# CDInterfaceModule Low Level Design

## Overview

Within the key generation suite of applications, there is a need to write various files out to CDR Media and to be able to interrogate the status of media in the drive.

Entrust needs to be able to interrogate the status of currently loaded media and to copy a specified directory to that media. It also needs to discover the drive letter the media is mounted on and eject the media.

The ECKG application which will be supplied as part of the suite has similar needs to Entrust. It will also need to guide the user through the process of creating two CDs for each key set.

## Implementation

Microsoft Powershell has been selected as the software language to provide the required functionality. The language is well supported, and the code relatively easy to follow. This has the added advantage that there is more knowledge within existing support teams for Powershell scripts than for the previous version of this application, which was implemented in C#.

To write files to CDR media on a Windows server, the application uses the Microsoft IMAPI interface. This provides the core set of functions needed to interact with the media and the drive the media is loaded into. The script wraps the IMAPI calls with error checking and feedback logic.

For a detailed understanding of the IMAPI interface, readers are recommended to follow the Microsoft documentation at <https://docs.microsoft.com/en-us/windows/win32/imapi/portal> (should they change the location of those documents – search for ‘Image Mastering API’)

## Powershell

The functionality is provided as a Powershell module, with a psm1 extension, and a module manifest (psd1 extension) is created when the module is deployed/installed.

Powershell will autoload modules that are located in directories listed in the PSModulePath ($env:PSModulePath), when a reference is made to a function from that module. This means that by using ‘CDInterface’ in a session you trigger this autoload action and are able to use the command. Powershell refers to these kind of scripts as ‘CmdLets’.

Modules for use by ‘All Users’ are located in C:\Program Files\Windows Powershell\Modules.

By structuring the script in this way any calling applications don’t need to know the location of the script code to use it. It also means that the functionality is visible to Powershell functions such as ‘Get-Module -ListAvailable’.

New versions of the script can be released and deployed to servers, and Powershell will ensure users get the latest version via autoload.

The module also makes use of the built-in ’Write-Verbose’ CmdLet to provide additional output as it goes through the steps. This is enabled using the -verbose argument. It is aimed at users running the script from the command line.

The script is enabled as an advanced script, so can also be passed common arguments implemented by Powershell. The ‘Show-Command CDInterface’ CmdLet brings up a GUI dialog which allows all the possible arguments to be viewed. The command line needed to run CDInterface can then be created by populating that dialog. This would be useful for first time users, and those that need to use it infrequently, alongside the help page.

## Media writing with IMAPI

For a full understanding of IMAPI the Microsoft documentation is the best resource, but the information presented here should suffice for gaining a general picture of the application and how it is structured.

This document will start with a logical picture of the overall process of media writing, and then provide further detail of each step in that process.

The 2nd version of the IMAPI interface is being used, which has been current since around 2008. The first version was much less feature rich. In some cases the version appears in the names used for objects in the API.

The main action implemented by the script is to burn a CD with files, the logical steps are as follows

1. Create an in memory file system (FS)
2. Copy the files you want on your media to FS
3. Build an ISO image of FS in memory
4. Connect to the writeable media
5. Write the ISO file to the media

The steps involve manipulating 4 objects exposed through to the script by IMAPI.

* IMAPI2.MsftDiscMaster2 – Exposes a list of writeable drives
* IMAPI2.MsftDiscRecorder2 – Exposes an individual drive
* IMAPI2.MsftDiscFormat2Data – Exposes the media in a particular drive
* IMAPI2FS.MsftFileSystemImage – Exposes the in memory file system

These objects are then used by the script to follow the steps required to burn a CD, and provide the other functionality needed.

## Logical structure

The application follows this sequence

* Collect and validate user input
* Determine the Action
* Complete the Action

The main logic for achieving this is implemented in the CDInterface function. The script also contains a few small additional functions to group together logic and keep the flow within the main function clearer. The functions in the script are

|  |  |
| --- | --- |
| Function | Purpose |
| Get-Usage | Displays Usage information |
| Write-Response | Returns data to the caller |
| Get-version | Returns the version of the script |
| CDInterface | The main function |

The CDInterface function is exported from the script, which makes it available to use in a Powershell session.

The script follows a single execution path. As it follows this execution path, it checks when it has the information for completing a given action and returns the appropriate response.

Errors can occur throughout the execution path, for errors checked for in the script code itself and ‘ERROR’ response is generated along with an explanatory line. The explanatory output line can be suppressed.

Powershell can also detect errors, and the error output in these cases is presented to the caller. Generally the script should handle normal runtime errors in a controlled way (by using an ERROR response) but user data entry errors can generate native Powershell output.

If there are no errors then a success response is returned, also with an explanatory line.

The sequence described above will now be considered in more detail.

### Collect and validate

Powershell provides good support for command line arguments. Two main ways are used to collect user input, Switches and Variables.

* Switches – a switch is either present or not on the command line. E.g. -production
* Variables – Variable arguments must always have a value. E.g. -cdlabel “myLabel”

Powershell itself handles validation of these arguments based on the way they are configured in the script.

There is one piece of additional logic, with the arguments, that the script handles itself. This is that when the user specifies the -writetomedia argument they must also specify a -cdlabel argument.

A number of separate operations are provided by the script, we can group these into Actions and Options where

* ACTIONS – Define an operation to be carried out.
* OPTIONS – Provide additional information for the Actions.

#### Actions

|  |  |  |  |
| --- | --- | --- | --- |
| Action | Argument | Options | Description |
| Get Drive State | -getdrivestate | -recorderIndex | Get the status of the drive and the media loaded |
| Copy to Media | -writetomedia | -cdlabel  -recorderIndex |  |
| Drive Letter | -driveletter | -recorderIndex | Display the first character of the drive mount point |
| Show Version | -version |  | Displays the version of the module |
| List drives | -list |  | Display a list of writeable drives on the server. |
| Get media type | -getmediatype | -recorderIndex | Display the type of media loaded |
| Get Media Types | -getmediatypelist |  | Display a list of media types |
| Show Usage | -help |  | Displays Available Actions and Options. |
| Drive Letter | -driveletter | -recorderIndex | Displays the drive letter the drive is mounted on |

#### Options

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Option | Argument | Works for | Description | Default |
| CD Label | -cdlabel | -writetomedia | The label to use on the media |  |
| Verbosity | -verbose | All Actions | Enable more detailed output | Non-Verbose |
| Single line response | -onlysingleline | All Actions | Suppresses output of the 2nd informational line |  |

### Determine the Action

The arguments passed in to the script, are defined in the param() variable. Powershell sets these internal variables based on the command line, along with any rules that have been specified in the param() variable.

#### Switch parameters

A parameter defined as a [switch], for example $production, would set the $production variable to $true if the corresponding argument (-production) is present on the command line.

#### Variable parameters

A parameter defined as a [string], for example $cdlabel, would set the $cdlabel variable to the value provided after the corresponding argument (-cdlabel “MyLabel”). If the value is not present Powershell will generate an error. The variable is $false if the argument (-cdlabel) is not provided.

#### Settings parameters

The module includes a settings file, CDInterfaceModuleSettings.json. This is installed alongside the module file. The values in the file control some of the behavior of the module.

|  |  |  |
| --- | --- | --- |
| Setting | Description | Default |
| sizeOfSector | Use to calculate if the directory selected will fit onto a CD. | 1800 |
| recorderIndex | The index of the drive to use | 0 |
| production | If true CDInterface runs in Production mode. CDRs only. | True |
| mediaTypeForProduction | Defines the media type code in Production mode. Default 2 | 2 |
| noCloseMedia | Stop media being ‘closed’ | false |
| noEjectMediaAfterWrite | Stop the tray ejecting after write | false |

### Complete the Action

Based on the Action required the script will flow through a number of steps to achieve that action, a lot of these steps are common between the Actions.

To follow the execution path sequence, the Actions of the script will be described in the order that they are completed by the script.

The execution of the CDInterface function begins by setting up the parameters it uses, then checking it has been given an action. It then validates that it has all the options needed.

#### Show version

If CDInterface has been invoked with the -version argument then the script outputs the version information, by calling the Get-Version function, and finishes.

Typical output



#### Help page

If CDInterface is invoked with the -help argument, then the script outputs the help page, by calling the Get-Usage function, and finishes.

A typical output

Text

Description automatically generated

A similar response is also generated no arguments are given, but in addition it produces an ERROR response.

#### Get Media Type List

If CDInterface is invoked with the -getmediatypelist argument, then the script outputs the list of media types supported. This Action is intended to support use in development scenarios. The is a list held in the script itself covering common media types. The position of the media type in the list is also displayed, and is needed to set the mediaTypeForProduction setting.

A typical output

A picture containing shape

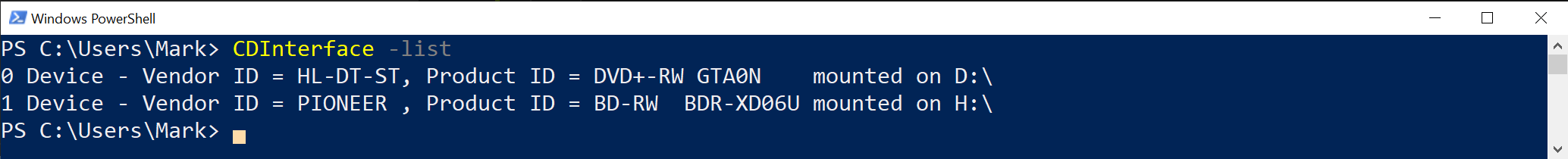
Description automatically generated

#### List Drives

The user specifies this Action with the -list argument.

This action needs to interact with a drive which is needed by most of the Actions. The script gets the list of available drives by instantiating an IMAPI2.MsftDiscMaster2 object. This appears to the script as an array of strings which contain a representation of the drive that Windows understands.

Typical output



Here the -list argument was used on a server with two writeable drives. The string values identify the manufacturer of the drive and the product identifier, along with the drive mount.

The script then needs to instantiate an IMAPI.MsftDiscRecorder2 object, and to initialize the object.

The script initializes the object by calling the InitializeDiscRecorder() method. Now the IMAPI2.MsftDiscRecorder2 object knows about the drive and has a connection to it.

It needs to do this for each drive to get the corresponding information about the drive.

The VendorId, ProductId and VolumePathNames properties of the recorder object, then provide the text for the output.

If you enable verbose output, you can see the full drive information in the internal format.

Text

Description automatically generated with low confidence

The purpose of the list option is to let the user select a particular drive to work with on other actions. The first field in the line (in the example 0 or 1) would be passed in using the -recorderIndex option to select a drive to work with.

By default an Action will always happen on the first available drive. This is controlled by the recorderIndex setting, which defaults to 0.

Setting recorderIndex to a drive that doesn’t exist generates an ERROR response.

If there are no writeable drives the script will generate an ERROR response.

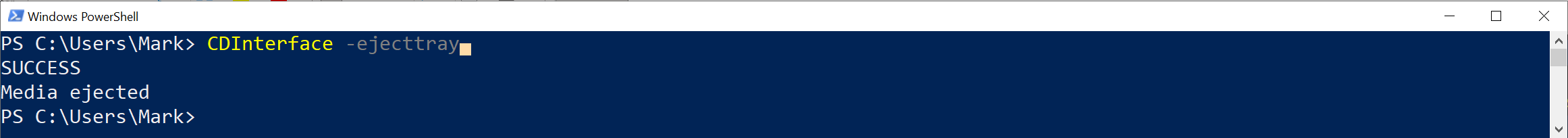
#### Eject Tray

This Action is specified with the -ejecttray argument. The script has control of the drive, and is able to eject the media, once it has instantiated the IMAPI2.MsftDiscMaster2 and IMAPI2.MsftDiscRecorder2 objects.

The script execution path at the point this action is checked for, is the same as that described above.

It has an initialized recorder object, and carries out that action by calling its EjectTray() method. It then provides the appropriate response and goes no further.

Typical output



#### Drive Letter

This Action is specified with the -driveletter argument.

Disk drives in Windows are mounted alongside the hard drive for example, and assigned a drive letter. Generally the drive letters are allocated in sequence. Historically floppy disc drives occupied A and B, then C was used for the first hard drive. In small systems D is then used for the first media drive and so on. Any network drives the server has connections to generally are selected by the user but Windows connects the hardware in the machine first.

This Action also requires an initialized IMAPI2.MsftDiscRecorder2 object. The mapping is provided by VolumePathNames property of the recorder object.

The Action completes by returning the first character of the drive mapping.

Typical output



#### Get Media Type

This Action is specified with the -getmediatype argument.

In development scenarios it is useful to be able to identify the type of media loaded into the drive. This Action and those that follow now need an additional IMAPI2.MsftDiscFormat2Data object instantiated. The script creates that and connects it to the recorder object.

The script now knows what type of media is present. It gets the media type code from the CurrentPhysicalMediaType property of the format object, and displays this code along with a string representation of the type.

Typical output



#### Get drive state

This Action is specified by passing the -getdrivestate argument.

At the point in the script where this Action is first checked for there are now 3 IMAPI objects instantiated and initialized. There are a number of ways the script determines the drive state.

The first thing it checks, using the CurrentPhysicalMediaType property of the format object to see if there is media loaded. For this Action the script would complete successfully with the NO\_DISC response at this point, if there is no media loaded.

The script continues if there is media loaded, and checks if it is actually Writeable media using the IsCurrentMediaSupported() method of the format object.

Following this a check is made using the value of the CurrentPhysicalMediaType property, based on the production setting, if the type of media loaded is allowed. In production environments only one type of media is supported – CDR. In non-production environments any writeable media that the drive itself supports is allowed for convenience.

When used in a production environment (the production setting is true) the script enforces use of Blank CDR media only. Internal to IMAPI this is media type 2. The type of media this logic checks for is controlled by the ‘mediaTypeForProduction’ setting. All media types supported by IMAPI can be used.

If the media typeloaded does not match the mediaTypeForProduction when running in Production mode, then INVALID\_MEDIA response is generated.

The final check is to make sure the media itself is blank, which is indicated by the MediaHeuristicallyBlank property. If the media is blank then a BLANK\_CD response is returned and execution finishes. If not empty a NON\_WRITEABLE\_MEDIA response is returned.

Example outputs

No disc loaded



Blank CD loaded



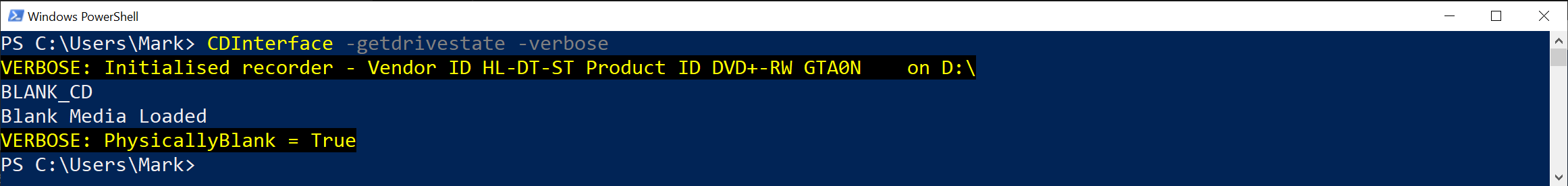
Unsupported media loaded



Wrong media type in production mode



By adding the -verbose option the output also shows if the media is physically blank, i.e. for erasable media, it has never been written to.



#### Copy to media

This Action is specified by using the -writetomedia argument, and also requires the -cdlabel argument.

The script flow for the previous Action, getdrivestate, did various checks for the type of media loaded. The only drive state that allows the Copying Action to continue during these checks is BLANK\_CD – if anything other than a BLANK\_CD is loaded the response will be ERROR and an explanation of what went wrong.

The same logic used to constrain the media type is run for this Action as used for -getdrivestate, the only difference is that incorrect media type, results in an ERROR response.

In order to write data to media, the script now needs an additional IMAPI object, to allow it to get the files to be copied into the right structure for writing. It instantiates an IMAPI2FS.MsftFileSystemImage object to do this.

The script sets the FileSystemsToCreate property to 7 which will ensure the widest read compatibility of the resulting media. This value specifies to IMAPI that it should create a disc image supported by systems that expect ISO9660, Joliet or UDF formatted discs. More detail on this can be found at <https://docs.microsoft.com/en-us/windows/win32/imapi/disc-formats>

It sets the VolumeName property to the value provided in the -cdlabel argument.

The script then uses the AddTreeWithNamedStreams() method of the Root property of the file system object to collect all the files contained within the path specified by the value of the -writetomedia argument into the file system object. This is creating an in memory copy of all the files that will be written. If errors occur during this process the ERROR response will be generated with an explanation.

All being well, an ISO image of the files in memory is now created using the CreateResultImage() method of the file system object. This step converts the data into the structure as it will appear on the media once it has been written.

File systems for files on a Hard drive in an operating system like windows, are generally laid out differently from the way they are on CD media. There is different meta data, defined by the file system, surrounding the file data itself. The ISO organization defines these file systems in detail. An ‘ISO image’ just refers to a single file (which if saved to disc would itself be a file with an .iso extension). The script doesn’t need to save this file, but will write its contents to the media instead.

ISO files are usually quite large files, so behind the scenes, Windows will stream the data out of the ISO image over to the drive itself via internal buffers. This is all hidden via the IMAPI interface, but the names of the methods used to achieve the write, reveal the streaming behaviour used.

Now that the script is ready to create the media, it makes a call to ensure no other process will disturb the write operation. It calls the AcquireExclusiveAccess() method on the recorder object, and passes it the client name of ‘CDInterface’ – any other process can then see which process is blocking its access to the drive.

The main work of writing to the media is now carried out by calling the Write() method on the format object, and passing it the ImageStream property of the file system object. The write process itself could now take several minutes depending on size of the data to be written.

Once the write operation completes, the script releases the lock it holds on the drive, using the ReleaseExclusiveAcccess() method of the recorder object.

The script now ‘Ejects’ the media, to be collected by the user, by calling the EjectTray() method of the recorder object. The eject stage can be bypassed using the -noejectafterwrite option.

If any errors occur during the write operation the ERROR response will be generated with an explanation.

Example outputs

Normal output

With -writetomedia specified without the -cdlabel option



## Other Information

### Verbosity

The -verbose argument can be added to all Actions, it will tell Powershell to output all the messages generated by Write-Verbose CmdLet in the script. This is generally only useful in a support environment.

### Responses

The script uses a common function, Write-Response, to send data back to the caller. In this way the actual format of the responses could be altered easily in one place rather than having to change it in many places in the script.

The script usually returns 2 lines of output, with the first line containing the response string, the second an explanatory message. The 2nd line can be suppressed with the -onlySingleLine argument. For example the -getdrivestate action normally responds



Adding the -onlysingleline argument suppresses the informational line



If used on the command line the output should be reasonably easy to understand, and if called from within another application, code can safely determine its behaviour using the contents of the first response line. The second line could be displayed to the user, if a UI is present for example.

### Settings file

A number of options will want to be fixed configuration when the module is used in a Production environment.

A settings file can be used to set the default values for these options.

The settings file, CDInterfaceModuleSettings.json is installed alongside the Module file.

### Invoking from Java

A command line application is also provided, showing how to invoke CDInterface from Java. The basic approach is to invoke a Powershell session and then pass that session the same arguments as you would on the command line. Output from the module is passed back to Java through the Standard Output.

The example code just echoes the modules responses to the screen, but in practice would be used to control the interaction between the user and the media.

With a Java JDK installed in your development environment, you can easily execute this code from Visual Studio Code by right-clicking in the source file and selecting ‘Run Java’.