODE,0L)) {
     msgout("RS232C Open Failed.",""); return(FALSE);}
   bps=B9600;                    /*-- 通信速度設定 --------- */
   if(DosDevIOCtl(0L,&bps,SETBAUD,1,hcom)) {
     msgout("baud rate set err.",""); return(FALSE); }
                                 /*-- 回線制御設定 --------- */
   lc.b=DATA7; lc.p=EVENP; lc.s=STOP1;
   if(DosDevIOCtl(0L,&lc,SETLCTL,1,hcom)) {
     msgout("line ctrl set err.",""); return(FALSE); }
                                 /*-- xon/xoff設定 ---------- */
   if(DosDevIOCtl(&dcb,0L,GETDCB,1,hcom)) {
     msgout("dev info get err.",""); return(FALSE); }
   if(XON) dcb.FlowRep |= XONOFF;  /* xon/xoff有効   */
   else    dcb.FlowRep &= ~XONOFF; /* xon/xoff無効   */
   if(DosDevIOCtl(0L,&dcb,SETDCB,1,hcom)) {
     msgout("dev info set err.",""); return(FALSE); }
   return(TRUE); }
/*-- rsterm():RS232C終了 ---------- */
void rsterm(void) {
   DosDevIOCtl(0L,0L,1,0x0b,hcom);/* DEV_FLUSHINPUT*/
   DosClose(hcom); }
/*-- n=rshit():RS232C受信文字数入力 ---------- */
int rshit(void){ struct _b { USHORT cch, cb; } b;
   DosDevIOCtl(&b,0L,0x68,1,hcom); return(b.cch); }
/*-- n=rsgets(b,cb):RS232C入力 ---------- */
int rsgets(CHAR *b, int cb) {
   USHORT nr; DosRead(hcom,b,cb,&nr); return(nr); }
/*-- rsputs(b,nb):RS232C出力 ---------- */
void rsputs(CHAR *b, int nb) {
   USHORT nw; DosWrite(hcom,b,nb,&nw); }
```

図-7 RS 232 C 関係のサブルーチン

用の特殊なメッセージが用意されている．

セマフォ・メッセージを WinPostMsg で転送すると，実際にはメッセージ・キューに格納されず，それが送られたこととその第 1 パラメータ *mp 1* のビットごとの OR が記憶される（したがってオーバフローの心配はない）．セマフォ・メッセージには，WM_SEM 1 から WM_SEM 4 まで 4 種類あり，メッセージ・キューから取り出されるときの優先順位が高いほうから WM_

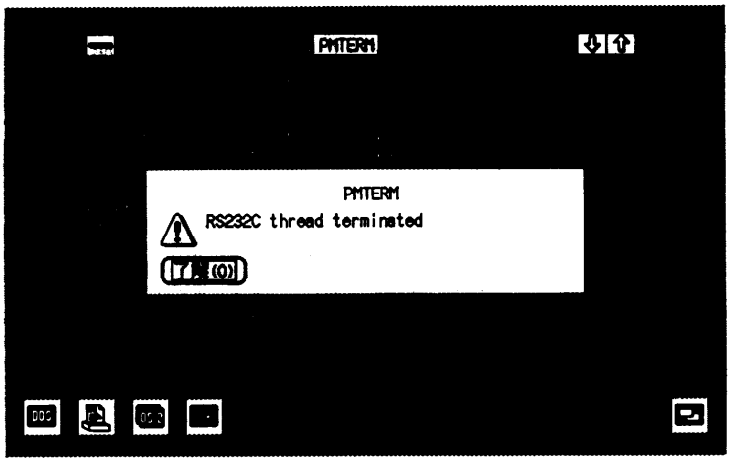

図-8　図-6 の実行例（クローズする直前）

SEM 1, WM_SEM 2, WM_PAINT, WM_SEM 3, WM_TIMER, WM_SEM 4 となっている．ここで WM_PAINT はウィンドウの再描画時に発生するメッセージで，WM_TIMER はタイマからのメッセージである．そのほかの一般のメッセージは，すべて WM_SEM 1 より下で，WM_SEM 2 より高い優先順位をもっている．図-6 では，WM_ADJUST_ORG メッセージとして最も優先順位の低い WM_SEM 4 を使用することにした．なお，WM_ADJUST_ORG メッセージは，主スレッドにおいてウィンドウのサイズが変更されたときにも呼ばれている（WM_SIZE の処理を参照）．

以上ここで示した非メッセージ・スレッドの利用法は，非同期通信回線以外のデバイス入力を PM 応用プログラムで利用したい場合にも役に立つであろう．なお，図-6 の端末プログラムはそれなりに使えるが，さらに，AVIO プレゼンテーション空間を大きくとり，スクロールバーで上下左右にスクロール可能にするなどの改善の余地が多々残されている．興味ある読者はそのような改善をされるとよいだろう．

4.6 PM 応用プログラム間でのデータ転送

複数の PM 応用プログラム間でも，メッセージ・スレッド間の場合のようにメッセージ転送ができると便利である．しかしプログラムが異なると，Win PostMsg で相手のウィンドウのハンドルを指定したり，データをパラメータとして渡すことは容易ではない．そこで PM では，PM 応用プログラム間のデータ転送のために，クリップボード (clipboard) と DDE (dynamic data exchange) が提供されている．

クリップボードは，カット，コピー，ペースト，クリアといったユーザ操作によって，応用プログラム間でデータの転送を行うものである．クリップボードを使うプログラムは，これらの操作をユーザが指定できるように，標準的なメニューを用意しておく必要がある．ユーザは，カットかコピー操作でクリップボードと呼ばれる共有メモリにデータを転送し，ペースト操作でクリップボードからデータを受け取る．データの形式としては，テキスト，ビットマップ，メタファイルの 3 種類が扱える．

クリップボードは，ユーザ駆動でプログラム間のデータ転送を行うものであるが，DDE の場合は，プログラム間で自動的にデータが転送される．DDE では，最初にデータ転送の要求を出したほうをクライアント，その要求を受けたほうをサーバと呼ぶ．データ転送形態としては，一括データ転送，継続的データ転送，コマンド転送の 3 種類がある．

DDE は，PM のメッセージ転送を応用プログラム間でも可能にしたという点が優れた特徴となっている．今回は残念ながら，DDE の利用についての詳しい説明やプログラム例は割愛させていただくが，PM におけるマルチプロセス・プログラミングを考えるときには，DDE の利用法とその実現方法は非常に興味深いテーマとなろう．

4.7 ま と め

今回は，メッセージ駆動型と呼ばれる PM 応用プログラムの概要と，ウィンドウ間のメッセージ転送の概要を述べた後に，マルチウィンドウの並行処理の方法，処理時間がかかるメッセージの並行処理の方法，非メッセージ・スレッドの活用方法などについて例題を示しながら詳しく解説した．しかし，複数の PM 応用プログラムを使ったマルチプロセス・プログラミングについては，十分解説することができず申し訳なく思っている．なお最後に，今回使用した例題のヘッダファ

```
/*-- sample4.h:例題で使用したヘッダファイル----------*/
#include <stdio.h>
#include <stdlib.h>
 /* 以下はソフトウェア開発キットに付属のos2.hを使用  */
#define  INCL_BASE
#define  INCL_WIN
#define  INCL_AVIO
#include <os2.h>
    /*----- クライアントウインドウ関数 ----------*/
MRESULT EXPENTRY MainWndProc(HWND,USHORT,MPARAM,MPARAM);
MRESULT EXPENTRY Sub_WndProc(HWND,USHORT,MPARAM,MPARAM);
    /*----- ユーザ定義のマクロ ----------*/
#define hTitleBar(hFrame) /*タイトルバーのハンドル*/ ¥
     WinWindowFromID( hFrame, FID_TITLEBAR )
#define setTitleTxt(hFrame,txt) /*タイトルバー設定*/ ¥
     WinSetWindowText( hTitleBar(hFrame), txt)
#define hAppIcon  /*標準アイコンのハンドル*/ ¥
  WinQuerySysPointer(HWND_DESKTOP,SPTR_APPICON,FALSE)
#define setStdIcon(hFrame) /*標準アイコンの設定*/ ¥
  WinSendMsg(hFrame,WM_SETICON,hAppIcon,NULL)
#define msgout(msg,title) /*メッセージの表示と確認*/ ¥
  WinMessageBox(HWND_DESKTOP,HWND_DESKTOP,msg,title,¥
          0,MB_OK|MB_ICONEXCLAMATION)
     /* 新しいマウス座標のx部を変数ptl.xへセット */
#define set_new(ptl,x) ptl.x = MOUSEMSG(&msg)->x
/*== pmmaketh:PM版スレッド生成ルーチン ========*/
#define STKSIZ 10240 /* スタック領域の大きさ(8KB以上) */
TID  pmmaketh( PFNTHREAD entry ) {
   USHORT rc; TID tid; CHAR *sb, s[60];
          /*-- スレッドのスタック領域を確保 ----------*/
   if( (sb=(CHAR *)malloc(STKSIZ)) == NULL ) {
      msgout("stack allocation failed.","pmmaketh");
      return(0);
   }        /*-- スレッドを生成 ----------*/
   rc = DosCreateThread( entry, &tid, sb+STKSIZ );
   if(rc){ sprintf(s,"RETURN CODE = %d",rc);
      msgout(s,"DosCreateThead failed."); return(0);
   }
   return(tid); /* 注：生成したスレッドのＩＤを返す */
}
/*=== mkwnd:標準的なウインドウ生成ルーチン =======*/
BOOL mkwnd(HAB hab, PSZ Class, /* ウインドウクラス名 */
        PFNWP WndProc,          /* ウインドウ関数名 */
        PHWND phFrame, PHWND phClient, PULONG pFlg){
      /*-- ウインドウクラスとウインドウ関数の登録 -------*/
   WinRegisterClass(hab,Class,WndProc,CS_SIZEREDRAW,0);
      /*-- 標準フレームウインドウの生成 ----------*/
   *phFrame=WinCreateStdWindow(HWND_DESKTOP,
        WS_VISIBLE, pFlg, Class, NULL, 0L, NULL, 0, phClient);
   if( *phFrame == NULL ){  /* エラーを表示し確認     */
        msgout("Window Creation Failed.",Class);
        return(FALSE);
   } else {    /* タイトルバーと標準アイコンの設定      */
        setTitleTxt(*phFrame,Class);
        setStdIcon(*phFrame); return(TRUE);
   }
}
```

図-9 今回の例題で使用したヘッダファイル

イルを図-9 に示す．本講座のソースプログラムは，JUNET で配布する予定である．

PM における並行プログラミングでは，メッセージ駆動型のしくみを理解することが肝要である．メッセージ駆動型のプログラミング・モデルは，本質的に高い並行処理能力を有すると言われている．PM は，OS/2 という本格的な並行プログラミング環境を提供する OS の上におかれているため，そのような高い並行処理を実現してみるにはよい環境にあると思われる．何よりも，パソコンという手軽な計算機で，このような本格的な並行プログラミングが手軽にできるようになったというのが喜びであろう．筆者のつたない文章で十分にそのあたりが伝えられたかどうか，はなはだ疑問であるが，本講座で一人でも多くの人に，並行プログラムのおもしろさが伝えられれば幸いである．

最後に，本講座を開講するに当たっていろいろお世話になった上智大学理工学部の伊藤潔先生，OS/2 の勉強の機会を与えてくださった東京大学大型計算機センターの石田晴久先生に感謝いたします．また多忙の中，短期間で原稿を読んでいただいた査読委員の方々，いろいろ無理を言って大変ご迷惑をおかけした情報処理学会の学会誌編集担当者に，深くお詫びと感謝の意を表します．

参考文献

［本講座第 1 回（1990 年 10 月）］の補足

［PM マルチスレッド・プログラムに関して］

1) Charles Petzold : PM におけるマルチスレッド・テクニック，日本版 Microsoft Systems Journal (Oct. 1989)，アスキー．
なお，PM マルチスレッド・プログラムについては，同じ著者による “Programming the OS/2 Presentation Manager”, Microsoft Press (1989) にも詳しく解説されている．

［DDE に関して］

2) Susan Franklin, Tony Peters : プレゼンテーションマネジャにおける DDE の技術的な考察，日本版 Microsoft Systems Journal (Dec. 1989)，アスキー．

（平成 3 年 2 月 1 日受付）

鷹野　澄（正会員）

昭和 27 年生．昭和 50 年静岡大学工学部電気工学科卒業．昭和 55 年東京大学大学院工学系研究科電子工学専門課程博士課程修了．工学博士．昭和 55 年東京大学大型計算機センター助卅，昭和 58 年東京大学地震研究所講師（地震予知観測情報センター）．情報理論，オペレーティングシステム，プログラミング言語，ネットワーク，データベースおよび地震予知情報システムなどの研究・開発・運用に従事．著書「MS-DOS」「OS/2」．情報システム研究会幹事，電子情報通信学会，IEEE，ACM，人工知能学会，地震学会など各会員．