Windows CE Programming For Pocket PC In A Nutshell

This document tries to explain in a small number of pages the essentials of the Windows application model on Windows, particularly Windows CE on the Pocket PC platform. More specifically, the course uses Windows Mobile 2003 as running on the HP IPAQ 4150, 4350, and 5550 Pocket PCs.

What the heck is Windows CE?

The core Windows API at this point in time is called Win32. A 64 bit version of the Windows API is in the works. There are currently three extant Windows operating systems and one dead version

- Dead 16 bit versions. These include Windows 1.0 (1985) through Windows 3.1 (1992). These support a version of the Windows API called Win16. In many ways, the core Windows application model has not changed much since those days.
- Dying 32 bit versions. These include Windows 95, 98, and ME. They support Win32, but are largely transitional at this point in time.
- Extant 32 bit versions. These include NT (1992-2000), 2000, XP, and the next version of NT, whatever it will be called. Note that Embedded Windows XP is basically just XP but can be custom-configured for specific uses.
- Windows CE. A version of windows designed for use in embedded systems that are very small and require real-time behavior. CE is wildly customizable and targets several different processors. There is a tool called Platform Builder which lets you create a custom CE build for a specific device. The Pocket PC system (aka Windows Mobile) is a custom CE build combined with device drivers for the relatively standardized Pocket PC hardware and other services.

Unlike the others, Windows CE is a real-time operating system, giving the following properties:

- Bounded interrupt service time. We can, at least in principle, determine the worst-case running time of any event or combination of events in the system.
- Fixed priority scheduler. Each thread, including in-kernel threads, runs at particular priority level, and the highest priority runnable thread is always the one that is run.
- Priority inheritance to defeat priority inversion.

Note that because CE has these properties, it is possible to build a hard real-time system on top of it. However, it does not mean it is easy. For example, since the device drivers created for Pocket PC are proprietary, we do not know the bounds.

It is important to note that CE does not implement the full Win32 API, only about half of it in terms of API functions. Various API functions have parameters that are ignored.

It's generally pretty easy to take a C or C++ Windows CE program and recompile it (with some changes) for use on, say, XP. This makes for a convenient debugging approach. If there is something you don't understand and the CE debugger is not helping, you might want to move it to XP and use the more power facilities there. Moving programs that exploit the full Win32 API to CE is much harder.

What is "dot net" ?

You may have heard a lot of news about ".NET" and in particular the ".NET Compact Framework" for Pocket PC, Windows CE, SPOT, and others. On a small device, it's easiest to think of .NET as language level virtual machine with a just-in-time (JIT) compiler combined with a large library and a run-time system that resembles an operating system. What this means is that you can write a program in any language, compile it down to the .NET stack machine architecture. That program can then be run on any system, including a pocket pc, as long as the program only uses run-time system services and the library functions that are implemented on the system. The JIT compiler will translate frequently used code paths into native code on the underlying machine, eventually resulting in the program running at near-native speeds. The ".NET Compact Framework" implements a subset of the full services and library functions.

Although the .NET Compact Framework runs on top of Windows CE and is installed on your Pocket PCs, we will not be using it for our projects because it does not provide the core properties shown above to applications. In other words, although it runs on top of this real-time operating system, it itself does not provide real-time properties to its applications. This may change, particularly as .NET is being pushed into smaller and smaller devices (look for "SPOT" if you're interested), more and more languages are ported to it, and non-Microsoft implementations such as Mono become common.

What about QNX/Linux/etc?

Although QNX, Linux, and others have been ported, particularly to early Pocket PCs, and there is no reason to believe that they could not be ported to modern Pocket PCs, we are unaware of any modern ports. We investigated QNX pretty thoroughly, for example. Linux is not currently a real-time operating system, and extensions that do provide the properties noted above require that programmers essentially develop at the kernel module level, which erases much of the allure of Linux.

What about uC/OS-II, eCos, Fusion, etc.?

These presently don't support Pocket PCs and their peripherals as well as Windows CE. You'll have the opportunity to work with a light-weight real-time embedded systems in the Motes lab assignments.

What's in a Pocket PC?

Pocket PCs are based on the ARM series of 32 bit low power processors, lately called Intel XScale. Although they typically run at 400 MHz, these processors are really only about as fast as a mid '90s Pentium (Pentium "one"). The typical Pocket PC has 64 MB of RAM, of which a program typically can use about 32 MB. Again, this is quite constrained compared to a modern PC. The Pocket PCs you will use in this class have

802.11b and Bluetooth wireless radio network interfaces as well as infrared optical network interfaces. They also have stereo sound out output (speaker and headphone jack) and monophonic sound input (microphone only). The display screen is a touch-sensitive 240x320 LCD, about 1/25 as many pixels as a modern desktop display.

To install new programs or debug a program on pocket pc, it must be docked with a desktop PC running ActiveSync.

Compiler toolchain

To program a Windows CE application, one can use either Embedded Visual C++ or Visual Studio .NET. EVC++ lets us create native code that directly uses the Win32 API as implemented by Windows CE. VS.N creates .NET stack machine code that uses the .NET API. We will be using EVC++ in this course. The two toolchains look very similar, having a friendly, project-oriented graphical interface that is built on top of command-line tools.

At the lowest level that we will use, a windows application is a C or C++ program that directly calls the Win32 C API. Microsoft also has a venerable C++ class library called MFC that can be used to ease development. We will not be using more recent extensions such as ATL or COM components.

An important feature of EVC++ is that it can write code for you. When programming Win32 directly or via MFC, EVC++ can automatically generate an initial skeleton for your application and then create new code for you as you add dialog boxes or new windows messages. This is also a wonderful learning aid.

Win32's philosophy

The Win32 API is massive, but a typical program will only use a few dozen functions and most of those involve drawing. If you're familiar with Unix programming, you should be aware that Win32 has a few philosophical differences

- Windows was designed for graphical (GUI or WIMP as you like) programs from the ground up.
- The GUI features and basic OS features are intertwined in Windows, while on Unix, X is quite separate from the kernel.
- Everything is **not** a file in Windows.
- A Windows HANDLE is not really a file descriptor despite what it looks like.
- Creating a process is very expensive. There is no fork and emulating one has an exorbitant cost. Threads are relatively cheap and work in only one way.
- Where the Unix way is to have a simple orthogonal API (e.g., the near-universality of open/read/write/seek/close) with all the infrequently used stuff hidden (e.g., the horror that is ioctl), the Windows way is to have clearly visible (and different!) function call for each operation on each kind of object. While people can argue about this forever, the upshot is that some things that are straightforward in one OS become extremely painful in the other and vice versa.

High-level application model

A single-threaded Windows application makes progress by receiving and processing messages sent to it by the operating system, other applications, or itself. Each message is typically bound for a window the application has open. The whole system's windows are arranged in a tree that reflects their containment relationship. Even controls are windows. For example, the OK button in a dialog box is a window. You may want to play with the spy++ application to see your window tree.

Each window has associated with it a window procedure[1], which is typically a giant switch statement that processes the messages the window cares about and passes the remainder back to the system for default processing. A message bound for a particular window may either be delivered directly to the window procedure (i.e., the system can call the window procedure) or indirectly by queuing it for the application which then will dequeue the message and invoke the window procedure. The queue is sorted by priority.

In a single-threaded application, or one in which only a single thread handles the GUI (which is what you want to do) no windows procedure need be reentrant. However, it should process its message quickly and queue a new message for itself to continue work later if necessary.

In many ways, a single-threaded application is similar in flavor to a select-based server on Unix. The server is single threaded and calls select to get events to respond to. The Windows application processes queued messages. In both cases, it is the programmer's responsibility to respond quickly to events/messages and to avoid blocking system calls. If processing a message takes too long or blocks, the program appears sluggish.

A single-threaded application is also very similar to an event-driven simulator.

A common response to a message is to redraw the window or parts of it. Drawing is done using an API called GDI. For the most part, GDI is straightforward, but one confusing thing is that one does not draw on a window but rather on a "device context" which can acquired from a window or created in memory.

For the first 8 years of its life, Windows only supported single threaded applications. Win32 allows straightforward creation of threads and they are commonly used. Although threads can interact with windows directly and even draw on them, you want to avoid this. Instead, use semaphores and shared memory to exchange data between threads and have background threads post messages to the application to signal conditions.

Looking at messages

A windows message consists of:

- A handle to the targeted window
- The message type (a number)

[1] Strictly speaking, the window procedure is associated with a window *class*. All windows of a particular class have the same window procedure.

- A 32 bit message parameter
- A second 32 bit message parameter

Based on the type of the message and the targeted window, the window procedure or the application can cast the parameters to appropriate types. For example, the parameters are often used to send a pointer to something. You can use the spy++ application to see what messages are being delivered to an application or a window. You will be surprised at the rate and diversity of message arrivals.

Here is the prototype of a window procedure:

```
LRESULT CALLBACK WndProc(HWND hWnd,     // window handle
                         UINT message,  // message type
                         WPARAM wParam, // 32 bit parameter
                         LPARAM lParam) // 32 bit parameter
```

There are hundreds of message types. Some important ones are:

- WM_CREATE – the very first message a window receives (see CreateWindow())
- WM_DESTROY – the very last message a window receives (see DestroyWindow())
- WM_INITDIALOG – first message
- WM_ACTIVATE – the window has come to the foreground or gone to the background (see ShowWindow())
- WM_PAINT – the system has erased some part of the window and wants the window to redraw that part. (See InvalidateRect())
- WM_TIMER – a timer (see SetTimer()) has expired
- WM_LBUTTONDOWN, WM_LBUTTONUP, WM_MOUSEMOVE, etc – mouse/stylus related events
- WM_CHAR, WM_KEYDOWN, WM_KEYUP, etc – keyboard related events
- WM_COMMAND – menu selection (or dialog box action)
- WM_USER (and WM_USER+1, etc) – a user-created message type.

Here is a very short window procedure:

```
{
  HDC hdc;
  int wmId, wmEvent;
  PAINTSTRUCT ps;

  switch (message)
     case WM_COMMAND:     // it's a menu command
        wmId    = LOWORD(wParam); // get the menu item
        wmEvent = HIWORD(wParam); // get what happened to
        // we'll just assume it's been selected
        switch (wmId) {
            case IDM_HELP_ABOUT:
```

```
            // display the about box
            // more about this later
            DialogBox(g_hInst, // this thread
                      (LPCTSTR)IDD_ABOUTBOX, // resource
                       hWnd,           // parent window
                      (DLGPROC)About); // window proc
            return 0; // success
          break;
…
…
 case WM_PAINT: // Need to redraw the window
    // We could look in the message to figure out what
    // region we need to redraw, but we'll just redraw
    // it all
    RECT rt;
    // Get a device context to draw on
    hdc = BeginPaint(hWnd, &ps);
    // Get a rectangle giving us the whole client
    // area of the window (ie, no borders, titles, etc
    GetClientRect(hWnd, &rt);
    // Draw "hello" right in the middle of everything
    // _T() means "make this Unicode"
    // Windows CE does not support ASCII
    DrawText(hdc,
             _T("Hello"),
             _tcslen(_T("Hello")),
             &rt,
             DT_SINGLELINE | DT_VCENTER | DT_CENTER);
    // Finish
    EndPaint(hWnd, &ps);
    return 0; // success
 break;
```

Dialog boxes and resources

A dialog box is basically just a window. Like any other window, it needs to have a window procedure. Unlike most windows, however, we usually don't draw it in WM_PAINT or handle all the controls. Instead, we lay it out using an editor and have Windows draw it for us and operate the controls.

The dialog box is a *resource* that is linked into the executable and loaded from it when needed. For example, in the above window procedure, when we receive the IDM_HELP_ABOUT, we call the windows DialogBox function to create and run the dialog box for us. We give this function an identifier for the dialog box resource we want to use, IDD_ABOUTBOX. Using the resource editor (ResourceView tab in EVC++) we can draw the dialog box, add controls to it, and name every element (typically using the IDD_* (dialog box) and IDC_* (dialog box control) convention.

Menus and their elements (IDM_*) are also resources that are created and named using the resource editor.

Other common types of resources are accelerators (short cut keys for menu items or controls), icons, and strings (for internationalization). We won't use any of these.

You may wonder why it is that we need to have a window procedure for a dialog box at all. There are basically two reasons. First, we often want to copy data out of controls in the dialog box into variables in our program, or vice versa. Windows does not do this automatically for us. If we catch the dialog box creation and destruction messages, we can copy in and out at these points. The second reason is that we may want to run the dialog box non-modally. DialogBox() runs the dialog box modally. In other words, it runs to the exclusion of every other part of the program's user interface. It is not until we close the dialog box that we can use anything outside of it. This is not usually an issue with a small screen device, but it is definitely an issue on desktops.

Instances and posted versus sent messages

Windows are registered by an application instance. A good way to think about this is an application instance is the thread of the application that will be responsible for handling the user interface and the windows message pump. The application may have many other threads, and there be multiple copies of the application running, each with its own instance thread.

One reason you want to have all of the GUI handled by one thread is that it is the thread that creates a window that must be its message pump. But you usually want a new thread to do something more interesting than that.

What the instance thread usually does is register the different window classes the application will use, create the first window of the application, and then settle down to a life of being the message pump, running this code:

```
while (GetMessage(&msg, NULL, 0, 0))  {
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
  }
}
```

What this does is repeatedly dequeue the highest priority message (GetMessage), translate a low level message into one or more high level messages that get put back in to the queue (TranslateMessage), and then call the right window procedure (DispatchMessage) with the message. TranslateAccelerator deals with keyboard shortcuts and is just a detail from our point of view.

Message translation is interesting. The windows model is to send all messages to the application, even very low level things like "someone just moved the mouse in your title bar" and "the user has pressed the ctrl key down but nothing else so far". Most of the time, your windows procedure just ignores them and passes them to DefWndProc(). However, if you do ever want to get at this low level info, all you have to do is add a case statement and you're there.

You can send your own messages, from any thread using the SendMessage and PostMessage calls:

```
LRESULT SendMessage/PostMessage( HWND hWnd,
                                 UINT Msg,
                                 WPARAM wParam,
                                 LPARAM lParam)
```

SendMessage basically finds the appropriate window procedure based on the window handle and immediately calls it. It blocks until the window procedure returns. PostMessage finds the instance thread (the thread that created the window), queues the message to it if possible, and then immediately returns.

Why should you care? If you have only a single thread, the issue with SendMessage is reentrancy. Suppose you SendMessage from a window procedure on some window that is also associated with the same window procedure. This is like your window procedure calling itself. It better be designed to handle this or bad juju will happen.

If you are calling SendMessage in thread1 and thread2 can also execute in the same window procedure, the window procedure needs to be able to handle concurrent execution. If it doesn't, well, you're probably in race condition city.

Our general advice is to use PostMessage whenever possible. Why would you ever want to use SendMessage? One reason is priority. Recall that the message queue is a priority queue. The windows message have careful priorities associated with them. However, the user messages (WM_USER+) do not. SendMessage is a way to jump ahead of the queue. Another situation in which SendMessage makes some sense is when you know for other reasons that reentrancy or concurrency is not a concern (say, you and your child windows were created by the same thread) and you want to force an order of operations (update your child windows in right-to-left order, for example). Of course, SendMessage is also much, much cheaper than PostMessage since it's basically a virtual function call, so it might make sense when resource constrained.

Threads and real-time

Any thread in Windows CE can create a thread at any time using CreateThread():

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
```

```
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

This function looks terrifying, but it's actually pretty simple in most cases, particularly on CE, which ignores most of the parameters. Here's a create thread call that's used in example_bluetooth to create a Bluetooth handler thread:

DWORD WINAPI BTHandlerThread(LPVOID) { does something }

CreateThread(NULL,0,BTHandlerThread,0,0,0);

This creates a new thread and starts it executing in the BTHandlerThread function. When the function returns, the thread terminates. Here's why this is so simple:

- lpThreadAttributes: Windows CE has no real security model, so the thread security attributes are ignored even if given.
- dwStackSize: Windows CE decides how big the stack will be at compile time, so the stack size parameter is ignored[2]
- lpParameter: It's not usually helpful to pass a single 32 bit value to the thread, hence it's zeroed out here.
- dwCreationFlags: I want to create the thread in a runnable state, hence the zero for creation flags
- lpThreadID: I don't need the thread id since I assume it will terminate by itself and I can always join with it through a message anyway. For most thread-related calls I can use the handle instead of the thread id anyway.

As mentioned earlier, CE has a fixed priority scheduler. The highest priority runnable thread in the system is always run. If there is more than one thread at that priority level, they are run round-robin.

Here is how you change the priority of a thread:

```
BOOL SetThreadPriority(HANDLE hThread, int nPriority)
```

[2] There are a several gotchas here, btw. First, if the stack size limit (set with the /STACK flag to the linker with a default of 1 MB) is too small, you lose. Second, if an individual stack frame is larger than a page (2 KB, usually), you lose. Third, if you blow past the guard page (2 KB), you lose. Basically, it appears as if CE gives you a tiny stack to begin with and will grow it by one page when you hit the guard page. CE does not do optimistic allocation. If you set the stack size limit to 1 MB and you have 32 MB of memory, you can have only 32 threads.

In the beginning, CE had 8 priority levels (THREAD_PRIORITY_IDLE to THREAD_PRIORITY_TIME_CRITICAL). It now has 256 levels, with the lower the number, the higher the priority. The convention is:

- 0..96 – higher priority than default CE device drivers. This is where your time critical sensor device drivers go.
- 97..152 - default device driver range
- 153..247 – real-time application threads, but not a device driver
- 248..255 – none real-time, application threads (the UI, for example)

The previous 8 priority levels are mapped to 248..255. Threads begin their lives at THREAD_PRIORITY_NORMAL (251).

Notice that the fixed priority scheduler is very strict and does not care about fairness or other concerns that are integrated in to dynamic priority scheduler of a modern general purpose operating system. If you have a thread that does not thing but computation and you set it to, say 153, leaving your UI thread at 251, your UI thread will starve. You'll also starve out every thread above 153 in the whole system. There is also nothing stopping you from setting priority to whatever you want. If you want your UI thread to run at higher priority than the raster driver, by all means go ahead. To get a sense of what runs at what level, check out http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceddk40/html/cxconReal-TimePrioritySystemLevels.asp.

Waiting for events, semaphores, and non-blocking I/O

The Win32 mechanisms for waiting for events is somewhat convoluted compared to Unix. There are signals, windows messages, events, and probably more. In large part, these things are not orthogonal. For example, the WaitForSingleObject() and WaitForMultipleObjects() functions look superficially similar to the Unix select() function, but there are many kinds of handles you can't wait for with them, and it can be difficult to disambiguate which event actually occurred on a handle.

One problem with this is how to determine whether an I/O will block or not. Although Win32 supports non-blocking I/O (called "overlapped I/O"), the CE implementation currently does not. It is still a bit unclear to your instructors how exactly to determine whether an I/O on a handle for a single byte will block or not in CE. The obvious work around is to create a thread to do the dirty work and not worry about whether it will block or not, but this is less than satisfying.

Win32 has lots of IPC mechanisms. Counting semaphores are available both for process synchronization and thread synchronization.

CE implements priority inheritance. This means, for example, that if you're holding a resource like a semaphore and some higher priority thread is blocked waiting for it, CE will temporarily increase your priority to that of the blocked thread to encourage you to release the semaphore sooner.

Communication

Win32 includes "winsock" and so does Windows CE. Winsock is a relatively straightforward implementation of the familiar Berkeley sockets API with some extensions that you don't really have to worry about. For IP communication over 802.11 networks, the Pocket PC just works as expected for typical unicast UDP and TCP. We provide you with a wrapper library to simplify things even further.

Our Pocket PCs also support Bluetooth radio communication, and we will need to use it since the Motes do not speak 802.11 and the Pocket PCs don't (yet) speak Zigbee. So, we are jury-rigging Bluetooth adaptors on the serial ports of some motes and gatewaying the two networks through Bluetooth.

Bluetooth is an absolute and horrific disaster both in terms of protocol and implementation on all three platforms we have looked at. It's hard to believe that someone would build a connection-oriented network layer protocol in this day an age, much less one that requires seconds to establish a connection over a whole foot of empty space. If you're interested in our gripes about Bluetooth and several particular implementations, we'd be (un)happy to talk offline. In the meantime, we have found an acceptable SDK and created a wrapper library that at least makes it tolerable. We can't fix its insane slowness when establishing connections, however.