U3 Platform 1.0 SDK

DAPI Reference Guide

Version 1.0.1
September 2005
Revision 1.0

DOCUMENT CONTROL INFORMATION

Document No.: 04-UM-0605-00

	Title	Name	Date
Issued by:	U3 Chief Architect	Daniel Goodman	September 2005
Edited by:			

Table of Contents

1 Introduction

This Reference Guide presents a detailed description of each function of the U3 Device API, including syntax, arguments, return codes, possible error conditions and C examples for each API function. It is intended to be used by developers writing applications for U3 smart devices and also non U3 smart devices.

- Chapter 1 provides a general introduction to the U3 smart devices and the U3 Device API.
- Chapter 2 presents information about DAPI environment, components and data types.
- Chapter 3 describes the session management functions in detail.
- Chapter 4 describes the device query functions in detail.
- Chapter 5 describes the private area functions in detail.
- Chapter 6 describes the standard cookie functions in detail.
- Chapter 7 contains reference material, including a list of DAPI error codes.

For DAPI usage information and task-oriented examples please refer to the *Developers Guide*, which is part of the SDK.

The following sections briefly describe the U3 smart device characteristics and the Device API capabilities.

1.1 Overview of U3 smart Devices

A U3 smart device is a configurable USB flash drive (UFD) that currently supports up to two independent memory domains (support for additional domains will be provided in future releases). Each memory domain can be configured as one of the following:

- Persistent read-only domain that emulates a CD-ROM
- Removable domain (removable disk)

1.1.1 Persistent Read-Only Domains (Virtual CD-ROM)

A persistent read-only domain is a domain that emulates CD-ROM functionality. The domain provides full CD-ROM functionality, including auto-run. The CD-ROM domain can be used to store pre-defined data on a U3 device that cannot be changed by a user.

The virtual CD-ROM domain is created, configured and managed using the U3 Tool. The tool allows ISO images to be burnt to the virtual CD-ROM domain. This domain can also be locked with a key that must be provided to reconfigure the domain or burn a new image. For information about handling the CD domain, refer to the *Hardware Development Kit* documentation.

1.1.2 Removable Domain Areas

A removable domain can be configured to contain a public and/or private area. A public area acts like a standard USB flash device. The private area implements password-based access control and it is not accessible until a successful login has occurred with the correct password. The login and logout functions are accessible through DAPI.

Users can view either the public or the private area on a removable domain at one time, as follows:

- If the domain contains only a public area, it becomes visible immediately when the device is inserted into the USB port.
- If the domain contains only a private area, it becomes visible only after logging in through DAPI. Until then, it behaves like a device with no media inserted.
- If the domain contains both public and private areas, the public area is initially visible, for example, on drive E. When users log into the private area, it replaces the public area and only the private area is visible on drive E. When users log out of the private area, the public area again becomes visible on drive E, replacing the private area.

1.1.3 Sample Configurations

U3 smart devices support the following configurations:

- Removable (all public) + CDROM
- Removable (all private) + CDROM

1.1.4 Cookies

Cookies are used to store data in a special hidden data area of the device. They are accessible only by DAPI. Applications can store private data in this hidden area.

Cookies are addressed by a unique section and entry pair, in the same way that entries in Windows INI files are addressed.

1.2 Device API

The Device API (DAPI) is a set of methods that allows the user to interact programmatically with a U3 or non U3 smart devices to:

- get configuration, capability and status information about the U3 and non U3 smart devices in a machine (see Chapter 4 Device Query Functions).
- login and logout from private areas (see Chapter 5 Private Area Functions).
- read and write to standard cookies (see Chapter 6 Standard Cookie Functions).

2 Overview

This chapter describes the operating environment and functionality of the DAPI.

2.1 Development Environment

- Supports all Win32 programming languages, including Microsoft Visual C++ 6 and .Net 2003 that support Unicode strings.
- Only Unicode strings are supported.
- Automatically supports admin and non admin users.
- Supports multiple instances, in the same application and across multiple applications.

2.2 Functionality

The Device API (DAPI) is a C-Style API packaged in a DLL. The DLL ships with a .lib and .def file. The DLL may be loaded multiple times within the same process or across multiple applications. Each may create its own unique DAPI session. Each DAPI session maintains its' own set of unique handles. Handles are not transferable between sessions.

Other characteristics of DAPI include:

- Every function returns a HRESULT value.
- DAPI is thread-safe.
- All string types are Unicode.
- DAPI generates events and allows the user to query both U3 and non U3 smart devices.

2.2.1 Privileged Operating Modes

DAPI supports admin and non-admin user privilege modes. When in non-admin mode, some of DAPI's functions are not supported. If a function is not supported in user mode, it will return the DAPI HRESULT code DAPI_E_CMD_NOT_SUPPORTED_IN_NON_ADMIN_MODE. See Section 7 DAPI HRESULT Codes for more details.

2.3 DAPI Components

The Device API package contains the following set of files:

Table 1: Device API Components

Item	Description
u3dapi10.dll	The DAPI DLL
u3dapi10.lib	The library file required to link to the DLL
u3dapi10.h	The DAPI header file. Throughout this document, this file will be referred to as dapi.h.
u3dapi10.def	The .def file listing all DAPI functions and their ordinals.

Note: The u3dapi10.dll file is signed with the “U3 LLC” Authenticode certificate.

2.4 Data Types

The structures and parameters used throughout this document use specific identifiers to define the intended data type. The implementation of DAPI uses the following data type definitions

Table 2: DAPI Data Types

Data Type	Description
DWORD	Unsigned 32-bit integer value
int	Signed 32-bit integer value
WORD	Unsigned 16-bit integer value
Handle	Signed 32-bit integer value
BYTE	8-bit unsigned integer value
wchar_t	16-bit Unicode character. The standard character type used by DAPI
DWORD64	64-bit unsigned integer
HSESSION	A 32-bit handle referring to an instance of a DAPI session. Each instance maintains it own set of callback and device handles. When a session is destroyed, all associated callback and device handles are destroyed as well.
HCALLBACK	A 32-bit handle referring to a registered callback function. If the same function is registered twice, there are two different handles.
HDEVICE	A 32-bit handle that references a detected USB device. Handles refer to both U3 and nonU3 smart devices.
HRESULT	The Windows HRESULT errors codes are used for all function return codes. For U3-specific return codes, the FACILITY_ITF feature is used, with error codes starting from 0x200. See Section 7 “DAPI HRESULT Codes” for more details.

All constants used throughout this document are defined in the dapi.h file.

3 Session Management Functions

The DAPI session management functions are used to create the DAPI session and register the application to receive DAPI events. A session must be created with the DAPI DLL to enable it to begin listening for USB device events. Once a session has been created, the callback function can be registered to receive DAPI events. The device handle provided to the callback for each event makes it possible to use all other DAPI functions and refer to devices.

The session management functions consist of the following commands:

- dapiCreateSession
- dapiDestroySession
- dapiRegisterCallback
- dapiUnregisterCallback
- dapiGetVersion

The session management commands also define the format of the callback function to be used with the dapiRegisterCallback function. This is defined in Section 3.3 Callback Function Template for Device Events.

3.1 dapiCreateSession

Description

Creates a new DAPI session and returns a handle to the session. Sessions handles are required to register a DAPI event callback.

Sessions manage and maintain callback and device handles. Once a session is created, the DAPI DLL will begin to monitor USB devices.

Syntax

```
HRESULT dapiCreateSession(HSESSION * hSession)
```

Input Parameters

`hSession`

A pointer to a HSESSION variable.

Output Parameters

`hSession`

The handle to the newly created session. If the function fails, this should be equal to ILLEGAL_HSESSION (see dapi.h).

Return Values

Table 3: dapiCreateSession Return Values

HResult	Value	Description
S_OK	0x00000000	Session successfully created
E_POINTER	0x80004003	The pointer to session handle is null
DAPI_E_OS_NOT_SUPPORTED	0x80040250	The operating system is not supported by DAPI.
DAPI_E_WIN_2K_SP4_HOTFIX_MISSING	0x80040251	DAPI requires that the latest version of Windows 2000 Service Pack 4 must be applied to this machine.

See Also

dapiDestroySession, dapiRegisterCallback

Notes

- At least one session must be created before DAPI begins to identify and create events for USB devices.
- The session handle returned can be used to register a callback function.

Example

The following example code demonstrates how to create and destroy a session. In addition it defines and registers a callback function.

To simplify the example, this sample application waits for a specific text input in order to maintain the application in its listening state. It is recommended that alternatively the callback be registered in another thread.

```
#include "dapi.h"
#include <conio.h>
#include <ctype.h>

    // A sample callback function implementation.
    // If any DAPI functions must be called, they should be done outside
    // of the callback function.
void _stdcall EventCallback(HDEVICE hDev, DWORD eventType, void* pEx)
    {
                // log device insert and eject events
    switch(eventType)
          {
          case DAPI_EVENT_DEVICE_CONNECT:
                printf("Device %2d was connected\n", hDev);
                break;
          case DAPI_EVENT_DEVICE_DISCONNECT:
                printf("Device %2d was disconnected\n", hDev);
                break;
          }
    }

    // Basic function to create a session and register a callback.
HSESSION startDAPI(void)
    {
    // create the session
    HSESSION hSession = ILLEGAL_HSESSION;

    if(S_OK == dapiCreateSession(&hSession))
          {

          // register the callback for all devices
          HDEVICE hCallback = ILLEGAL_HCALLBACK;
          DAPI_CALLBACK pCallback = (DAPI_CALLBACK)&EventCallback;

          if(S_OK != dapiRegisterCallback(
                            hSession,  // handle to the session
                            NULL,      // all devices
                            pCallback, // pointer to EventCallback(...)
                            NULL,      // no pEx data,
                            &hCallback))
                {
```

```
                        // callback registration failed, let us destroy
                        // the session now.
                  dapiDestroySession(hSession);
                  hSession = ILLEGAL_HSESSION;
                  }
            }
      return hSession;
      }

// this will destroy the session and all associated callback events
// handlers and device handles.

void stopDAPI(HSESSION hSession)
      {
      dapiDestroySession(hSession);
      }


// main entry point to application.
// Create the session and maintain the application waiting for key input
// before destroying the session and quiting.

int _tmain(int argc, _TCHAR* argv[])
      {
      wprintf(_T("DAPI version is %s\n"),dapiGetVersion());
      printf("Type q to quit\n");
      HSESSION hSession = startDAPI();

            // keep receiving events until the user has hit 'q'
      if(ILLEGAL_HSESSION != hSession)
            while('q' != _getch());

      stopDAPI(hSession);
      return 0;
      }
```

3.2 dapiDestroySession

Description

Destroys the DAPI session referenced by the session handle.

All callback and device handles associated with this DAPI session are also destroyed and invalidated. Any subsequent reference to these handles will return an E_HANDLE error.

The function dapiDestroySession should be called for each session before terminating a DAPI application.

Syntax

```
HRESULT dapiDestroySession(HSESSION hSession)
```

Input Parameters

`hSession`

The handle to a DAPI session.

Output Parameters

none

Return Values

Table 4: dapiDestroySession Return Values

HResult	Value	Description
S_OK	0x00000000	Session was successfully destroyed
E_HANDLE	0x80070006	The session handle is not valid

Remarks

- All callbacks associated with the session are unregistered and any resources used by them are freed.
- When a session is destroyed, the event DAPI_EVENT_DEVICE_DISCONNECT is not generated for connected devices associated with the session/callbacks (See Callback Function Template for Device Events).

See Also

dapiCreateSession

Example

See example in Section 3.1 dapiCreateSession

3.3 Callback Function Template for Device Events

Description

This section shows the template for the callback function used in conjunction with the dapiRegisterCallback function. The callback function is called every time a DAPI event occurs for matching devices.

This function must be implemented to received DAPI events and obtain device handles.

Syntax

```
void _stdcall dapiCallback(HDEVICE hDev,
                           DWORD   eventType,
                           void *  pEx)
```

Function Parameters

`hDev`

The handle to the device generating the event.

`eventType`

The following events are generated for all devices:

Table 5: Callback Events Generated for all Devices

Event Name	Value	Description
DAPI_EVENT_DEVICE_CONNECT	0x01	A supported USB device has been connected.
DAPI_EVENT_DEVICE_DISCONNECT	0x02	A device has been disconnected. The device handle is no longer valid.

The following events are generated for U3 smart devices:

Table 6: Callback Events Generated for U3 smart Devices

Event Name	Value	Description
DAPI_EVENT_DEVICE_UPDATE	0x03	Information about a domain on the device has been updated. A domain has been assigned a drive letter. See 4.3 dapiQueryDomainInformation.
DAPI_EVENT_DEVICE_LOGIN	0x04	A private area on the device has been successfully logged into. See Section 5.1 dapiGetPrivateAreaInfo.
DAPI_EVENT_DEVICE_LOGOUT	0x05	A private area on the device has been successfully logged out of. See Section 5.1 dapiGetPrivateAreaInfo.
DAPI_EVENT_DEVICE_WRITE_PROTECT_ON	0x06	Write protect has been enabled on a removable domain and the domain is

Event Name	Value	Description
		now read only.
DAPI_EVENT_DEVICE_WRITE_PROTECT_OFF	0x07	Write protect has been disabled on a removable domain.
DAPI_EVENT_DEVICE_RECONNECT	0x08	The device has been refreshed by windows. The current device configuration is no longer valid. See 4.3 dapiQueryDomainInformation.
DAPI_EVENT_DEVICE_NEW_CONFIG	0x09	The device has been reconfigured. The current device configuration is no longer valid. See Section 4.3 dapiQueryDomainInformation.

pEx

A pointer to the arbitrary data provided at callback registration. When an event is generated for a registered callback, the pEx data is sent to the callback with the event. If do data was defined at registration time, NULL will be passed.

Output Parameters

none

See Also

dapiQueryDomainInformation, dapiGetPrivateAreaInfo, dapiRegisterCallback.

Remarks

DAPI functions should not be called from within the event callback function. The callback function should return as quickly as possible and the device handle should be used either outside of the function or in a different thread.

Example

See example in Section 3.1 dapiCreateSession

3.4 dapiRegisterCallback

Description

Registers the callback function and returns a handle to the callback instance. Each callback is associated with a session. A callback function can be registered more than once; each successful registration returns its own callback handle.

Every instance of a registered callback maintains its own set of device handles. This is true even if the function is registered multiple times, regardless of the parameters.

Syntax

```
HRESULT dapiRegisterCallback(HSESSION *       hSession
                             wchar_t *        pszConnectionString,
                             DAPI_CALLBACK    pCallBack,
                             void *           pEx,
                             HCALLBACK *      hCallback)
```

The DAPI_CALLBACK function pointer is defined in dapi.h as

```
typedef void (_stdcall *DAPI_CALLBACK)  (HDEVICE    hDev,
                                         DWORD      eventType,
                                         void*      pEx);
```

Input Parameters

`hSession`

A handle to a valid DAPI session.

`pszConnectionString`

A pointer to the null-terminated Unicode connection string. When this parameter is set to a serial number of a device, only events from that device can cause the callback function to be called; events from other devices are filtered out. When this parameter is set to NULL, events from any device will cause the callback function to be called.

`pCallback`

A pointer to the callback function. See also Section 3.3 Callback Function Template for Device Events for the callback function template definition.

`pEx`

A pointer to arbitrary data. This pointer is passed to the callback function every time an event occurs. May be NULL.

Output Parameters

`hCallback`

Returns the handle that refers to the registered callback function. If function fails, the hCallback will be equal to ILLEGAL_HCALLBACK.

Return Values

Table 7: dapiRegisterCallback Return Values

HResult	Value	Description
S_OK	0x00000000	Callback function successfully registered
E_POINTER	0x80004003	The pointer to pCallback or hCallback is null.
E_HANDLE	0x80070006	Invalid session handle.
E_FAIL	0x80004005	Unable to register the callback. hCallback will be equal to ILLEGAL_HCALLBACK.

Remarks

- DAPI creates events for both U3 and non U3 smart devices.
- U3 smart Applications that are run by the U3 Launchpad are provided a connection string environment variable that refers to the device from which it was executed. See Section 9.4.4, *U3_DAPI_CONNECTION_STRING* of the Application Deployment Guide for more details.
- If a connection string is provided, only events generated by the device matching the connection string are sent. The callback can be registered multiple times with different or identical connection strings.
- If no connection string is provided, events for all devices (U3 and non U3 smart) are sent to the registered callback function.
- The callback function also allows the developer to define a pointer to arbitrary data (pEx); the pointer is passed to the callback function every time an event is generated as a result of this registration.

See Also

dapiUnregisterCallback, Callback Function Template for Device Events

3.5 dapiUnregisterCallback

Description

Unregisters a callback function, releasing both the callback handle and the device handles associated with the callback handle.

Syntax

```
HRESULT dapiUnregisterCallback(HCALLBACK hCallback)
```

Input Parameters

`hCallback`

The handle that refers to the callback function to be unregistered.

Output Parameters

`none`

Return Values

Table 8: dapiUnregisterCallback Return Values

HResult	Value	Description
S_OK	0x00000000	Callback function successfully unregistered
E_HANDLE	0x80070006	The callback handle is not valid.

See Also

dapiRegisterCallback

Remarks

- If the session has already been destroyed, the callback handle is not valid and an E_HANDLE error is returned.
- Using either the callback handle or device handles after the callback has been unregistered will generate an E_HANDLE error.

3.6 dapiGetVersion

Description

Returns a string with the current DAPI version.

Syntax

```
const wchar_t * const dapiGetVersion();
```

Return Value

- The return string has the format “x.y.z” ([1-99].[0-99].[0-999]).
- The first two values match the name of the DLL and its version.

4 Device Query Functions

The DAPI device query functions allow the application to query information about the device and its configuration. These functions return information for all supported USB flash drives, both U3 and non U3 smart devices. See the *Developer Guide* for additional examples using these functions. The device query functions consist of the following commands:

- dapiQueryDeviceCapability
- dapiQueryDeviceInformation
- dapiQueryDomainInformation

All functions require handles to devices that are obtained from callback events.

4.1 dapiQueryDeviceCapability

Description

Queries known capabilities of a detected device.

Syntax

```
HRESULT   dapiQueryDeviceCapability (HDEVICE hDev, DWORD nCapability)
```

Input Parameters

`hDev`

The handle to the device to be queried.

`nCapability`

The device capability to be queried. Can be one of the following:

Table 9: Supported Capabilities as defined in dapi.h

Capability	Description
DAPI_CAP_USB_HI_SPEED	The device is current running in USB 2.0 hi speed mode and has a high speed channel to the host. Bandwidth intensive operations may be performed against the device.
DAPI_CAP_U3	The device is first generation U3 compliant
DAPI_CAP_MSD	The device is a Mass Storage Device
DAPI_CAP_U3_COOKIE	The device supports U3 standard cookies
DAPI_CAP_U3_PCOOKIE	The device supports U3 protected cookies
DAPI_CAP_U3_PRIVATE_AREA	The device supports U3 private areas

Return Values

Table 10: dapiQueryDeviceCapability Return Values

HResult	Value	Description
S_OK	0x00000000	Capability is supported
S_FALSE	0x00000001	Capability is not supported
E_HANDLE	0x80070006	The device handle is not valid
E_INVALIDARG	0x80070057	The query type is unknown.
DAPI_E_COMM_FAIL	0x80040210	General communications error with the device.

Remarks

Protect cookie functions (DAPI_CAP_U3_PCOOKIES) are not supported in this version of the Device API.

Example

Calling the function in C:

```
// test that the device is U3 compatible
BOOL bU3 = (S_OK == dapiQueryDeviceCapability(hConnectedDev, DAPI_CAP_U3));
```

4.2 dapiQueryDeviceInformation

Description

This function provides information about the detected USB mass storage device. For non U3 compatible devices, not all fields will be returned with data.

Syntax

```
HRESULT   dapiQueryDeviceInformation (HDEVICE hDev,
                                      devInfo *  pInfo)
```

Input Parameters

`hDev`

The handle to the device to be queried, as returned by the function dapiCallback.

`pInfo`

A pointer to devInfo structure. The devInfo structure is defined in dapi.h:

```
struct devInfo {
    wchar_t     serialNumber[257];
    BYTE        uniqueID[20];
    wchar_t     vendorString[10];
    wchar_t     productString[18];
    wchar_t     firmwareVersion[7];
    DWORD       vendorID;
    DWORD64     deviceSize;
}
```

Return Information

`serialNumber`

The Unicode serial number of the device. Example “9A0A551017111BCB”

`uniqueID`

The byte array containing the U3 Unique ID of the device. The unique ID is a 20 byte array (160 bit). For non U3 compatible devices, an array of 0’s is returned.

`vendorString`

The Unicode device vendor string, for example “USB Corp”

`productString`

The Unicode device product name string, for example "Minutae"

`firmwareVersion`

The Unicode decimal formatted version number of the device firmware. Example “6.121”.

`vendorID`

The device vendor’s ID, for example 0x08EC

`deviceSize`

The device size in bytes

Return Values

Table 11: dapiQueryDeviceInformation Return Values

HResult	Value	Description
S_OK	0x00000000	Command succeeded
E_POINTER	0x80004003	pInfo pointer is null
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_COMM_FAIL	0x80040210	General communications error

Example

Calling the function in C:

```
devInfo deviceInfo;
HRESULT res;
BYTE    uniqueID[20];

res = dapiQueryDeviceInformation(hConnectedDev, &deviceInfo) ;

if (SUCCEEDED(res))
{
    memcpy(uniqueID, deviceInfo.uniqueID, 20);
}
```

4.3 dapiQueryDomainInformation

Description

This function returns the configuration of the domains on a U3 smart device. Non U3 smart devices are also supported and are treated as single domain devices.

The function can be used to query the number of domains on a device and also to fill in the domain information when an appropriately sized domain information array is provided.

To determine the number of domains on a device, the domain count parameter should be set to zero and a NULL domain information array pointer should be passed. The number of domains and hence the required array size is returned in the domain count parameter.

For each domain, the function will return a domain information structure. The structure lists the path to the domain (if known), flags describing the domain features and the size of domain on the device.

See Section 1.1 for information on domain types and device configuration.

When devices with multiple domains are inserted, Windows will sequentially assign each domain a drive letter or path. If the function is called before a drive letter has been assigned to all domains, the function will return a S_FALSE result, signifying the additional information will become available in the future. As drive letters are assigned to each domain, a DAPI_EVENT_DEVICE_UPDATE event is generated (see Section 3.3 Callback Function Template for Device Events). Domains that do not yet have drive letters assigned will have the DAPI_DI_NOT_READY flag set.

Syntax

```
HRESULT   dapiQueryDomainInformation (HDEVICE       hDev,
                                      domainInfo *  pInfo
                                      WORD *        nCount)
```

Input Parameters

`hDev`

The handle to the device to be queried.

`pInfo`

A pointer to an array of domainInfo structures, each of which has the form:

```
struct domainInfo {
  wchar_t   szPath[MAXPATH+1];
  DWORD64   size;
  DWORD     typeMask;
}
```

This parameter may be NULL if nCount is set to 0. The domainInfo structure is defined in dapi.h.

`nCount`

A pointer to a variable that holds the size of the array or number of domains on the device. If set to zero and pInfo is NULL, it will query the number of domains on the device and the required array size.

Output Parameters

`pInfo`

A pointer to the updated array of domainInfo structures.

`nCount`

The number of domains on the device. If pInfo is non null, it also indicates the number of elements in the array that were updated with domain information.

Domain Information

The domain information structure provides the following information for each domain

`szPath`

The Unicode drive letter or path of the domain. The path does not include the trailing '\' character. Example "E:"

`size`

The physical size of the domain in bytes. If the domain is a CDROM or a removable domain containing only one type of area then this is the size of area.

If a removable area contains both a public and private area, the size of public area is calculated by subtracting the size of the private area, returned by the getPrivateAreaInfo function, from this value.

`typeMask`

A bit array describing domain type and features. The following flags are supported:

Table 12: Supported Domain Type Flags

typeMask	Value	Description
DAPI_DI_READ_ONLY	0x00000001	The domain is read only. Write protection is enabled.
DAPI_DI_TYPE_CD	0x00000100	The domain type is a CD ROM
DAPI_DI_REMOVABLE_PUBLIC	0x00000200	A removable domain containing a public area whose size greater then 0 bytes.
DAPI_DI_REMOVABLE_PRIVATE	0x00000400	A removable domain containing a private area whose size greater then 0 bytes.
DAPI_DI_NOT_READY	0x40000000	Windows has not yet finished initializing the domain. The information is not yet available.
DAPI_DI_UNKNOWN	0x80000000	An unknown or unsupported U3 domain type.

Note: If the domain is a removable domain that contains both a private and public area, both the DAPI_DI_REMOVABLE_PRIVATE and DAPI_DI_REMOVABLE_PUBLIC flags are set in the type mask.

Return Values

Table 13: dapiQueryDomainInformation Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
S_FALSE	0x00000001	Command succeeded, however not all domains are initialized by Windows.
E_POINTER	0x80004003	pInfo is null when nCount > 0
E_HANDLE	0x80070006	The device handle is not valid
E_INVALIDARG	0x80070057	The nCount value does not match the domain count
DAPI_E_COMM_FAIL	0x80040210	General communications error. Wait and try again.

Remarks

- For non U3 smart devices, the following information will be returned regardless of the actual device configuration:
  - Domain count: 1
  - Domain 0:
    - Size: reported physical size of the device
    - Type: removable (public)
    - Path: first drive letter associated with the device
- A CD-ROM will always have the DAPI_DI_READ_ONLY flag set.
- If the removable area has write protect enabled, the DAPI_DI_READ_ONLY flag will be set.
- Some devices may return a DAPI_E_COMM_ERROR if it is currently being reconfigured or initialized. This may occur frequently if dapiQueryDomainInformation is called within a callback event, which is not recommended. Ideally dapiQueryDomainInformation should be called outside of the callback event.
- If DAPI_E_COMM_ERROR occurs, wait and call the function again.
- See the U3 Platform HDK for information on enabling and disabling write protect on a domain.

4.4 dapiEjectDevice

Description

Ejects a USB device.

Syntax

```
HRESULT   dapiEjectDevice (HDEVICE hDev)
```

Input Parameters

`hDev`

The handle to the device to be ejected.

Output Parameters

`none`

Return Values

Table 14: dapiEjectDevice Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
S_FALSE	0x00000001	The device could not be ejected.
E_HANDLE	0x80070006	The device handle is not valid

5 Private Area Functions

A Private Area (PRA) is an area on a removable domain that requires password authentication before the area can be mounted. PRAs are referred to by their path or drive letter. When a removable domain contains both a public and a private area, both will share the same drive letter, however only one can be mounted at a time. When the private area is in the logged in state, the private area is mounted and its files are visible. The public area is visible when the private area is logged out.

The PRA functions can be used to query information on the login status of PRAs, login and logout of the PRA, and change the password of the PRA.

Note: The login, logout and setPassword functions are for use with U3 devices configured using the U3 Platform SDK. These functions should never be used by applications on U3 smart drives.

The private area functions are supported when:

- The device supports the DAPI_CAP_U3_PRIVATE_AREA capability (see dapiQeuryDeviceCapability).
- The domain includes a PRA as signified by the DAPI_DI_REMOVABLE_PRIVATE flag (see dapiQueryDomainInformation).

The path to the PRA can be obtained using the dapiQueryDomainInformation function. All PRA paths must include a trailing '\' character.

PRAs support a maximum number of attempts (MaxNOA) values. This value is set at manufacture time. If the number of failed login attempts reaches MaxNOA then the private area can no longer be used until the private area is reset using the dapiSetPrivateAreaPassword. This will format the private area and remove all existing data.

The PRA functions use Unicode password strings. Care should be taken not to mix string types, e.g. ANSI, Unicode and MCBS, as it may cause failed logins and locked PRAs.

See the *SDK Developer Guide* for additional examples using these functions.

See the *U3 Platform HDK* for information on how to configure and create private area on U3 smart devices.

DAPI supports the following PRA commands:

- dapiGetPrivateAreaInfo
- dapiLoginPrivateArea
- dapiLogoutPrivateArea
- dapiSetPrivateAreaPassword

5.1 dapiGetPrivateAreaInfo

Description

Returns information about the private area. Private areas are referenced using their drive letters or path. The drive letter can be obtained via the dapiQueryDomainInformation function.

When a removable domain includes both public and private area, this function must be used in conjunction with the dapiQueryDomainInformation function to determine the size of the public area.

Syntax

```
HRESULT     dapiGetPrivateAreaInfo(HDEVICE   hDev,
                                   const wchar_t * szPath,
                                         praInfo * pInfo)
```

Input Parameters

`hDev`

The handle to the device.

`szPath`

The path to the domain to be queried, as returned by dapiQueryDomainInformation. The path must include the trailing '\' character.

`pInfo`

A pointer to a pre-defined private area information structure (praInfo), which has the form:

```
struct praInfo{
  DWORD         maxNOA;
  DWORD         currNOA;
  DWORD64       size;
  DWORD         status;
  whchar_t      szHint[32+1];
  }
```

The praInfo structure is defined in dapi.h.

Output Parameters

`none`

Private Area Information

The updated praInfo structure returned by the function returns the following information about the private area.

`maxNOA`

The maximum number of failed login attempts allowed for the domain before lockout. The maxNOA value is set at manufacture time.

`currNOA`

Current number of failed login attempts. If this value is the same as the MaxNOA value then the private area is 'locked' and no further login attempts will succeed. This value will be set to zero every time a successful login occurs. Use dapiSetPrivateAreaPassword to reset a locked private area.

`size`

The size of the private area in bytes. This parameter refers to the physical size of the private area on the device and not its size according to the file system.

`status`

The current status of the private area whose values can be:

Table 15: Private Area Status Values

typeMask	Value	Description
DAPI_PRA_NOT_CONFIGURED	0x00	There is private area associated with the path or the domain does not include a private area.
DAPI_PRA_LOGGED_OUT	0x01	The private area is currently logged out
DAPI_PRA_LOGGED_IN	0x02	The private area is available

Note: If the status of the private area is DAPI_PRA_LOGGED_OUT and the domain also contains a public area (DAPI_PI_REMOVABLE_PUBLIC), then the public area is currently visible.

`szHint`

The Unicode password hint string for the private area.

Return Values

Table 16: dapiGetPrivateAreaInfo Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szPath does not refer to a domain on the device.
E_POINTER	0x80004003	pInfo is null
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_DOMAIN_TYPE_ERROR	0x80040207	The domain referenced by the path argument is not a removable domain.
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	The device does not support DAPI_CAP_U3_PRIVATE_AREA
DAPI_E_COMM_FAIL	0x80040210	General communications error
DAPI_E_HINT_FAIL	0x80040214	Error reading the hint data. Remaining data is still valid.

See Also

dapiQueryDomainInformation

5.2 dapiLoginPrivateArea

Description

Logs into the private area making the private area visible to the user. If the domain includes a public area, the public area will no longer be visible.

Note: The function is for use only with U3 devices configured using the U3 Platform HDK. These functions should never be used by applications on U3 smart drives. Their use could cause data loss or application instability

See notes below for detecting when a private area is 'locked'

Syntax

```
HRESULT   dapiLoginPrivateArea(HDEVICE    hDev,
                        const wchar_t * szPath,
                        const wchar_t * szPassword,
                              BOOL      bLock)
```

Input Parameters

`hDev`

The handle to the device.

`szPath`

The path to the domain for login, as returned by dapiQueryDomainInformation. The path must include the trailing '\' character.

`szPassword`

The Unicode password for logging into the private area.

`bLock`

If true, it will try and request an exclusive media lock on the public area before switching to the private area, thus ensuring that there are no open files on the public area. If set to false, the login proceeds in any case, and will swap the current public area with the logged-in private area.

This value should always be true to ensure no data loss. If a lock could not be obtained a DAPI_E_MEDIA_LOCK result is returned.

Output Parameters

`none`

Return Values

Table 17: dapiLoginPrivateArea Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szPath does not refer to a domain on the device.
E_HANDLE	0x80070006	The device handle is not valid

HRESULT	Value	Description
DAPI_E_DOMAIN_TYPE_ERROR	0x80040207	The domain referenced by the path argument is not a removable domain.
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	The device does not support DAPI_CAP_U3_PRIVATE_AREA
DAPI_E_COMM_FAIL	0x80040210	General communications error or private area is locked. See notes below
DAPI_E_MEDIA_LOCK_FAIL	0x80040212	The private area could not be locked.
DAPI_E_ADV_ERROR	0x80040261	Advanced communications error

Remarks

1. If the private area is already logged in this function will return S_OK.
2. If the private area is 'locked', the login command will return a DAPI_E_COMM_FAIL error. To test if the private area is locked, call dapiQuertDomainInfo and test if the MaxNOA value equals the CurrNOA value. If so, dapiSetPrivateAreaPassword must be used to reset the private area.

See Also

dapiQueryDomainInformation, dapiGetPrivateAreaInfo, dapiSetPrivateAreaPassword.

5.3 dapiLogoutPrivateArea

Description

Logs out from the private area. If the domain also contains a public area, the public area will become visible.

Note: The function is for use only with U3 devices configured using the U3 Platform HDK. These functions should never be used by applications on U3 smart drives. Their use could cause data loss or application instability

Syntax

```
HRESULT   dapiLogoutPrivateArea(HDEVICE        hDev,
                          const wchar_t *      szPath,
                                BOOL           bLock)
```

Input Parameters

`hDev`

The handle to the device, as returned by the function dapiCallback.

`szPath`

The path to the domain for logout, as returned by dapiQueryDomainInformation. The path must include the trailing '\' character.

`bLock`

If true, it will try and request an exclusive media lock on the private area before logging out, thus ensuring that there are no open files on the private area.

If set to true, the logout fails if a file is open on the current private area. If set false, the login proceeds in any case, and swaps the logged-out private area with the public area if available.

This value should always be true to ensure no data loss. If a lock could not be obtained a DAPI_E_MEDIA_LOCK result is returned.

Output Parameters

none

Return Values

Table 18: dapiLogoutPrivateArea Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szPath does not refer to a domain on the device.
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_DOMAIN_TYPE_ERROR	0x80040207	The domain referenced by the path argument is not a removable domain.

HRESULT	Value	Description
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	The device does not support DAPI_CAP_U3_PRIVATE_AREA
DAPI_E_COMM_FAIL	0x80040210	General communications error
DAPI_E_MEDIA_LOCK_FAIL	0x80040212	The private area could not be locked.
DAPI_E_ADV_ERROR	0x80040261	Advanced communications error

See Also

dapiQueryDomainInformation, dapiLoginPrivateArea, dapiQueryPrivateAreaInfo.

5.4 dapiSetPrivateAreaPassword

Description

Sets a new password and password hint for the private area. This function can also be used to reconfigure the private area should the password be lost or private area is 'locked' due to MaxNOA being exceeded.

Note: The function is for use only with U3 devices configured using the U3 Platform HDK. These functions should never be used by applications on U3 smart drives. Their use could cause data loss or application instability

The function provides the following options:

Table 19: dapiSetPrivateAreaPassword Function Options

Function Option	Parameter Settings	
Set new password and hint	szOldPassword szNewPassword szNewHint	old password new password new hint
Set new hint	szOldPassword szNewPassword szNewHint	old password old password new hint
Reset and format private area. **Note**: All data is destroyed.	szOldPassword szNewPassword szNewHint	NULL new password new hint

The reset private area option can be used if

1. The password is forgotten by the user
2. The private area is locked as MaxNOA has been exceeded.

Note: This option reformats the private area, deleting all the data. This feature is not available in user mode.

Syntax

```
HRESULT   dapiSetPrivateAreaPassword(HDEVICE   hDev,
                                     wchar_t * szPath,
                                     wchar_t * szOldPassword,
                                     wchar_t * szNewPassword,
                                     wchar_t * szNewHint,
                                     BOOL      bLock)
```

Input Parameters

`hDev`

The handle to the device, as returned by the function dapiCallback.

`szPath`

The path to the domain, as returned by dapiQueryDomainInformation. The path must include the trailing '\' character.

`szOldPassword`

The current Unicode password for private area.
Note: If this is parameter is NULL, the private area will be formatted and all data will be destroyed.

`szNewPassword`

The new Unicode password for logging in to the private area. If just updating the hint, then this value should be the same as the szOldPassword value.

`szNewHint`

The new Unicode hint for the password. The maximum string length is 32 Unicode characters.

`bLock`

[This value is only used when resetting a private area]

If set true it will try and obtain a media lock on the private area before reformatting it. If any application currently has any open file handles on the area, the function will not proceed. If set false, the private area will be formatted regardless of any open file handles.

Output Parameters

none

Return Values

Table 20: dapiSetPrivateAreaPassword Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szPath does not refer to a domain on the device.
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_DOMAIN_TYPE_ERROR	0x80040207	The domain referenced by the path argument is not a removable domain.
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	The device does not support DAPI_CAP_U3_PRIVATE_AREA
DAPI_E_COMM_FAIL	0x80040210	General communications error
DAPI_E_MEDIA_LOCK_FAIL	0x80040212	The device could not be locked.
DAPI_E_ADV_ERROR	0x80040261	Advanced communications error
DAPI_E_HINT_FAIL	0x80040214	Error reading the hint data. Remaining data is still valid.
DAPI_E_FORMAT_FAIL	0x80040213	The new private area could not be formatted. When the user logs in they will be prompted by Windows to format the area,

Remarks

- The area lock test is carried out only when a new private area is created. Otherwise, the `bLock` parameter is ignored.
- If the command is called with an incorrect szOldPassword, the currNOA value will be incremented with every call.

See Also

dapiQueryDomainInformation, dapiQueryPrivateAreaInfo

6 Standard Cookie Functions

Standard cookies allow applications store information on the U3 device that is not accessible via the file system. Cookie space is limited and developers should follow the usage guidelines specified in the U3 smart Application Certification Self-Test Criteria document (available as part of the U3 Logo Compliance Kit). See the Developer Guide for examples using standard cookies.

Cookies behave in a similar way to Windows INI file entries. Each entry has a section name (szSection) and an entry name (szEntry). Both of these values are required to reference a cookie.

Standard cookies support text and binary data cookies. A text and a binary data cookie with the same section and entry values may coexist and be read and written independently. However calling delete cookie will delete all cookies matching the section and entry values. It is recommended not to overload cookies.

Each type of cookie supported by U3 devices support their own name spaces. Any other type of cookie that has the same section and entry name values will refer to the cookie of that type and not the standard cookies creates with these functions.

Cookie Data Limitations

The following data limitations are defined for cookies:

Table 21: Standard Cookie Data Limitations

Data	Limit
Section name length	256 16-bit Unicode characters
Entry name length	256 16-bit Unicode characters
Cookie data size	Limited to available space in the cookie store (Hidden Area)

Cookie Functions

The following cookie function commands are supported:

- dapiWriteTextCookie
- dapiWriteBinaryCookie
- dapiReadTextCookie
- dapiReadBinaryCookie
- dapiDeleteCookie

6.1 dapiWriteTextCookie

Description

Creates and populates a standard text cookie in the cookie area. If the cookie exists then the existing data will be overwritten.

Syntax

```
HRESULT   dapiWriteTextCookie(HDEVICE    hDev,
                        const wchar_t * szSection,
                        const wchar_t * szEntry,
                        const wchar_t * szValue)
```

Input Parameters

`hDev`

The handle to the device.

`szSection`

The null terminated Unicode section name of the cookie. Max length is 256.

`szEntry`

The null terminated Unicode entry name of the cookie. Max length is 256.

`szValue`

The null-terminated Unicode string to write to the cookie.

Output Parameters

none

Return Values

Table 22: dapiWriteTextCookie Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szSection or szEntry values are too long, null or empty.
E_POINTER	0x80004003	szValue is null
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_HIDDEN_AREA_FULL	0x80040230	No more room to write the cookie.
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	Device does not support standard cookies (DAPI_CAP_U3_COOKIE)

Remarks

A text cookie and a binary cookie can be written to the same section and entry.

See Also

dapiReadTextCookie, dapiDeleteCookie

6.2 dapiWriteBinaryCookie

Description

Creates a standard cookie with binary data in the cookie area. If the cookie exists then the existing data will be overwritten.

Syntax

```
HRESULT   dapiWriteBinaryCookie(HDEVICE  hDev,
                          const wchar_t * szSection,
                          const wchar_t * szEntry,
                          const BYTE    * pBuffer,
                                DWORD     nBufLength)
```

Input Parameters

`hDev`

The handle to the device.

`szSection`

The null terminated Unicode section name of the cookie. Max length is 256.

`szEntry`

The null terminated Unicode entry name of the cookie. Max length is 256.

`pBuffer`

Pointer to the buffer containing raw binary data.

`nBufLength`

Size in bytes of data to be copied from the buffer pointed to by pBuffer.

Output Parameters

none

Return Values

Table 23: dapiWriteBinaryCookie Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szSection or szEntry values are too long, null or empty.
E_POINTER	0x80004003	pBuffer is null
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_HIDDEN_AREA_FULL	0x80040230	No more room to write the cookie.
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	Device does not support standard cookies (DAPI_CAP_U3_COOKIE)

Remarks

A text cookie and a binary cookie can be written to the same section and entry.

See Also

dapiReadBinaryCookie, dapiDeleteCookie

6.3 dapiReadTextCookie

Description

Reads a data from text cookie in the cookie area. This function can be called to first determine the size of the buffer required to hold the text data.

Syntax

```
HRESULT   dapiReadTextCookie(HDEVICE    hDev,
                       const wchar_t * szSection,
                       const wchar_t * szEntry,
                       const wchar_t * szValue,
                             DWORD   * nBufLength)
```

Input Parameters

`hDev`

The handle to the device.

`szSection`

The null terminated Unicode section name of the cookie. Max length is 256.

`szEntry`

The null terminated Unicode entry name of the cookie. Max length is 256.

`szValue`

Pointer to a Unicode string buffer. To determine the size of the required buffer set this value to NULL.

`nBufLength`

Pointer to variable holding the size of the szValue buffer. The string length of zero should be passed, along with a NULL szValue in order to receive the required buffer length.

Output Parameters

`szValue`

Pointer to the returned Unicode string if valid for the operation.

`nBufLength`

The length of the Unicode string copied to the buffer. If this was specified as zero and szValue was set to NULL when the function was called then this will contain the required buffer size to hold the cookie data including the null terminating character.

Return Values

Table 24: dapiReadTextCookie Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szSection or szEntry values are too long, null or empty.

HRESULT	Value	Description
E_POINTR	0x80004003	szValue is null and nBufLength > 0
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	Device does not support standard cookies (DAPI_CAP_U3_COOKIE)
DAPI_E_DATA_NOT_FOUND	0x80040231	The cookie could not be found
DAPI_E_BUFFER_TOO_SMALL	0x80040240	The data to be copied is larger then the supplied buffer size.

Remarks

To determine the size of the required buffer, call the function with nBufLength = 0 and szValue=NULL. The required buffer length will be return in the nBufLength parameter.

See Also

dapiCallback

Example

Calling the function in C++:

```
DWORD nBufLen = 0;
HRESULT res;
wchar_t * pBuffer = NULL;

// determine the required buffer length by calling with NULL szBuffer and
nBuflen = 0.
res = dapiReadTextCookie(hConnectedDev, szSection,
    szEntry, NULL,
    &nBufLen)

if(SUCCEEDED(res))
{
pBuffer = new wchar_t[nBufLen];
wmemset(pBuffer, _TCHAR("\0"),nBufLen);
res = dapiReadTextCookie(hConnectedDev, szSection, szEntry, pBuffer,
                         &nBufLen);
}

// process HRESULT and use cookie data

delete [] pBuffer;
```

6.4 dapiReadBinaryCookie

Description

Reads data from a binary cookie in the cookie area. This function can be called to first determine the size of the buffer required to hold the binary data.

Syntax

```
HRESULT   dapiReadBinaryCookie(HDEVICE    hDev,
                         const wchar_t * szSection,
                         const wchar_t * szEntry,
                               BYTE    * pBuffer,
                               DWORD   * nBufLength)
```

Output Parameters

`hDev`

The handle to the device.

`szSection`

The null terminated Unicode section name of the cookie. Max length is 256.

`szEntry`

The null terminated Unicode entry name of the cookie. Max length is 256.

`pBuffer`

Pointer to a byte array of minimum size nBufLength to receive the binary cookie data. To determine the size of the required buffer set this value to NULL.

`nBufLength`

Pointer to variable holding the size of pBuffer. A buffer of zero should be passed, along with a NULL pBuffer pointer in order to receive the required buffer size.

Input Parameters

`pBuffer`

Pointer to the returned binary data buffer if valid for the operation.

`nBufLength`

The size of the data to the buffer. If set to zero and pBuffer is NULL, then the required size of the buffer in bytes will be returned.

Return Values

Table 25: dapiReadBinaryCookie Return Values

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
E_INVALIDARG	0x80070057	szSection or szEntry values are too long, null or empty.
E_POINTER	0x80004003	pBuffer is null

HRESULT	Value	Description
E_HANDLE	0x80070006	The device handle is not valid
DAPI_E_DATA_NOT_FOUND	0x80040231	The cookie could not be found
DAPI_E_CMD_NOT_SUPPORTED	0x80040280	Device does not support standard cookies (DAPI_CAP_U3_COOKIE)
DAPI_E_BUFFER_TOO_SMALL	0x80040240	The data to be copied is larger then the supplied buffer size.

See Also

dapiDeleteCookie, dapiWriteBinaryCookie

6.5 dapiDeleteCookie

Description

Deletes a single standard cookie entry or all standard cookies in a section. If no entry name is supplied, all standard cookies in the section will be deleted.

If a text and binary cookie both have the same section and entry values, both cookies will be deleted.

Syntax

```
HRESULT   dapiDeleteCookie(HDEVICE    hDev,
                    const wchar_t * szSection,
                    const wchar_t * szEntry)
```

Input Parameters

`hDev`

The handle to the device.

`szSection`

The null terminated Unicode section name of the cookie. Max length is 256.

`szEntry`

The null terminated Unicode entry name of the cookie. Max length is 256. May be NULL to signify delete section.

Output Parameters

`none`

Return Values

Table 26: dapiDeleteCookie Return Values

HRESULT	Value	Description
S_OK	00000000	Command succeeded
E_INVALIDARG	80070057	szSection values are too long, null or empty. szEntry may be too long or empty.
E_HANDLE	80070006	The device handle is not valid
DAPI_E_DATA_NOT_FOUND	80040231	The cookie could not be found
DAPI_E_CMD_NOT_SUPPORTED	80040280	Device does not support standard cookies (DAPI_CAP_U3_COOKIE)

7 DAPI HRESULT Codes

This chapter contains listing of DAPI return codes.

7.1 HRESULT Codes

Table 27: HRESULT Codes

HRESULT	Value	Description
S_OK	0x00000000	Command succeeded
S_FALSE	0x00000001	Command succeeded however the result is negative.
E_POINTER	0x80004003	Null pointer parameter
E_FAIL	0x80004005	The command was unable to be executed.
E_HANDLE	0x80070006	Invalid handle parameter.
E_INVALIDARG	0x80070057	One of the function arguments is invalid.
DAPI_E_STATE_ERR	0x80040200	The current command cannot be executed as the current state of the device will not support the command. Try again.
DAPI_E_CONFIGURATION_ERR	0x80040201	The configuration options supplied are either not valid or not supported.
DAPI_E_PATH_ERR	0x80040202	The path does not match domain on the device.
DAPI_E_GENERAL_ERR	0x80040203	A recoverable error has occurred.
DAPI_E_INTERNAL_ERR	0x80040204	A recoverable error has occurred.
DAPI_E_STRING_CONVERSION_ERR	0x80040205	An unsupported non Unicode strings was provided.
DAPI_E_DOMAIN_TYPE_ERR	0x80040206	The domain referenced by the path argument is not a removable domain.
DAPI_E_DOMAIN_PATH_ERR	0x80040207	The supplied path does not point to a valid domain for the operation.
DAPI_E_COMM_FAIL	0x80040210	General communications error
DAPI_E_MEDIA_LOCK_FAIL	0x80040212	The device could not be locked.

HRESULT	Value	Description
DAPI_E_FORMAT_FAIL	0x80040213	DAPI was unable to format the private area and reconfiguring the area. The configuration succeeded, however the private area must be manually formatted.
DAPI_E_HINT_FAIL	0x80040214	Error reading the hint data. Remaining data is still valid.
DAPI_E_FS_REFRESH_FAIL	0x80040216	DAPI was unable to refresh the file system information
DAPI_E_NOT_CONNECTED	0x80040220	The device is not connected.
DAPI_E_NOT_READY	0x80040221	The device is either not initialized, or has not yet completed a previous operation.
DAPI_E_NOT_WRITABLE	0x80040222	The target area or device is read only.
DAPI_E_PA_CONFIGURED	0x80040223	The private area was already configured.
DAPI_E_CD_LOCKED	0x80040224	The operation could not be performed on the CD-ROM domain. Credentials must be supplied to unlock the CD-ROM first.
DAPI_E_HIDDEN_AREA_FULL	0x80040230	No more room available to write the cookie type to the hidden area.
DAPI_E_DATA_NOT_FOUND	0x80040231	The cookie could not be found
DAPI_E_OPERATION_NOT_COMPLETED	0x80040232	The operation was not completed. Try again.
DAPI_E_READ_PWD_ERR	0x80040234	Bad read password
DAPI_E_WRITE_PWD_ERR	0x80040235	Bad write password
DAPI_E_COOKIE_EXISTS	0x80040236	The cookie already exists
DAPI_E_COOKIE_CONFIG_ERR	0x80040237	The combination of null read and write passwords is not supported
DAPI_E_PASSWORD_TOO_LONG	0x80040238	Password is too long
DAPI_E_BUFFER_TOO_LARGE	0x80040239	The data to be written is larger then the allocated space.
DAPI_E_BUFFER_TOO_SMALL	0x80040240	The data to be copied is larger then the destination buffer size.
DAPI_E_CMD_NOT_SUPPORTED	0x80040241	The device is not U3; device does not support capability; the function is no longer supported by DAPI.

HRESULT	Value	Description
DAPI_E_CMD_NOT_SUPPORTED_IN_NON_ADMIN_MODE	0x80040242	The command cannot be executed in user mode. Run this command with administrator privileges.
DAPI_E_OS_NOT_SUPPORTED	0x80040250	The operating system is not supported by DAPI.
DAPI_E_WIN_2K_SP4_HOTFIX_MISSING	0x80040251	The latest Windows 2000 Service Pack 4 must be applied to this machine.
DAPI_E_ADV_ERR	0x80040260	Advanced communications error

U3 Platform 1.0 SDK, DAPI Reference Guide

U3
303 Twin Dolphin Drive, 6th Floor
Redwood City, CA 94065
1-800-837-3654, info@u3.com

For more information, visit www.u3.com