Implementing Windows Send/Post/ PeekMessage on a Macintosh

A cross platform chore without any fun at all

Michael Rutman <mailto:/ /moose@manicmoose.com>

What is Send/Post/ PeekMessage?

On the Mac, the programmer writes a single event-handling loop that dispatches events to various parts of the program, such as menu handlers, keyboard handlers, and window handlers. In Windows, all event handling is centered around the window. Each event, called a message in Windows terminology, goes through the message handler. Window programmers, create an event handler loop for every window. All events, called messages under Windows, go through the message handler for a window, these include timer messages as well as paint messages. PeekMessage is used to grab the next message waiting for any particular window.

Any window can handle messages. In addition, any message can be sent to any window, in either the same or a different application. Each window is uniquly identified, therefore, the underlying OS can tell if a message is being sent to the same application, or a different application. All messages are sent with either PostMessage or SendMessage. PostMessage just puts the message in the queue, SendMessage posts the message, then blocks the windows thread waiting for a response.

One ramification, even if you don't have a window visible, you still need to have an invisible window to receive timer and quit messages.

Why would I want to use this scheme on a Macintosh?

In our project, we had an existing windows application we were porting to the Macintosh. The choice was to either rewrite the entire project, or to implement the Windows message dispatch scheme on the Macintosh.

It is hard to justify replacing the standard Macintosh inter application communications with a Windows mechanism, especially when our implementation used AppleEvents as the underlying protocol. However, once the mechanism was written, it did show a layer of simplicity that I have never seen with raw AppleEvents.

Sending a message to any window only requires knowing one value, the window ID. If the window ID is known, then any application can send a message to any window anywhere on the system.

Creating a window

Creating a window does not mean creating a Macintosh window. The window created is only used as a communication channel, not for displaying anything. One of the annoying aspects of Windows programming is the bluring of terminology. A window under Windows may mean a window on the screen, or it could just be a loop waiting for a timer event.

Each window belongs to a class of windows. Registering a class is as simple as

filling in the fields of a structure and passing it to RegisterClass. Windows uses an Application instance value, instead we used the Macintosh's process ID's to specify which application owned which window.

Each class also needs a unique class name. In this case, we created an instance variable, and set it to a constant string with the process ID added on to the end. Later, when the class needs to be unregistered, we still have the string stored. As long as the string is unique, it can be anything at all. Using the process ID will insure that each process will be able to register a window.

In addition, we extended the WNDCLASS structure to allow registering a Macintosh Event Handling interface and some user data.

Messages will get dispatched to the method specified in the lpfnWndProc field. Each class has one handler, but the handler can tell which window received the message.

```
HINSTANCE hInstance = GetCurrentTask();
WNDCLASSwndclass;

MakeClassString( mWindowClass, kClassName
);
wndclass.style         = 0;
wndclass.lpfnWndProc   =
(WNDPROC)thisMessageHandler;
wndclass.cbClsExtra    = 0;
wndclass.cbWndExtra    = 0;
wndclass.hInstance     = hInstance;
wndclass.hIcon         = NULL;
wndclass.hCursor       = NULL;
wndclass.hbrBackground = NULL;
wndclass.lpszMenuName  = NULL;
wndclass.lpszClassName =
(LPSTR)mWindowClass;
wndclass.eventCallback = HandleEvent;
wndclass.eventUserData = this;

RegisterClass(&wndclass);
```

Once a class is registered, then a window can be created with the following call:

```
mWindowsWindow = CreateWindow(
  (LPCSTR)mWindowClass,
  (LPCSTR)"Client window", /* title */
  WS_OVERLAPPEDWINDOW | WS_VSCROLL,
    /* type */
  15, 15,  /* x,y */
  40, 100,/* w,h */
  (HWND)NULL, (HMENU)NULL,
  hInstance, NULL);
```

Later, code for implementing these routines will be discussed.

Window message Dispatching

Despite the window class knowing the routine for dispatching a message, there still has to be a main message handler, although, it is a simple routine.

```
if ( PeekMessage ( &msg, mWindowID, 0, 0,
  PM_REMOVE ))
  {
  TranslateMessage( &msg );
  DispatchMessage( &msg );
  }
```

PeekMessage is a combination of PeekEvent and WaitNextEvent. With the PM_REMOVE flag, a message is copied and removed from the queue. Without the PM_REMOVE flag, a message is only copied, not removed.

TranslateMessage is used to convert key values in key messages to the local system. It handles dead keys and international issues. The documentation makes it clear that you should always call it before dispatching a message.

DispatchMessage sends a message to the correct dispatch handler, as specified when the class was registered, in this case, thisMessageHandler. There is one message queue per application, so some process has to pull messages off the queue and dispatch them to the correct window. This includes events not normally associated with a display window.

Window Message Handler

DispatchMessage calls the handler, thisMessageHandler, with a message for the window. The handler is declared as

```
LRESULT CALLBACK DLL_API
  thisMessageHandler ( HWND hwnd, UINT msg,
  WPARAM wParam, LPARAM lParam );
```

Neither the CALLBACK nor the DLL_API keywords have any meaning, and are #defined to nothing, The LRESULT is defened to be a signed long. HWND is a blind value representing the window ID, msg is an enumeration of message types, and wParam, a 16 bit (32 bits under Win32) , and lParam, a 32 bit parameter. These are defined by the msg type. It should be noticed, each message can only pass 48 bits (64 bits under Win32) of information with it, and that may be a bit less than most Macintosh programmers are used to.

The method itself is a switch statement with each message type handled differently. Simple ones, such as COMM_CLEANUP, are only one call, comm_server_cleanup((HTASK)wParam)); in this case. More complicated ones, such as COMM_NET_ACTIVITY, get quite large with subtypes of net activity and file descriptors passed in.

Sending messages

While most messages are generated by the system, there are times when one part of an application, or another application, needs to send a message to a particular window. As long as the windows ID is known, sending the message is as simple as a single call. For example, in the Macintosh winsock layer we created, we have the call:

```
PostMessage(
  ((ws_Endpoint_t *)contextPtr)->hwnd,
  COMM_NET_ACTIVITY, FD_CONNECT,
  (LPARAM)contextPtr );
```

Now, the various parameters in this PostMessage are application dependant, but suffice it to say, the first parameter is the window we are calling, the second is the message type, the third is the wParam and the fourth is the lParam. In this one case, we had to change the API definition of a message type. Under Windows, a communications endpoint is defined as a 16 bit file identifier, on the Macintosh, it's a 32 bit pointer. The COMM_NET_ACTIVITY message type defines the wParam as the file descriptor, but that clearly would not work on the Macintosh. As we are generating the message as well as reading them, we were able to change the definition in both places.

In the PostMessage case, the message is added to the queue, and the application continues on. Sometimes, a response is required, and that's what SendMessage is for. The calling application will block until the receiving app returns a 32 bit result. Again, this limit may be a bit more than Macintosh developers are used to.

Implementing windows

Again, we have a terminology problem. When I say implementing windows, it's a bit ambiguous which windows I mean. In this case, I only mean implementing the windows message system, not the graphical windows, nor the entire Windows OS.

An important issue is sharing memory. Each application using the message queues needs to share memory so they can access the queues themselves. This, unfortunately, put the queues in whatever heap was launched first. One solution would be to use an extension, but for this particular project, that was neither required nor desired.

Another important aspect is the memory management. Some of the PostMessages have to happen at interupt time, and therefore, a memory scheme that can be called at interupt time is vital. We found that few structures need to be created at interupt time, so we used OTAllocMem

for message queueing. It is very important to not flood this. OTAllocMem will allocate several K of memory, but not large quantities. Calls to the OTAllocMem family are defined by amalloc, acalloc, afree, and so on. All these calls are straightforward memory allocation wrappers.

First, some data structures are needed to handle the queues:

```
typedef struct _msg_t {
  struct _msg_t *next;
  struct _msg_t *prev;
  MSG msg;
} msg_t;

typedef struct _wnd_t {
  struct _wnd_t *next;
  struct _wnd_t *prev;
  void *class;
  HWND wnd;
  HTASK task; // this just makes it easier!
} wnd_t;

typedef struct _wnd_class_t {
  struct _wnd_class_t *next;
  struct _wnd_class_t *prev;
  WNDCLASS wndclass;
  wnd_t wnds;
} wnd_class_t;
```

MSG, HWND, HTASK, and WNDCLASS are all standard Windows structures. The above structures are used to create a circular linked list of windows structures. Static variables are defined to head the queues.

```
static msg_t msgs = { &msgs,&msgs };
static wnd_class_t classes =
  {&classes, &classes};
static HWND timer_wnd;
static TMTask timer_task;
```

Also, some accessor methods are used to make finding particular windows and classes easier:

```
static wnd_class_t *find_class(
  const char *class)
{
  wnd_class_t *scan;

  scan = classes.next;
  while (scan != &classes) {
    if (!strcmp(
        scan->wndclass.lpszClassName,
        class))
      return(scan);
    scan = scan->next;
  }
  return(NULL);
}

static wnd_class_t *find_class_by_task(
  HTASK theTask )
{
  wnd_class_t *scan;

  scan = classes.next;
  while (scan != &classes)
    {
    if ( scan->wndclass.hInstance ==
        theTask )
      return(scan);
    scan = scan->next;
    }

  return(NULL);
}

static wnd_t *find_wnd(HWND wnd)
{
  wnd_class_t *scan_class;
  wnd_t *scan;

  scan_class = classes.next;
  while (scan_class != &classes) {
    scan = scan_class->wnds.next;
    while (scan != &(scan_class->wnds)) {
      if (wnd && scan->wnd == wnd)
        return(scan);
      scan = scan->next;
    }
    scan_class = scan_class->next;
  }

  return(NULL);
}

static wnd_class_t *find_class_from_wnd(
  HWND wnd)
{
  wnd_class_t *scan_class;
  wnd_t *scan;

  scan_class = classes.next;
  while (scan_class != &classes) {
    scan = scan_class->wnds.next;
    while (scan != &(scan_class->wnds)) {
      if (wnd && scan->wnd == wnd)
        return(scan_class);
      scan = scan->next;
    }
    scan_class = scan_class->next;
  }

  return(NULL);
}
```

The accessors are using a few of the internal windows fields, but what they are doing is pretty straight forward.

Two other utility functions, insque and remque, are used to attach the different structures together.

A few other utility methods are used to convert Macintosh formats into Windows formats. For example, Window's tasks are 64 bit longs and Macintosh processes are a 64 bit structure; Macintosh ticks are 1/60 of a second and Windows uses microseconds. Those types of routines.

```
HTASK GetCurrentTask ( void )
{
  ProcessSerialNumber PSN;
  HTASK task_id;

  GetCurrentProcess ( &PSN );
  task_id = (PSN.highLongOfPSN << 32) +
    PSN.lowLongOfPSN;
  return task_id;
}
unsigned long GetCurrentTime(void)
{
  return ((unsigned long)TickCount() *
1666)/100; // 16.66msecs/tick
}
```

The next chuck of code is the implementation of Register/ UnregisterClass

```
ATOM RegisterClass(
  const WNDCLASS FAR*wndclass)
{
  wnd_class_t *new;
  ATOM retval = 0;

  if (find_class(wndclass->lpszClassName)
      == NULL) {
    new = acalloc(1, sizeof(wnd_class_t));
    if (new) {
      memcpy(&(new->wndclass), wndclass,
        sizeof(WNDCLASS));
      new->wnds.next = &(new->wnds);
      new->wnds.prev = &(new->wnds);
      new->next = new;
      new->prev = new;
      insque((queue_node_t *)new,
        (queue_node_t *)&classes);

      retval = 1;
    }
  }

  return(retval);
}
```

```
BOOL UnregisterClass(LPCSTR class,
  HINSTANCE hinst)
{
  wnd_class_t *scan;
  BOOL retval = FALSE;

  scan = find_class(class);
  if (scan && scan->wnds.next ==
      &(scan->wnds)) {
    remque((queue_node_t *)scan);
    afree(scan);

    retval = TRUE;
  }

  return(retval);
}
```

The keyword ATOM is used to define an atomic operation, one that no other process can interupt. For once, we are fortunate that the Macintosh uses a cooperative multi-processing environment, and again, ATOM is #defined out.

As well as registering classes, routines for creating and destroying windows are needed.

```
HWND CreateWindow(LPCSTR class,
  LPCSTR name, DWORD style, int x, int y,
  int w, int h, HWND parent, HMENU menu,
  HINSTANCE hinst, void FAR* arg)
{
  wnd_class_t *scan;
  wnd_t *new;
  HWND retval = NULL;

  scan = find_class(class);
  if (scan != NULL) {
    new = amalloc(sizeof(wnd_t));
    if (new) {
      new->class = scan;
      new->wnd = new;
      new->task = GetCurrentTask();
      new->next = new;
      new->prev = new;
      insque((queue_node_t *)new,
        (queue_node_t *)&(scan->wnds));

      retval = new->wnd;
    }
  }

  return(retval);
}
```

```
HWND DestroyWindow( HWND hWnd )
{
  wnd_class_t *scan;
  wnd_t *the_wind;
  HWND retval = NULL;

  the_wind = find_wnd( hWnd );
  if (the_wind != NULL)
    {
    the_wind->next->prev = the_wind->prev;
    the_wind->prev->next = the_wind->next;
    afree( the_wind );
    }

  return(retval);
}
```

As the only use of our windows is a communication channel, most of the parameters are ignored. In the example that used them, real values were passed in, but those came from the porting of Windows code. The task is really the current process id, and it is used to direct messages to the correct application.

Implementing Post/ SendMessage

All messages sent have to be handled either as an inter-application call, or a local call. First, there is a very important rule because a SendMessage call is always blocking, never send a message to the current application, only post. Fortunately, Windows has to follow this rule also.

```
LRESULT SendMessage(HWND wnd,
  wnd_msg_t msg, WPARAM wparam,
  LPARAM lparam)
{
  MSG   theMsg;
  BOOL  retval = FALSE;
  DWORD cur_time;

  OSErr       anErr;
  Boolean     isSame;
  Size        actualSize;
  LRESULT     AEresult = true;
  DescType    typeCode;
  wnd_class_t *scan;

  GetDateTime ( &cur_time );
  theMsg.hwnd = wnd;
  theMsg.message = msg;
  theMsg.wParam = wparam;
  theMsg.lParam = lparam;
  theMsg.time = cur_time;
  theMsg.hInstance = GetCurrentTask();

  scan = find_class_from_wnd( wnd );
  if ( GetCurrentTask() ==
      scan->wndclass.hInstance )
    return DoMessageNow( &theMsg );
  else
    return SendMessageByAE( &theMsg );
}
```

Basically, the message is placed inside a message structure, then either immediately executed, or sent to the other application. The determination is made by comparing the current task to the task saved when the window was created.

Immediate execution is handled by DoMessageNow

```
staticLRESULT DoMessageNow(const MSG *msg)
{
  wnd_class_t *scan_class;
  scan_class = find_class_from_wnd(
    msg->hwnd );
  if (scan_class &&
      scan_class->wndclass.lpfnWndProc)
    return
      ( scan_class->wndclass.lpfnWndProc)
      ( msg->hwnd, msg->message,
      msg->wParam, msg->lParam );
}
```

The class of the window contains the message handler, so a simple call to the class accessor finds if there is a callback routine. If there is a callback, the message is dispatched.

If the message has to be dispatched to another application is done by SendMessageByAE

```
static LRESULT
SendMessageByAE( MSG *msg )
{
  AppleEvent           event, reply;
  OSErr                anErr;
  AEAddressDesc        procDesc;
  Size                 actualSize;
  LRESULT              AEresult = true;
  DescType             typeCode;
  ProcessSerialNumber  psn;
  wnd_t                *scan;
  long                 psn_part;

  scan = find_wnd ( msg->hwnd );
  if ( !scan )
    return 0;

  psn_part = (long)(scan->task>>32);
  psn.highLongOfPSN = psn_part;
  psn_part = (long)(scan->task &
    0x00000000ffffffff);
  psn.lowLongOfPSN = psn_part;
  if ( !psn.highLongOfPSN &&
      !psn.lowLongOfPSN ) {
    DebugStr("\pCould not find process");
    return 0;
  }
  anErr = AECreateDesc(
    typeProcessSerialNumber, (Ptr)&psn,
    sizeof(psn), &procDesc);

  if (!anErr)
    anErr = AECreateAppleEvent(kBGApp,
      'Msg ', &procDesc,
      kAutoGenerateReturnID,
      kAnyTransactionID, &event);
  if (!anErr)
    anErr = AEPutParamPtr( &event, '----',
      'Msg ', msg, sizeof(MSG) );

  if (!anErr)
    anErr = AESend( &event, &reply,
      kAECanInteract + kAECanSwitchLayer +
      kAEWaitReply, kAENormalPriority,
      kNoTimeOut, nil, nil );

  if ( !anErr )
    {
    anErr = AEGetParamPtr( &reply, '----',
      'Rply', &typeCode, &AEresult,
      sizeof(LRESULT), &actualSize );
    if (anErr)
      DebugStr("\pCould not get reply");
    }

  if (!anErr)
    AEDisposeDesc( &event );

  return AEresult;
}
```

The process of the receiving application is found, and an AppleEvent containing a MSG structure is sent. The receiving application is responsible for processing the AppleEvent and generating a return value. The sending application will block, receiving no events, not even update events.

PostMessage works with a different mechanism. PostMessage can be called at interupt time, when AppleEvents are not allowed to be called. However, PostMessage doesn't block, and there is no guarantee when the event will be processed. Therefore, we just put the posted events in a queue using OpenTransport's interupt safe memory allocation routines.

```
BOOL PostMessage(HWND wnd, wnd_msg_t msg,
  WPARAM wparam, LPARAM lparam)
{
  msg_t *next_msg;
  BOOLretval = FALSE;
  DWORD cur_time;
  wnd_t *scan;

  next_msg = OTAllocMem(sizeof(msg_t));
  if (next_msg) {
    GetDateTime ( &cur_time );
    next_msg->msg.hwnd = wnd;
    next_msg->msg.message = msg;
    next_msg->msg.wParam = wparam;
    next_msg->msg.lParam = lparam;
    next_msg->msg.time = cur_time;
    scan = find_wnd ( wnd );
    if ( scan )
      next_msg->msg.hInstance = scan->task;
    else
      next_msg->msg.hInstance =
        GetCurrentTask();
    next_msg->next = next_msg;
    next_msg->prev = next_msg;
    insque((queue_node_t *)next_msg,
      (queue_node_t *)&msgs);
    retval = TRUE;
  }

  return(retval);
}
```

To contrast the two, SendMessage has to block until it returns, so it either short-circuits the dispatch mechanism and immediately calls its message handler, or it sends an AppleEvent to the correct applica-

tion. PostMessage just takes the message, adds it to the global message queue, and continues on. PeekMessage will need to handle both mechanisms for receiving events.

Implementing PeekMessage

First, PeekMessage checks to see if any PostMessages have been put on the queue. These usually come from interupts, and are usually of high priority. If we have any posted messages, they are always handled first. This causes any sent messages to be delayed, and also blocks the sender for longer.

If no messages have been posted, the next event is taken from the event queue and handled, either as an AppleEvent, or passed to the Macintosh Event Handler registered with this class.

```
BOOL PeekMessage(MSG *msg, HWND wnd,
  UINT start, UINT end, UINT action)
{
  msg_t        *next_msg;
  BOOL         retval = FALSE;
  EventRecord  event;
  Boolean      nullEvent;
  HTASK        thisProcess, msgProcess;
  OSErr        theErr;

  thisProcess = GetCurrentTask();
  next_msg = msgs.next;
  while (next_msg != &msgs)
    {
    msgProcess = next_msg->msg.hInstance;
    if ( ( thisProcess ==
        next_msg->msg.hInstance ) &&
        ( ( wnd == NULL ) ||
        ( wnd == next_msg->msg.hwnd ) ) )
      {
      memcpy(msg, &(next_msg->msg),
        sizeof(MSG));
      if (action == PM_REMOVE)
        {
        remque((queue_node_t *)next_msg);
        OTFreeMem(next_msg);
        }
      retval = TRUE;
      break;
      }
    next_msg = next_msg->next;
    }

  if ( !retval )
    {
    wnd_class_t  *theClass;
    nullEvent = WaitNextEvent( everyEvent,
      &event, kSleepValue, nil );
    theClass = find_class_by_task(
      GetCurrentTask() );

    if ( theClass &&
        (theClass->wndclass.eventCallback))
      (*theClass->wndclass.eventCallback)
        ( &event, nullEvent,
        theClass->wndclass.eventUserData );
    else if ( !nullEvent )
      switch (event.what)
        {
        case kHighLevelEvent:
          AEProcessAppleEvent( &event );
          break;
        case updateMask:
          BeginUpdate(
            (WindowPtr)event.message );
          EndUpdate(
            (WindowPtr)event.message );
          break;
        }
    }

  return(retval);
}
```

Only messages queued for this process and for the window asked for are returned. Passing a null value for window will return any message for this process. If there is a message waiting, it is returned with the value of true.

If no message is queued up, then we check for an event. As AppleEvents come through the event queue, and there is no easy way to get just AppleEvents without throwing out other events, we have to grab all events. If there is a registered callback, then the event is passed to the callback, otherwise, we check for AppleEvents and update events. Update events have to be handled minimally to clear the update region, otherwise, the application will be flooded with update events.

Handling the AppleEvent involves an AppleEvent handler installed at application initialization time:

```
AEInstallEventHandler( kBGApp, 'Msg ',
  NewAEEventHandlerProc( MsgHandler ), 0,
  false );
```

and a message handler:

```
OSErr
MsgHandler( AppleEvent *event,
  AppleEvent *reply, long refcon )
{
  long         theTickcount;
  EventRecord  anEvent;
  MSG          msg;
  Size         actualSize;
  LRESULT      result;
  DescType     typeCode;
  OSErr        anErr;

  anErr = AEGetParamPtr( event, '----',
    'Msg ', &typeCode, &msg, sizeof(MSG),
    &actualSize );

  if (actualSize != sizeof(MSG))
    DebugStr("\pbad event came in");

  if (!anErr)
    result = DoMessageNow( &msg );

  if (reply->descriptorType != 'null')
    {
    anErr = AEPutParamPtr( reply, '----',
      'Rply', &result, sizeof( LRESULT ) );
    if (anErr)
      DebugStr("\pCould not make a reply");
    }

  return noErr;
}
```

The MsgHandler pulls a message structure from the AppleEvent, calls DoMessageNow, just like SendMessage would have done with a local call, and puts the return value in the reply field. If an AppleEvent is received, a false is returned from PeekMessage, meaning no more processing is required.

Implementing the rest

Every PeekMessage should be in the following loop:

```
if ( PeekMessage ( &msg, mWindowID, 0, 0,
      PM_REMOVE ))
    {
    TranslateMessage( &msg );
    DispatchMessage( &msg );
    }
```

The TranslateMessage for our purposes does nothing, and is implemented as:

```
BOOL TranslateMessage(const MSG *msg)
{
  return(TRUE);
}
```

The DispatchMessage is also straightforward. It is only called when PeekMessage has returned a message for this process, and therefore, it only needs to call DoMessageNow.

```
LRESULT DispatchMessage(const MSG *msg)
{
  if ( GetCurrentTask() == msg->hInstance )
    DoMessageNow( msg );
  else
    DebugStr( "\pDispatch should only have
      gotten a same process message");
  return 0;
}
```

A little debugging code is added for a sanity check.

Conclusions

It was a lot of work to get the underlying structure to work, but once done, a lot of code was ported without many other changes. For a small project, it would have been a lot better to rewrite everything, but for the project we did, it was well worth the time to be able to port the Windows code directly.