Remote API (RAPI)

The Remote API (RAPI) provides a set of helper functions that enable a desktop-based application to execute code on a connected Pocket PC device. Once a function has returned, the results are sent back to the PC. In essence, RAPI is a type of one-way Remote Procedure Call (RPC)?the client (your desktop application) makes a request to the server (a connected Pocket PC device) to execute some functionality, and returns the results to it.

RAPI was originally designed as a way to manage a Pocket PC device from the desktop. It includes functions that enable an application to query the file system, registry, and device databases, as well as get information about the Pocket PC's system configuration. You can even create your own functions, which can be run over the RAPI APIs.

You will quickly notice that most of the functions in RAPI look similar to the functions in the standard Pocket PC and Windows 32 API. In fact, they typically have the same definition and number of parameters, as well as the same return values, as a standard desktop function call. The only difference is that they all are prefixed with the letters Ce. For example, the RAPI function `CeFindFirstFile()` is the same as the desktop `FindFirstFile()` API, except that it will enumerate the files on a connected Pocket PC device, rather than those on the desktop. This being the case, I will not provide a detailed description of each function available via the RAPI API.

Because RAPI is run on the desktop, you must ensure that the computer running your application has the latest version of ActiveSync installed on it. This will ensure that `rapi.dll` (which is required for your application to work) is present on the desktop, as you may not distribute `rapi.dll` on your own. You can call RAPI from console applications, window applications, and even a .NET assembly.

In order to use the Remote API within your applications, you need to include the `rapi.h` header file in your project, as well as link with the `rapi.lib` library (note that because this is a desktop library, it is located in the `...\vc98\Pocket PC 2002\support`

`\ActiveSync\lib` directory in the folder where you have installed Embedded Visual C++).

Using RAPI

Before you can use any of the RAPI functions, you must first initialize Windows CE's remote services and establish a communications link with a connected device by calling either the `CeRapiInit()` or `CeRapiInitEx()` functions.

The simplest way to start RAPI is by calling the synchronous (i.e., blocking) function `CeRapiInit()`, which is defined as follows:

```
HRESULT CeRapiInit();
```

Once the function is called, it will immediately attempt to establish a connection to a Pocket PC device, and will not return control to your application until either a connection has been made or the function fails. `CeRapiInit()` will return `E_SUCCESS` if a successful connection can be made and RAPI has initialized without a problem; otherwise, you will be returned `E_FAIL`. If RAPI has already been initialized, you will receive `CERAPI_E_ALREADYINITIALIZED` as the return value.

The following short code sample shows you how to use the `CeRapiInit()` function:

```
HRESULT hr = S_OK;
hr = CeRapiInit();

if(FAILED(hr)) {
   if(hr == CERAPI_E_ALREADYINITIALIZED)
      OutputDebugString(TEXT("RAPI has already been
          initalized"));
   return FALSE;
}
```

Although using the `CeRapiInitEx()` function is a bit more involved, you will find that it provides you with a greater amount of control because it is asynchronous (the function will return to you immediately). This means, of course, that you will have to periodically check the RAPI event handle you are returned in order to find out when it has become signaled.

The `CeRapiInitEx()` function is defined as follows:

```
HRESULT CeRapiInitEx(RAPIINIT *pRapiInit);
```

The only parameter that the function takes is a pointer to a `RAPIINIT` structure, which contains information about the RAPI event handle and its status. The structure is defined as follows:

```
typedef struct _RAPIINIT {
   DWORD cbSize;
   HANDLE heRapiInit;
   HRESULT hrRapiInit;
} RAPIINIT;
```

The first field, `cbSize`, should be set before calling `CeRapiInitEx()` with the size of the `RAPIINIT` structure. This is followed by `heRapiInit`, which will be filled in with the event

`hrRapiInit`, will be filled in with the return code from `CeRapiInitEx()` once `heRapiInit` becomes signaled.

The following code snippet shows how you can use the `CeRapiInitEx()` function to initialize your RAPI connection by using the `WaitForSingleObject()` function to monitor the RAPI event handle:

```
HRESULT hr = S_OK;
RAPIINIT rapiInit;

memset(&rapiInit, 0, sizeof(RAPIINIT));
rapiInit.cbSize = sizeof(RAPIINIT);

hr = CeRapiInitEx(&rapiInit);
if(FAILED(hr)) {
   if(hr == CERAPI_E_ALREADYINITIALIZED)
      OutputDebugString(TEXT("RAPI has already been
      initalized"));
   return FALSE;
}

// Wait for RAPI to be signaled
DWORD dwResult = 0;
dwResult = WaitForSingleObject(rapiInit.heRapiInit, 5000);
if(dwResult == WAIT_TIMEOUT || dwResult == WAIT_ABANDONED) {
   // RAPI has failed or timed out. Proceed with cleanup
   CeRapiUninit();
   return FALSE;
}

if(dwResult == WAIT_OBJECT_0 && SUCCEEDED(rapiInit.hrRapiInit)) {
   // RAPI has succeeded.
   OutputDebugString(TEXT("RAPI Initialized."));

   // Do something here
}
```

When working with applications that use RAPI, it is important to remember that you are relying on a network connection between the desktop and a device. When a function fails, an error can occur in either the RAPI layer or the function itself. You can determine where the error has actually occurred by calling into the `CeRapiGetError()` function, which is defined as follows:

```
HRESULT CeRapiGetError(void);
```

The function takes no parameters, and will return a value other than 0 if RAPI itself was responsible for the function failing. If `CeRapiGetError()` returns 0, however, you know that the error occurred in the actual remote function, and you can use the `CeGetLastError()` function to determine the error code, as shown in the following example:

```
DWORD dwError = CeRapiGetError();

if(dwError == 0) {
   // The error did not occur in RAPI, find out what the
   // remote function returned
   dwError = CeGetLastError();
}
```

A few functions (`CeFindAllDatabases()`, `CeFindAllFiles()`, and `CeReadRecordProps()`) will allocate memory on the desktop when they are called. In order to properly free this memory, you can use the following function:

```
HRESULT CeRapiFreeBuffer(LPVOID);
```

The only parameter you need to pass in is a pointer to the buffer that was allocated. If the function succeeds, then you will be returned a value of `S_OK`.

When you have finished using RAPI, you must also make sure that you properly shut down the remote connection services. To do so, you can simply use the following function:

```
HRESULT CeRapiUninit();
```

The function takes no parameters, and will return a value of `E_FAIL` if RAPI has not been previously initialized.

File System RAPI Functions

Table 9.2 lists the Remote API functions for working with the Pocket PC file system.

Table 9.2. RAPI File System Functions

Remote File System Functions

Remote File System Functions

```
[View full width]

BOOL CeCloseHandle(HANDLE);
BOOL CeCopyFile(LPCWSTR, LPCWSTR, BOOL);
BOOL CeCreateDirectory(LPCWSTR,
➥LPSECURITY_ATTRIBUTES);
BOOL CeDeleteFile(LPCWSTR);
BOOL CeFindAllFiles(LPCWSTR, DWORD, LPDWORD,
  LPLPCE_FIND_DATA);
BOOL CeFindClose(HANDLE);
BOOL CeFindNextFile(HANDLE, LPCE_FIND_DATA);
BOOL CeGetFileSize(HANDLE, LPDWORD);
BOOL CeGetFileTime(HANDLE, LPFILETIME, LPFILETIME,
  LPFILETIME);
BOOL CeMoveFile(LPCWSTR, LPCWSTR);
BOOL CeReadFile(HANDLE, LPVOID, DWORD, LPDWORD,
  LPOVERLAPPED);
BOOL CeRemoveDirectory(LPCWSTR);
BOOL CeSetEndOfFile(HANDLE);
BOOL CeSetFileAttributes(LPCWSTR, DWORD);
BOOL CeSetFileTime(HANDLE, LPFILETIME, LPFILETIME,
  LPFILETIME);
BOOL CeWriteFile(HANDLE, LPCVOID, DWORD, LPDWORD,
  LPOVERLAPPED);
DWORD CeGetFileAttributes(LPCWSTR);
DWORD CeSetFilePointer(HANDLE, LONG, PLONG,
➥DWORD);
HANDLE CeCreateFile(LPCWSTR, DWORD, DWORD,
  LPSECURITY_ATTRIBUTES, DWORD, DWORD, HANDLE);
HANDLE CeFindFirstFile(LPCWSTR, LPCE_FIND_DATA);
```

The following example shows how you can use the Remote API's `CeFindAllFiles()` (as well as `CeRapiFreeBuffer()`) function on the desktop to easily retrieve a list of the wave files that are located on the device:

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.