Open Watcom Windows Programming Interface (WPI)

Originally written by WATCOM International Corp.
Revised by Open Watcom contributors

Table of Contents

Open Watcom Windows Programming Interface (WPI)

1 Open Watcom Windows Programming Interface (WPI)

Paul Fast, December 23, 1993

Open Watcom Windows Programming Interface (WPI) is a developers tool to aid in porting applications from Microsoft Windows to IBM Presentation Manager (PM). The goal in the creation of WPI (pronounced wippee) is to supply programmers with a set of macros and library routines which allow them to quickly convert an application which already exists in Windows code to PM. It should be mentioned that this will not necessarily give a very efficient PM program; only a working executable in short period of time. Using WPI also allows the developer the luxury of having only one copy of the source instead of a Windows version and a PM version. The WPI interface has evolved only as functionality is required; future projects may use Windows features which require additional enhancements to WPI.

WPI is explicitly designed to port Windows functionality to OS2/PM (not the other way around). Because of this, converted projects may require additional system dependent code to take advantage of native OS/2 features (such as drag and drop or notebooks in OS/2).

This document is intended to be used by developers porting applications from Windows to PM as an introduction to using WPI. It notes situations which are similar to Windows code and some which are quite different from Windows code.

1.1 What is WPI?

WPI is a combination of macros and library routines. The WPI project is in ./bld/wpi and contains subdirectories for source code, header files, and libraries. The header file wpi.h includes the type header file wpitypes.h and either wpi_win.h or wpi_os2.h depending on the platform for which you wish to compile. The header files wpi_os2.h and wpi_win.h contain the macros and prototypes for functions which correspond to Windows API calls when compiling under the Windows platform and PM API calls when compiling for the OS2 platform. It should be noted that these files are not necessarily complete. These files will constantly be modified and further macros will no doubt be added as their need arises. Therefore, if a particular function is required by the developer and it does not already exist, it is the job of that developer to add it to both wpi_win.h and wpi_os2.h. The source code for the library routines are in wpi_win.c and wpi_os2.c.

To distinguish between the Windows and PM platforms, the Open Watcom C compiler provides a __OS2_PM__ macro which indicates that the program is being compiled for OS2 Presentation Manager. To separate Windows and PM code something like the following is needed:

```
#ifdef __OS2_PM__

        // PM code

#else

        // Windows code

#endif
```

The file wpitypes.h contains the conversion of types from Windows to PM. There are some cases in which Windows types have equivalent PM data types, such as HDC and HWND. These obviously are not redefined. In other cases, a non existent equivalent can easily be added. For example the data type ULONG in PM is defined as unsigned long for Windows in wpitypes.h. It should be noted that not all data type definitions are as neat and clean as this; however, an attempt has been made to make this as invisible to the developer as possible.

The files wpi_win.h and wpi_win.c contain macros and routines which translate to Windows API calls under the Windows platform and wpi_os2.h and wpi_os2.c translate to PM API calls under the PM platform. Some of these routine will have the same parameter list as the Windows version of the API and others may appear drastically different than their Windows counterpart. Many of the Windows APIs do not have corresponding PM APIs and in these cases, many lines of code are required to mimick the Windows call. In cases where more than one line of code is required to convert the Windows code to PM, the routines have been coded as library functions instead of macros. All WPI functions are prefixed by _wpi_ and contain no uppercase characters. Furthermore, an attempt has been made to keep the macro names as similar to the Windows functions as possible. Hence, BeginPaint is called _wpi_beginpaint. In cases where the Windows API has an exact equivalent in PM, the Windows function is defined as is for PM. For example, GetClientRect has an PM equivalent of WinQueryWindowRect and hence GetClientRect has been defined as WinQueryWindowRect for PM. This means that some Windows API names can remain the way they are. Even if an API has an exact equivalent and is defined that way for PM, it is also defined in the form _wpi_functionname for both Windows and PM. It is up to the developer as to whether they wish to use the Windows name or the WPI name.

1.2 Converting Windows Applications

The nature of the application will dictate the amount of code reusability available through WPI (and hence the usefulness of WPI). If the application deals with elements that are similar on both the Windows and PM platform then perhaps the developer can reuse 90% or more of their code. In cases where Windows and PM handle things quite differently, reuseability may only reach 70-80%. The developer should determine places in their Windows application that may differ quite drastically in PM such as file storage formats or window classes or messages. These will be the most difficult modules of code to port.

The method the developer uses to port is really up to him or her. The most straightforward way is to (after reading this document) choose a module in the applications code that is small or has relatively few major Windows/PM differences and begin there. The file wpi.h will need to be included (it includes the other WPI header files) in every module that uses WPI macros. Every function call that is a Windows API will need to be replaced with the equivalent _wpi_functionname macro. The simplest way to determine which macro to use is to search for the API name in wpi.h. It should be relatively easy to determine the name of the macro and whether or not the parameter list has changed. The OS2 Toolkit contains on-line help which describes the PM APIs. This will also prove to be very useful since some macros may take parameters that are required and only used in the PM calls.

The user may find it easier to have separate code for the initialization of their program. While Windows has a WinMain key word, PM applications simply begin with the normal C main. Moreover, PM requires the user to initialize the windows and create a message queue. Although _wpi_createwindow and _wpi_registerclass are provided in WPI, the user may find the code to register and create windows different enough to warrant separate Windows and PM code. It is safest to look at the WPI code to see if it performs as you wish.

One final word on converting code. Some applications have unique memory allocation functions they use in order to track memory. Since WPI occassionally allocates memory, users may wish to use their own memory allocation and freeing routines. They can do so by defining the symbol _wpi_malloc and _wpi_free to be their own routines. If they are not defined, WPI sets them to the default of malloc and free. All memory allocations and frees are performed with _wpi_malloc and _wpi_free (the exception is in _wpi_selectobject which uses alloca. This routine is discussed later).

1.3 Differences in Philosophies

Before beginning the PM conversion, you should become aquainted with a few differences between the philosophies of Windows and PM. This document does not cover all of them; only the ones that pertain to WPI.

1.4 PM Windows

To begin with, windows are put together slightly differently. A typical Windows window has a title bar with a system menu and application menu, border and a client area. These are all part of the window indicated by one HWND. In PM, each of these components has its own HWND and all are considered to be children of the frame window HWND. Hence, the client area has its own HWND as does the menu bar, the scroll bars, etc. Drawing in the client area should be done by referencing the HWND of the client; however, visibility, destruction and menu operations need to reference the HWND of the frame. To overcome some of these problems, the macros _wpi_getclient(hwnd) and _wpi_getframe(hwnd) have been defined which do nothing in Windows and return the client window handle of the supplied frame handle and the frame handle of the given client (respectively). Moreover, through WPI, HMENU has been defined as an HWND and routines have been created which allow the user to use an HMENU as it is in Window.

1.5 Instances and Anchor Blocks

Windows programs have the concept of an instance which is required by some functions to differentiate between possibly numerous instances of the program. PM does not have an instance identifier, however it does have an anchor block. In WPI, instance handles and anchor blocks are associated in the data type WPI_INST. This type should be used instead of the INSTANCE type for declaring INSTANCE variables in Windows. For PM, it is defined as a follows:

```
typedef struct {
        HAB             hab;
        HMODULE mod_handle;
} WPI_INST;
```

The hab is the anchor block that is required for many of the function like creating presentation spaces and registering windows. The mod_handle field is required when working with DLLs. It is particularly required for loading accelerator tables, menus and other resources. If you are not working with DLLs, you

may simply set this field to NULL. PM functions that require HABs will have macros that accept a WPI_INST and the hab field will be extracted from the structure. The following routines handle the WPI_INST data type:

_wpi_setwpiinst(hab, mod_handle, &wpi_instance)

Use this function to initialize a WPI_INST. For Windows set hab to the instance and set mod_handle to NULL.

_wpi_issameinst(inst1, inst2)

This compares the two instances and returns whether or not they are the same.

_wpi_setmodhandle(name, inst)

This sets the module handle only, of a WPI_INST. This function does nothing in Windows.

_wpi_setanchorblock(hwnd, inst)

This sets the anchor block only, of a WPI_INST given a window handle. For Windows, this sets the instance.

1.6 Coordinates and Rectangles

Another major difference between Windows and PM is the coordinate system used for the desktop. In Windows the default for the origin is the top left of the screen with the axes extending positively to the right and down. In PM, the origin is the bottom left and the axes extend positively to the right and up. Therefore, the coordinates for Windows will differ from that of PM. In fact, PM has a slightly different definition for their rectangle data type which also reflects its coordinate origin. Where Windows has:

```
typedef struct {
        int     left;
        int     top;
        int     right;
        int     bottom;
} RECT;
```

PM has

```
typedef struct {
        LONG    xLeft;
        LONG    yBottom;
        LONG    xRight;
        LONG    yTop;
} RECTL;
```

Aside from the different type names the issues that needed resolving through WPI were the differences in field names and the different types for the fields. The different structure field names make accessing individual fields in the rectangle structure awkward and the different types of the fields once they are accessed is important to bear in mind. WPI defines a data type WPI_RECT which is RECT in Windows and RECTL in PM. To get and set the values of the structure, a call must be made to a routine (see below) supplying the rectangle structure and the values for the fields. A data type called WPI_RECTDIM has been created which is the type of the fields of the structure. WPI_RECTDIM is defined as LONG in PM and int

in Windows. The following routines handle rectangles and assume that left, top, right, and bottom are defined as WPI_RECTDIM:

_wpi_setrectvalues(&r, left, top, right, bottom)

In Windows this simply sets the values of the fields. In PM this will set top to yBottom and bottom to yTop. This function can be useful when you have Windows code which subtracts bottom from top to get window heights.

_wpi_setwrectvalues(&r, left, top, right, bottom)

This will set the values of the relative fields in both Windows and PM (ie no switching is performed between top and bottom).

_wpi_getrectvalues(r, &left, &top, &right, &bottom)

This will retrieve the values of the rectangle structure in Windows as expected and will assign the value of yBottom to top and yTop to bottom. Again, this is because of the difference in the coordinate systems.

_wpi_getwrectvalues(r, &left, &top, &right, &bottom)

This will retrieve the values of the fields in both Windows and PM as expected (ie no switching is performed between top and bottom).

Because some developers may wish to work strictly with the int data type, the following equivalent functions have been created which do the same as the above routines, but accept as parameters int instead of WPI_RECTDIM:

_wpi_setintrectvalues(&r, left, top, right, bottom)

_wpi_setintwrectvalues(&r, left, top, right, bottom)

_wpi_getintrectvalues(r, &left, &top, &right, &bottom)

_wpi_getintwrectvalues(r, &left, &top, &right, &bottom)

Using these macros will at first be annoying; however the need to have a WPI_RECT lies in the fact that many macros will require a Windows RECT for the Windows API and a PM RECTL for the equivalent PM API. Hence, there is really no getting around the matter. Note especially the difference between _wpi_setrectvalues and _wpi_setwrectvalues; and _wpi_getrectvalues and _wpi_getwrectvalues. For Windows, top < bottom is normal for a rectangle. However, for PM bottom < top is normal for a rectangle. Hence by switching the values of top and bottom in the assignment, we guarantee that if top < bottom in Windows, then bottom < top in PM. It is recommended that coordinates that are saved be stored in the windows format and then be converted to PM coordinates before displaying or drawing.

The WPI_RECT structure also has a sister WPI_POINT data type. This is defined as POINT in Windows and POINTL in PM. This type should be used to replace the POINT structure in the Windows code. This will assure the user that Windows APIs will use the POINT structure and PM ones, the POINTL structure. The fields of this structure are named the same (x and y); however, the types for Windows are int and for PM, LONG. While there is a _wpi_setpoint macro in WPI, it is not essential to use it because of the common field names.

In addition, the following routines work with rectangles, points and coordinates:

_wpi_getwidthrect(r)

This will return the width of the rectangle.

_wpi_getheightrect(r)

This will return the height of the rectangle.

_wpi_cvth_y(y, height)

This will convert the value of y for a window in Windows coordinates to PM coordinates. Height is the height of the window and the new value is returned.

_wpi_cvth_pt(&pt, height)

This converts the y value of a WPI_POINT in a window from Windows coordinates to PM coordinates. Height is the height of the window.

_wpi_cvth_rect(&rect, height)

This converts the top and bottom values of a WPI_RECT from Windows coordinates to PM coordinates. Height is the height of the window and is assumed to be a LONG.

_wpi_cvth_wanchor(y, window_cy, parent_cy)

This converts an anchor point (y) from Windows to PM coordinates. The anchor point is the coordinates at which a window is displayed on the desk top (top left for Windows and bottom left for PM). All values are assumed to be LONG.

_wpi_cvts_y(y)

This converts a value relative to the desktop in Windows coordinates to that of PM. The new value is returned (ie. same as _wpi_cvth_y except it is relative to the desktop).

_wpi_cvts_pt(&pt)

Same as _wpi_cvth_pt except that the point is assumed to be relative to the screen (desk top window).

_wpi_cvts_rect(&rect)

Same as _wpi_cvth_rect except that the values are assumed to be relative to the screen (desk top window).

_wpi_cvts_wanchor(y, window_cy)

Same as _wpi_cvth_wanchor except the anchor point is assumed to be relative to the screen (desk top window). Values are expected to be LONG.

_wpi_cvtc_y(hwnd, y)

Same as _wpi_cvth_y except the routine only takes the window handle (it calculates the window height). The new value is returned and y is expected to be LONG.

_wpi_cvtc_rect(hwnd, &rect)

Same as _wpi_cvth_rect except the routine calculates the window height itself.

The _wpi_cvt* macros are used for converting coordinates between the two target systems. For PM, calling the macros once will convert the y values passed in from Windows coordinates to the equivalent PM coordinates. In some instances, coordinates may be stored differently for the different platforms (for example when the coordinates are established by a detection of a WM_MOUSEMOVE). Be sure the _wpi_cvt* routines are only used when the values will be in Windows coordinates. For example, drawing a rectangle according to the coordinates determined from a WM_MOUSEMOVE would not warrant a _wpi_cvt* call. For Windows, these macros do not alter the y values. The _wpi_cvt*_wanchor macros can be used to place a window on the desktop. The point passed in can be in Windows coordinates and the macro will convert the point to PM coordinates.

1.7 Presentation Spaces and Device Contexts

PM has the concept of a presentation space which can be used to display graphics. However, the presentation space does not replace the notion of a device context, because PM also has device contexts. Since Windows does not have a presentation space, a WPI_PRES has been introduced to allow the use of presentation spaces. For Windows, a WPI_PRES is simply an HDC and for PM it is an HPS (presentation space handle). The equivalent of creating a memory device context in Windows is to create a memory presentation space. In creating a compatible presentation space in PM, an HDC is required and must be deleted. WPI handles this. Throughout this document the term presentation space (or pres) is used and will refer to an HPS in PM and an HDC in Windows. Hence when phrases such as "drawing on a pres" arise, it is implied that drawing is happening on an HPS in PM and an HDC in Windows. Moreover, a compatible pres corresponds to a compatible HPS in PM and a compatible (memory) HDC in Windows. When HDC is refered to (unless specifically for Windows) it will imply both HDC for Windows and HDC for PM.

The following are some common macros used with presentation spaces and device contexts:

_wpi_getpres(hwnd)

This returns the presentation space associated with the window handle. For Windows this is simply GetDC. hwnd can be HWND_DESKTOP (which is the same as NULL).

_wpi_releasepres(hwnd, pres)

This releases the presentation space associated with the given window. In Windows this is a ReleaseDC.

_wpi_createcompatiblepres(pres, hab, &hdc)

For PM this macro creates and returns a presentation space that is compatible with the one given. It also creates a device context which is required when deleting the presentation space. For normal drawing, the hdc is not used. For Windows this simply performs a CreateCompatibleDC.

_wpi_deletecompatiblepres(mempres, hdc)

This function deletes the presentation space and for PM, the device context handle.

Creating a compatible presentation space handle may appear a little confusing. First, recall that the Windows steps to creating a compatible presentation space are as follows:

```
hdc = GetDC( hwnd );
 ...
memdc = CreateCompatibleDC( hdc );
        /*
         * Either draw on the memdc or the hdc
         */
 ...
ReleaseDC( hwnd, hdc );
 ...
DeleteDC( memdc );
 ...
```

When using WPI, a presentation space handle and a device context appear to be created. In Windows, the compatible DC is returned by the routine and hdc is simply set to NULL (ie. it is not used). When deleting the compatible pres in Windows, nothing happens to the hdc parameter since it has been set to NULLonly the compatible DC is deleted (stored in mempres). In PM, the device context handle is only used when deleting the presentation space handle. Under both platforms, the pres parameter passed into _wpi_createcompatiblepres must be valid and all drawing must take place on the returned memory presentation space handle, or pres. The equivalent to the above example would be:

```
pres = _wpi_getpres( hwnd );
 ...
mempres = _wpi_createcompatiblepres( pres, Instance, &hdc );
        /*
         * Either draw on pres or mempres
         */
 ...
_wpi_releasepres( hwnd, pres );
 ...
_wpi_deletecompatiblepres( mempres, hdc );
 ...
```

1.8 Graphics

In Windows, the graphics interface is known as the GDI (Graphics Device Interface). The equivalent in PM is the GPI (Graphical Programming Interface). The graphics interface includes objects such as pens, brushes, fonts, bitmaps, palettes and presentation space attributes.

In PM as in Windows, each presentation space has its own set of attributes. When a presentation space is created, it has a default set of attributes which are used unless they are specifically changed. Among these attributes are line, area, image, and character attributes. Each of these attribute types has a corresponding structure which contains fields to describe the type. PM provides APIs to change the values of the individual fields of the attributes. Also, there is a GpiSetAttrs API in which the attribute type (line, area, etc.) must be specified and can be set.

1.9 Colours

There is a slight difference between colours in Windows and PM. In Windows, colours are usually referenced by a COLORREF variable which contains an RGB value. Moreover, Windows APIs with a COLORREF parameter expect the variable to be an RGB value. This is not necessarily the case in PM. In PM, a presentation space can either be in index mode or RGB mode. If a presentation space is in RGB mode, all references are expected to be RGB values. However, this is NOT the default mode. The default

(index mode) implies that all references to colours are indices into a colour table associated with the presentation space. When in index mode, all API references to colours are expected to contain indices and when in RGB mode, all APIs expect RGB values. Attempting to use RGB values in index mode will produce unpredictable results. Furthermore, PM will interpret index values as RGB values when in RGB mode.

Since the user is converting from Windows to PM, their Windows code will contain RGB colour values. Since RGB is not the default mode, the user will need to switch to RGB mode whenever RGB colour values are used in a newly created presentation space. The WPI macro _wpi_torgbmode(pres) will set the given presentation space to RGB mode. This macro does nothing in Windows. A very easy mistake to make when converting code is to forget to set the presentation space to RGB mode.

WPI defines COLORREF to be ULONG for PM which is how colours are stored in that environment. Hence, the user can leave Windows COLORREF variables as COLORREF variables (alternatively, they can declare colour variables as WPI_COLOUR) in their converted code. Moreover, the RGB macro used in Windows to create an RGB value is also available with WPI in PM. Occurrences like: RGB(red, green, blue) need not change.

1.10 Pens and Brushes

Windows includes the data types HPEN and HBRUSH which PM does not. However, through WPI, HPEN and HBRUSH can be used in PM. In order to create and use an HPEN or HBRUSH, WPI allocates memory for a structure which describes its attributes. When selecting an HPEN or HBRUSH into a presentation space, space must be allocated to store the old pen or brush and the attributes of the pen or brush being selected are set for the current pres. Upon deleting an HPEN or HBRUSH, WPI frees the memory it has allocated. When a WPI routine returns (or accepts as a parameter) an HPEN or HBRUSH, it is really returning an address to the object structure (this address has been defined as a WPI_HANDLE).

Creating a pen using WPI is the same as creating a pen in Windows. Certain attributes must be set such as the pen type (solid or dashed), the pen thickness and the pen colour. The _wpi_createpen macro accepts the same parameters as Windows' CreatePen and returns the newly created pen. The pen type should be specified using the Windows pen type definitions (these are of the form PS_* and some types may not yet be converted in WPI). After allocating memory for the object structure, the PM routine sets the field values of the structure as indicated by the parameters to the function (the structure corresponding to an HPEN in PM is a LINEBUNDLE; however, this should be invisible to the user).

Similarly, brushes are created as they are in Windows. The _wpi_createsolidbrush, for example accepts the colour of the solid brush and returns the brush being created. Again, the PM version of the function allocates memory for the object structure and then sets the fields of the structure as indicated by the parameters to the function (the structure corresponding to an HBRUSH in PM is an AREABUNDLE; again, this is invisible to the user).

Recall that both HPEN and HBRUSH are addresses of a object structures in PM. Hence when a pen or brush is created, WPI allocates space for the structure before setting the fields of the structure and returning a pointer to the structure. Naturally, when deleting an object WPI frees the memory associated with the object. Moreover, when selecting an object into a presentation space, the normal Windows behaviour is to return the old object. So WPI needs to allocate space for the old object and must know when to free that memory. To accomplish this, WPI uses alloca which allocates enough space for the old object and the automatically frees the memory when the routine selecting the object is exited. By employing this method, most Windows code is convertable by simply using the _wpi_selectobject routine. The drawback to this method is that the old object returned from _wpi_selectobject cannot be a global variable. To accommodate this problem, WPI also has the following routines:

_wpi_selectpen(pres, hpen)

This routine selects the pen associated with hpen into the presentation space. Space for the old pen is allocated and returned from the function. The old pen can be global if necessary.

_wpi_getoldpen(pres, holdpen)

This routine sets the presentation space attributes for the old pen and frees the memory associated with the old pen.

_wpi_selectbrush(pres, hbrush)

This routine is the same as _wpi_selectpen except for brushes.

_wpi_getoldbrush(pres, holdbrush)

This routine sets the presentation space attributes for the old brush and frees the memory associeated with the old brush.

The difference between _wpi_selectobject and _wpi_selectpen cannot be over-emphasized! The points to consider when using _wpi_selectobject are:

it is more generic (the same function selects pens, brushes, bitmaps, old pens, old brushes etc...) and looks more like windows code

the following code is possible because all old objects will be freed when myproc is exited:

```
void myproc( void ) {
        oldpen = _wpi_selectobject( hpres, hpen1 );
        ...
        _wpi_selectobject( hpres, hpen2 );
        ...
        _wpi_selectobject( hpres, oldpen );
}
```

if the old object is a global variable, set in one routine and selected back into the pres in another routine, _wpi_selectobject cannot be used because alloca will free the memory when exiting the routine in which the old object was created.

The points to consider when using _wpi_selectpen or _wpi_selectbrush are:

the old object returned from the select can be used as a global variable because the select does a normal allocate (using _wpi_malloc) and the memory will not be freed until a _wpi_getold*

each _wpi_selectpen (or _wpi_selectbrush) must have a corresponding _wpi_getoldpen before the next _wpi_selectpen for that presentation space handle; so the following code is not correct because memory for the second _wpi_selectpen will never be freed:

```
void myproc( void ) {
        oldpen = _wpi_selectpen( hpres, hpen1 );
        ...
        _wpi_selectpen( hpres, hpen2 );
        ...
        _wpi_getoldpen( hpres, oldpen );
}
```

One final note about selecting pens and brushes: the routines are not interchangable. So if a pen has been selected with _wpi_selectobject the old pen should be selected with _wpi_selectobject, not with _wpi_getoldpen! Similarly, selecting a pen with _wpi_selectpen necessitates _wpi_getoldpen to select the old pen back into the presentation space.

Finally, deleting a pen or brush is similar to deleting the object in Windows. For PM, the WPI macro frees the space associated with the pointer to the structure. The following WPI routines handle creating and deleting pens and brushes:

_wpi_createpen(type, width, colour)

In Windows, this creates a pen with the specified pen style (PS_*), width and colour. In PM this allocates space for the object structure and sets the type, width and foreground colour for the pen. The pen is returned by the function.

_wpi_createnullpen( )

Creates and returns a NULL pen (ie. invisible pen). Windows version simply gets the NULL stock pen.

_wpi_createnullbrush( )

Creates and returns a NULL brush. Windows version simply gets the NULL stock brush.

_wpi_createsolidbrush(colour)

In Windows this returns a solid brush with colour colour. In PM this allocates space for the object structure and sets the foreground colour for the brush.

_wpi_createpatternbrush(bitmap)

Returns a pattern brush using bitmap as the pattern.

_wpi_deletepen(pen)

In Windows this deletes the pen object. In PM this frees the memory associated with pen.

_wpi_deletebrush(brush)

In Windows this deletes the brush object. In PM this frees the memory associated with brush.

_wpi_deletenullpen(pen)

For PM this deletes the NULL pen. This does nothing in Windows.

_wpi_deletenullbrush(brush)

For PM this deletes the NULL brush. This does nothing in Windows.

_wpi_selectobject(pres, hobject)

This routine selects a pen, brush or bitmap into the presentation space. The PM version performs an alloca to allocate space for the old object which is returned.

There may be more functions than are presented here. If the desired routine does not appear in this list, search for it in wpi_os2.h and if it does not exist, add it. Here is a typical example of selecting pens and brushes into a presentation space.

Windows code:

```
pen1 = CreatePen( PS_SOLID, 0, BLACK );
pen2 = CreatePen( PS_SOLID, 0, WHITE );
brush1 = CreateSolidBrush( RED );
 ...
oldpen = SelectObject( hdc, pen1 );
oldbrush = SelectObject( hdc, brush1 );
 ...
SelectObject( hdc, pen2 );
 ...
SelectObject( hdc, oldpen );
SelectObject( hdc, oldbrush );
DeleteObject( pen1 );
DeleteObject( pen2 );
DeleteObject( brush1 );
```

WPI code:

```
pen1 = _wpi_createpen( PS_SOLID, 0, BLACK );
pen2 = _wpi_createpen( PS_SOLID, 0, WHITE );
brush1 = _wpi_createsolidbrush( RED );
 ...
oldpen = _wpi_selectobject( pres, pen1 );
oldbrush = _wpi_selectobject( pres, brush1 );
 ...
_wpi_selectobject( pres, pen2 );
 ...
_wpi_selectobject( pres, oldpen );
_wpi_selectobject( pres, oldbrush );
_wpi_deletepen( pen1 );
_wpi_deletepen( pen2 );
_wpi_deletebrush( brush1 );
```

If the old objects are global:

```
pen1 = _wpi_createpen( PS_SOLID, 0, BLACK );
pen2 = _wpi_createpen( PS_SOLID, 0, WHITE );
brush1 = _wpi_createsolidbrush( RED );
 ...
Oldpen = _wpi_selectpen( pres, pen1 );
Oldbrush = _wpi_selectbrush( pres, brush1 );
 ...
_wpi_getoldpen( pres, Oldpen );
Oldpen = _wpi_selectpen( pres, pen2 );
 ...
_wpi_getoldpen( pres, Oldpen );
_wpi_getoldbrush( pres, Oldbrush );
_wpi_deletepen( pen1 );
_wpi_deletepen( pen2 );
_wpi_deletebrush( brush1 );
```

Notice that with a few exceptions, the code looks very similar to that of Windows. In general, pens and brushes can be used the same way they are in Windows.

1.11 Bitmaps

The use of bitmaps is quite similar between Windows and PM. Although both platforms have an HBITMAP data type, WPI stores bitmaps as object structures similar to HPENs and HBRUSHes. Hence when a bitmap is created and a handle returned, the returned value should not be used for pure PM code, but only in WPI code. For example, the return value from _wpi_createcompatiblebitmap is an address to a structure describing the bitmap (this address is defined as a WPI_HANDLE). This return value (the WPI_HANDLE) can be passed to WPI functions expecting a bitmap handle, but the return value should not be used in PM specific code (because the return value is a WPI_HANDLE and not an HBITMAP). The bitmap data types declared in WPI are as follows:

```
WPI Data Type   Windows Data Type   PM Data Type
-------------   -----------------   ------------
WPI_HBITMAP     HBITMAP             WPI_HANDLE
```

Like HPEN and HBRUSH, bitmaps can be selected into memory presentation spaces with the _wpi_selectobject routine and the old bitmap will be allocated and returned from the routine. For example:

```
oldbitmap = _wpi_selectobject( mempres, hbitmap );
 ...
_wpi_selectobject( mempres, oldbitmap );
```

Again, if the old bitmap is used outside the routine selecting the bitmap into the space, _wpi_selectbitmap and _wpi_getoldbitmap can be used. Note once again that each _wpi_selectbitmap must have a corresponding _wpi_getoldbitmap before another bitmap can be selected into the presentation space. So if the old bitmap was global the code would look like:

```
Oldbitmap = _wpi_selectbitmap( mempres, hbitmap );
 ...
_wpi_getoldbitmap( mempres, Oldbitmap );
```

The following WPI routines handle bitmaps:

_wpi_createcompatiblebitmap(pres, width, height)

This routine returns a bitmap compatible with the given presentation space (HDC for Windows) and having the specified dimensions.

_wpi_createbitmap(width, height, planes, bitcount, &bits)

This routine returns a bitmap with the attributes given in the parameter list. Like Windows, if bits is NULL then the bitmap is left uninitialized.

_wpi_deletebitmap(bmp)

Deletes the given bitmap. The bitmap must have been created by a WPI function.

_wpi_getbitmapbits(hbitmap, size, &bits)

Performs the same action as the Windows GetBitmapBits. The bitmap must be created by a WPI routine.

_wpi_setbitmapbits(hbitmap, size, &bits)

Performs the same action as the Windows SetBitmapBits. The bitmap must be created by a WPI routine.

_wpi_selectobject(hpres, hobj)

This will select the bitmap (or pen or brush) into the presentation space. For PM, the old bitmap gets space allocated for it and is returned. It will be freed when the routine is exited.

_wpi_selectbitmap(hpres, hbitmap)

This will select the bitmap into the presentation space. In PM, space is allocated for the old bitmap handle (which is return) and will not be freed until a call to _wpi_getoldbitmap.

_wpi_getoldbitmap(hpres, holdbitmap)

This will select the old bitmap into the presentation space. For PM this frees the memory associated with the old bitmap.

This again, is merely a subset of the bitmap functions available in WPI. Many other Windows APIs have been converted to WPI. If the desired function is not present in this list then search wpi_os2.h.

Mention should be made here of how monochrome bitmaps are converted to colour bitmaps. The user should read the section on BitBlt (Windows) and GpiBitBlt (PM) to find out how the conversion is done for each platform. Unfortunately Windows and PM perform in the opposite manner. Windows converts all white pixels (1's) in the monochrome bitmap to the destination DC background colour and all black pixels (0's) in the monochrome bitmap to the destination DC foreground colour. PM converts all white pixels to the destination presentation space foreground colour and black pixels to the background colour. Hence in WPI, a macro called _wpi_preparemono can be used to prepare the destination background and foreground colours to allow the user to copy a monochrome bitmap to a colour bitmap. The macro is used as follows:

```
_wpi_preparemono( destpres, black_pixel_colour, white_pixel_colour
);
```

This will allow the user to set the presentation space to its proper colours for both platforms. The Windows default DC attributes should be such that a monochrome bitmap copied to a colour bitmap will in fact be a colour bitmap appearing the same as the monochrome bitmap. These default settings cannot be assumed for PM. Hence this macro should be used whenever copying from a monochrome to a colour bitmap.

A few new types have been declared in wpitypes.h that relate to bitmaps. They are as follows and can be used to pass to some of the macros that require different types for Windows and PM:

WPI Data Type	Windows Data Type	PM Data Type
WPI_BITMAP	BITMAP	PM1632_BITMAPINFOHEADER2
WPI_BITMAPINFO	BITMAPINFO	PM1632_BITMAPINFO2
WPI_BITMAPINFOHEADER	BITMAPINFOHEADER	PM1632_BITMAPINFOHEADER2
WPI_BMPBITS	LPSTR	PBYTE

Since the bitmap structures for 16 bit PM are different than that for 32 bit PM, a file called pm1632.h handles the differences (pm1632.h will be discussed later). There are also functions available to retrieve bitmap information:

_wpi_getbitmapparms(hbitmap, &cx, &cy, &planes, &bitcount, &bitspixel)

Takes the bitmap handle and returns the information about the bitmap. NULL can be passed to any parameters not desired. The bitspixel parameter will always be set to 0 in PM because that information is not available under PM.

_wpi_getbitmapstruct(hbitmap, &bitmap_info)

This routine fills the bitmap_info structure (should be type WPI_BITMAP) according to the attributes of the given bitmap handle. This routine may be usefull when a structure is needed to pass to another WPI routine.

1.12 Using WPI to Draw

Once pens, brushes and bitmaps have been selected into their proper presentation spaces, drawing can begin. Many of the Windows drawing functions have been converted in WPI to provide functionality for PM. While some of these routine are quite similar to the Windows versions, others differ dramatically. The following is a list of some of these routines:

_wpi_moveto(pres, &pt)

Moves to the point on pres indicated by pt.

_wpi_lineto(pres, &pt)

Draws a line from the current position to the indicated point.

_wpi_setpixel(pres, x, y, colour)

Performs the same as the Windows SetPixel routine.

_wpi_getpixel(pres, x, y)

Performs the same as the Windows GetPixel.

_wpi_rectangle(pres, left, top, right, bottom)

Draws a rectangle. Note that top is both top in Windows and PM (ie not bottom in PM). Hence _wpi_cvt* macros may prove useful before using this macro.

_wpi_ellipse(pres, left, top, right, bottom)

Draws an ellipse inside the box implied by the dimensions. Note again that top < bottom for Windows and top > bottom for PM is assumed.

_wpi_arc(pres, x1, y1, x2, y2, x3, y3, x4, y4)

Draws an arc (as it does in Windows) defined by the given points.

_wpi_bitblt(dest, x1, y1, cx, cy, src, x2, y2, rop)

Identical to Windows BitBlt. PM version assumes (x1, y1) is actually the bottom left corner of the area and uses Windows predefined ROP codes.

_wpi_patblt(dest, x1, y1, cx, cy, rop)

Identical to Windows PatBlt. Same comments as _wpi_bitblt.

_wpi_stretchblt(dest, x1, y1, cx1, cy1, src, x2, y2, cx2, cy2, rop)

Identical to Windows StretchBlt. Same comments as _wpi_bitblt.

Note the difference in the first two routines listed. They take a WPI_POINT instead of x and y values. It should also be emphasized that the rectangle and ellipse macros for PM expect top > bottom. This is important because of the way PM draws these images. Windows draws up to but not including the right and bottom coordinates. PM actually includes the right and bottom coordinates. WPI attempts to handle this difference and in so doing, requires that top actually be the top. Note the difference between the Windows code and converted code in this example:

Windows Code:

```
Rectangle( hdc, 10, 0, 100, 50 );
```

WPI Code:

```
top = 0;
bottom = 50;
// Assume height has been set to the height of pres
top = _wpi_cvth_y( 0, height );
bottom = _wpi_cvth_y( 50, height );
_wpi_rectangle( pres, 10, top, 100, bottom );
```

The user will be required to add the conversion of the height (depending on the circumstances) in order to assure that top > bottom. Currently, ellipses are not drawn very accurately in PM. This is due to the way in which PM draws ellipses. The most trouble occurs when the dimensions of the bounding rectangle of the ellipse contain even values. Work is underway to rectify this problem.

Since the blt functions take only an origin and then the width and height, the origin is assumed to be in PM format for PM and Windows format for Windows. This is particularly useful in copying bitmaps since the origin (0, 0) can be used under both platforms.

1.13 Palettes

Currently, WPI does not support many palette operations. The macro _wpi_selectpalette(pres, hpal) will select a palette into the given presentation space. Moreover, the data type HPALETTE in PM is defined to be HPAL. If the code being converted contains references to palettes, expect to make additions to WPI.

1.14 Fonts

Fonts can be complicated objects to deal with in PM. PM does not have an HFONT data type nor a LOGFONT data type. The font data types declared in WPI are as follows:

WPI Data Type	Windows Data Type	PM Data Type
HFONT	HFONT	LONG
WPI_FONT	HFONTF	ATTRS*
WPI_LOGFONT	LOGFONT	FONTMETRICS
WPI_TEXTMETRIC	TEXTMETRIC	FONTMETRICS

In Windows, one can create a font and once a font handle is acquired, use that font handle as a parameter (or any other kind of variable) to be selected into a device context for text output. However, creating a font in PM requires a presentation space and once that presentation space is released, the created font is gone. So, a division arises between the two cases. When a font is desired and a presentation space exists, the following macros can be used:

_wpi_createfont(pres, &wlfont, &hfont)

For Windows this is the same as CreateFontIndirect with wlfont as the logfont. The font handle is returned in hfont. For PM this creates a font from the specifed WPI_LOGFONT and returns the font handle (the value: 1) in hfont. The font can only be used in the given presentation space.

_wpi_getdeffm(wlfont)

Sets the WPI_LOGFONT to its default values.

_wpi_deletefont(hfont)

Deletes the font for Windows and resets the font identifier for PM.

_wpi_getsystemfont()

Returns the system font for Windows and the default font identifier for PM.

_wpi_selectfont(pres, hfont, &oldfont)

Selects the given font into the given presentation space.

_wpi_getoldfont(pres, oldfont)

Selects the old font into the given presentation space.

In addition to these macros, there are macros to set the font to italics, bold, strikeout, and more. Moreover, macros exist to set the font height, width and pointsize. Due to the number of macros available, they are not all listed here. To find the proper macro, search for _wpi_setfont in wpi_os2.h until the required macro has been found.

All of the above macros work with the HFONT data type. They all assume that text output will occur with the presentation space used in the macro calls. However, there may be situations in which the user wishes to create a font and use it with a presentation space created at a later time. This is the purpose of the WPI_FONT data type. The following macros handle WPI_FONTs:

_wpi_createwpifont(&wlfont, wfont)

Creates a WPI_FONT. For Windows, this is the same as a normal font creation.

_wpi_selectwpifont(pres, wfont)

Windows version selects the font into the DC. PM allocates memory for the old font and sets the font of the given pres to be that of wfont.

_wpi_getoldwpifont(pres, oldfont)

Restores the font of pres to be the old font and frees the old font memory.

_wpi_deletewpifont(wfont)

Deletes the font associated with wfont.

A WPI_FONT for PM is actually a pointer. The creation macro allocates space for the structure on the PM platform and the deletion frees the space. As with pens and brushes, selecting the font into the presentation space allocates space for the old font structure so it is essential that the old font be selected back into the presentation space to free the memory associated with it. A quick look at _wpi_selectwpifont for PM will show that the font is actually being created in this routine. This is because the font is always tied to the presentation space. Once again, before calling the creation macro, the user may use the _wpi_setfont* macros to set the attributes of the desired font.

1.15 Window Functions

There are some similarities between Windows window procedures and PM window procedures and hence converting these is relatively straightforward. The format and main philosophy of windows procedures is the same on both platforms and much is accomplished by simply defining a new set of types or function names.

1.16 Window APIs

Since many of the windows related functions differ only in name between the two platforms, WPI allows the user to use the Windows name when calling the API. The following is a list of some of the window related macros and the names available to the user:

```
DefWindowProc(hwnd, msg, wp, lp)     _wpi_defwindowproc(hwnd, msg,
wp, lp)
```

Takes default window procedure action.

```
ShowWindow(hwnd, state)              _wpi_showwindow(hwnd, state)
```

Shows the window according the given state. Windows predefined states are used.

```
GetClientRect(hwnd, &wrect)          _wpi_getclientrect(hwnd, &wrect)
```

Gets the rectangle dimensions of the given window. The function fills the WPI_RECT variable.

```
DestroyWindow(hwnd)                  _wpi_destroywindow(hwnd)
```

Destroys the given window.

```
SetWindowText(hwnd, str)              _wpi_setwindowtext(hwnd, str)
```

Sets the caption of the given window.

```
MessageBox(hpar, txt, title, style) _wpi_messagebox(hpar, txt,
title, style)
```

Displays a message box as it does in Windows. Most of the MB_* styles have been converted, using the Windows naming convention.

```
GetMenu(hwnd)                         _wpi_getmenu(hwnd)
```

Gets the menu of the window. For PM this window handle must be a frame window.

The user may decide whether or not to keep the Windows names or use the available WPI names.

1.17 Window Creation

Window creation is somewhat different in PM than Windows. The user may wish to use separate PM code from their Windows code in order to gain clarity and the flexibility when creating windows. In WPI however, an attempt has been made to convert the window creation routines. The two main macros are _wpi_registerclass and _wpi_createwindow. The _wpi_createwindow routine may prove useful for creating a standard window style, however it is limiting in some areas including the fact that it will not allow initialization data to be passed in the WM_CREATE message. The user should examine this routine to verify its usefulness for the situation at hand.

1.18 Window Procedures

Window procedures are also similar in both Windows and PM. Under the WPI scheme, windows functions should be declared as follows:

```
MRESULT CALLBACK WindowsProc( HWND hwnd,
                              WPI_MSG msg,
                              WPI_PARAM1 wparam,
                              WPI_PARAM2 lparam );
```

Most of the messages are similar in name and where they differ, the Windows convention has been used. The user should note that messages such as WM_MOUSEMOVE that store coordinates will store the coordinates according to the platform under which they are running (ie Windows will return the mouse coordinates with the top left as the origin and PM with the bottom left as the origin).

When creating windows, PM does not specify the background colour of the window. This suggests that no default painting goes on when a PM window procedure receives a WM_PAINT. The user may be required to add the following to their WM_PAINT messages:

```
case WM_PAINT:
        _wpi_beginpaint( hps, NULL, &wrect );
        ...
#ifdef __OS2_PM__
        WinFillRect( hps, wrect, background_clr );
#endif
        ...
        _wpi_endpaint( hwnd, pres, &wrect );
```

1.19 Resources

Open Watcom has its own resource compiler, so Open Watcom does not require the use of the OS2 toolkit resource compiler. RC files in PM look similar to those of Windows. The user should look at the PM help files or an available example in order to convert their RC files.

1.20 Dialogs

Dialogs, like other windows have some definite similarities between Windows and PM. Because of these similarities, a program has been created to transform Microsoft Dialog Editor DLG files to PM DLG files. This program is called parsedlg.exe and can be found in r:\cmds. To use this program simply type:

parsedlg infile.dlg outfile.dlg

The user will need to compile their PM application with the newly parsed DLG file. Note that this new dialog should look the same as the Windows dialog. The user does not need to convert any coordinates. One should also note that in Windows a dialog can either be referenced by either a unique string name or a unique integer (this gets passed into the DialogBox API). In PM however, a dialog box can be referenced only by a unique integer. If a string is currently used, the user will need to change this to an integer.

Once the dialog file is parsed, the user can convert the code generating the dialog procedure. Initializing a dialog box procedure would look like the following under WPI:

```
fp = _wpi_makedlgprocinstance( DlgProc, Instance );
ret_val = _wpi_dialogbox( hparent, fp, Instance, DLG_ID, 0L );
_wpi_freedlgprocinstance( fp );
```

The dialog procedure itself looks much like a Windows dialog procedure excluding the WPI conversion names:

```
DLG_RESULT CALLBACK DlgProc( HWND hwnd,
                             WPI_MSG msg,
                             WPI_PARAM1 wparam,
                             WPI_PARAM2 lparam );
```

The user may continue to use the constants IDOK and IDCANCEL since they are defined in WPI to be consistent with the constants used by the parsedlg.exe program. Unique to PM is the use of a default dialog procedure. Where Windows usually returns FALSE, PM returns WinDefDlgProc with the same parameters as the procedure from which it is returned. Hence the user will want to return _wpi_defdlgproc(hwnd, msg, wparam, lparam) for unprocessed dialog messages. This will simply return FALSE for Windows.

The following is an example of a dialog procedure for which there are a number of tricky points between Windows and PM. This is done to illustrate some of the differences between Windows and PM as well as where to spot trouble.

```
WPI_DLGRESULT CALLBACK dlg_sample( HWND dlg_hld,
                                   WPI_MSG msg,
                                   WPI_PARAM1 parm1,
                                   WPI_PARAM2 parm2 )
{
    int                 initial_col;
    int                 tmp;

    if( msg == WM_INITDIALOG ) {

        /* PM expects a pointer to the data    */
        /* while Windows just expects the data */

#ifdef __OS2_PM__
        {
        int                     *ptr;
        ptr = (int *) parm2;
        initial_col = *ptr;
        }
#else
        initial_col = parm2;
#endif
        return( (WPI_DLGRESULT) TRUE );


        /* The following line is necessary because for PM */
        /* because some messages which are caught under   */
        /* WM_COMMAND in PM are not in Windows and vice   */
        /* versa.  Hence, use this routine to trap the    */
        /* WM_COMMAND message.                                          */

    } else if( _wpi_dlg_command( dlg_hld, &msg, &parm1, &parm2 ) ) {

        if( _wpi_getid( parm1 ) == IDOK ) {
            _wpi_enddialog( dlg_hld, TRUE );
            return( (WPI_DLGRESULT) TRUE );

        } else if( _wpi_getid( parm1 ) == IDCANCEL ) {
            _wpi_enddialog( dlg_hld, FALSE );
            return( (WPI_DLGRESULT) TRUE );

        } else if( _wpi_isbuttoncode( parm1, parm2, LBN_SELCHANGE )
&&
                            _wpi_getid( parm1 ) == DLG_SAMPLE_BUTTON
) {

            /* Many messages in PM pack the information   */
            /* in different places than in Windows        */
            /* hence the need for _wpi_getid, and new     */
            /* parameters for the following messages      */
            /* cf. _wpi_getdlgitemlbtext etc. as well     */

            tmp = (int) _wpi_senddlgitemmessage( dlg_hld,
                    DLG_GPCMN_GROUP, LB_GETCURSEL, LIT_FIRST, NULL
);

            _wpi_senddlgitemmessage( dlg_hld, DLG_GPCMN_GROUP,
                        LB_SETCURSEL, initial_col, LIT_SELECT );
```

```
                initial_col = tmp;
            }

        } else {
            return( _wpi_defdlgproc( dlg_hld, msg, parm1, parm2 ) );
        }

        return( (WPI_DLGRESULT) FALSE );
    }

        .
        .
        .

        /* PM expects a pointer to the data    */
        /* while Windows just expects the data */

    #ifdef __OS2_PM__
            dlg_ret = _wpi_dialogbox( frame_win_hld, proc, Inst, tpl,
    &id );
    #else
            dlg_ret = _wpi_dialogbox( frame_win_hld, proc, Inst, tpl, id
    );
    #endif
        .
        .
        .
```

1.21 Owner-Drawn Controls

This section covers only buttons and listboxes since menus have not been researched or tested. There is some fully functioning code in the datactl library (cf. Dan Pronovost).

To begin with, owner-drawn buttons get drawn by responding to a WM_CONTROL (WM_DLGCOMMAND) message while everything else should respond to a WM_DRAWITEM message. The following section of code illustrates how to determine what type to draw:

```
 ...
    _wpi_getcursorpos( &q_pt );
    _wpi_setanchorblock( hwnd, inst );
    time = _wpi_getcurrenttime( inst );
    _wpi_setqmsgvalues( &qmsg, hwnd, msg, parm1, parm2, time, q_pt
);

    /*
     * make absolutely sure that any buttons handle the
WM_QUERYDLGCODE
     * message and return DLGC_BUTTON
     */
    is_button = ((int)_wpi_sendmessage(hwnd, WM_QUERYDLGCODE, &qmsg,
0L)

& DLGC_BUTTON);
    is_button = is_button &&
        ( SHORT2FROMMP( parm1 ) == BN_PAINT ) && ( msg ==
WM_DLGCOMMAND );

    if( is_button ) {
                .
                .
        < draw the button >
                .
                .
    } else if( msg == WM_DRAWITEM ) {
                .
                .
        < draw the listbox >
                .
                .
    }
                .
                .
```

Buttons are sent a USERBUTTON structure in the second parameter. The following section of code illustrates how to obtain information from this structure:

```
        .
        .
USERBUTTON              *b2;

b2 = (USERBUTTON *) parm2;
win_hld = b2->hwnd;
pres = b2->hps;
id = SHORT1FROMMP( parm1 );
tmp_word = LOWORD( b2->fsState );
disabled = ( tmp_word == BDS_DISABLED );
selected = ( tmp_word == BDS_HILITED );
tmp_word = LOWORD( b2->fsStateOld );
old_selected = ( tmp_word == BDS_HILITED );
select_changed = ( selected == old_selected );
has_focus = ( GetFocus() == win_hld );
_wpi_getclientrect( win_hld, &rect );
        .
        .
< draw the button based on above information >
        .
        .
// tells the system you did the highlighting
b2->fsStateOld = b2->fsState;

// tells the system you drew the item
to_ret = (WPI_DLGRESULT) TRUE;
        .
        .
```

Listboxes are sent an OWNERITEM structure in the second parameter. The following section of code illustrates how to obtain information from this structure:

```
        .
        .
poi = (OWNERITEM *) parm2;
win_hld = poi->hwnd;
pres = poi->hps;
id = poi->idItem;
rect = poi->rclItem;
disabled = !WinIsControlEnabled( win_hld, id );
selected = poi->fsState;
focus_changed = !( poi->fsState == poi->fsStateOld );
select_changed = !focus_changed;
        .
        .
< draw the listbox based on above information >
        .
        .

// tells the system you did the highlighting
poi->fsState = poi->fsStateOld = 0;

// tells the system you drew the item
to_ret = (WPI_DLGRESULT) TRUE;
        .
        .
```

1.22 Menus

Menus are handled in the WM_DRAWITEM message, similar to listboxes, and also use the OWNERITEM structure but take advantage of the attribute and old attribute fields (MIA_). The following routines handle menu operations:

_wpi_getmenu(hwnd)

This returns the handle of the menu. Note that for PM, hwnd must be a frame window handle.

_wpi_getcurrentsysmenu(hwnd)

Returns the current system menu for the given frame window handle.

_wpi_checkmenuitem(hmenu, id, fchecked, fby_pos)

This checks a menu item identified by id. The fchecked variable should be either TRUE or FALSE (whether the item should be checked or unchecked) and by_pos indicates whether the id indicates the position (TRUE) or the actual identifier.

_wpi_enablemenuitem(hmenu, id, fenabled, fby_pos)

Similar to _wpi_checkmenuitem except it indicates whether the item should be enabled (TRUE) or grayed (FALSE).

Once again, this is only a subset of the available menu functions.

1.23 Other Platform Considerations

There is a good possibility that the Windows program being converted to PM has already been converted to 32 or 64 bit Windows (or will be converted to 32 or 64 bit Windows). Furthermore, there is the possibility that a PM application is desired for both 16 and 32 bit platforms. These can produce somewhat precarious situations; however, there should not be too many problems if the user is careful. To begin with, wpitypes.h includes wi163264.h if compiling for Windows and pm1632.h if compiling for PM. These header files contain macros which translate between 16 and 32 and 64 bit Windows; and 16 and 32 bit PM.

Where possible, if adding to WPI the user should be certain that the Windows version will be compatible with wi163264.h and the PM version with pm1632.h. This means for example, that it may be appropriate for a WPI macro on the Windows side to refer to another macro defined in wi163264.h.

1.24 Adding To WPI

A few considerations should be made when modifying WPI.

Note first of all that all references to malloc should use the _wpi_malloc symbol since the user may decide to define their own memory allocating routine.

Remember that if a routine is changing there could be several parties affected by the change.

When adding routines, be sure to consider how the code may affect 32 or 64 bit Windows or may be different for 16 and 32 bit PM. Be sure to check wi163264.h and pm1632.h to make sure one of the contained macros is not needed. It is best after adding a routine to WPI to create the library for all of 16 and 32 and 64 bit Windows, and 16 bit PM, and 32 bit PM, even if you are not using all levels.

One should be certain when adding macros to WPI that no size values are hard coded. Whenever possible the sizeof operator should be used since structures may be different sizes in 16 and 32 and 64 bit Windows; and 16 and 32 bit PM.

When using macros be generous in the use of parentheses. Bracket pointers and structures in case expressions are passed to the macros. In general, if the macro requires more than 1 line of code or if the routine needs to return a value, add the routine to the library.