Application Manager Software Design

Revision: 1.0
December 2001

1 Introduction to Palm programming in brief

1.1 The Soul of Palm System

1.1.1 FormType

FormType is the main structure to record the operation of every form. Form is the UI of application, describing the main frame of each screen. Each application must have at least one form, and most of the applications have more than one.

FormType not only contains every component restored in the FormObjListType, but also records all changes of UI.

Nevertheless, not all windows are forms, such as Alert and Progress. An Alert is just like a simple Form that only allows Messages and Buttons, returning the index value of Button that user presses. Progress Dialog performs a lengthy progress.

Below is the type definition of **FormType**:

```
typedef struct {
    Word                formId;  // The ID of form
    FormAttrType        attr;       // Attribute of form
    Word                focus;    // Save the focus ID of object
    Word                defaultButton;     // Default button of form
    Word                helpRscId;    // The ID of help resource
    Word                menuRscId; // The ID of menu resource
    Word                numObjects;       // Number of objects
    FormObjListType*    objects;// Pointer to Object structure
    WinHandle           bitsBehindForm; // Save the bits behind the form
    WindowType          window;       // Windows UI of form
    FormEventHandlerPtr handler;       // Pointer to handler of form
```

```
} FormType;
```

Objects are the UI elements of the form. The combination of objects shows the GUI of designer. The types of object can be one of FormObjectType, which is also described below:

```
typedef union {
        void*                   ptr;
        FieldType*              field;
        ControlType*            control;
        ListType*               list;
        TableType*              table;
        FormBitmapType*         bitmap;
        FormLineType*           line;
        FormFrameType*          frame;
        FormRectangleType*      rectangle;
        FormLabelType*          label;
        FormTitleType*          title;
        FormPopupType*          popup;
        FrmGraffitiStateType*   grfState;
        FormGadgetType*         gadget;
        ScrollBarType *         scrollBar;
} FormObjectType;
```

1.1.1.1 Fields: Field object displays one or more lines of text.

1.1.1.2 Control: Control Object allows user to interact with when you add the following into the forms in your application. CtlHandleEvent handles events in control objects.

There are several types of control objects:

- Button: Button displays a text label in a box. The default style for a button in a text string is centered within a rounded rectangle.
- Popup Trigger: A popup trigger displays a text label and a triangle in the left that signifies the control initiates a popup list.
- Selector Trigger: A selector trigger displays a text label surrounded by a gray rectangular frame.
- Repeating Button: A repeat control looks like a button. In contrast to buttons, however, users can repeatedly select repeat controls if they don't lift the pen when the control has been selected.
- Push Buttons: Push buttons look like buttons, but the frame always has square corners. Touching a push button with the pen inverts the bounds. (Like ratio buttons in Windows)
- Check Box: Check boxes display a setting, either on or off. Touching a check box with the pen toggles the setting.

1.1.1.3 List: The list object appears as a vertical list of choices in a box. The current selection of the list is inverted.

1.1.1.4 Table: Tables support multi-column displays. The table object is used to organize several types of UI objects.

1.1.1.5 Bitmap: A bitmap is a graphic that can display on the screen.

1.1.1.6 Label: The label resource displays none editable text or labels on a form.

1.1.1.7 Title: Title resource is the black bar of form that usually indicates the application name and view.

1.1.1.8 Scroll Bars: Scroll bars can attach to fields, tables, or lists, and the system sends the appropriate events when the user interacts with the scroll bar.

1.1.2 EventType

The **EventType** structure contains all the data associated with a system event. All event types have some common data. Most events also have data specific to those events. The specific data uses a union that is part of the EventType data structure. The union can have up to 8 words of specific

data. The common data is documented below the structure.

```
typedef struct {
        eventsEnum          eType;
        Boolean             penDown;
        UInt8               tapCount;
        Int16               screenX;
        Int16               screenY;
        union {
        ...
        } data;
} EventType;
```

1.1.2.1 eType One of the eventsEnum constants. Specifies the type of the event.

1.1.2.2 penDown true if the pen was down at the time of the event, otherwise false. tapCount is the number of taps received at this location. This value is used mainly by fields. When the user taps in a text field, two taps selects a word, and three taps selects the entire line.

1.1.2.3 screenX Window-relative position of the pen in pixels (number of pixels from the left bound of the window).

1.1.2.4 screenY Window-relative position of the pen in pixels (number of pixels from the top left of the window).

1.1.2.5 data is the specific data for an event, if any. The data is a union, and its exact contents depend on the eType field. Such as ctlSelect event will record the controlID, pointer of ControlType, the status of current control, and the value of control.

1.2 Operation of Palm System

1.2.1 Event Loop

Palm is an event-driven system; all activation is performing by a serial of events. Each application has one event loop to process each received event. In the event loop, the application fetches events from the event queue and dispatches them. A typical event loop looks like:

```
static void EventLoop (void)
{
    UInt16 error;
    EventType event;
    do {
            EvtGetEvent (&event, evtWaitForever);
            PreprocessEvent (&event);
            if (SysHandleEvent (&event)) continue;
               if (MenuHandleEvent (NULL, &event, &error))       continue;
            if (ApplicationHandleEvent (&event))    continue;
            FrmDispatchEvent (&event);
    } while (event.eType != appStopEvent);
}
```

In the event loop, the application iterates through these steps:

1.2.1.1 Fetch an event from the event queue.

1.2.1.2 Call PreprocessEvent to allow the designer to process the event before the system event handler or the menu event handler display any UI objects.

1.2.1.3 Call SysHandleEvent to give the system an opportunity to handle the event. The system handles event like power on/ power off, Graffiti input, tapping silk-screened icons, or pressing buttons.

During the call to SysHandleEvent, the user may also be informed about low-battery warnings or may find and search another application. Note that in the process of handling an event, SysHandleEvent may generate new events and put them on the queue. For example, the system handles Graffiti input by translating the pen events to key events. SysHandleEvent returns true if the event was completely handled, that is, no further processing of the event is needed.

1.2.1.4 Call MenuHandleEvent to handle the menu events:

1.2.1.5 If the user has tapped in the area that invokes a menu, MenuHandleEvent brings up the menu.

1.2.1.6 If the user has tapped inside a menu to invoke a menu command, MenuHandleEvent removes the menu from the screen and puts the events that result from the command onto the event queue.

1.2.1.7 MenuHandleEvent returns TRUE if the event was completely handled.

1.2.1.8 If MenuHandleEvent didn't completely handle the event, the application calls ApplicationHandleEvent, a function your application has to provide itself. ApplicationHandleEvent handles only the frmLoadEvent for that event; it loads an activates application form resources and sets the event handleer for the active form. If ApplicationHandleEvent didn't completely handle the event, the application calls FrmDispatchEvent. FrmDispatchEvent first sends the event to the application's event handler for the active form. This is the event handler routine that was established in ApplicationHandleEvent. Thus the application's code is given the first opportunity to process events that pertain to the current form. The application's event handler may completely handle the event and return true to calls from FrmDispatchEvent.In that case, FrmDispatchEvent returns to the application's event loop. Otherwise, FrmDispatchEvent calls FrmHandleEvent to provide the system's default processing for the event.

1.2.1.9 Event loop will not stop to fetch event until received an appStopEvent. Only the active form should process events.

There are five categories of event manager functions that describe below:

System Event Manager Functions

Main Event Queue Management

EvtGetEvent EvtEventAvail

EvtSysEventAvail EvtAddEventToQueue

EvtAddUniqueEventToQueue EvtCopyEvent

Pen Queue Management

EvtPenQueueSize EvtDequeuePenPoint

EvtDequeuePenStrokeInfo EvtFlushNextPenStroke

EvtFlushPenQueue EvtGetPen

EvtGetPenBtnList

Key Queue Management

EvtKeyQueueSize EvtEnqueueKey

EvtFlushKeyQueue EvtKeyQueueEmpty

Handling pen strokes and key strokes

EvtEnableGraffiti EvtProcessSoftKeyStroke

Handling power on and off events

EvtResetAutoOffTimer EvtWakeup

- Main Event Queue Management: This category provides the event queue manager functions, such as EvtGetEvent can get one event each time.
- Pen Queue Management: This category provides pen relative event manager functions, such as EvtGetPen can get the one pen event each time.
- Key Queue Management: This category provides key relative event manager functions.
- Handling pen strokes and keystrokes: This category provides Graffiti relative event manager functions.
- Handling power on and off event: This category provides power relative event manager functions.

1.2.2 Type of Events

Palm OS ® events are structures that the system passes to the application when the user interacts with the graphical user interface. Below shows the types used by Palm OS events. We will not mention the detail of event type; you can reference the “Palm OS Reference” to get the detail description of each event type.

```
typedef enum eventsEnum {
    nilEvent = 0,                   // system level
    penDownEvent,                   // system level
    penUpEvent,                     // system level
    penMoveEvent,                   // system level
    keyDownEvent,                   // system level
    winEnterEvent,                  // system level
    winExitEvent,                   // system level
    ctlEnterEvent,
    ctlExitEvent,
    ctlSelectEvent,
    ctlRepeatEvent,
    lstEnterEvent,
    lstSelectEvent,
    lstExitEvent,
    popSelectEvent,
    fldEnterEvent,
    fldHeightChangedEvent,
    fldChangedEvent,
    tblEnterEvent,
    tblSelectEvent,
    daySelectEvent,
```

```
menuEvent,
appStopEvent = 22,              // system level
frmLoadEvent,
frmOpenEvent,
frmGotoEvent,
frmUpdateEvent,
frmSaveEvent,
frmCloseEvent,
frmTitleEnterEvent,
frmTitleSelectEvent,
tblExitEvent,
sclEnterEvent,
sclExitEvent,
sclRepeatEvent,
tsmConfirmEvent = 35,       // system level
tsmFepButtonEvent,              // system level
tsmFepModeEvent,                    // system level

// add future UI level events in this numeric space
// to save room for new system level events
menuCmdBarOpenEvent = 0x0800,
menuOpenEvent,
menuCloseEvent,
frmGadgetEnterEvent,
frmGadgetMiscEvent,

// <chg 2-25-98 RM> Equates added for library events
firstINetLibEvent = 0x1000,
firstWebLibEvent = 0x1100,
```

```
    // <chg 10/9/98 SCL> Changed firstUserEvent from 32767 (0x7FFF) to
    // 0x6000
    // Enums are signed ints, so 32767 technically only allowed for ONE
    // event.
    firstUserEvent = 0x6000,
    sysExitEvent = 0x9999
} eventsEnum;
```

1.2.3 Entry of application---PilotMain

PilotMain is the entry function of Palm application. A typical PilotMain looks like the code showing below:

```
DWord PilotMain (Word cmd, Ptr cmdPBP, Word launchFlags)
{
    int     error;

    if (cmd == sysAppLaunchCmdNormalLaunch) {
            error = StartApplication();
            if (error) return error;
            EventLoop();
            StopApplication ();
    }

    return 0;
}
```

1.2.3.1 An application launches when it receives a launch code. Launch codes are a means of communication between the Palm OS and the application (or between two applications).

1.2.3.2 StartApplication initialize the system before the application start to work.

1.2.3.3 EventLoop fetches events from the event queue and dispatches them.

1.2.3.4 An application shuts itself down when it receives the event appStopEvent. Note that this is an event, not a launch code. The application must detect this event and terminate. (You'll learn more about events in the next chapter.) When an application stops, it is given an opportunity to perform cleanup activities including closing databases and saving state information. In the stop routine, an application should first flush all active records, then close the application's database, and finally save those aspects of the current state needed for startup.

1.3 Event handler of application

When we change the display form in an application, we usually call **FrmGotoForm** to reach the goal. It will derive three events--- frmCloseEvent, frmLoadEvent, and frmOpenEvent.

frmCloseEvent will due to close current form, frmLoadEvent let designer can change the current event handler. We call **FrmSetEventHandler** to change the event handler.

```
FrmSetEventHandler(form, (FormEventHandlerPtr)
DateViewForm1000HandleEvent);
```

frmOpenForm let designer can initialize proper situation when open a new form. Below is a sample event handler in Date application, this handler process frmOpenEvent, lstSelectEvent, ctlSelectEvent, ctlRepeatEvent. You can add the code to process the different event depending on your need.

```
Boolean     DateViewForm1000HandleEvent(EventPtr    event)
{
      FormPtr     form=FrmGetActiveForm();
      ListPtr
      list=FrmGetObjectPtr(form,FrmGetObjectIndex(form,MonthList1007));
      FieldPtr
      field=FrmGetObjectPtr(form,FrmGetObjectIndex(form,YearField1011));
      SWord       selectedMonth;
      Word        year;
      Char        buffer[9];
      Boolean     handled=false;

      if    ( event->eType == frmOpenEvent ) {
            VCPCtlSetLabel (YearLabel1001, StrIToA (buffer, currentYear));
            //CtlSetLabel(monthTrigger,monthLabel[currentMonth-1]);
            LstSetSelection(list,currentMonth-1);
            FrmDrawForm(form);
            DrawMonth (currentYear, currentMonth);
            MonthDrawTodayInversionCell (currentYear, currentMonth,
            currentDay);
            handled=true;
      } else    if ( event->eType == lstSelectEvent ) {                  // use list UI
            selectedMonth=LstGetSelection(list);
            currentMonth=selectedMonth+1;
            DrawMonth (currentYear, currentMonth);
            MonthDrawTodayInversionCell (currentYear, currentMonth,
            currentDay);
            FldReleaseFocus(field);
            handled=true;
      } else    if ( event->eType == ctlSelectEvent ) {
            switch (event->data.ctlSelect.controlID) {
            case OKBtn1012:
                  year=GetYear(form,field);
                  if(year <= UpperLimitedYear && year >= LowerLimitedYear) {
                        currentYear = year;
                        VCPCtlSetLabel (YearLabel1001, StrIToA (buffer,
                        currentYear));
                        DrawMonth (currentYear, currentMonth);
```

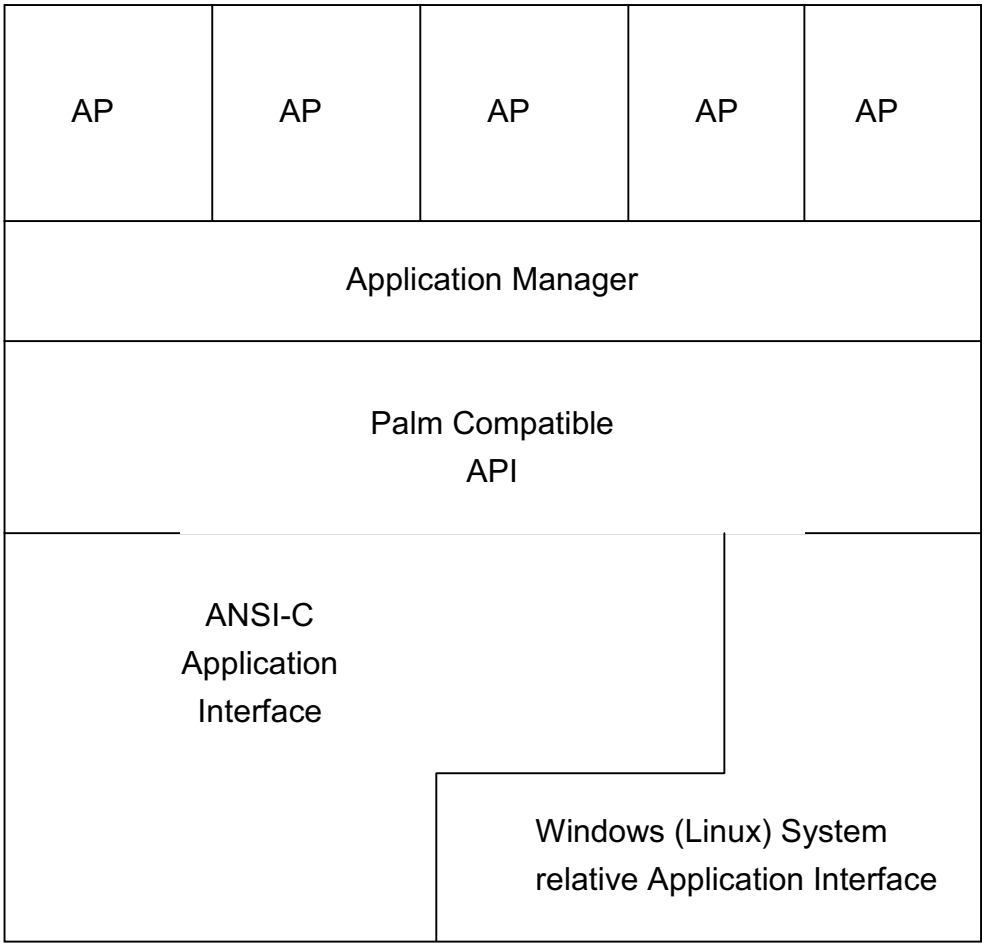
```
                        MonthDrawTodayInversionCell (currentYear,
                        currentMonth, currentDay);
                        } else {
                        FrmAlert(IllegalAlert1100);
                }
                FldFreeMemory(field);
                FldDrawField(field);
                FldGrabFocus(field);
                handled=true;
                break;
            }
        } else   if ( event->eType ==   ctlRepeatEvent) {
            FldReleaseFocus(field);
            switch (event->data.ctlRepeat.controlID) {
                case LeftRepeatButton1002:
                    currentYear--;
                    if(currentYear < LowerLimitedYear)
                    currentYear=LowerLimitedYear;
                    VCPCtlSetLabel (YearLabel1001, StrIToA (buffer,
                    currentYear));
                    DrawMonth (currentYear, currentMonth);
                    MonthDrawTodayInversionCell (currentYear,
currentMonth,
                    currentDay);
                    handled = 0;
                    break;
                case RightRepeatButton1003:
                    currentYear++;
                    if(currentYear > UpperLimitedYear)
                    currentYear= UpperLimitedYear;
                    VCPCtlSetLabel (YearLabel1001, StrIToA (buffer,
                    currentYear));
                    DrawMonth (currentYear, currentMonth);
                    MonthDrawTodayInversionCell (currentYear,
currentMonth,
                    currentDay);
                    handled = 0;
                    break;
```

```
                    case YearLeftRepeatButton1004:
                        currentYear -= 10;
                        if(currentYear < LowerLimitedYear)
                        currentYear=LowerLimitedYear;
                        VCPCtlSetLabel (YearLabel1001, StrIToA (buffer,
                        currentYear));
                        DrawMonth (currentYear, currentMonth);
                        MonthDrawTodayInversionCell (currentYear,
currentMonth,
                        currentDay);
                        handled = 0;
                        break;
                    case YearRightRepeatButton1005:
                        currentYear +=10;
                        if(currentYear > UpperLimitedYear)
                        currentYear= UpperLimitedYear;
                        VCPCtlSetLabel (YearLabel1001, StrIToA (buffer,
                        currentYear));
                        DrawMonth (currentYear, currentMonth);
                        MonthDrawTodayInversionCell (currentYear,
currentMonth,
                        currentDay);
                        handled = 0;
                        break;
                    default:
                        break;
                }
        }

        return(handled);
}
```

2 Introduction to the architecture of Application Manager

2.1 Architecture of Application Manager



2.2 Application Manager file listing

Below is the file list of Application Manager, each file collects the functions of each category of Palm API set

2.2.1 Palm API File

Memory.c String.c EventLib.c FrmDDK.c Parser.c Database.c Control.c List.c Table.c Category.c Field.c Menu.c Window.c Rectangle.c Font.c File.c VCommon.c Time.c Text.c Graffiti.c SysManager.c Network.c

2.2.2 Hand Writing Relative File

HandWriting.c HWUtility.c

2.2.3 System Processing File

VSysSDK.c WisSDK.c VPdaSDK.c

2.2.4 System Relative File

VxDDK.c VMsDDK.c

For example FRMDDK.c collects the functions leading by Frm, HandWriting.c and HWUtility.c is the hand writing function. VPDASDK.c is the SDK function for database processing, VSYSSDK.c is system relative SDK, VWisSDK.c is other SDK collection function, VMSDDK.c is Windows relative SDK function run in Windows system, VxDDK.c is linux relative SDK function run in WisCore APDA platform. Only VMSDDK.c and VxDDK.c is non-shared file, the others are the same run on Windows and WisCore APDA platform.

2.3 Resource Compiler

Before we start an application, we need one Parser to parse the resource file, we design a one-pass parser to parse the resource file. This parser can parse Alert, Menu, Form and control descript in Form. There has **GetWord** function to abstract the word from resource file. **GetCoordinate** will get the coordinate of control. Each control has its own sub-parser function to get the specific attribute.

2.4 User Interface Storage

We use FormPtr, MenuPtr to store all form and menu defined by each application. In our system, we regard Alert as a specific Form. All UI components is including in FormType structure.

2.5 Current execution information

There are ActiveForm and ActiveMenu structure to store the current From and Menu. When application change the current form, that is call the FrmGotoForm,

FrmAlert, or FrmPopupForm. The ActiveForm and ActiveMenu will update to the current active Form and Menu.

3 Interworking of Application Manager and applications

3.1 Windows

An application is a DLL file in WisCore Simulator. It's dynamic being loaded when need. When user select an application to execute. We use **LoadLibrary** to load the correspondent DLL file. If Load successful, we will get the instance of this DLL file, then we call **GetProcAddress** to get the process address of **PilotMain**, then we check whether the resource file is correct.

While all is ready, the server transfers the handle to application by calling **PilotMain**. All API is put in another DLL file, which will be loaded when server is start up, The application will share the same DLL while running.

When the application is completed execution, the handle will give back to the server, prevent the application didn't release all allocated memory. We call **FrmCloseAllForms** to confirm releasing the memory, then redraw the application manager's screen and setting the correct menu, and free the library memory allocated.

```
appInst = LoadLibrary (appList[index].appFileName);
if ( appInst ) {
        WISPilotMain = (PilotMain *) GetProcAddress(appInst, "PilotMain");
        if ( WISPilotMain ) {
                if ( VSetRCPFile (appList[index].rscName) ) {
                        VCPDbCloseDB ();
                        WClearScreen (SCREEN_DISPLAY);
                        (*WISPilotMain) ((Word)sysAppLaunchCmdNormalLaunch,
                        &abx, (Word)0);
                        FrmCloseAllForms ();
                        APM_DrawScreen (true);
                        WSetMenu (menuP);
                }
        }
```

```
        FreeLibrary (appInst);
        appInst = NULL;
}
```

3.2 Linux

3.2.1 Start an application

When user tap on the icon of application, it means that user wish to execute the application. Server will call **vfork** to fork a process space and call **execv** to run the application.

```
if ( vfork () == 0) {
char*args[4];

args[0] = appList[index].appFileName;
args[1] = appList[index].rscName;
args[2] = NULL;
args[3] = NULL;

if ( execv (appList[index].appFileName, args) == -1 ) {
      if( semctl( sem_callid, 0, SETVAL, 1) != 0 ) {
            perror( "SERVER: semctl" );
      }
}

exit (1);
} else {
struct sembuf lock_sembuf[1];

// set semaphore operation option
lock_sembuf[0].sem_num     =   0;
lock_sembuf[0].sem_op      =   -1;
lock_sembuf[0].sem_flg     =   0;

// wait client finished
if( semop( sem_callid, lock_sembuf, 1) == -1 ) {
```

```
        perror("SERVER: semop");
    }
    APM_DrawScreen (true);
    WSetMenu (menuP);
    }
```

3.2.2 Provide service to application

3.2.2.1 We create a function table looks like below,

```
struct GrFunction {
        void            (*func)(void);
} GrFunctions[] = {
////////////////////////////////////////////////////////////
// Alarm Manager Library, completed 4/4 functions now.
////////////////////////////////////////////////////////////
        PipeAlmGetAlarm,                //      0
        PipeAlmGetProcAlarm,            //      1
        PipeAlmSetAlarm,                //      2
        PipeAlmSetProcAlarm,            //      3
}
```

3.2.2.2 When the system startup, we build a API function table through a bridge function that name leads by Pipe, this function will write all API function address in share memory call pipe_ram that we get when initialize LCD display device, this share memory is located immediately after the LCD display memory.

```
void PipeStrCompare (void)

{

        WriteParameter (StrCompare, sizeof(void*), pStrCompare);

}
```

3.2.2.3 WriteParameter will do the job that writes the API function address to share memory. APIADDRESSOFFSET is the starting address of

this share memory. WtransFunction will calculate the offset of current function that will be written. ptr is the address of API function that is processing, and length is the length of function address.

```
Boolean WriteParameter (void *ptr, int length, int offset)
{
      memcpy (&(pipe_ram[APIADDRESSOFFSET +
          WTransFunction(offset)*sizeof(void *)]),
      &ptr, length);
}
```

After WriteParameter, system will create an API service table as below.

0x00000000	AlmGetAlarm
0x00000004	AlmGetProcAlarm
0x00000008	AlmSetAlarm
0x00000E00	WinSetClip
0x00000E04	WinSetDrawMode
0x00000E08	WinSetForeColor

3.2.2.4 When Client needs to call the API function, client must get the API function address first. We provide a fake API function as a bridge to real API function, this shell function will get the real API function

address, then call the function address with input parameter to reach the goal.

```
UInt32 AlmGetAlarm (UInt16 cardNo, LocalID dbID, UInt32 *refP)
{
        UInt32 (*AlmGetAlarmP) (UInt16 cardNo, LocalID dbID,
        UInt32 *refP);

        GetParameter (&AlmGetAlarmP, sizeof(void *),
        pAlmGetAlarm);
        return      (*AlmGetAlarmP) (cardNo, dbID, refP);
}
```

3.2.2.5 We provide GetParameter to get the address of read API function address. The parameter of GetParameter is same as WriteParameter, but we copy the API function address from pipe_ram to ptr for client to call.

```
Boolean GetParameter (void *ptr, int length, int offset)
{
memcpy (ptr, &(pipe_ram[APIADDRESSOFFSET +
        WTransFunction(offset)*sizeof(void *)]), length);
        return      true;
}
```

3.2.3 Return the handle to server

How to wake up server when application stop running: We setup a semaphore mechanism to ensure the server being waken up when the application stop running. After one application is start to run, server will set the semaphore through **semop** to enter waiting state to wait **semctl** function call to wake it up.

```
lock_sembuf[0].sem_num    =   0;
lock_sembuf[0].sem_op     =   -1;
```

```
lock_sembuf[0].sem_flg     =    0;
if( semop( sem_callid, lock_sembuf, 1) == -1 ) {
      perror("SERVER: semop");
}
```

When application stop, it will call **semctl** to wake up server, and server will continue execution after the **semop** setting.

```
    if( semctl( sem_callid, 0, SETVAL, 1) != 0 ) {
    perror( "CLIENT: semctl" );
    }
```

4 Description on the Application Manager implementation

4.1 System Initialization

When Application Manager startup, it will initialize some device and resource that can provide service to application.

4.1.1 Wiscore platform

In Wiscore platform, system must initialize LCD display and touch pad device first, hand writing recorder must startup also for Graffiti recognizing system, then we prepare the API mapping function table to provide service to application, After that, we can draw the GUI of startup screen to start to work.

4.1.2 Windows platform

In Windows platform, we need to create a window as Wiscore Simulator's work space, hand writing recorder also needed in Windows platform, but we apply system's mouse event as pen input event.

4.1 Transfer serial touch pad input data to pen event

We know that there are no pen down, pen move, and pen up event when user taps any point on the screen, all we got is tap coordinate only. But we need to classify those serial input data to different pen event that application can process that event.

When we receive input data from touch pad, it means user start to tap on the screen, we send a pen down event and set penDown flag. We use clock to repartition the data we collect if user taps continuously on the screen, if yes, we send pen move event continuously, if penDown flag is set and we do not receive data from touch pad any more, that means user stop tapping on the screen, we send one pen up event to the event queue.

Below is the demo program:

```
Boolean GetTouchPadInput ()
{
    unsigned int count;
    short data[3];
    int bytes_read, i;

    // get touch pad data
    bytes_read = read(TouchPadDev,data,sizeof(data));
    if (bytes_read != sizeof(data)) {
        if ( !penDown ) {
           tPadX = -1;
           tPadY = -1;
           return     false;
        } else
           penDown = false;
    } else {
        long  stime, ctime=0;
        penDown = true;

        tPadX = 0;
        stime = clock ();
        do {
           bytes_read = read(TouchPadDev,data,sizeof(data));
           if ( bytes_read == sizeof(data) ) {
            oldPadX = data[0]&0xff0;
            oldPadY = (0xfff-data[1])&0xff0;
            if (tPadX == 0) {
               tPadX = oldPadX;
               tPadY = oldPadY;
            }
            if ( (Vabs(tPadX-oldPadX)<150) && (Vabs(tPadY-oldPadY)<150) ){
               tPadX = oldPadX;
               tPadY = oldPadY;
            }
           } else {
            break;
           }
```

```
            ctime = clock ();
        } while ( ctime < (stime+MINSYSTICKS) );
    }

    return true;
}
```

4.2 Most of Events are derived by pen event:

When we receive one pen event, we will check whether it hits any objects in the form. If it does, we will give the relative response, in the mean time we will also put the relative event, which is connected with this object, into event queue.

Below is part of the demo program:

```
        case frmFieldObj:
            CurEvent->eType = fldEnterEvent;
            CurEvent->data.fldEnter.fieldID = id;
            CurEvent->data.fldEnter.pField =
            ActiveForm->objects[ActiveObjIndex].object.field;
            ActiveForm->objects[ActiveObjIndex].object.field->attr.hasFocus =
            true;
            break;
        case frmListObj:
            CurEvent->eType = lstSelectEvent;
            CurEvent->data.lstSelect.listID = id;
            CurEvent->data.lstSelect.pList =
            ActiveForm->objects[ActiveObjIndex].object.list;
            CurEvent->data.lstSelect.selection = LstCheckListHit
            (CurEvent->data.lstSelect.pList, CurEvent->screenX,
            CurEvent->screenY);
            ActiveForm->objects[ActiveObjIndex].object.list->currentItem =
            CurEvent->data.lstSelect.selection;
            VRedrawControl (ActiveForm->objects[ActiveObjIndex]);
            break;
```

4.3 Drawing Function

4.3.1 Windows

All drawing functions are using Windows drawing function, like MoveTo, LineTo, SetPixel, BitBlt. All operation following Windows SDK definition.

4.3.2 Linux

There is **APDA_PutPixel2** function we design to draw a pixel on LCD display, All drawing functions apply this draw pixel function to draw the element they need. There are two modes to draw a pixel, DRAW_SET is drawing a pixel direct on the screen, DRAW_XOR is drawing a pixel in XOR mode.

```
static void APDA_PutPixel2 (Display* display, int x, int y, int color)
{
    if (display &&
        display->area.uLeft <= x && x <= display->area.uRight &&
        display->area.uTop    <= y && y <= display->area.uBottom) {
         register unsigned offset = (y << 5) - (y << 1) + (x >> 3);
         register unsigned char bit = 0x80 >> (x & 7), v = *((unsigned
               char*)(display->pPrivate) + offset);
         color = (color & 1) << (7 - (x & 7));
         switch (copyMode) {
               case DRAW_SET : v &= ~bit;
               case DRAW_XOR : v ^= color;
         }
         *((unsigned char*)(display->pPrivate) + offset) = v;
    }
}
```

4.4 Font

Font is also a free source font. Font structure is defined as a MWCFONT structure. It define font name, maximum width of characters, font height, ascent height of font, characters range defined in this font, bits data of character, and data offset of character.

Below is the MWCFONT structure:

```
/* built-in C-based proportional/fixed font structure*/
typedef struct {
        char *          name;       /* font name*/
        int             maxwidth;   /* max width in pixels*/
        int             height;     /* height in pixels*/
        int             ascent;     /* ascent (baseline) height*/
        int             firstchar;  /* first character in bitmap*/
        int             size;       /* font size in characters*/
        MWIMAGEBITS*    bits;       /* 16-bit right-padded bitmap data*/
        unsigned short* offset;     /* 256 offsets into bitmap data*/
        unsigned char*  width;      /* 256 character widths or 0 if fixed*/
} MWCFONT, *PMWCFONT;
```

The bits data of character is looks like below, It is combined with twelve 16-bits data, that drawing a character in one 16x12 matrix.

Below is a ‘!’ character’s font data:

```
/* Character (0x21):
    bbw=1, bbh=8, bbx=1, bby=0, width=3
    +----------------+
    |                |
    |                |
    | *              |
    | *              |
    | *              |
    | *              |
    | *              |
    | *              |
    |                |
    | *              |
    |                |
    |                |
    +----------------+ */
0x0000,
```

```
0x0000,
0x4000,
0x4000,
0x4000,
0x4000,
0x4000,
0x4000,
0x0000,
0x4000,
0x0000,
0x0000,
```

According those data, we designed a character drawing function to draw a character on the screen. The first step is calculating the offset of drawing character, and then gets the width and bits data of character, then according those bits data, draw the corresponding pixel on the screen.

```
Int16 VDrawChar (WChar ch, Coord x, Coord y)
{
        unsigned short  offset;
        int             width;
        unsigned short  byte;
        IMAGEBITS*      bits;
        int             i, j;

        offset = currentFont->offset[(ch-0x20)];
        width = currentFont->width[(ch-0x20)];
        bits = helvR10_bits + (ch-0x20)*12;

        for ( i = 0; i < currentFont->height; i++, bits++ ) {
                byte = ((*bits)&0xff00);
                if ( !byte )
                        continue;
                for ( j = 0; j < 8; j++ ) {
                        if ( byte & 0x8000 ) {
                        VDrawPixel (x+j,y+i,tColor);
                        }
```

```
                    byte = byte << 1;
                }
            }

            return width+1;
        }
```

4.5 Synchronization

4.5.1 Import mail

We apply CMapiSession to import mail from Outlook Express. The sample program listing below:

```
BOOL bMapiInstalled = session.MapiInstalled();
//Logon to MAPI
bLoggedOn = session.Logon(_T("WisCore"));

DmOpenRef       MailDb=NULL;
Word            mailIndex=0;
UInt32          recSize, totalSize=0;
MailDb=OpenPdb();
if ( !MailDb )      return;
if (bLoggedOn) {
    //Send the message
    BOOL bSent = session.ReadFirst(MessageID, &msg);
   m_Body=msg.m_sBody;
   m_Subject=msg.m_sSubject;
   m_From=msg.m_sFrom;
   m_FromAddress=msg.m_sFromAddress;
   m_To=msg.m_sTo;
   m_ToAddress=msg.m_sToAddress;
```

```
    while ( (bSent = session.ReadNext(MessageID, &msg))) {
        m_Body=msg.m_sBody;
        m_Subject=msg.m_sSubject;
    }
}
```

4.5.2 Mail encoding system

The encoding system we support include:

- QuotedPrintable
- Base64
- UUEncode
- HTML
- ASCII

4.5.3 Synchronization

We apply CreateFile to create a communication resource to synchronization data. Through the COM port, we can send and receive data from PDA.

```
CreateFile( szPort, GENERIC_READ | GENERIC_WRITE,
            0,                          // exclusive access
            NULL,                        // no security attrs
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL |
            FILE_FLAG_OVERLAPPED, // overlapped I/O
            NULL )) == (HANDLE) -1 )
```

5 Information on the X Scribble project

5.1 Introduction of Graffiti

We get a free source called X-Scribble to implement the Graffiti hand writing recognition. The X-Scribble is a stroke recognition system, so it needs a stroke database while compare the input stroke data.

There are three categories of recognition database in X-Scribble--- Letter, Digital, and Punctuation. Each category has its own database to store the recognition data.

We make a little modification of X-Scribble. The original source will call malloc more than thousands of times to allocate memory storing the database of hand writing data, and then will result in insufficient memory because **malloc** will allocate 4K in each time.

All X-Scribble data information is gotten from http://www.handhelds.org

5.2 How Graffiti Works

5.2.1 Initialize recorder

Setup the Graffiti system and start to receive data from touch pad input.

5.2.2 Translate Pen input to Graffiti Input

When get pen down event in hand writing area, we will set the HandWritingStart flag to note user starting to input data for recognition. After this, all pen move event will be regarded as hand writing data till receive pen up event, it denotes user completion input hand writing data and ready to recognize data.

5.2.3 Recognize

Call **hwAddPoint** to add point into recognition points array, this array stores consecutive points as hand writing data. When finishing input hand writing data. Call **hwRecognizeStart** to start recognition input character.

Recognition engine will filter out close point to avoid too much input data wasting recognition time, the second step will normalize input data, then according the input state grfInputState the engine will select different recognition database, and recognize input data through this database. The recognition engine will give scores when recognize the input data with each character in selected database. The lowest score character will be the recognize one.

```
void hwAddPoint (int x, int y);  // x, y is the input coordinate of hand writing
Char hwRecognizeStart (UInt8 grfInputState);     // grfInputState is the input
                                                 // category of character
```

5.2.4 Finalize recorder

Release allocated memory.

5.3 Database of hand writing data

Below is an example of hand writing database. The first line of database file is the number of classes(character) in the database, then is the alphabet of those database, after this is the vector and matrix of each character, The final section is the sample coordinates of each character. One character not only has one sample coordinate set, For example ‘0’ has 4 sample coordinate sets, ‘1’ has 8 samples coordinate sets. Each sample coordinate set lead by a number of coordinates.

Below is part of sample database

20 classes
0
1
2
3
4
6
7
8
9
B
N
A
S
R
P
U
V
W
L
5
V 12 -0.868832 0.336823 54.4382 0.824697 5.95937 -0.639416 0.364462 128.224 7.0839 7.21703 3.27009 0.265
M 12 12
0.0706877 0.178766 -1.55779 0.0170591 -1.18709 0.0988158 -0.0405307 -5.34896 -0.261723 -0.331571 -0.346411 0.000916717
0 0.456038 -4.0199 0.0469372 -3.26572 0.295152 -0.127233 -14.1596 -0.686385 -0.865744 -0.895775 0.00100384
0 0 44.9158 -0.766481 29.6618 -3.04356 1.50025 156.169 6.50061 9.22175 9.16945 0.0277157
0 0 0 0.0187331 -0.47486 0.0654544 -0.0372046 -2.78889 -0.0949287 -0.153827 -0.142259 -0.00178416
0 0 0 0 37.9436 -4.69544 2.31214 126.788 5.98374 7.10354 6.89876 0.0680542
0 0 0 0 0 0.657636 -0.339774 -14.574 -0.645471 -0.767322 -0.70343 -0.013669
0 0 0 0 0 0 0.179529 7.32567 0.306387 0.376506 0.335367 0.00796206
0 0 0 0 0 0 0 578.293 24.3964 33.3165 32.5936 0.201431
0 0 0 0 0 0 0 0 1.12731 1.43227 1.4347 0.00532357
0 0 0 0 0 0 0 0 0 1.94989 1.9333 0.0079861

0 0 0 0 0 0 0 0 0 0 1.93982 0.00475851
0 0 0 0 0 0 0 0 0 0 0 0.0005
……
4 0
28 111 162 110 161 109 160 108 160 107 160 105 162 102 163 100 166 98 170 97 175 97 180 97 187 100 192 105 196 111 198 118 197 126 194 133 189 138 183 142 177 142 170 139 165 134 161 127 159 120 159 113 160 108 162 104 166

27 40 155 39 154 38 154 37 153 35 153 32 155 30 157 28 160 27 164 26 169 27 176 29 182 32 188 36 192 40 194 45 193 50 190 54 184 58 178 61 171 62 164 60 158 56 154 50 152 44 153 37 155 32 159

29 102 43 103 43 102 43 101 43 99 45 97 47 95 50 94 55 93 60 94 65 95 68 97 73 101 76 106 77 112 77 117 75 122 73 126 69 128 64 128 59 127 53 124 48 120 43 114 40 109 39 104 39 100 41 98 43 97 46

26 28 39 27 38 27 39 26 40 24 41 22 44 21 48 20 52 19 58 20 64 22 69 25 73 29 76 34 77 38 75 43 72 47 68 50 63 51 57 51 51 49 46 46 41 42 38 37 37 32 37 29 39