CHAPTER

Overview of the Pocket PC Environment

IN THIS CHAPTER:
The Basic User Interface
The Architecture of Windows CE
Inside the GWE Subsystem
Reviewing the Graphics Device Interface (GDI)
Logical Design of a Windows Program
Summary
Windows CE manages every version of the Pocket PC (PPC). Elements of Windows CE display the windows, service the mouse clicks implemented as stylus taps for the Pocket PC, and update the display. However, a program written by a developer performs all the work to coordinate these activities by the elements of Windows CE. This chapter introduces the elements of Windows CE and describes the manner in which a program operates to interact with those elements.

**NOTE**

*Windows CE is the most underappreciated operating system today.*

The Windows CE operating system possesses a number of important and very impressive features. This operating system is a scaled down version of Windows 2000! The primary elements of Windows 2000 are the same elements that appear in Windows CE. Moreover, a Pocket PC application interacts with these elements in the same way that a desktop application interacts with Windows 2000. Windows CE offers the richest array of reusable interface components of any other embedded operating system, including the Palm OS and embedded Unix (which offers exactly none). This array of components is absolutely necessary to compensate for the smaller physical real estate of the Pocket PC display screen. Most real applications require reliable multithreaded applications. Within Windows CE, a program can easily spawn new threads and can ensure reliable access to shared data through the proper usage of provided synchronization primitives. Palm OS does not support multiple application threads. The various flavors of embedded Unix support multiple processes, not threads, that consume excessive overhead and significantly degrade the performance of an application. And, for all this capability, the memory footprint of Windows CE is fairly small—around 4MB of memory. So, for a typical Pocket PC, an application has available a large amount of memory, typically 8MB, for both the program executable code and its attached data. Many full-featured Pocket PC applications consume as little as 64KB of memory. All of the operating system support is memory efficient, enabling an application to exhibit extremely high levels of performance. A carefully constructed Pocket PC application just zips along without any perceptible delay in user response times.

**TIP**

*Techniques and reusable software presented in the chapters of this book help an application to obtain the smallest memory footprint and the best possible performance.*
The Basic User Interface

This section introduces the basic elements of the user interface that an application provides when executing on the Pocket PC. One purpose of this section is to introduce some basic elements of user interfaces that appear throughout this book. Another purpose is to provide an initial justification for the proper usage of these basic elements that compensate for the limited screen space of the Pocket PC.

NOTE
As with any programming capability, hundreds of various approaches may be used to construct a user interface. The simple concepts introduced in this section are the result of implementing dozens of programs under Windows CE and have proved to be most effective in utilizing the limited available screen space.

When any application executes under Windows CE, the user sees a specific interface. The basic element of this interface is a window. An example of a very simple user interface and window appears in Figure 1-1. Each of the fundamental elements of a window appears in this figure: the caption bar, the menu bar and associated menu items, and the client area.

Starting at the top of the window appears the caption bar. In the caption bar, the program places a text string that characterizes the purpose of the program. Another use of the caption bar is to serve as a navigation aid. As the user traverses through a complex user interface, the application modifies the text that appears in the caption bar to reflect the relative location within the user interface hierarchy. With a Pocket PC application, the user interface must necessarily be a complex hierarchy due to the limited real-estate of the display device. If the caption area is not used as a navigation aid in the manner described previously, the user is likely to rapidly become lost during program usage.

Below the caption bar is the menu bar. This menu contains menu items. In Figure 1-1, a single menu item appears. This menu item contains the caption Quit. When the user taps this menu item with the stylus, the response of the program is to terminate execution and to remove the window from the physical display. Using a menu item as the only element for exiting the program minimizes the screen space lost to this important feature. For the most part, in the programs in this book, the menu bar serves only to support transitions through a hierarchically organized user interface. Again, this approach minimizes the usage of the limited screen space.

The major portion of the window appears below the menu bar. This client area is the portion of the window that a program manages and utilizes. An application can
display controls, such as buttons, in this area. This area is also the portion of the window in which all drawing operations occur. Bitmap images also appear within this area when displayed by a program. Typically, the client area of a program is 145 pixels wide by 145 pixels high. When compared with a standard desktop display size of 1,024 pixels by 760 pixels, this drawing area is very small indeed!

**TIP**

*Using owner-drawn controls, tab pages, and other controls, a program can provide a hierarchical user interface that can prove to be very effective. Chapter 8 provides examples and reusable code for constructing just such an interface.*

### The Architecture of Windows CE

This section describes the overall architecture of the Windows CE operating system. A description of each of the components of the operating system appears. A demonstration of the interaction of these components assumes the form of a file-creation example. This section also clearly describes the connection between the construction of Windows 2000 and Windows CE.
The layered design approach that Windows CE implements appears in Figure 1-2. This figure shows three important layers. The Applications layer represents any PPC client application written by a developer. Of course, an application client provides a specific set of capabilities and features of interest to a particular user, such as an analysis and database program for evaluating defects in a manufacturing process. Every client application interacts with the primary element of the next lower layer, the Graphics, Windowing, and Event Subsystem (GWES). GWES appears as a protected subsystem in Figure 1-2. A protected subsystem provides a controlled interface between all applications and the underlying operating system features. Consistent with this interface role, the GWES subsystem performs two important functions: relaying user input to the application and translating client program output to the display hardware and to the Windows CE operating system. GWES accomplishes these functions through interaction with the various elements at the next lower layer of the operation system, the CE Executive.

In fact, the CE Executive consists of a number of primary components, some of which interact with each other during normal operations. These elements also appear in Figure 1-2 within the CE Executive box: Object Manager, Process Manager, Memory Manager, Input/Output (I/O) Manager, and the ubiquitous Kernel.
The CE Executive elements that appear in Figure 1-2 are not the only elements of the Windows CE operating system. These elements are simply those that perform the primary work of the operating system.

Windows CE is, in fact, an object-based operating system. Every resource, such as a process, thread, or file, that a program creates or accesses appears as a translucent object to the application. The Object Manager performs the crucial task of relating the data structures in the operating system memory to a translucent object handle or identifier. In this way, a client application can only access the resource through a set of controlled methods that require the object handle and that validate input. Additionally, the client application cannot directly modify the operating system data structures. The ultimate result of all these object-based protections is to impart a greater reliability to the client application and to the operating system itself.

When a user initiates the execution of a client application, the Process Manager comes into play. This component of the operating system creates an initial thread for the application, called the primary thread, and establishes a number of important data structures for the application, such as the initial memory heap. In fact, the Process Manager creates these resources through interactions with other components: the Kernel and the Memory Manager.

The allocation, deallocation, and tracking of all available physical memory are the functions that the Memory Manager performs. When the Process Manager or a client application requests memory, the Memory Manager finds the available memory, marks the memory as being allocated, and assigns the memory to the application. Upon release of the memory, this component simply reverses the processes, releasing the memory for use by other client applications.

All input and output resources and operations to physical devices, such as files, the serial port, and the network port, are under the control of the Input/Output Manager. When a client requests access to a file, the I/O Manager performs a set of operations similar to those performed by the Memory Manager. For the Pocket PC, the file space comes from the available memory inside the Pocket PC rather than an external device. Therefore, the File Manager interacts with the Memory Manager to allocate the memory and then creates the data structures to manage the file, such as the pointer to the current location being accessed in the file.

The most important member of the Windows CE operating system is the Kernel. The primary job of the Kernel is to manage and schedule the set of existing executing threads. When the Process Manager requests the creation of an application's primary thread, the Kernel actually establishes the thread and creates the necessary data
structures. Of course, allocation of the data structures occurs by interaction with the Memory Manager. Two sets of data structures are necessary for each thread. One data structure enables the Kernel to maintain the state of the executing thread during CPU sharing by maintaining values such as the program counter of the thread code. The other primary data structure is the stack that the thread uses to allocate and manage local variables.

Also a part of the Kernel is the thread scheduler. This nifty little piece of code ensures that each thread receives a fair share of the available CPU cycles according to the priority of the thread. After a thread receives a specific number of CPU cycles, called a time slice, an interrupt handler transfers control to the thread scheduler inside the Kernel. The Kernel decides the next thread to execute and transfers control to that thread after saving the current thread state to the thread’s data structure, which is managed by the Kernel.

NOTE
The components described previously and depicted in Figure 1-2 are the same components that appear in the Windows 2000 Executive. These CE Executive objects are scaled-down versions of the exact same code that appears inside Windows 2000.

By placing these scaled versions inside Windows CE in the memory of a Pocket PC, Microsoft provides a fully functioning, very powerful operating system for program support.

File creation by a client application provides an example of the typical interactions among the components of the Windows CE layered operating system design. An operational trace of these component interactions appears in Figure 1-3. Each row in this figure represents an interaction between two elements of the layered design. The Source column initiates the interaction. The respondent to the interaction appears in the Destination column. Informative details about the interaction are in the Description column.

Initially, a Pocket PC client application starts the ball rolling by executing the CreateFile method of the Win32 Application Programming Interface (API). Inside this API method call is an interaction with the GWE Subsystem. In response, the GWE Subsystem passes the request to the I/O Manager of the Windows CE Executive. GWES simply validates and passes the necessary arguments to the IOCreateFile method supported by the I/O Manager. The I/O Manager has to accomplish two tasks. Initially, the I/O Manager interacts with the Object Manager to create a specific file resource object. Creating the file object results in the file object entering into the global namespace managed by the Object Manager. After the I/O Manager receives the handle to the file object, the next task consists of requesting physical memory
Pocket PC Developer’s Guide

for storing the file contents. Memory allocation is the responsibility of the Memory Manager, so the I/O Manager interacts with the Memory Manager to accomplish this task.

The remaining interactions inside Figure 1-3 consist of the return of the file object handle up the call chain to the client application. All subsequent operations on the file, such as reading, writing, and testing for end to file, utilize the returned file handle. This file handle clearly identifies the open file to the components of the CE Executive. The client application can only perform operations on the file that are supported by methods that accept this file handle as the first argument.

Inside the GWE Subsystem

A Pocket PC client application interacts with the Windows CE operating system through the Graphics, Windowing, and Event (GWE) Subsystem. An understanding of the internal organization and operation of this subsystem enables the application programmer to effectively design the client program. Figure 1-4 illustrates an architectural characterization of the internals of the GWE Subsystem. The important elements of this architecture are the queues and the GDI, WINDOW, and USER components.

The first important element that appears within GWES is the System Queue. All device drivers place a message into this queue that contains information characterizing a specific user interaction. The USER component of the GWE
Subsystem conveys this message into a thread message queue. Embedded within the USER component executes a special thread named the Raw Input Thread (RIT). This thread simply monitors the System Queue for input messages. Whenever a message appears in this queue, the RIT retrieves the message, identifies the destination thread message queue, and transfers the message to this destination queue.

Recall that the Kernel component of the Windows CE Executive creates a primary thread for an application; this thread receives a dedicated thread message queue. Based on a window handle embedded within the message by the device driver, the RIT of the USER component can easily determine the destination thread message queue.

Code within the client Pocket PC application retrieves the message from the thread message queue. This code then performs an appropriate response for the application. If the response involves updating a window or any controls displayed by the window, the code executes methods supported by the WINDOW component of GWES. Responses that constitute drawing operations interact with the Graphics Device Interface (GDI) element within GWES.
TIP

A detailed discussion of the features and capabilities of GDI appears in “Reviewing The Graphics Device Interface,” later in the chapter.

Figure 1-5 provides an example of the interaction among the various elements of the GWES and a Pocket PC client application. This sample operational flow begins with a mouse click by the user inside a window displayed by the client application.

When an actual application user clicks the mouse button, an interrupt occurs within the Windows CE operating system. This interrupt transfers execution control to the mouse device driver. Upon initiation, the mouse driver determines the current window with the input focus. The driver composes a message containing the receiving window handle, the location of cursor in the client area of the window, and the date and time of the mouse click. With all the data packaged into a message, the driver then enters the message into the System Queue. Now, the Raw Input Thread in the USER component of GWES enters the action. Eventually, the scheduler inside the Kernel transfers control to the RIT. This thread sits in a loop that simply monitors the System Queue. The RIT processes messages from this queue on a first-come, first-served basis. Eventually, the mouse message traverses to the head of the System Queue. When this condition occurs, the RIT removes the mouse message from the head of the System Queue.

Upon removing the message from the System Queue, the RIT parses the message, extracts the target window, determines the thread message queue using the window handle, and transfers the message to the receiving thread message queue. The primary thread of the Pocket PC client application also sits in a loop that removes and processes messages from its thread message queue. When the mouse message appears at the head of this queue, the client Pocket PC application removes the

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse Device Driver</td>
<td>System Queue</td>
<td>User clicks on mouse</td>
</tr>
<tr>
<td>System Queue</td>
<td>USER Component</td>
<td>Raw input thread in USER reads</td>
</tr>
<tr>
<td>USER Component</td>
<td>Thread Msg Queue</td>
<td>Raw input thread in USER writes</td>
</tr>
<tr>
<td>Thread Msg Queue</td>
<td>App Primary Thread</td>
<td>Application reads and processes</td>
</tr>
<tr>
<td>App Primary Thread</td>
<td>GDI Component</td>
<td>Application draws to client area</td>
</tr>
</tbody>
</table>

Figure 1-5  An operational view of user interaction
message and performs application-specific processing. This processing may consist of drawing a line from a start location to the location clicked by the user or any number of other responses, obviously dependent on the features and capabilities supported by the client application.

In order for a client Pocket PC application to successfully interact within the context of operations just described, a specific design approach is necessary for each and every application. The logic for this design approach is as follows:

display the main window;
while not finished
{
    get message from thread message queue;
    process message retrieved from thread message queue;
}

This special design approach is known as event-oriented programming. A Pocket PC client application responds to events, usually in the form of user interactions. Key presses, mouse clicks, and a whole range of events can occur and cause messages to appear in the application’s thread message queue.

**Reviewing the Graphics Device Interface (GDI)**

When a client application chooses to render graphics into the client area, GDI services the graphics command. Elements associated with GDI services appear in Figure 1-6.

When the services of GDI are used, a major distinction exists between graphics drawing and graphics displaying. Graphics drawing consists of issuing commands to draw an object within a virtual drawing space using a specific set of drawing tools. Initiating these actions is the domain of the program. On the other hand, graphics displaying is the province of GDI. Graphics displaying consists of the actual activity performed by Windows CE and the underlying device drivers to physically display the image described in a virtual drawing space.

Graphics drawing involves a number of important concepts, such as a virtual drawing space, drawing operations, and drawing tools.

A client program performs all drawing in a virtual or logical drawing space. This drawing space is quite large. Arguments to the various drawing commands are 32-bit integers, providing a range along each coordinate system from approximately $-2^{31}$ to $+2^{31}$. Large numbers indeed!
Available to a program through GDI is a set of drawing operations or commands, such as line, rectangle, ellipse, and rounded rectangle. Also accessible is a nice set of drawing tools, such as pens and brushes, that possess a wide range of attributes under program control.

When performing a drawing operation, the program first collects together a set of drawing tools into a toolbox. This toolbox possesses a rather stuffy formal name—device context. Submitting a drawing command consists of packaging together the toolbox, the drawing operation, and any other arguments necessary for the drawing operation. The application submits this information to GDI through a specific set of methods that provide the available drawing operations.

Typically, a Pocket PC client program issues a sequence of these drawing commands. As the program issues these commands, GDI enters them into an internal cache or buffer. When the program signals that all commands have been issued, the actual display activity begins. A sequence of mapping and clipping operations transforms the drawing commands from inside the logical drawing space into the client area inside the physical space of the application window.

Figure 1-6  Elements of the Windows CE graphics model
Assuming drawing commands survive all the mapping and clipping activities, GDI then performs the drawing commands in the video hardware’s frame buffer. However, GDI does not perform the actual drawing. In reality, GDI negotiates with the video device driver to have the actual drawing commands translated into pixels inside a video frame buffer. The device driver translates the individual pixel colors into a color combination that the hardware can display and then transfers the pixels into the video frame buffer. The display hardware accesses the pixels in the video frame buffer, causing the results to display on the physical screen inside the client area of the window.

Using a virtual display space as the arguments to all drawing methods enables a client application to be totally independent from any knowledge regarding the physical display characteristics of the hardware. When a client application moves to another manufacturer’s Pocket PC, the developer may not need to modify the program to incorporate the new display sizes or colors. This translation usually occurs within the device driver, enabling application developers to target multiple Pocket PCs without significant rewrite.

**CAUTION**

If the physical size of the display area changes, the client application developer must resize the display and drawing arguments to consume the smaller or larger space available. However, if the application moves from a black-and-white display to a color display, changes to the drawing code of the application are usually unnecessary.

As described a few paragraphs ago, a device context is really a toolbox containing the current set of tools to be used by GDI in processing a particular drawing operation. Figure 1-7 contains a nominal list of the tools maintained in a device context for GDI. These tools consist of pens, brushes, fonts, and a host of other drawing resources and parameters.

In addition to the toolset maintained in the device context, this figure provides the default values associated with each tool in the toolset. However, an application’s user interface would likely be pretty boring if all text and graphics were drawn with a black pen.

Therefore, for each of these tools, a wide range of options is available to the programmer. When an application wants to use some version of a tool or resource other than the default, a simple method call replaces the default tool description with a new tool characteristic. For instance, a client application might indicate that drawing operations should use a red pen rather than a black pen. After drawing operations are completed with the red pen, the client application then restores the
drawing pen in the device context to its default value. This restoration is necessary because a limited number of these device contexts are maintained for the users of GDI, so the device contexts are reused across applications.

Another important facet of tools inside the device context toolbox is that the tools are virtual tools, because the device driver gets to decide the exact meaning of each requested tool.

Suppose for a moment that an application replaces the black pen with a red pen. In this case, a red drawing may not actually appear on the screen. This effect results from the hardware-independent nature of GDI. GDI and the device driver work together to determine the meaning for red based on the capabilities of the video
display hardware. If it is a monochrome display, a black line appears. Therefore, in this situation, a virtual red pen is in reality a black pen. However, the client program still executes without breaking and without changes, although the video hardware fails to support red pixels. Without this hardware-independence support by Windows CE and the GDI, an programmer would have to change all the red drawing commands into black drawing commands or drawing commands supported by each Pocket PC’s video display hardware.

Logical Design of a Windows Program

Recall that a Pocket PC client program uses a specific program design. The general logic of this program is quite simple:

display the main window;
while not finished {
    get message from thread message queue;
    process message retrieved from thread message queue;
}

If responding to a message involves the services of GDI, the application packages the necessary drawing tools into a device context and issues the appropriate drawing commands in virtual space.

In concept, this logic appears to be quite simple. Unfortunately, a client application implements this simple logic in real software. In order to implement this simple logic, an program employs a specific programming dance. General API functions supporting this logic appear in Figure 1-8.

Especially important to correctly implement are the interactions with USER, the primary thread message queue, and the message switch that must be coded into the program.

When constructing the Pocket PC client application, two specific components are necessary:

► **WinMain** Includes the message input support for your application
► **WndProc** Serves as the response handler for individual messages
The design requirements for any Pocket PC application indicate that each application must explicitly include these two elements. Moreover, these programs use a required signature. The signature of a function involves its name, return data type, and formal argument list. A Windows CE program fails to compile if either of these elements is absent. Compilation also fails if these pieces do not satisfy the required signatures.

Initialization of the application involves the registration of the window procedure named WndProc. In the WinMain component, a message loop invokes the API function named GetMessage. This method actually asks USER for the next message in the primary thread message queue. Inside the loop, WinMain simply passes the retrieved message to the API method called DispatchMessage. For the most part, this small function unpacks the message and passes the message data to the registered WndProc procedure.

**Processing Messages Within a Program**

Inside the WndProc, a switch statement routes execution to a message-specific handler. The code written inside the message handler invokes the API functions necessary to communicate with GDI. All messages to an application’s main window appear in your primary thread message queue. Many of these messages involve default behavior enforced by USER. Placing the message in the message queue enables a program to override this default behavior. If the client application chooses to stay with default processing, the default case in the switch statement uses the method DefWindowProc to pass the message back to USER for nominal processing.
According to the online documentation, DispatchMessage causes USER to invoke the registered WndProc. Supposedly, USER wants to save the program the effort of unpacking message arguments and finding the address of the registered WndProc. However, this explanation from the on-line documentation fails to state the real reason for forcing the client application to execute DispatchMessage. Subsequent to calling the registered WndProc, DispatchMessage checks to make sure that the client program has handled the mechanics of painting correctly. If the program mishandles painting, DispatchMessage does the job correctly.

Every interaction with a client program, either by the application user or by the Windows CE operating system itself, involves sending a message to the client program. Literally, USER supports several hundred messages. To simplify the task of using messages, Figure 1-9 includes the most common messages processed by a Pocket PC client application.

Message codes (or symbols) appear along the left side of Figure 1-9. Each message enters into your primary thread message queue under specific conditions. The right side of the figure gives the message initiation conditions for these messages.

Most of the messages represent hardware- or display-level activity. The most interesting of these messages are those representing user interaction: WM_COMMAND and WM_NOTIFY. These messages indicate user activity.

![Figure 1-9: Representing common events as messages](image-url)
with a specific component of the graphical user interface. For example, when the user selects a specific menu item from a pop-up submenu, this message enters the rear of the primary thread message queue. Eventually, this message arrives at a specific message handler encoded in the registered WndProc. Code inside the message handler responds to the indicated submenu item.

In essence, the message stream processed by the WndProc exhibits a clear lifecycle pattern. Upon initiation, the client application receives a WM_CREATE message. User interactions during actual program execution result in messages involving changes in the state of the main window (WM_MOVE and WM_SIZE) or messages that signal specific user interactions (WM_COMMAND, WM_KEYDOWN, and WM_LBUTTONDOWN). Immediately prior to termination of the program, the WndProc receives a WM_DESTROY message. WM_CREATE and WM_DESTROY are particularly useful in enabling the client application to acquire and release resources, such as access to hardware, files, and databases.

**Updating the Client Area of Your Window**

Under this event- or message-oriented mode of developing Pocket PC programs, a specific programming paradigm forms the basis for updating the client area of the main application window. Figure 1-10 illustrates the mechanics involved in updating the window client area.

Updating the window client area usually involves servicing multiple messages with explicit signaling by the client application. A special method, named InvalidateRect, provides the signal to update the client area based on the results of processing a specific message.

![Figure 1-10](Image)
Consider the example in Figure 1-10 in which a user presses the mouse button. Eventually, the WndProc receives the WM_LBUTTONDOWN message. The message handler in the WndProc extracts the mouse cursor location from the message and then caches this location into a buffer for later retrieval. Prior to exiting the handler, the WndProc signals that a window update is necessary. A special API function, InvalidateRect, provides this capability. Eventually, the WndProc receives a WM_PAINT message, indicating that a painting signal was generated. The paint message handler inside the WndProc retrieves the mouse location from its cache, using the location values as arguments in drawing commands to GDI.

Summary

This chapter explains the basic organization and operation of Windows CE and the design requirements for an application to operate within that architecture. Knowledge of the concepts discussed is crucial to understanding the discussions in the following chapters. Here are the specific concepts of importance:

- Windows CE is simply a scaled-down version of Windows 2000.
- Programs receive messages as abstractions of input processing.
- Programs issue virtual drawing commands using boxes of virtual tools as generalizations of output processing.
- Abstractions of input and output processing enable the writing of platform-independent programs.
- Pocket PC client applications require a WinMain and a WndProc with message handlers.
- Message processing often requires processing across multiple events with explicit signaling.