|  |
| --- |
| Microsoft Surface |
| Driver Module Framework (DMF) |
| v2.4.1 |

|  |
| --- |
| Sam Tertzakian and Rajesh Gururaj  9-19-2018 |

**Document Change Log**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Author** | **Description** |
| November 18, 2017 | 2.0 | Sam Tertzakian | Initial version (derived from original DMF documentation). |
| January 2, 2018 | 2.0.8 | Sam Tertzakian | Incorporate changes by Charlie Arcuri. |
| January 2, 2018 | 2.0.9 | Sam Tertzakian | Incorporate changes by Sweta Ananth. |
| February 1, 2018 | 2.1.1 | Sam Tertzakian | Incorporate changes by Scott Durland. Add diagrams. |
| February 13, 2018 | 2.1.2 | Sam Tertzakian | Edit authors. |
| February 17, 2018 | 2.2.0 | Sam Tertzakian | Document Transport Modules. |
| March 8, 2018 | 2.2.1 | Sam Tertzakian | Update for reduced Client API surface and Feature Modules. |
| March 21, 2018 | 2.2.2 | Sam Tertzakian | Update for updated Module declaration macros. Update to correct errors/omissions in version 2.2.1. |
| April 5, 2018 | 2.3.0 | Sam Tertzakian | Update for Module Libraries. |
| July 23, 2018 | 2.3.1 | Sam Tertzakian | Updated for recent DMF changes. |
| August 10, 2018 | 2.3.3 | Sam Tertzakian | Updated to correct some errors. |
| September 12, 2018 | 2.4.0 | Sam Tertzakian | Updated to explain new APIs for creating Child Modules. Update examples to match current version of code. Update to add new APIs. |
| September 19, 2018 | 2.4.1 | Sam Tertzakian | Update for updated WPP tracing methods. |

**Document Conventions**

|  |  |
| --- | --- |
| **Convention** | **Meaning** |
| DMF | Driver Module Framework |
| WDF | Windows Driver Framework (KMDF or UMDF). |
| WDM | Windows Driver Model. The layered driver model that the Windows operating system uses to organize device drivers. |
| Module | A DMF Module. A set of code that has a specific structure that is compatible with the DMF framework. |
| Child Module | A Module that is instantiated by another Module. |
| Parent Module | A Module that instantiates a Child Module. |
| Client Driver | A WDF device driver that uses DMF. |
| Client | The user of a Module; the code being served by a Module. It can either be a Parent Module or a Client Driver. |
| Library | A library of DMF Modules. It differs from “library” which is an arbitrary .lib file which contains reusable code in a binary format. |
| Module Include File | An .h file that contains all the information needed for a Client to instantiate and use a Module. |
| Library Include File | An .h file that contains all the information needed for a Client to instantiate and use Modules in a specific Library. |

**Documentation Guide**

This document is part of a family of documents that explain DMF…

|  |  |
| --- | --- |
| **Document Title** | **Purpose** |
| *Device Driver Writing Considerations: An Introduction to DMF* | This document discusses issues device driver authors need to consider when writing device drivers. Furthermore, this document lays out the case for why DMF exists and why it can help device driver programmers. |
| *Driver Module Framework (DMF)* (this document) | This document explains the DMF API and how to use it. As you read this document, please have access to the source code so you can have a better view of how the APIs are used. |
| *BranchTrack* | This document explains what BranchTrack is and how to use it. (BranchTrack requires DMF.) As you read this document, please have access to the source code so you can have a better view of how the APIs are used. |

Contents

[1 What is the Driver Module Framework (DMF)? 11](#_Toc525141943)

[1.1 Goals of DMF 11](#_Toc525141944)

[1.2 How does DMF Achieve the Above Goals? 11](#_Toc525141945)

[2 How DMF Works in a Client Driver 13](#_Toc525141946)

[2.1 Differences between a DMF driver and a traditional (non-DMF) Driver. 13](#_Toc525141947)

[2.1.1 Traditional Driver 13](#_Toc525141948)

[2.1.2 DMF Driver 13](#_Toc525141949)

[2.1.3 Differences 14](#_Toc525141950)

[2.2 DMF Container Driver 15](#_Toc525141951)

[2.3 DMF Non-Container Driver 16](#_Toc525141952)

[2.4 Examples of Modules 17](#_Toc525141953)

[2.4.1 Buffers 17](#_Toc525141954)

[2.4.2 Data Structures 17](#_Toc525141955)

[2.4.3 Task Execution 17](#_Toc525141956)

[2.4.4 Targets 17](#_Toc525141957)

[2.4.5 User Notification 17](#_Toc525141958)

[2.4.6 Driver Patterns 17](#_Toc525141959)

[2.5 Properties of Modules 18](#_Toc525141960)

[2.5.1 DMFMODULE is of type WDFOBJECT 18](#_Toc525141961)

[2.5.2 Module is Object Oriented 18](#_Toc525141962)

[2.5.3 Modules Have a Predefined Lifecyle 18](#_Toc525141963)

[2.5.4 Modules Have a Hierarchy 18](#_Toc525141964)

[2.5.5 Properties of Modules 19](#_Toc525141965)

[2.6 Properties of a DMF Driver 20](#_Toc525141966)

[2.7 Using DMF in an Existing Driver or a Driver that has its own Device Context 21](#_Toc525141967)

[2.7.1 Hook DMF into the driver. 21](#_Toc525141968)

[2.7.2 Initialize DMF 24](#_Toc525141969)

[2.8 Using DMF in a Driver That Does Not Have Its Own Device Context 25](#_Toc525141970)

[2.9 Instantiating DMF Modules 27](#_Toc525141971)

[2.10 Instantiating Dynamic Modules 29](#_Toc525141972)

[2.10.1 Synchronous Dynamic Instantiation 29](#_Toc525141973)

[2.10.2 Asynchronous Notification Dynamic Instantiation 30](#_Toc525141974)

[2.10.3 Destroying a Dynamic Module 31](#_Toc525141975)

[2.11 Transport Modules 32](#_Toc525141976)

[2.12 Using Modules 33](#_Toc525141977)

[2.13 WPP Tracing Module Traces From A Client Driver 35](#_Toc525141978)

[2.13.1 Filtering WPP Tracing 36](#_Toc525141979)

[3 The Structure of a Module 37](#_Toc525141980)

[3.1 The Module .h File 38](#_Toc525141981)

[3.2 The Module .c File 40](#_Toc525141982)

[3.2.1 Section 1: File Header 40](#_Toc525141983)

[3.2.2 Section 2: DMF Include 41](#_Toc525141984)

[3.2.3 Section 3: WPP Definitions 41](#_Toc525141985)

[3.2.4 Section 4: Module Private Enumerations and Structures 41](#_Toc525141986)

[3.2.5 Section 5: Module Private Context 42](#_Toc525141987)

[3.2.6 Section 6: Module Macros 43](#_Toc525141988)

[3.2.7 Section 7: Module Private Code 44](#_Toc525141989)

[3.2.8 Section 8: Module WDF Callbacks 46](#_Toc525141990)

[3.2.9 Section 9: Module DMF Callbacks 51](#_Toc525141991)

[3.2.10 Section 10: Module Descriptors 54](#_Toc525141992)

[3.2.11 Section 11: Public Calls by Client (Includes Module Create Function) 55](#_Toc525141993)

[3.3 The Module \_Public.h File 57](#_Toc525141994)

[3.4 The Module .txt File 58](#_Toc525141995)

[3.5 The Module .mc File 61](#_Toc525141996)

[3.6 The Module’s Create Function 62](#_Toc525141997)

[3.6.1 Contents of a Module’s Create Function 62](#_Toc525141998)

[3.6.2 Annotated Module Create function 63](#_Toc525141999)

[3.7 How to Create One or More Child Modules 65](#_Toc525142000)

[3.7.1 Contents of a Module’s ChildModulesAdd Callback 65](#_Toc525142001)

[3.7.2 Annotated ChildModulesAdd Callback 66](#_Toc525142002)

[4 DMF Library Include File 68](#_Toc525142003)

[4.1 Library Include File 68](#_Toc525142004)

[4.1.1 Using the Library Include File 68](#_Toc525142005)

[4.1.2 Sample Library Include File 69](#_Toc525142006)

[4.2 Module Dependencies 70](#_Toc525142007)

[5 DMF Client Driver API Reference 71](#_Toc525142008)

[5.1 Client Driver DMF Structures 72](#_Toc525142009)

[5.1.1 PDMFDEVICE\_INIT 73](#_Toc525142010)

[5.1.2 DMF\_EVENT\_CALLBACKS 74](#_Toc525142011)

[5.2 Client Driver DMF Initialization Macros 75](#_Toc525142012)

[5.2.1 DMF\_DEFAULT\_DRIVERENTRY 76](#_Toc525142013)

[5.2.2 DMF\_DEFAULT\_DEVICEADD 77](#_Toc525142014)

[5.2.3 DMF\_DEFAULT\_DRIVERCLEANUP 78](#_Toc525142015)

[5.3 Client Driver DMF Initialization Functions 79](#_Toc525142016)

[5.3.1 DMF\_DmfDeviceInitAllocate 80](#_Toc525142017)

[5.3.2 DMF\_DmfDeviceInitFree 82](#_Toc525142018)

[5.3.3 DMF\_DmfDeviceInitHookPnpPowerEventCallbacks 83](#_Toc525142019)

[5.3.4 DMF\_DmfDeviceInitHookPowerPolicyEventCallbacks 84](#_Toc525142020)

[5.3.5 DMF\_DmfDeviceInitHookFileObjectConfig 85](#_Toc525142021)

[5.3.6 DMF\_DmfDeviceInitHookQueueConfig 86](#_Toc525142022)

[5.3.7 DMF\_DmfDeviceInitSetEventCallbacks 87](#_Toc525142023)

[5.3.8 DMF\_ModulesCreate 88](#_Toc525142024)

[5.4 Client Driver DMF Callbacks 89](#_Toc525142025)

[5.4.1 EVT\_DMF\_DEVICE\_MODULES\_ADD 89](#_Toc525142026)

[6 DMF Client API Reference 91](#_Toc525142027)

[6.1 DMF Client Structures 92](#_Toc525142028)

[6.1.1 DMF\_CONFIG\_[ModuleName] 92](#_Toc525142029)

[6.1.2 DMF\_MODULE\_ATTRIBUTES 92](#_Toc525142030)

[6.1.3 DMF\_MODULE\_EVENT\_CALLBACKS 93](#_Toc525142031)

[6.2 DMF Client API 94](#_Toc525142032)

[6.2.1 DMF\_CONFIG\_[ModuleName]\_AND\_ATTRIBUTES\_INIT 94](#_Toc525142033)

[6.2.2 DMF\_[ModuleName]\_AND\_ATTRIBUTES\_INIT 95](#_Toc525142034)

[6.2.3 DMF\_MODULE\_EVENT\_CALLBACKS\_INIT 96](#_Toc525142035)

[6.2.4 DMF\_DmfModuleAdd 97](#_Toc525142036)

[6.2.5 DMF\_ModuleDestroy 100](#_Toc525142037)

[7 DMF Module API Reference 101](#_Toc525142038)

[7.1 Module Enumerations 102](#_Toc525142039)

[7.1.1 DmfModuleOpenOption 102](#_Toc525142040)

[7.2 Module Structures 104](#_Toc525142041)

[7.2.1 DMF\_MODULE\_DESCRIPTOR 104](#_Toc525142042)

[7.2.2 DMF\_ENTRYPOINTS\_DMF 106](#_Toc525142043)

[7.2.3 DMF\_ENTRYPOINTS\_WDF 107](#_Toc525142044)

[7.3 Module WDF Callbacks 109](#_Toc525142045)

[7.3.1 DMF\_[ModuleName]\_ModulePrepareHardware 110](#_Toc525142046)

[7.3.2 DMF\_[ModuleName]\_ModuleReleaseHardware 111](#_Toc525142047)

[7.3.3 DMF\_[ModuleName]\_ModuleD0Entry 112](#_Toc525142048)

[7.3.4 DMF\_[ModuleName]\_ModuleD0EntryPostInterruptsEnabled 113](#_Toc525142049)

[7.3.5 DMF\_[ModuleName]\_ModuleD0Exit 114](#_Toc525142050)

[7.3.6 DMF\_[ModuleName]\_ModuleD0ExitPreInterruptsDisabled 115](#_Toc525142051)

[7.3.7 DMF\_[ModuleName]\_ModuleDeviceIoControl 116](#_Toc525142052)

[7.3.8 DMF\_[ModuleName]\_ModuleInternalDeviceIoControl 117](#_Toc525142053)

[7.3.9 DMF\_[ModuleName]\_ModuleSelfManagedIoCleanup 118](#_Toc525142054)

[7.3.10 DMF\_[ModuleName]\_ModuleSelfManagedIoFlush 119](#_Toc525142055)

[7.3.11 DMF\_[ModuleName]\_ModuleSelfManagedIoInit 120](#_Toc525142056)

[7.3.12 DMF\_[ModuleName]\_ModuleSelfManagedIoSuspend 121](#_Toc525142057)

[7.3.13 DMF\_[ModuleName]\_ModuleSelfManagedIoRestart 122](#_Toc525142058)

[7.3.14 DMF\_[ModuleName]\_ModuleSurpriseRemoval 123](#_Toc525142059)

[7.3.15 DMF\_[ModuleName]\_ModuleQueryRemove 124](#_Toc525142060)

[7.3.16 DMF\_[ModuleName]\_ModuleQueryStop 125](#_Toc525142061)

[7.3.17 DMF\_[ModuleName]\_ModuleRelationsQuery 126](#_Toc525142062)

[7.3.18 DMF\_[ModuleName]\_ModuleUsageNotificationEx 127](#_Toc525142063)

[7.3.19 DMF\_[ModuleName]\_ModuleArmWakeFromS0 128](#_Toc525142064)

[7.3.20 DMF\_[ModuleName]\_ModuleDisarmWakeFromS0 129](#_Toc525142065)

[7.3.21 DMF\_[ModuleName]\_ModuleWakeFromS0Triggered 130](#_Toc525142066)

[7.3.22 DMF\_[ModuleName]\_ModuleArmWakeFromSxWithReason 131](#_Toc525142067)

[7.3.23 DMF\_[ModuleName]\_ModuleDisarmWakeFromSx 132](#_Toc525142068)

[7.3.24 DMF\_[ModuleName]\_ModuleWakeFromSxTriggered 133](#_Toc525142069)

[7.3.25 DMF\_[ModuleName]\_ModuleFileCreate 134](#_Toc525142070)

[7.3.26 DMF\_[ModuleName]\_ModuleFileCleanup 135](#_Toc525142071)

[7.3.27 DMF\_[ModuleName]\_ModuleFileClose 136](#_Toc525142072)

[7.3.28 DMF\_[ModuleName]\_ModuleQueueIoRead 137](#_Toc525142073)

[7.3.29 DMF\_[ModuleName]\_ModuleQueueIoWrite 138](#_Toc525142074)

[7.4 Module DMF Callbacks 139](#_Toc525142075)

[7.4.1 DMF\_[ModuleName]\_ChildModulesAdd 140](#_Toc525142076)

[7.4.2 DMF\_[ModuleName]\_ResourcesAssign 141](#_Toc525142077)

[7.4.3 DMF\_[ModuleName]\_Open 142](#_Toc525142078)

[7.4.4 DMF\_[ModuleName]\_Close 145](#_Toc525142079)

[7.4.5 DMF\_[ModuleName]\_NotificationRegister 147](#_Toc525142080)

[7.4.6 DMF\_[ModuleName]\_NotificationUnregister 149](#_Toc525142081)

[7.4.7 DMF\_[ModuleName]\_Destroy 152](#_Toc525142082)

[7.5 Module API 153](#_Toc525142083)

[7.5.1 The Module Create Function 154](#_Toc525142084)

[7.5.2 DECLARE\_DMF\_MODULE 155](#_Toc525142085)

[7.5.3 DECLARE\_DMF\_MODULE\_NO\_CONFIG 156](#_Toc525142086)

[7.5.4 DMF\_ENTRYPOINTS\_DMF\_INIT 157](#_Toc525142087)

[7.5.5 DMF\_ENTRYPOINTS\_WDF\_INIT 158](#_Toc525142088)

[7.5.6 DMF\_MODULE\_DESCRIPTOR\_INIT 159](#_Toc525142089)

[7.5.7 DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT 161](#_Toc525142090)

[7.5.8 DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE 162](#_Toc525142091)

[7.5.9 DMF\_CONFIG\_GET 163](#_Toc525142092)

[7.5.10 DMF\_CONTEXT\_GET 164](#_Toc525142093)

[7.5.11 DMF\_ModuleCreate 165](#_Toc525142094)

[7.5.12 DMF\_ModuleOpen 166](#_Toc525142095)

[7.5.13 DMF\_ModuleClose 167](#_Toc525142096)

[7.5.14 DMF\_ModuleAcquire 168](#_Toc525142097)

[7.5.15 DMF\_ModuleRelease 169](#_Toc525142098)

[7.5.16 DMF\_ModuleDestroy 170](#_Toc525142099)

[7.5.17 DMF\_[ModuleName]\_TransportMethod 171](#_Toc525142100)

[8 Feature Module Access API 172](#_Toc525142101)

[8.1.1 DMF\_FeatureModuleGetFromDevice 173](#_Toc525142102)

[8.1.2 DMF\_FeatureModuleGetFromModule 174](#_Toc525142103)

[9 General DMF API 175](#_Toc525142104)

[9.1 Module Parent Device and Client Context 176](#_Toc525142105)

[9.1.1 DMF\_ParentDeviceGet 177](#_Toc525142106)

[9.2 Portable API 178](#_Toc525142107)

[9.2.1 Portable API Structures and Enumerations 179](#_Toc525142108)

[9.2.2 DMF\_Portable\_EventCreate 180](#_Toc525142109)

[9.2.3 DMF\_Portable\_EventSet 181](#_Toc525142110)

[9.2.4 DMF\_Portable\_EventReset 182](#_Toc525142111)

[9.2.5 DMF\_Portable\_EventWaitForSingle 183](#_Toc525142112)

[9.2.6 DMF\_Portable\_EventClose 184](#_Toc525142113)

[9.2.7 DMF\_Portable\_LookasideListCreate 185](#_Toc525142114)

[9.2.8 DMF\_Portable\_LookasideListCreateMemory 186](#_Toc525142115)

[9.3 Utility API 187](#_Toc525142116)

[9.3.1 DMF\_Utility\_AclPropagateInDeviceStack 188](#_Toc525142117)

[9.3.2 DMF\_Utility\_ActivityIdFromDevice 189](#_Toc525142118)

[9.3.3 DMF\_Utility\_ActivityIdFromRequest 190](#_Toc525142119)

[9.3.4 DMF\_Utility\_DelayMilliseconds 191](#_Toc525142120)

[9.3.5 DMF\_Utility\_EventLogEntryWriteDriverObject 192](#_Toc525142121)

[9.3.6 DMF\_Utility\_EventLogEntryWriteDriver 194](#_Toc525142122)

[9.3.7 DMF\_Utility\_EventLogEntryWriteDevice 196](#_Toc525142123)

[9.3.8 DMF\_Utility\_EventLogEntryWriteDmfModule 198](#_Toc525142124)

[9.3.9 DMF\_Utility\_EventLogEntryWriteUserMode 200](#_Toc525142125)

[9.3.10 DMF\_Utility\_IsEqualGUID 201](#_Toc525142126)

[9.3.11 DMF\_Utility\_UserModeAccessCreate 202](#_Toc525142127)

[10 DMF Coding Conventions 203](#_Toc525142128)

[10.1 Conventions 203](#_Toc525142129)

[10.1.1 Do: 203](#_Toc525142130)

[10.1.2 Do Not: 203](#_Toc525142131)

[11 Additional Information 204](#_Toc525142132)

[11.1 Git Repository 204](#_Toc525142133)

[11.2 Contributors 204](#_Toc525142134)

# What is the Driver Module Framework (DMF)?

DMF is a framework designed for Windows device driver developers.

## Goals of DMF

The goals of DMF are:

* Give programmers a framework so they can design and implement encapsulated blocks of code, termed **Modules,** inside their device drivers. These Modules act like small device drivers inside the driver that is built and installed.
* True code sharing made possible by reusable Modules. *Eliminate “copy (from original driver) and change (in new driver)” method of sharing code*.
* Related to the above item, one of the goals of DMF is to reduce the number of lines of code a programmer needs to write in a driver. The goal for programmers to only need to write “business logic” of the driver. In some cases, this number of lines of unique code (that performs the “business logic” of the driver) has been shown to be as low as **0.07% of the total lines of code in the driver**.
* Continue the layering architecture of Windows Kernel inside the device driver.
* Use object-oriented programming constructs without being forced to use an object-oriented programming language. Modules have attributes that are similar to attributes that C++ classes have: Private data, private methods, public methods, a constructor and destructor. Modules also support the idea of inheritance. However, they have additional attributes that make these “objects” suitable for device driver programming.
* Make it easier for individual contributors to work in parallel on different aspects of a given driver’s tasks. This is possible because each Module is totally self-contained, and the programmer only needs to consider that Module.
* Make device driver programming for Windows easier, cheaper and more fun, while at the same time increasing code quality.
* Help programmers address device driver development issues discussed in *Device Driver Consideration: An Introduction to DMF*.

## How does DMF Achieve the Above Goals?

DMF tries to achieve the goals above by doing the following:

* DMF, regardless of the Module’s functionality, provides a consistent interface to create and manage the Module.
* DMF allows programmers to reuse existing Modules directly, subclass Modules, modify existing Modules and create new Modules.
* DMF provides runtime lifetime management of the Modules as well as features like memory allocation, synchronization and others.
* DMF automatically dispatches all the callbacks from WDF into the device driver to each of its instantiated Modules.
* DMF allows Modules to easily contain other Modules which in turn can contain other Modules.
* DMF allows all of the driver’s functionality to be in one or more Modules. In this case, no driver specific code is needed.
* Every Module is directly usable by any driver that uses DMF or by another Module.
* DMF allows for a Module to have its own context (similar to a device drivers Device Context) and Module specific configuration.
* Modules can perform any function. A Module can expose data structures (e.g. hash table), programming patterns (e.g. continuous reader or IOCTL handling), devices (e.g. virtual keyboard) and hardware buses (e.g. I2c or HID).
* DMF is **not** a wrapper around WDF. DMF uses WDF to create objects (Modules) from common *driver patterns*.
* DMF makes it easy for common device driver programming patterns to be converted into objects, and thus Modules which can then be easily reused.

Henceforth, in this document…

* “Small driver” referenced above is called a **Module**. It can be considered to be an object per the object-oriented programming model.
* The “(device) driver” is called the Client Driver.
* A Client is **either** a Client Driver or another Module. A Module is agnostic about whether its parent is a Client Driver or another Module.

# How DMF Works in a Client Driver

This section provides a bird’s eye view of how DMF fits in the device driver ecosystem.

## Differences between a DMF driver and a traditional (non-DMF) Driver.

These diagrams pictorially show the differences between a hypothetical driver that uses DMF and a hypothetical driver that does not.

### Traditional Driver



### DMF Driver

|  |  |
| --- | --- |
|  |  |

*Note: The Modules listed above are a subset of the all the available Modules. Any Client Driver can use any number of Modules. Currently, about 50 Modules are available. Also, it is easy (and encouraged) for programmers to write new Modules.*

### Differences

This table lists the differences between the two drivers:

|  |  |  |
| --- | --- | --- |
| **Task** | **Traditional** | **DMF** |
| Data used by Driver (Device Context) | All data is held in the Device Context and accessible to all code. | Each Module can only access its own data. The Client specific code and other Modules cannot access other Modules’ data. Modules cannot access the Client Driver’s data. |
| Callbacks | Callbacks are sent by WDF to the Client Driver which then must execute proper code for each code block as needed. | Callbacks are sent by WDF to DMF. DMF dispatches those callbacks to each Module as well as the Client Driver. Each Module is responsible for handling the callback for its own needs. The Client Driver and other Modules do not have any idea if or how Modules handle callbacks. |
| Locking | The Client Driver is responsible for locking and synchronizing all the code blocks and the Device Context. The Client Driver is responsible for creating the locks. | The Client Driver is only responsible for locking its own code and data in its Device Context. Each Module has its own lock and is responsible for locking its own code and data. |
| Code Visibility | Any code can easily call any code block in any order. | Client Driver can **only** call the top-level Modules. Child Modules can **only** be called directly by immediate Parent Modules. |
| WDF/WDM access | Full access. | Full access. |
| Code reuse | Difficult and requires careful planning. Interfaces will most likely not be the same. | Easy and requires does no additional planning. Interfaces are already defined and always the same. |

Of course, it is possible to design a traditional driver that acts like a DMF driver. It requires careful planning and expert device driver programmers. Also, it requires a lot of code to be written that would not normally be present in a device driver. For this reason, drivers are rarely designed with all the above features that a DMF driver has.

## DMF Container Driver

In this mode, there is no specific Client Driver code other than to instantiate DMF Modules and potentially handle Module specific callbacks. There is no Device Context. The DMF Modules act *like* small drivers and perform all their work (driver’s “business logic”) individually without knowledge of the other Modules. All the Modules receive WDF callbacks as if they were a normal WDF driver. *Note: This type of Client Driver has* ***no*** *Device Context. However, each Module has its own Private Context.*

WDF (Lower Edge)

DMF Module(s) (1 to n)

WDF (Upper Edge)

DMF Framework

## DMF Non-Container Driver

In this mode, the Client Driver has a Device Context. The Client Driver instantiates Modules and uses them to perform work (device driver’s “business logic”). But there is also unique driver code that is not part of any Module. It is a classic WDF driver except that it uses DMF Modules to offload some of its code and work. *Note: This type of Client Driver has a Device Context*.

WDF (Upper Edge)

DMF Framework

DMF Module(s) (1 to n)

WDF (Lower Edge)

Unique Client Driver code

(In the above diagrams, the blue boxes contain code that performs the work (“business logic”) of the Client Driver.)

## Examples of Modules

Here are examples of Modules. This is only a partial list of generic Modules as the number of Modules keeps increasing. More Modules exist that perform more complex tasks that are driver/platform specific. For the complete list and an explanation of what each Module does, see the document, *DMF Modules Overview*.

### Buffers

* DMF\_BufferPool
* DMF\_BufferQueue
* DMF\_PingPongBuffer
* DMF\_RingBuffer
* DMF\_ThreadedBufferQueue

### Data Structures

* DMF\_HashTable

### Task Execution

* DMF\_QueuedWorkitem
* DMF\_ScheduledTask
* DMF\_Thread

### Targets

* DMF\_AcpiTarget
* DMF\_ContinuousRequestTarget
* DMF\_DeviceInterfaceTarget
* DMF\_GpioTarget
* DMF\_HidTarget
* DMF\_I2cTarget
* DMF\_ResourceHub
* DMF\_SelfTarget
* DMF\_SerialTarget
* DMF\_SpiTarget

### User Notification

* DMF\_NotifyUserWithEvent
* DMF\_NotifyUserWithRequest

### Driver Patterns

* DMF\_AcpiNotification
* DMF\_AlertableSleep
* DMF\_CrashDump
* DMF\_IoctlHandler
* DMF\_Pdo

## Properties of Modules

This is an overview of what DMF Modules are:

### DMFMODULE is of type WDFOBJECT

A Module is declared in code using DMFMODULE. Like WDFDEVICE which inherits from WDFOBJECT, DMFMODULE also inherits from WDFOBJECT. Therefore, DMFMODULE has all the characteristics of a WDFOBJECT such as Reference Count, Context Space, Parent Object and so on. In addition, functions that act upon WDFOBJECT also act upon DMFMODULE.

### Module is Object Oriented

Modules have private and public data and methods much like C++ objects. Module specifically contain:

* Module Config: A structure specific to the Module which allows the Client to configure the Module using Module specific parameters.
* Module Methods: Allows a Client to execute Module specific functions using Module and Method specific parameters.
* Module Callbacks: Callback functions from DMF and/or WDF into the Module so it can perform actions necessary at that time.
* Module Attributes: Indicates the Module’s parent.
* Private Context: Contains data that the Module needs while it is instantiated. This is similar to a device driver’s Device Context.
* Private Methods: Code which is internal to the Module and not exposed to Clients. These are private and are called by the Module’s Methods and internal callbacks.

### Modules Have a Predefined Lifecyle

* Create: DMF resources are allocated and prepared.
* Open: Module specific resources are allocated and prepared. Also, the Module’s work may happen here.
* Use: Modules can perform work completely on their own as well as handle requests and do work on behalf of the Client.
* Close: Module specific resources are released.
* Destroy: DMF resources are released.

### Modules Have a Hierarchy

* Modules can only communicate with each other when they are in a parent child relationship.
* Modules can never communicate with each other when they are siblings.
* Modules can be extended by using other Modules and using inheritance.
* Client driver may not use Modules that are not properly instantiated.

### Properties of Modules

The above generic properties that make Modules object oriented are used to give Modules these properties that are specifically useful in drivers:

1. Clients access Modules via Module instance handles. Each Module is responsible for managing its own state using its own Module Private Context (similar to a Device Context). However, this Module Private Context is only accessible and visible by the Module itself. Neither the Client nor any other Module has access to that Module Private Context.
2. Each Module receives all WDF callbacks, such as EvtDevicePrepareHardware.
3. Modules can, and usually do, use other Child Modules. This allows the layered architecture in WDM to exist inside the Client Driver. (The driver’s author designs the layering.)
4. DMF is responsible for the direct creation and destruction of all Modules. DMF is responsible for allocating memory for DMF specific resources for each instantiated Module. DMF also allocates the Module’s Private Context.
5. DMF is responsible for dispatching all WDF callbacks that WDF sends to the Client Driver to each Module and its Child Modules recursively. The dispatching to Modules happens before the callback is dispatched to the Client Driver’s callbacks. (Client Driver WDF callbacks occur after all the instantiated Modules’ WDF callbacks have occurred except during the unwinding path, in which case they happen in reverse order.)
6. Modules may only talk to each other via a parent-child relationship. Modules are specifically prevented from communicating in a sibling-sibling relationship. This is necessarily true because of point 1.
7. DMF provides a common manner to instantiate and initialize a Module. This makes it easier and less error prone for the driver writer to incorporate a Module. It also makes it easier for the Module creator to create the Module as there are many existing samples to learn from and the interface to Clients is well defined.
8. The Client does not need to lock Module data because each Module is responsible for locking its own data using its own locks (which are provided for each Module by DMF).
9. The Client does not need to handle asynchronous arrival/removal of Module resources because each Module is responsible for properly handling asynchronous arrival/removal of its underlying resources. DMF has APIs that help the Module author handle these issues robustly.
10. A Module can instantiate up to 16 Child Modules.
11. DMF Modules can be written and used in C or C++.
12. DMF Modules can be written and used in both Kernel and User-mode drivers.

## Properties of a DMF Driver

DMF drivers have the following properties:

1. DMF Drivers consist of zero or more (usually more than zero) instantiated Modules.
2. There may or may not be Client Driver specific code that interacts with the instantiated Modules.
3. DMF Drivers are WDF Drivers that add the DMF framework by adding a few extra steps when the Client Driver creates its WDFDEVICE.
4. Using predefined macros, it is possible to easily create a DMF Driver without writing any of the code to initialize the driver and/or the Client Driver’s WDFDEVICE.
5. DMF Drivers can be any type of device driver including Function Driver, Filter Driver, Bus Driver, Kernel Mode Driver or User Mode Drivers.
6. A Client Driver can instantiate any number of Modules.
7. DMF Drivers do not prevent the Client Driver from using any WDF or WDM primitives.

## Using DMF in an Existing Driver or a Driver that has its own Device Context

Aside from including the appropriate headers and libraries, there are four specific steps to using DMF in a driver that has its own Device Context:

1. Hook DMF into the driver.
2. Initialize DMF.
3. Instantiate DMF Modules.
4. Use Modules.

This section explains steps 1 and 2 in detail with examples. Later sections explain steps 3 and 4 which are common for all drivers that use DMF. They are “Instantiating DMF Modules” and “Using DMF Modules”.

### Hook DMF into the driver.

It is necessary to “hook” DMF into the driver so that DMF can perform two important tasks:

* Dispatch all WDF callbacks to each instantiated DMF Module while still allowing the Client Driver’s callbacks to execute.
* Manage the creation, destruction and operation of each instantiated Module.

To accomplish task 1 above, DMF needs to tell WDF to call DMF for all possible WDF callbacks and it needs to know what callbacks the Client Driver supports. This is accomplished using a set of calls into DMF that map directly to the types of WDF callbacks that exist.

Follow these steps:

First, include the DMF support as well as the definitions needed to use Modules in a specific Library by including the relevant Library Include File. The Library Include File, DmfModules.Library.h, contains all the definitions needed to use DMF as well as all the definitions for the Modules that come with the Framework.

#include "DmfModules.Library.h

Next, write the DriverEntry function as you would normally—it has nothing that is DMF specific.

In the Client Driver’s AddDevice callback, allocate an opaque DMF structure that is used later during initialization:

PDMFDEVICE\_INIT dmfDeviceInit;

dmfDeviceInit = DMF\_DmfDeviceInitAllocate(DeviceInit);

This structure is used to gather information about how the Client Driver operates and is used later to initialize DMF.

Next, the Client Driver must tell DMF what WDF callbacks it will use. There are four sets of WDF callbacks, each set corresponding to a DMF call.

* Pnp Power Events
* Pnp Power Policy Events
* File Object Events
* Default Queue Events

Each type of callback must be hooked. This is because DMF needs to chain the Module’s callbacks to the Client Driver’s callbacks. Note: The Client Driver must first call the DMF API **before** calling the WDF API to set the Client Driver callbacks.

Pnp Power Event callbacks:

WDF\_PNPPOWER\_EVENT\_CALLBACKS\_INIT(&Pnp);

Pnp.EvtDeviceD0Entry = DmfSampleEvtDeviceD0Entry;

Pnp.EvtDeviceD0Exit = DmfSampleEvtDeviceD0Exit;

Pnp.EvtDevicePrepareHardware = DmfSampleEvtDevicePrepareHardware;

Pnp.EvtDeviceReleaseHardware = DmfSampleEvtDeviceReleaseHardware;

DMF\_DmfDeviceInitHookPnpPowerEventCallbacks(dmfDeviceInit,

&Pnp);

WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit,

&Pnp);

Pnp Power Policy callbacks:

DMF\_DmfDeviceInitHookPowerPolicyEventCallbacks(dmfDeviceInit,

NULL);

File Object Event callbacks:

DMF\_DmfDeviceInitHookFileObjectConfig(dmfDeviceInit,

NULL);

In the above example, the Client Driver does not register for any Power Policy or PnP Power Event callbacks, but the DMF API must still be called. This is mandatory because the Client Driver does not know what callbacks are supported by each Module.

Note: The Client Driver must call the above three functions regardless of whether or not the Client Driver supports the corresponding callbacks.

There is one more set of important callbacks that must be hooked: The WDFQUEUE callbacks for the default queue:

WDF\_IO\_QUEUE\_Config\_INIT\_DEFAULT\_QUEUE(&queueConfig,

WdfIoQueueDispatchParallel);

queueConfig.PowerManaged = WdfTrue;

queueConfig.EvtIoDeviceControl = DmfSampleEvtIoDeviceControl;

queueConfig.EvtIoInternalDeviceControl = DmfSampleEvtIoDeviceControl;

DMF\_DmfDeviceInitHookQueueConfig(dmfDeviceInit,

&queueConfig);

ntStatus = WdfIoQueueCreate(device,

&queueConfig,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&queue);

If the Client Driver has no queue, there is no need to call this function. In this case, DMF will create a default queue so that any Modules that support IOCTLs (which need the default queue) will function properly.

### Initialize DMF

Using the above steps, DMF is hooked into the Client Driver but is not yet initialized. The next step is to initialize DMF.

DMF\_EVENT\_CALLBACKS dmfEventCallbacks;

DMF\_EVENT\_CALLBACKS\_INIT(&dmfEventCallbacks);

dmfEventCallbacks.EvtDmfDeviceModulesAdd = DmfDeviceModulesAdd;

DMF\_DmfDeviceInitSetEventCallbacks(dmfDeviceInit,

&dmfEventCallbacks);

ntStatus = DMF\_ModulesCreate(device,

&dmfDeviceInit);

if (!NT\_SUCCESS(ntStatus))

{

goto Exit;

}

The above code does the following:

1. Set a callback that DMF will call when it is ready to instantiate Modules. This tells DMF to know what Modules to instantiate.
2. Initialize DMF itself.

In the above example, DmfDeviceModulesAdd is the function that DMF will call to get the list of Modules that the Client Driver will use. This function is where the Client Driver configures each of the Modules that it uses.

Now the Client Driver is ready to instantiate Modules. See the section “Instantiating DMF Modules”.

## Using DMF in a Driver That Does Not Have Its Own Device Context

The preferred way to use DMF is for all work in a Client Driver to be done by the Modules and corresponding Client Driver callbacks. In this case, the Client Driver is a “container” of Modules. These types of Client Drivers do not have their own Device Context. They do not need to create the typical DriverEntry and Device Add functions and all the Client Driver code is in a single file called DmfInterface.c.

All DMF drivers are WDF drivers, therefore they must follow WDF rules. One of those rules is to declare function prototypes for callbacks prior to their use so that SAL can properly analyze code. In the code snippet below you can see those lines.

DMF provides macros that make it simple to initialize a DMF driver. These macros provide standard code that most DMF drivers need to initialize WDF and connect WDF to DMF. See the section, “Client Driver Initialization Macros”. In some cases, the code the macros provide may not be adequate. In that case it is not necessary to use the macros. In that case, the programmer must write the WDF initialization code directly and write the code to connect WDF to DMF. See “Client Driver DFM Initialization Functions”.

These macros are:

* DMF\_DEFAULT\_DRIVERENTRY
* DMF\_DEFAULT\_DRIVERCLEANUP
* DMF\_DEFAULT\_DEVICEADD

To initialize a DMF driver that does not have its own Device Context, follow these steps:

1. First, include the DMF support as well as the definitions needed to use Modules in a specific Library by including the relevant Library Include File. The Library Include File, DmfModules.Library.h, contains all the definitions needed to use DMF as well as all the definitions for the Modules that come with the Framework.
2. Declare WDF function prototypes.
3. Write the comment that the WPP scanner needs.
4. Use the above macros to define DriverEntry, Driver Cleanup and Device Add.

The following code example shows the above steps:

#include "DmfModules.Library.h"

DRIVER\_INITIALIZE DriverEntry;

EVT\_WDF\_DRIVER\_DEVICE\_ADD StorageFwUpdateDeviceAdd;

EVT\_WDF\_OBJECT\_CONTEXT\_CLEANUP StorageFwUpdateDriverContextCleanup;

EVT\_DMF\_DEVICE\_MODULES\_ADD DmfDeviceModulesAdd;

/\*WPP\_INIT\_TRACING(); (This comment is necessary for WPP Scanner.)\*/

#pragma code\_seg("INIT")

DMF\_DEFAULT\_DRIVERENTRY(DriverEntry,

StorageFwUpdateDriverContextCleanup,

StorageFwUpdateDeviceAdd)

#pragma code\_seg()

#pragma code\_seg("PAGED")

DMF\_DEFAULT\_DRIVERCLEANUP(StorageFwUpdateDriverContextCleanup)

DMF\_DEFAULT\_DEVICEADD(StorageFwUpdateDeviceAdd,

DmfDeviceModulesAdd)

#pragma code\_seg()

In this case DMF will perform all the hooking in addition to creating the driver’s WDFDEVICE and initializing DMF. Now, you only need to write the function that instantiates the Modules. In the above example, it is called DmfDeviceModulesAdd.

If your file is a C++ file, you need to surround the prototypes with extern “C” as in this example:

#include "DmfModules.Library.h"

extern "C"

{

DRIVER\_INITIALIZE DriverEntry;

EVT\_WDF\_DRIVER\_DEVICE\_ADD StorageFwUpdateDeviceAdd;

EVT\_WDF\_OBJECT\_CONTEXT\_CLEANUP StorageFwUpdateDriverContextCleanup;

EVT\_DMF\_DEVICE\_MODULES\_ADD DmfDeviceModulesAdd;

}

/\*WPP\_INIT\_TRACING(); (This comment is necessary for WPP Scanner.)\*/

#pragma code\_seg("INIT")

DMF\_DEFAULT\_DRIVERENTRY(DriverEntry,

StorageFwUpdateDriverContextCleanup,

StorageFwUpdateDeviceAdd)

#pragma code\_seg()

#pragma code\_seg("PAGED")

DMF\_DEFAULT\_DRIVERCLEANUP(StorageFwUpdateDriverContextCleanup)

DMF\_DEFAULT\_DEVICEADD(StorageFwUpdateDeviceAdd,

DmfDeviceModulesAdd)

#pragma code\_seg()

Now the Client Driver is ready to instantiate Modules. See the section “Instantiating DMF Modules”.

## Instantiating DMF Modules

This section is common to all types of DMF drivers. Most DMF drivers will instantiate at least one Module. If you have followed the steps above, DMF will call the callback function that allows the Client Driver to list and configure the Modules to be used:

This is the signature for that function:

#pragma code\_seg("PAGED")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

VOID

DmfDeviceModulesAdd(

\_In\_ WDFDEVICE Device,

\_In\_ PDMFMODULE\_INIT DmfModuleInit

)

Recall that most Modules have Module specific configuration parameters. It is the job of the Client Driver to set the configuration of each Module and then call a DMF API that adds that Module to a list of Modules that will be instantiated. To do so, follow this example:

First, declare a single local structure (on the stack) that stores the Module Attributes. Note: You only need a single instance of this structure for all modules that are instantiated.

DMF\_MODULE\_ATTRIBUTES moduleAttributes;

Next, declare a Module specific Config structure for each Module that will be instantiated. Look at the Module’s header file to know what structure to use. Here is shown the Config structure for the BufferList Module. Not all Modules require a Config structure and not all parameters of a Config structure are necessarily required. See the Module documentation for details.

DMF\_CONFIG\_BufferPool bufferPoolModuleConfig;

Next, initialize and populate the Module’s Config and Attributes structure. It takes as parameters the Module specific Config structure and the Module Attributes structure.

DMF\_CONFIG\_BufferPool\_AND\_ATTRIBUTES\_INIT(&moduleConfigBufferPool,

&moduleAttributes);

moduleConfigBufferPool.BufferPoolMode = BufferPool\_Mode\_Source;

moduleConfigBufferPool.Mode.SourceSettings.EnableLookAside = TRUE;

moduleConfigBufferPool.Mode.SourceSettings.BufferCount = BRANCHTRACK\_NUMBER\_OF\_BUFFERS;

moduleConfigBufferPool.Mode.SourceSettings.PoolType = NonPagedPoolNx;

moduleConfigBufferPool.Mode.SourceSettings.BufferSize = moduleConfigHashTable.MaximumKeyLength;

moduleConfigBufferPool.Mode.SourceSettings.BufferContextSize = 0;

Finally, add the Module to the list of Modules that will be instantiated. This API allows the Client Driver to save a copy of the Module handle for later use. If the Client Driver has no Device Context or does not need to save the Module handle, then the last argument can be NULL.

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&moduleContext->DmfObjectBufferPool);

Follow the above steps for each Module that is to be instantiated. Here are more examples:

// AcpiTarget

// ----------

//

DMF\_CONFIG\_AcpiTarget\_AND\_ATTRIBUTES\_INIT(&moduleConfigAcpiTarget,

&moduleAttributes);

moduleConfigAcpiTarget.DsmRevision = 1;

moduleConfigAcpiTarget.Guid = GUID\_DSM\_PCIE;

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&deviceContext->DmfModuleAcpiTarget);

// CrashDump

// ---------

//

DMF\_CONFIG\_CrashDump\_AND\_ATTRIBUTES\_INIT(&moduleConfigCrashDump,

&moduleAttributes);

moduleConfigCrashDump.ComponentName = CRASH\_DUMP\_COMPONENT\_NAME;

// This driver only uses its FIFO Crash Dump buffer.

//

moduleConfigCrashDump.BufferSize = CRASH\_DUMP\_DATA\_FIFO\_SIZE\_OF\_BUFFER;

moduleConfigCrashDump.BufferCount = CRASH\_DUMP\_DATA\_FIFO\_NUMBER\_OF\_BUFFERS;

// The ring buffer which contains the log.

//

moduleConfigCrashDump.RingBufferDataGuid = DmfSampleDriver\_CrashDump\_GUID\_Fifo;

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&deviceContext->DmfModuleCrashDump);

// HidPortableDeviceButtons

// -----------------------

//

DMF\_CONFIG\_HidPortableDeviceButtons\_AND\_ATTRIBUTES\_INIT(&moduleConfigHidPortableDeviceButtons,

&moduleAttributes);

moduleConfigHidPortableDeviceButtons.ProductId = HIDID\_PRODUCT\_VHF\_BUTTONS;

moduleConfigHidPortableDeviceButtons.VendorId = HIDID\_VENDOR\_MICROSOFT;

moduleConfigHidPortableDeviceButtons.VersionNumber = 0x0002;

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&deviceContext->DmfModuleHidPortableDeviceButtons);

// RingBuffer

// ----------

//

DMF\_CONFIG\_RingBuffer\_AND\_ATTRIBUTES\_INIT(&moduleConfigRingBuffer,

&moduleAttributes);

moduleConfigRingBuffer.ItemCount = RING\_BUFFER\_MAXIMUM\_NUMBER\_OF\_ENTRIES;

moduleConfigRingBuffer.ItemSize = sizeof( BUTTON\_DATA );

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&deviceContext->DmfModuleButtonFifo);

That is all you need to do to instantiate a Module. When this callback function returns, DMF will actually instantiate the Modules that have been added to the list.

If the Client Driver has a Device Context and it saved a copy of the Module handle, then the Client Driver can start using the Modules directly. See the section below called, “Using Modules”. If the Client Driver has no Device Context, then the Modules are not directly used; they just operate on their own.

## Instantiating Dynamic Modules

In some rare cases, a Client may instantiate a Module directly without using the DmfModulesAdd callback. For example, a Client (driver or Module) may want to use a Module to perform a query before instantiating other Modules. DMF supports this ability. However, there is one restriction: It is only possible to open a Module dynamically if the Module does not support WDF callbacks. This makes sense because if the Module is created, opened and used dynamically it means it can happen at any time. In this case, the Module cannot operate as intended by the Module author because there is no guarantee that its WDF callbacks will be called.

To open a Module dynamically, follow these steps:

### Synchronous Dynamic Instantiation

*Use these steps to open a Module that does not register for a DMF\_[ModuleName]\_NotificationRegister callback*.

1. Declare a DMF\_MODULE\_ATTRIBUTES structure.
2. Declare a DMF\_CONFIG\_[ModuleName] if the Module has a Config structure.
3. Populate the above two structures properly. (Parent object is the Client Driver’s WDFDEVICE.)
4. Declare a DMFMODULE object.
5. Prepare to call the Module’s Create function by using DMF\_CONFIG\_[ModuleName]\_AND\_ATTRIBUTES\_INIT() or DMF\_[ModuleName]\_AND\_ATTRIBUTES\_INIT(). *See the section, Dynamic Module Access API*.
6. Call the Module’s Create function using the above two structures and retrieve a DMFMODULE and save it in the variable declared in step 4. See the section, *The Module Create Function*.
7. Call the Module’s Methods as needed passing the DMFMODULE from step 6.
8. Finally, destroy the DMFMODULE using DMF\_Module\_InstanceDestroy().

### Asynchronous Notification Dynamic Instantiation

*Use these steps to dynamically instantiate a Module that registers for a DMF\_[ModuleName]\_NotificationRegister callback.*

1. Declare a DMF\_MODULE\_ATTRIBUTES structure. You must set its DMF\_MODULE\_EVENT\_CALLBACKS structure’s EvtModuleOnDeviceNotificationOpen callback (and optionally the EvtModuleOnDeviceNotificationPostOpen callback).
2. Declare a DMF\_CONFIG\_[ModuleName] if the Module has a Config structure.
3. Populate the above two structures properly. (Parent object is the Client Driver’s WDFDEVICE.)
4. Declare a DMFMODULE object.
5. Prepare to call the Module’s Create function by using DMF\_CONFIG\_[ModuleName]\_AND\_ATTRIBUTES\_INIT() or DMF\_[ModuleName]\_AND\_ATTRIBUTES\_INIT(). See the section, *Dynamic Module Access API*.
6. Call the Module’s Create function using the above two structures and retrieve a DMFMODULE and save it in the variable declared in step 4. See the section, *The Module Create Function*.
7. ***Wait for the callback set in step 1 to occur.*** Now, the rest of the driver can use the Module. Note: The Client may also register for notification that the underlying device is removed and stop using the Module. If the Client calls a Module’s method while or after the underlying target has been removed, an error is gracefully returned to the Client.
8. Call the Module’s Methods as needed passing the DMFMODULE from step 6.
9. Finally, destroy the DMFMODULE using DMF\_Module\_InstanceDestroy().

### Destroying a Dynamic Module

It is not necessary to immediately close and destroy the Module after its Methods are called. But, it is the responsibility of the Client Driver to do so before the driver corresponding WDFDEVICE is deleted.

Here is an example of the above sequence showing the Dmf\_AcpiTarget Module dynamically instantiated:

NTSTATUS

ConfigurationDetermine(

DEVICE\_CONTEXT\* DeviceContext

)

{

NTSTATUS ntStatus;

DMF\_MODULE\_ATTRIBUTES moduleAttributes;

DMF\_CONFIG\_AcpiTarget acpiTargetModuleConfig;

DMFMODULE dmfModuleAcpiTarget;

ULONG returnBufferSize;

WDF\_OBJECT\_ATTRIBUTES attributes;

dmfModuleAcpiTarget = NULL;

WDF\_OBJECT\_ATTRIBUTES\_INIT(&attributes);

attributes.ParentObject = DeviceContext->WdfDevice;

DMF\_CONFIG\_AcpiTarget\_AND\_ATTRIBUTES\_INIT(&acpiTargetModuleConfig,

&moduleAttributes);

acpiTargetModuleConfig.DsmRevision = 1;

acpiTargetModuleConfig.Guid = GUID\_DSM\_CONFIGURATION;

// Create a Dynamic Module.

//

ntStatus = DMF\_AcpiTarget\_Create(DeviceContext->WdfDevice,

&moduleAttributes,

&attributes,

&dmfModuleAcpiTarget);

if (!NT\_SUCCESS(ntStatus))

{

goto Exit;

}

// Get configuration information using the Dynamic Module’s Methods.

//

ntStatus = DMF\_AcpiTarget\_InvokeDsm(dmfModuleAcpiTarget,

CONFIGURATION\_INFORMATION,

0,

&DeviceContext->ConfigurationInformation,

&returnBufferSize);

if (!NT\_SUCCESS(ntStatus))

{

DeviceContext->ConfigurationInformation = 0;

}

Exit:

if (dmfModuleAcpiTarget != NULL)

{

DMF\_Module\_Destroy(dmfModuleAcpiTarget);

}

return ntStatus;

}

## Transport Modules

One of the most powerful features of DMF is that it allows programmers to easily implement an important programming paradigm, namely the idea of a Protocol-Transport. Using this paradigm, one can implement the important object-oriented programming pattern known as Composition.

The Protocol-Transport paradigm is useful when one has a Protocol that is to be implemented on several types of Transports. It is useful, in this case, to write the Protocol one time using a common lower interface. Then, each of the multiple Transports are written using the same upper interface.

In DMF, this is accomplished using Parent-Child Modules. In this case, the Protocol is located in the Parent Module. The Transports are located in multiple Child Modules.

Normally, Child Modules are statically created by the Parent during Module creation. The Client of the Parent has no knowledge of the Child Modules instantiated by the Parent Module.

However, using DMF’s Transport Module concept, the Client has the ability to determine a Child Module that the Parent Module will instantiate. In this way, the Client decides the underlying transport the Parent Module uses.

A Transport Module is the same as any other Modules, except that it implements a Method using a specific signature. Transport Modules can be used both as normal (non-Transport Modules) and as Transport Modules.

A Parent Module that needs to use a Transport Module must always use a Transport Module. In this case, when the Module is created the MODULE\_OPTIONS\_TRANSPORT\_REQUIRED bit must be set in the Module Options. When this bit is set, just after the Module (and any Child Modules) have been created, DMF will call a callback function that allows the Client of the Module to create the required Transport Module. DMF will then add that additional Child Module to the list of Child Modules.

Finally, when the Parent Module needs to call the Transport Module, it does so using the Transport Module’s Transport Method. This Method has the same signature for all Transport Modules which allows the C compiler to compile correctly.

## Using Modules

Once a Module is instantiated there are three ways the Client uses the Module.

1. The Module operates on its own by directly receiving WDF callbacks via DMF.
2. The Module operates on its own by directly receiving DMF callbacks that are sent by DMF.
3. The Clients calls Module Methods. These Methods are specific to the Module and perform work that only that Module understands.

Note that the options above are not mutually exclusive. Modules can expose any combination of DMF and WDF callbacks as well as Module methods.

The Client need not be concerned about items 1 and 2 above. Those two paths are the responsibility of the Module author. It goes without saying that this also means that the Module author does not need to be concerned with what the Client or any other Module does. All actors are only concerned and responsible for their own code and there exists a strong boundary between Modules and Client code. Simply instantiating the Module allows paths 1 and 2 to happen automatically and independently without further interaction from the Client.

The Client Driver’s direct interaction with Module is via the Module’s Methods (if any) and Client Callbacks (if any). If the Module has no Methods or Client Callbacks, then the Client Driver does not, and cannot, interact with the Module directly.

When a Client wants to know what Module Methods the Module exposes, the author simply refers to the Module’s .h file, which has the publicly available Methods listed. Each Module also has a corresponding .txt file that explains all the Methods the Module exposes.

All Module Methods require that the Module’s handle be sent as the first parameter. The Client receives this handle when the Module is instantiated. It is up to the Client to store this handle for later use.

Here is an example of a Module Method:

NTSTATUS

DMF\_RingBuffer\_Write(

\_In\_ DMFMODULE DmfModule,

\_In\_reads\_(SourceBufferSize) UCHAR\* SourceBuffer,

\_In\_ ULONG SourceBufferSize

);

This Method receives the instantiated Module handle for a Module that exposes a FIFO. It also accepts an address of a buffer and the length of that buffer. This Method reads the next entry in the Module’s FIFO and returns it to the caller.

Here is an example of a how a Client uses a Method:

ntStatus = DMF\_RingBuffer\_Write(deviceContext->DmfModuleButtonFifo,

( PUCHAR )&buttonData,

sizeof( buttonData ));

The first parameter is the Module’s handle. This handle is obtained when the Module is instantiated. The rest of the parameters are specific to that Module’s Method. Methods can have any number of parameters.

It is not possible to use a Module’s Method without a valid handle to an instance of the Module. This guarantees that when the Module’s Method is called, the Module’s internal data structures are ready to be used. It is the responsibility of the Module’s author to make sure that as long as a valid Module handle is passed, the Method must operate gracefully.

Note the separation of work between Modules and the Clients. This separation limits the number of code paths and eliminates dependencies that may be difficult to see or understand. In turn, it makes code easier to write, understand, maintain and reuse.

## WPP Tracing Module Traces From A Client Driver

DMF Modules emit WPP trace statements using WPP logging. The Client driver can cause those statements to emit along with the Client driver’s own tracing. To do so, the Client driver must define a WPP\_CONTROL\_GUID and WPP Control Bit in addition to the WPP\_CONTROL\_GUID and WPP Control Bits defined by the Client driver.

Specifically, the Client driver must define the above in this manner:

// NOTE: Each Client must create a unique GUID. Do not use the GUID in this sample.

//

WPP\_DEFINE\_CONTROL\_GUID( \

DmfLibraryTraceGuid, (74DC7AB3,690B,480A,87A4,3627882A831C), \

WPP\_DEFINE\_BIT(DMF\_TRACE) \

) \

For example, here is how a Client driver defines the above GUID along with the Client driver’s GUIDs.

// Define the tracing flags.

//

// Tracing GUID - {1AE05C6D-9140-41DC-BFBD-C8C4391FD95F}

//

#define WPP\_CONTROL\_GUIDS \

WPP\_DEFINE\_CONTROL\_GUID( \

DmfLibraryTraceGuid, (74DC7AB3,690B,480A,87A4,3627882A831C), \

WPP\_DEFINE\_BIT(DMF\_TRACE) \

) \

WPP\_DEFINE\_CONTROL\_GUID( \

DmfSampleDriverTraceGuid, {1AE05C6D,9140,41DC,BFBD,C8C4391FD95F} \

WPP\_DEFINE\_BIT(MYDRIVER\_ALL\_INFO) \

WPP\_DEFINE\_BIT(TRACE\_DRIVER) \

WPP\_DEFINE\_BIT(TRACE\_DEVICE) \

WPP\_DEFINE\_BIT(TRACE\_QUEUE) \

WPP\_DEFINE\_BIT(TRACE\_USP) \

)

// This comment block is scanned by the trace preprocessor to define our

// Trace function.

//

// USEPREFIX and USESUFFIX strip all trailing whitespace, so we need to surround

// FuncExit messages with brackets

//

// begin\_wpp config

// FUNC Trace{FLAG=MYDRIVER\_ALL\_INFO}(LEVEL, MSG, ...);

// FUNC TraceEvents(LEVEL, FLAGS, MSG, ...);

// FUNC FuncEntry{LEVEL=TRACE\_LEVEL\_VERBOSE}(FLAGS);

// FUNC FuncEntryArguments{LEVEL=TRACE\_LEVEL\_VERBOSE}(FLAGS, MSG, ...);

// FUNC FuncExit{LEVEL=TRACE\_LEVEL\_VERBOSE}(FLAGS, MSG, ...);

// FUNC FuncExitVoid{LEVEL=TRACE\_LEVEL\_VERBOSE}(FLAGS);

// FUNC TraceError{LEVEL=TRACE\_LEVEL\_ERROR}(FLAGS, MSG, ...);

// FUNC TraceInformation{LEVEL=TRACE\_LEVEL\_INFORMATION}(FLAGS, MSG, ...);

// FUNC TraceVerbose{LEVEL=TRACE\_LEVEL\_VERBOSE}(FLAGS, MSG, ...);

// FUNC FuncExitNoReturn{LEVEL=TRACE\_LEVEL\_VERBOSE}(FLAGS);

// USEPREFIX(FuncEntry, "%!STDPREFIX! [%!FUNC!] --> Entry");

// USEPREFIX(FuncEntryArguments, "%!STDPREFIX! [%!FUNC!] --> Entry <");

// USEPREFIX(FuncExit, "%!STDPREFIX! [%!FUNC!] <-- Exit <");

// USESUFFIX(FuncExit, ">");

// USEPREFIX(FuncExitVoid, "%!STDPREFIX! [%!FUNC!] <-- Exit");

// USEPREFIX(TraceError, "%!STDPREFIX! [%!FUNC!] ERROR:");

// USEPREFIX(TraceEvents, "%!STDPREFIX! [%!FUNC!] ");

// USEPREFIX(TraceInformation, "%!STDPREFIX! [%!FUNC!] ");

// USEPREFIX(TraceVerbose, "%!STDPREFIX! [%!FUNC!] ");

// USEPREFIX(FuncExitNoReturn, "%!STDPREFIX! [%!FUNC!] <--");

// end\_wpp

### Filtering WPP Tracing

Due to the fact that much of WPP functionality is determined at compile time, it is not possible to easily add fine control of WPP tracing of code inside DMF Modules using WPP Control Bits. Thus, the single bit, DMF\_TRACE, emits all DMF related tracing.

It is possible to turn on/off DMF tracing by enabling/disabling the DMF tracing GUID in TraceView.exe. It is also possible to filter tracing by filtering on the many available fields available for each trace.

# The Structure of a Module

The structure of a Module is consistent among all Modules. This consistency is important because the Module is designed to be easily shareable and readable by many people.

Every Module has three mandatory files:

* Module’s .c file
* Module’s .h file
* Module’s .txt file

In addition, Modules may have two optional files:

* Module’s \_Public.h file
* Module’s .mc file

## The Module .h File

This file contains information that allows Clients to use the Module. Specifically, it contains the following in this order:

1. Enumerations and structures used in the Module Config.
2. The Module Config.
3. The Module macros’ definitions.
4. Prototypes of the Module’s Methods.

Private definitions used only by the Module are not placed in this file. Definitions necessary for user-mode interaction (such as IOCTLS and their corresponding data structures) are not placed in this file.

In the following example, note the use of the DECLARE\_DMF\_MODULE macro. This macro automatically defines all the public functions that Clients use to instantiate the Module.

/\*++

Copyright (c) Microsoft Corporation. All rights reserved.

Licensed under the MIT license.

Module Name:

Dmf\_ResourceHub.h

Abstract:

Companion file to Dmf\_ResourceHub.c.

Environment:

Kernel-mode Driver Framework

User-mode Driver Framework

--\*/

#pragma once

#if !defined(DMF\_USER\_MODE)

// Client Driver callback to get TransferList from Spb.

//

typedef

\_Function\_class\_(EVT\_DMF\_ResourceHub\_DispatchTransferList)

\_IRQL\_requires\_max\_(DISPATCH\_LEVEL)

\_IRQL\_requires\_same\_

NTSTATUS

EVT\_DMF\_ResourceHub\_DispatchTransferList(\_In\_ DMFMODULE DmfModule,

\_In\_ SPB\_TRANSFER\_LIST\* SpbTransferListBuffer,

\_In\_ size\_t SpbTransferListBufferSize,

\_In\_ USHORT I2CSecondaryDeviceAddress,

\_Out\_ size\_t \*TotalTransferLength);

typedef enum

{

Reserved = 0,

I2C,

SPI,

UART

} DIRECTFW\_SERIAL\_BUS\_TYPE;

// Client uses this structure to configure the Module specific parameters.

//

typedef struct

{

// TODO: Currently only I2C is supported.

//

DIRECTFW\_SERIAL\_BUS\_TYPE TargetBusType;

// Callback to get TransferList from Spb.

//

EVT\_DMF\_ResourceHub\_DispatchTransferList\* EvtResourceHubDispatchTransferList;

} DMF\_CONFIG\_ResourceHub;

// This macro declares the following functions:

// DMF\_ResourceHub\_ATTRIBUTES\_INIT()

// DMF\_CONFIG\_ResourceHub\_AND\_ATTRIBUTES\_INIT()

// DMF\_ResourceHub\_Create()

//

DECLARE\_DMF\_MODULE(ResourceHub)

// Module Methods

//

#endif // !defined(DMF\_USER\_MODE)

// eof: Dmf\_ResourceHub.h

//

## The Module .c File

Every Module has all its code in a single .c or .cpp file. This file also contains all the private code and data structures the Module uses.

Each Module consists of ***these sections in this order***: (Note: Even if the Module has no code in a section, always include the section headers.)

* File Header
* DMF Include
* WPP Definitions
* Module Private Enumerations and Structures
* Module Private Context
* Module Macros
* Module Private Code
* Module WDF Callbacks
* Module DMF Callbacks
* Module Descriptors
* Public Calls by Client (Includes Module Create Function)
  + Module Create Function (more in section 3.6)
  + Module Methods

### Section 1: File Header

The file header section contains the copyright notice, the file name, a short description of the purpose of the Module and the environment it is designed to work in.

/\*++

Copyright (c) Microsoft Corporation. All rights reserved.

Licensed under the MIT license.

Module Name:

Dmf\_ResourceHub.c

Abstract:

Resource Hub support code.

Environment:

Kernel-mode Driver Framework

User-mode Driver Framework

--\*/

### Section 2: DMF Include

Every Module always includes the include file for the Library in which the Module is located. This include file always includes the DMF core include files as well as the Library include file for the Library it depends on. Since every Library is a superset of all the Libraries it depends on, the Client need only include the name of a single Library:

// DMF and this Module's Library specific definitions.

//

#include "DmfModules.Library.h"

#include "DmfModules.Library.Trace.h"

### Section 3: WPP Definitions

A corresponding include file which contains definitions needed for WPP tracing which must also be included. Finally, the corresponding .tmh file for the .c/.cpp file must be included:

#include "Dmf\_[ModuleName].tmh"

### Section 4: Module Private Enumerations and Structures

If the Module’s private code uses enumerations and data structures, they are placed here. These are not accessible outside of this Module. Non-DMF specific include files which contain definitions needed by the Module are also included here:

/////////////////////////////////////////////////////////////////////////////////////////////////

// Module Private Enumerations and Structures

/////////////////////////////////////////////////////////////////////////////////////////////////

//

#define RESHUB\_USE\_HELPER\_ROUTINES

#include "reshub.h"

#include <Spb.h>

typedef struct \_RESOURCEHUB\_FILEOBJECT\_CONTEXT

{

USHORT SecondaryDeviceAddress;

WDFMEMORY ConnectionProperties;

} RESOURCEHUB\_FILEOBJECT\_CONTEXT;

WDF\_DECLARE\_CONTEXT\_TYPE\_WITH\_NAME(RESOURCEHUB\_FILEOBJECT\_CONTEXT, ResourceHub\_FileContextGet);

### Section 5: Module Private Context

The Module’s Private Context is similar to a driver’s device context. This data structure contains all the information needed while the Module is running. For example, it may contain handles to Child Modules or timers or other WDF primitives as well as flags and buffers. This structure is only visible to this file. Place the definition of the Module’s Private Context in this section. It may use enumerations and structures defined above.

/////////////////////////////////////////////////////////////////////////////////////////////////

// Module Private Context

/////////////////////////////////////////////////////////////////////////////////////////////////

//

// Contains the WDF IO Target as well as all the structures needed for

// streaming requests.

//

typedef struct

{

WDFIOTARGET ResourceHubTarget;

} DMF\_CONTEXT\_ResourceHub;

### Section 6: Module Macros

This section contains definitions that standardize the way Modules access the Private Context and Config. Each Module always has exactly two macros that automatically define macros that allow the Module to access its Context and its Config. The macros are always used even if there is no Context and/or Config because the macros also define other functions that are used internally by DMF.

Note: Always use these macros in the Module’s .c file, not the Module’s .h file to ensure they are private to the Module. Clients should never have access to functions defined by these macros.

#### DMF\_MODULE\_DECLARE\_CONTEXT

This macro indicates that the Module has a Context (define just above this macro). It automatically defines a function called DMF\_CONTEXT\_GET() which allows Modules to access their Context given a DMFMODULE:

// This macro declares the following function:

// DMF\_CONTEXT\_GET()

//

DMF\_MODULE\_DECLARE\_CONTEXT(ResourceHub)

#### DMF\_MODULE\_DECLARE\_CONFIG

This macro indicates that the Module has a Config (defined in the Module’s .h file). It automatically defines a function called DMF\_CONFIG\_GET() which allows Modules to access their Config given a DMFMODULE:

// This macro declares the following function:

// DMF\_CONFIG\_GET()

//

DMF\_MODULE\_DECLARE\_CONFIG(ResourceHub)

#### DMF\_MODULE\_DECLARE\_NO\_CONTEXT

This macro indicates that the Module has no Context. Use this macro to make the Module standard and for possible additional features in the future. This macro also defines other definitions used internally by DMF.

// This Module has no Context.

//

DMF\_MODULE\_DECLARE\_NO\_CONTEXT(PdoCreate)

#### DMF\_MODULE\_DECLARE\_NO\_CONFIG

This macro indicates that the Module has no Config. Use this macro to make the Module standard and for possible additional features in the future.

// This Module has no Config.

//

DMF\_MODULE\_DECLARE\_NO\_CONFIG(Registry)

### Section 7: Module Private Code

This section contains functions that are called by the Module’s callbacks and Methods. This code contains the Module’s functionality and is very specific to this Module. Note there may be zero, one or more than one function in this section.

This code is only ever called by Module’s callbacks or Module Methods. DMF and Clients never call this code.

///////////////////////////////////////////////////////////////////////////////////////////////////////

// DMF Module Support Code

///////////////////////////////////////////////////////////////////////////////////////////////////////

//

#define INTERNAL\_SERIAL\_BUS\_SIZE(Desc) ((ULONG)(Desc)->Length + \

RTL\_SIZEOF\_THROUGH\_FIELD(PNP\_SERIAL\_BUS\_DESCRIPTOR, Length))

#include "pshpack1.h"

typedef struct \_DIRECTFW\_I2C\_CONNECTION\_DESCRIPTOR\_SUBTYPE

{

UINT32 ConnectionSpeed;

USHORT SecondaryDeviceAddress;

UCHAR VendorDefinedData[ANYSIZE\_ARRAY];

} DIRECTFW\_I2C\_CONNECTION\_DESCRIPTOR\_SUBTYPE;

#include "poppack.h"

#pragma code\_seg("PAGE")

NTSTATUS

ResourceHub\_ConnectResourceHubIoTarget(

\_Inout\_ DMFMODULE DmfModule,

\_In\_opt\_ PLARGE\_INTEGER Id,

\_Out\_ WDFIOTARGET \* RootHubIoTarget

)

/\*++

Routine Description:

Opens an I/O target to the Resource Hub.

If Id is left NULL, the target is opened to the Resource Hub directly, and can then

be used to query for connection properties.

If Id is specified, the target is opened to the RH with this Id as the

filename. Internally the RH redirects the I/O target such that requests made

against the target are sent to the device represented by that Id.

Arguments:

DmfModule - This Module's handle.

Id - Connection ID received as part of FileCreate callback or

PrepareHardware resources.

RootHubIoTarget - Opened I/O target to the Resource Hub, using a filename

constructed from the value Id.

Return Value:

NTSTATUS

--\*/

{

NTSTATUS ntStatus;

WDF\_OBJECT\_ATTRIBUTES attributes;

DECLARE\_UNICODE\_STRING\_SIZE(resourceHubFileName, RESOURCE\_HUB\_PATH\_SIZE);

WDF\_IO\_TARGET\_OPEN\_PARAMS openParameters;

PAGED\_CODE();

ASSERT(DmfModule != NULL);

ASSERT(RootHubIoTarget != NULL);

// Create an IO target to the controller driver via the resource hub.

//

WDF\_OBJECT\_ATTRIBUTES\_INIT(&attributes);

attributes.ParentObject = DmfModule;

ntStatus = WdfIoTargetCreate(DMF\_AttachedDeviceGet(DmfModule),

&attributes,

RootHubIoTarget);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "WdfIoTargetCreate fails: ntStatus=%!STATUS!", ntStatus);

\*RootHubIoTarget = NULL;

goto Exit;

}

// Create controller driver string from descriptor information.

//

if (Id != NULL)

{

RESOURCE\_HUB\_CREATE\_PATH\_FROM\_ID(&resourceHubFileName,

Id->LowPart,

Id->HighPart);

}

else

{

RtlInitUnicodeString(&resourceHubFileName,

RESOURCE\_HUB\_DEVICE\_NAME);

}

WDF\_IO\_TARGET\_OPEN\_PARAMS\_INIT\_OPEN\_BY\_NAME(&openParameters,

&resourceHubFileName,

STANDARD\_RIGHTS\_ALL);

// Open the controller driver / Resource Hub I/O target.

//

ntStatus = WdfIoTargetOpen(\*RootHubIoTarget,

&openParameters);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "WdfIoTargetOpen fails: ntStatus=%!STATUS!", ntStatus);

goto Exit;

}

Exit:

if (! NT\_SUCCESS(ntStatus) &&

(\*RootHubIoTarget != NULL))

{

WdfObjectDelete(\*RootHubIoTarget);

\*RootHubIoTarget = NULL;

}

FuncExit(DMF\_TRACE\_ResourceHub, "ntStatus=%!STATUS!", ntStatus);

return ntStatus;

}

#pragma code\_seg()

// NOTE: See Dmf\_ResourceHub.c to see the rest of the code in this section.

//

// …

### Section 8: Module WDF Callbacks

This section contains definitions of all the WDF callbacks that the Module supports. Only callbacks that the Module supports are listed. Unsupported callbacks use DMF’s generic version of those callbacks which in most (but not all) cases, simply perform validation in DEBUG build.

(*Callbacks in italics* indicate callbacks that are almost never used because DMF contains special support which generally eliminates the need for these specific callbacks.) See the table below for more information about the callbacks in *italics*.

This is the list of WDF callbacks that DMF supports:

* *\*\*\*DMF\_[ModuleName]\_PrepareHardware*
* *\*\*\*DMF\_[ModuleName]\_ReleaseHardware*
* DMF\_[ModuleName]\_D0Entry
* DMF\_[ModuleName]\_D0EntryPostInterruptsEnabled
* DMF\_[ModuleName]\_D0ExitPreInterruptsDisabled
* DMF\_[ModuleName]\_D0Exit
* *\*\*\*DMF\_[ModuleName]\_DeviceIoControl*
* *\*\*\*DMF\_[ModuleName]\_InternalDeviceIoControl*
* DMF\_[ModuleName]\_SelfManagedIoCleanup
* DMF\_[ModuleName]\_SelfManagedIoFlush
* DMF\_[ModuleName]\_SelfManagedIoInit
* DMF\_[ModuleName]\_SelfManagedIoSuspend
* DMF\_[ModuleName]\_SelfManagedIoRestart
* DMF\_[ModuleName]\_SurpriseRemoval
* DMF\_[ModuleName]\_QueryRemove
* DMF\_[ModuleName]\_QueryStop
* DMF\_[ModuleName]\_RelationsQuery
* DMF\_[ModuleName]\_UsageNotificationEx
* DMF\_[ModuleName]\_ArmWakeFromS0
* DMF\_[ModuleName]\_WakeFromS0Triggered
* DMF\_[ModuleName]\_ArmWakeFromSxWithReason
* DMF\_[ModuleName]\_DisarmWakeFromSx
* DMF\_[ModuleName]\_WakeFromSxTriggered
* DMF\_[ModuleName]\_FileCreate
* DMF\_[ModuleName]\_FileCleanup
* DMF\_[ModuleName]\_FileClose
* DMF\_[ModuleName]\_QueueIoRead
* DMF\_[ModuleName]\_QueueIoWrite

#### Rarely Used Callbacks

This section contains notes about the entries in the above table that are in *italics*.

|  |  |
| --- | --- |
| DMF\_[ModuleName]\_PrepareHardware  DMF\_[ModuleName]\_ReleaseHardware | Modules should generally not support DMF\_[ModuleName]\_PrepareHardware and DMF\_[ModuleName]\_ReleaseHardware. Instead, set the Module Open Option to indicate when the Modules’ DMF\_[ModuleName]\_Open and DMF\_[ModuleName]\_Close callbacks are be called. Also, Modules that need resources define DMF\_[ModuleName]\_ResourcesAssign.  For more information about why DMF\_[ModuleName]\_PrepareHardware and DMF\_[ModuleName]\_ReleaseHardware are rarely used, please see 7.3.1 and 7.3.2. Also, please see section 7.1.1 which describes Module Open Options. |
| DMF\_[ModuleName]\_DeviceIoControl  DMF\_[ModuleName]\_InternalDeviceIoControl | Instead of using these callbacks, it is much easier to use DMF\_IoctlHandler instead. This Module is designed to easily manage the routing of IOCTLs between the different Modules and Client driver code. Also, this Module will perform validation and access control based on table of supported IOCTLs. For more information please see DMF\_IoctlHandler.txt. Also, please see 7.3.7 and 7.3.8. |

///////////////////////////////////////////////////////////////////////////////////////////////////////

// Wdf Module Callbacks

///////////////////////////////////////////////////////////////////////////////////////////////////////

//

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

static

BOOLEAN

DMF\_ResourceHub\_ModuleFileCreate(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFDEVICE Device,

\_In\_ WDFREQUEST Request,

\_In\_ WDFFILEOBJECT FileObject

)

/\*++

Routine Description:

File Create callback. Use this callback to open a connection to ACPI.

Arguments:

DmfModule - The given DMF Module.

Device - WDF device object.

Request - WDF Request with IOCTL parameters.

FileObject - WDF file object that describes a file that is being opened for the specified request.

Return Value:

TRUE if this Module handled the callback; false, otherwise.

--\*/

{

BOOLEAN handled;

DMF\_CONTEXT\_ResourceHub\* moduleContext;

PUNICODE\_STRING fileName;

UNICODE\_STRING filePart;

LARGE\_INTEGER id;

NTSTATUS ntStatus;

WDFMEMORY connectionProperties;

VOID\* connectionPropertiesBuffer;

size\_t connectionPropertiesLength;

DIRECTFW\_SERIAL\_BUS\_TYPE serialBusType;

VOID\* typeSpecificData;

ULONG typeSpecificDataLength;

USHORT secondaryDeviceAddress;

size\_t filenameLength;

WDF\_OBJECT\_ATTRIBUTES attributes;

RESOURCEHUB\_FILEOBJECT\_CONTEXT\* fileContext;

PAGED\_CODE();

UNREFERENCED\_PARAMETER(Device);

FuncEntry(DMF\_TRACE\_ResourceHub);

ASSERT(Device == DMF\_AttachedDeviceGet(DmfModule));

ASSERT(Request != NULL);

ASSERT(FileObject != NULL);

moduleContext = DMF\_CONTEXT\_GET(DmfModule);

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&attributes,

RESOURCEHUB\_FILEOBJECT\_CONTEXT);

ntStatus = WdfObjectAllocateContext(FileObject,

&attributes,

(VOID\*\*)&fileContext);

fileContext = ResourceHub\_FileContextGet(FileObject);

handled = FALSE;

ntStatus = STATUS\_SUCCESS;

connectionProperties = NULL;

filenameLength = 0;

fileName = WdfFileObjectGetFileName(FileObject);

if ((fileName != NULL) &&

(fileName->Length != 0))

{

//If the string is null-terminated, Length does not include the trailing null character.

//So use MaximumLength field instead.

//

RtlInitEmptyUnicodeString(&filePart,

fileName->Buffer,

fileName->MaximumLength);

// The file-name part received may begin with a leading backslash

// in the form "\0000000012345678". If the first character is a

// backslash, skip it.

//

filePart.Length = fileName->Length;

if (filePart.Length >= sizeof(WCHAR) && filePart.Buffer[0] == L'\\')

{

++filePart.Buffer;

filePart.Length -= sizeof(WCHAR);

filePart.MaximumLength -= sizeof(WCHAR);

}

if (filePart.Length < sizeof(WCHAR))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "Invalid fileName parameter");

ntStatus = STATUS\_INVALID\_PARAMETER;

goto Exit;

}

filePart.MaximumLength /= sizeof(WCHAR);

ntStatus = RtlStringCchLengthW(filePart.Buffer,

filePart.MaximumLength,

&filenameLength);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "Invalid fileName parameter");

goto Exit;

}

// Retrieve ACPI resource descriptor for this connection from Resource Hub.

//

RESOURCE\_HUB\_ID\_FROM\_FILE\_NAME(filePart.Buffer,

&id);

ntStatus = ResourceHub\_QueryConnectionProperties(DmfModule,

&id,

FileObject,

&connectionProperties);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "ResourceHub\_QueryConnectionProperties fails: ntStatus=%!STATUS!", ntStatus);

goto Exit;

}

// Only I2C GenericSerialBus descriptors are supported. Extract the secondaryDevice address.

//

connectionPropertiesBuffer = WdfMemoryGetBuffer(connectionProperties,

&connectionPropertiesLength);

if (connectionPropertiesBuffer == NULL)

{

ntStatus = STATUS\_UNSUCCESSFUL;

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "No resources returned from RH query");

goto Exit;

}

ntStatus = ResourceHub\_ParseGenericSerialBusDescriptor(DmfModule,

connectionPropertiesBuffer,

connectionPropertiesLength,

&serialBusType,

&typeSpecificData,

&typeSpecificDataLength,

NULL);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "ResourceHub\_ParseGenericSerialBusDescriptor fails: ntStatus=%!STATUS!", ntStatus);

goto Exit;

}

if (serialBusType != I2C)

{

ntStatus = STATUS\_UNSUCCESSFUL;

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "GenericSerialBus descriptor subtype not I2C: 0x%x ntStatus=%!STATUS!", serialBusType, ntStatus);

goto Exit;

}

ntStatus = ResourceHub\_ParseI2CSerialBusDescriptorSubtype(DmfModule,

typeSpecificData,

typeSpecificDataLength,

&secondaryDeviceAddress);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "ResourceHub\_ParseI2CSerialBusDescriptorSubtype fails: ntStatus=%!STATUS!", ntStatus);

goto Exit;

}

// Success.

//

TraceEvents(TRACE\_LEVEL\_INFORMATION, DMF\_TRACE\_ResourceHub, "secondaryDeviceAddress=0x%X request=0x%p", secondaryDeviceAddress, Request);

fileContext->SecondaryDeviceAddress = secondaryDeviceAddress;

WdfRequestComplete(Request,

ntStatus);

handled = TRUE;

}

Exit:

if (connectionProperties != NULL)

{

WdfObjectDelete(connectionProperties);

}

FuncExit(DMF\_TRACE\_ResourceHub, "ntStatus=%!STATUS!", ntStatus);

return handled;

}

#pragma code\_seg()

// NOTE: See Dmf\_ResourceHub.c to see the rest of the code in this section.

//

// …

### Section 9: Module DMF Callbacks

This section contains definitions of all the DMF callbacks that the Module supports. See section “Module DMF Callbacks” for the full list.

///////////////////////////////////////////////////////////////////////////////////////////////////////

// DMF Module Callbacks

///////////////////////////////////////////////////////////////////////////////////////////////////////

//

IoctlHandler\_IoctlRecord ResourceHub\_IoctlSpecification[] =

{

{ IOCTL\_SPB\_EXECUTE\_SEQUENCE, sizeof(SPB\_TRANSFER\_LIST), 0, ResourceHub\_IoctlClientCallback\_SpbExecuteSequence }

};

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

VOID

DMF\_ResourceHub\_ChildModulesAdd(

\_In\_ DMFMODULE DmfModule,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* DmfParentModuleAttributes,

\_In\_ PDMFMODULE\_INIT DmfModuleInit

)

/\*++

Routine Description:

Configure and add the required Child Modules to the given Parent Module.

Arguments:

DmfModule - The given Parent Module.

DmfParentModuleAttributes - Pointer to the parent DMF\_MODULE\_ATTRIBUTES structure.

DmfModuleInit - Opaque structure to be passed to DMF\_DmfModuleAdd.

Return Value:

None

--\*/

{

DMF\_MODULE\_ATTRIBUTES moduleAttributes;

DMF\_CONFIG\_IoctlHandler ioctlHandlerModuleConfig;

UNREFERENCED\_PARAMETER(DmfParentModuleAttributes);

UNREFERENCED\_PARAMETER(DmfModule);

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_ResourceHub);

// IoctlHandler

// ------------

//

DMF\_CONFIG\_IoctlHandler\_AND\_ATTRIBUTES\_INIT(&ioctlHandlerModuleConfig,

&moduleAttributes);

// NOTE: No GUID is necessary because device interface is not created.

//

ioctlHandlerModuleConfig.AccessModeFilter = IoctlHandler\_AccessModeDefault;

ioctlHandlerModuleConfig.EvtIoctlHandlerAccessModeFilter = NULL;

ioctlHandlerModuleConfig.IoctlRecordCount = ARRAYSIZE(ResourceHub\_IoctlSpecification);

ioctlHandlerModuleConfig.IoctlRecords = ResourceHub\_IoctlSpecification;

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

NULL);

FuncExitVoid(DMF\_TRACE\_ResourceHub);

}

#pragma code\_seg()

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

\_Must\_inspect\_result\_

static

NTSTATUS

DMF\_ResourceHub\_Open(

\_In\_ DMFMODULE DmfModule

)

/\*++

Routine Description:

Initialize an instance of a DMF Module of type ResourceHub.

Arguments:

DmfModule - This Module's handle.

Return Value:

NTSTATUS

--\*/

{

NTSTATUS ntStatus;

DMF\_CONTEXT\_ResourceHub\* moduleContext;

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_ResourceHub);

ntStatus = STATUS\_SUCCESS;

moduleContext = DMF\_CONTEXT\_GET(DmfModule);

// Create SPB Resource Hub target to receive messages sent by ACPI.

//

ntStatus = ResourceHub\_RegisterForAcpiNotifications(DmfModule);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "ResourceHub\_RegisterForAcpiNotifications fails: ntStatus=%!STATUS!", ntStatus);

}

FuncExit(DMF\_TRACE\_ResourceHub, "ntStatus=%!STATUS!", ntStatus);

return ntStatus;

}

#pragma code\_seg()

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

static

VOID

DMF\_ResourceHub\_Close(

\_In\_ DMFMODULE DmfModule

)

/\*++

Routine Description:

Uninitialize an instance of a DMF Module of type ResourceHub.

Arguments:

DmfModule - This Module's handle.

Return Value:

NTSTATUS

--\*/

{

DMF\_CONTEXT\_ResourceHub\* moduleContext;

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_ResourceHub);

moduleContext = DMF\_CONTEXT\_GET(DmfModule);

FuncExitVoid(DMF\_TRACE\_ResourceHub);

}

#pragma code\_seg()

### Section 10: Module Descriptors

This section contains the Module descriptor buffers. They are global to this Module for two reasons:

1. To reduce stack space usage. This may be important when Modules contain many layers of Child Modules (especially if the sizes of these structures increase later).
2. So that the descriptor can be used by Methods to validate the formal DMFMODULE to verify the appropriate DMFMODULE is sent to the Module Methods by the Client.

Note the naming convention where the Module name is at the end of the name.

///////////////////////////////////////////////////////////////////////////////////////////////////////

// DMF Module Descriptor

///////////////////////////////////////////////////////////////////////////////////////////////////////

//

static DMF\_MODULE\_DESCRIPTOR DmfModuleDescriptor\_ResourceHub;

static DMF\_CALLBACKS\_DMF DmfCallbacksDmf\_ResourceHub;

static DMF\_CALLBACKS\_WDF DmfCallbacksWdf\_ResourceHub;

### Section 11: Public Calls by Client (Includes Module Create Function)

This section contains all the code that is callable by Clients:

1. Module Create Function (see the section “The Module’s Create Function”).
2. Module Methods.

The prototype for the Create function is in the Module’s .h file and is generated by a DMF macro. The prototypes for the Module Methods are also in the Module’s .h file.

///////////////////////////////////////////////////////////////////////////////////////////////////////

// Public Calls by Client

///////////////////////////////////////////////////////////////////////////////////////////////////////

//

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

\_Must\_inspect\_result\_

NTSTATUS

DMF\_ResourceHub\_Create(

\_In\_ WDFDEVICE Device,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* DmfModuleAttributes,

\_In\_ WDF\_OBJECT\_ATTRIBUTES\* ObjectAttributes,

\_Out\_ DMFMODULE\* DmfModule

)

/\*++

Routine Description:

Create an instance of a DMF Module of type ResourceHub.

Arguments:

Device - Client driver's WDFDEVICE object.

DmfModuleAttributes - Opaque structure that contains parameters DMF needs to initialize the Module.

ObjectAttributes - WDF object attributes for DMFMODULE.

DmfModule - Address of the location where the created DMFMODULE handle is returned.

Return Value:

NTSTATUS

--\*/

{

NTSTATUS ntStatus;

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_ResourceHub);

DMF\_CALLBACKS\_DMF\_INIT(&DmfCallbacksDmf\_ResourceHub);

DmfCallbacksDmf\_ResourceHub.DeviceOpen = DMF\_ResourceHub\_Open;

DmfCallbacksDmf\_ResourceHub.DeviceClose = DMF\_ResourceHub\_Close;

DmfCallbacksDmf\_ResourceHub.ChildModulesAdd = DMF\_ResourceHub\_ChildModulesAdd;

DMF\_CALLBACKS\_WDF\_INIT(&DmfCallbacksWdf\_ResourceHub);

DmfCallbacksWdf\_ResourceHub.ModuleFileCreate = DMF\_ResourceHub\_ModuleFileCreate;

DMF\_MODULE\_DESCRIPTOR\_INIT\_CONTEXT\_TYPE(DmfModuleDescriptor\_ResourceHub,

ResourceHub,

DMF\_CONTEXT\_ResourceHub,

DMF\_MODULE\_OPTIONS\_DISPATCH,

DMF\_MODULE\_OPEN\_OPTION\_OPEN\_PrepareHardware);

DmfModuleDescriptor\_ResourceHub.CallbacksDmf = &DmfCallbacksDmf\_ResourceHub;

DmfModuleDescriptor\_ResourceHub.CallbacksWdf = &DmfCallbacksWdf\_ResourceHub;

DmfModuleDescriptor\_ResourceHub.ModuleConfigSize = sizeof(DMF\_CONFIG\_ResourceHub);

ntStatus = DMF\_ModuleCreate(Device,

DmfModuleAttributes,

ObjectAttributes,

&DmfModuleDescriptor\_ResourceHub,

DmfModule);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "DMF\_ModuleCreate fails: ntStatus=%!STATUS!", ntStatus);

}

FuncExit(DMF\_TRACE\_ResourceHub, "ntStatus=%!STATUS!", ntStatus);

return(ntStatus);

}

#pragma code\_seg()

// Module Methods

//

// NOTE: This example does not list any Module Methods.

//

// eof: Dmf\_ResourceHub.c

//

## The Module \_Public.h File

This file contains only definitions necessary for user-mode programs or other kernel-mode drivers to interact with the Module. Specifically, this file contains the IOCTL definitions and any enumerations and structures that are transferred to/from the Module.

Usually this file has:

* The device interface GUID or the symbolic link name that the Module exposes.
* The IOCTL definitions for the IOCTLs that the Module supports.
* Enumerations and data structures transferred to/from the Module via IOCTLs.

This file must not contain any DMF specific references because this file may be used in environments where DMF is not available. For example, a user-mode application may include this file and try to compile it using only the Windows SDK.

If a Module does not expose a device interface or a symbolic link, then this file is not necessary.

## The Module .txt File

This file contains the documentation for the Module. Like the Module code, this file has a specific format so that all Modules are documented in a consistent manner.

This file is included in the DMF library’s project file. It is stored as plain text to make it easy to review in code reviews to make sure it is always up to date with the code.

Note that the exact length of the “=” which divide the sections is 132 characters. Please make sure all the text fits within 132 columns.

====[DMF Module Documentation]=====================================================================================================

Copyright (c) Microsoft Corporation. All rights reserved.

Licensed under the MIT license.

====[Module]=======================================================================================================================

DMF\_ResourceHub

====[Module Category]==============================================================================================================

Driver Patterns

====[Module Summary]===============================================================================================================

Provides support for extracting data from a Resource Hub and processing SPB IOCTL requests.

====[Module Configuration]=========================================================================================================

typedef struct

{

// Target bus type.

// TODO: Currently only I2C is supported.

//

DIRECTFW\_SERIAL\_BUS\_TYPE TargetBusType;

// Callback to get Transfer List from Spb.

//

EVT\_DMF\_ResourceHub\_DispatchTransferList\* EvtResourceHubDispatchTransferList;

} DMF\_CONFIG\_ResourceHub;

====[Module Enumeration Types]======================================================================================================

typedef enum

{

Reserved = 0,

I2C,

SPI,

UART

} DIRECTFW\_SERIAL\_BUS\_TYPE;

I2C:

I2C bus.

Spi:

SPI bus.

UART:

UART bus.

====[Module Structures]============================================================================================================

====[Module Callbacks]=============================================================================================================

\_IRQL\_requires\_max\_(DISPATCH\_LEVEL)

\_IRQL\_requires\_same\_

NTSTATUS

EVT\_DMF\_ResourceHub\_DispatchTransferList(\_In\_ DMFMODULE DmfModule,

\_In\_ SPB\_TRANSFER\_LIST\* SpbTransferListBuffer,

\_In\_ size\_t SpbTransferListBufferSize,

\_In\_ USHORT I2CSecondaryDeviceAddress,

\_Out\_ size\_t \*TotalTransferLength);

Callback to the Client when data is transferred via the Resource Hub.

Returns

-------

NTSTATUS

Parameters

----------

DmfModule:

An open DMF\_ResourceHub Module handle.

SpbTransferListBuffer:

The SPB transfer list. The Client parses this list and performs actions based on the data in that list.

SpbTransferListBufferSize:

The size in bytes of the buffer pointed to by SpbTransferListBuffer.

I2CSecondaryDeviceAddress:

I2C secondaryDevice address of the OpRegion that processed the SPB request.

TotalTransferLength:

Allows the Client to return the number of bytes processed by the Client to the issuer of the associated SPB request.

Remarks

-------

\*

-----------------------------------------------------------------------------------------------------------------------------------

====[Module Methods]===============================================================================================================

====[Module IOCTLs]================================================================================================================

IOCTL\_SPB\_EXECUTE\_SEQUENCE

--------------------------

Minimum Input Buffer Size:

sizeof(SPB\_TRANSFER\_LIST)

Minimum Output Buffer Size:

0

====[Module Remarks]===============================================================================================================

====[Module Children]==============================================================================================================

\* DMF\_IoctlHandler

====[Module Implementation Details]================================================================================================

====[Examples]=====================================================================================================================

====[To Do]========================================================================================================================

\* Add support for non-I2c buses.

====[eof: DMF\_ResourceHub]=========================================================================================================

## The Module .mc File

*NOTE: This concept will be eliminated in the near future. Instead callbacks to the Client will happen where Client can output event logging information itself.*

Modules may write to the Event Log. (There are some DMF APIs that simplify writing to the Event Log).

The .mc file contains definitions that must be copied into the Client Driver’s .mc file in order for the Event Log entries that the Module writes to appear properly formed in the Event Log.

Due to how the compiler works, it is not possible to directly include the .mc file into the Client Driver project.

If a Module does not write to the Event Log, this file is not necessary.

## The Module’s Create Function

Every Module has a Create function. This is the function that initializes the structures that are necessary for a Module to be instantiated. Every Module Create function contains code specific to the Module.

The Module’s Create function optionally performs another important task: It instantiates the Module’s Child Modules (if any).

When a Module is created in a Client Driver via the DmfModulesAdd callback, the Module’s Create function is called by DMF after DmfModulesAdd returns control to DMF.

Module Create functions are usually called by DMF, either when the Collection of Modules is created or when Child Modules are created. In some cases, it is possible for a Client, either a Module or Client driver, to call a Module’s Create function directly. In this case, the Module is referred to as a Dynamic Module.

See DMF\_[ModuleName]\_Create for more information about the Module’s Create function. This function is always the first function in the section named, “Public Calls by Client”.

### Contents of a Module’s Create Function

These are the steps a Module’s Create function should perform:

1. Define the Module’s DMF callbacks. This step is optional if the Module does not support DMF callbacks. If the Module has Child Modules, it is in this structure where the callback that adds Child Modules is set.
2. Define the Module’s WDF callbacks. This step is optional if the Module does not support WDF callbacks.
3. Define the Module’s descriptor. This step is mandatory for all Modules. This descriptor also holds the DMF and WDF callback structures that are defined in the above steps.
4. Define the Module’s Private Context. This step is mandatory for all Modules even if the Module does not define a Private Context. Use DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE if the Module has a Private Context. Use DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT if the Module does not have a Private Context.
5. Call the DMF API that creates an instance of the Module using all the definitions in the above steps.

The Module’s Create function should not talk to hardware. Generally speaking, it should only perform the above steps. Allocation of Module specific resources such as memory and timers should be done in the Module’s DMF\_[ModuleName]\_Open callback.

### Annotated Module Create function

Here is an example of a Module’s Create function that shows the above steps. All Modules follow this pattern omitting steps as needed.

Note that this is an example of a Module that creates Child Modules. The callback function that adds the Child Modules (DMF\_ResourceHub\_ChildModulesAdd) is annotated afterward.

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

\_Must\_inspect\_result\_

NTSTATUS

DMF\_ResourceHub\_Create(

\_In\_ WDFDEVICE Device,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* DmfModuleAttributes,

\_In\_ WDF\_OBJECT\_ATTRIBUTES\* ObjectAttributes,

\_Out\_ DMFMODULE\* DmfModule

)

/\*++

Routine Description:

Create an instance of a DMF Module of type ResourceHub.

Arguments:

Device - Client driver's WDFDEVICE object.

DmfModuleAttributes - Opaque structure that contains parameters DMF needs to initialize the Module.

ObjectAttributes - WDF object attributes for DMFMODULE.

DmfModule - Address of the location where the created DMFMODULE handle is returned.

Return Value:

NTSTATUS

--\*/

{

NTSTATUS ntStatus;

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_ResourceHub);

// Step 1: Define Module’s DMF callbacks.

//

DMF\_CALLBACKS\_DMF\_INIT(&DmfCallbacksDmf\_ResourceHub);

DmfCallbacksDmf\_ResourceHub.DeviceOpen = DMF\_ResourceHub\_Open;

DmfCallbacksDmf\_ResourceHub.DeviceClose = DMF\_ResourceHub\_Close;

DmfCallbacksDmf\_ResourceHub.ChildModulesAdd = DMF\_ResourceHub\_ChildModulesAdd;

// Step 2: Define Module’s WDF callbacks.

//

DMF\_CALLBACKS\_WDF\_INIT(&DmfCallbacksWdf\_ResourceHub);

DmfCallbacksWdf\_ResourceHub.ModuleFileCreate = DMF\_ResourceHub\_ModuleFileCreate;

// Steps 3 and 4: Define Module’s Descriptor.

//

DMF\_MODULE\_DESCRIPTOR\_INIT\_CONTEXT\_TYPE(DmfModuleDescriptor\_ResourceHub,

ResourceHub,

DMF\_CONTEXT\_ResourceHub,

DMF\_MODULE\_OPTIONS\_DISPATCH,

DMF\_MODULE\_OPEN\_OPTION\_OPEN\_PrepareHardware);

DmfModuleDescriptor\_ResourceHub.CallbacksDmf = &DmfCallbacksDmf\_ResourceHub;

DmfModuleDescriptor\_ResourceHub.CallbacksWdf = &DmfCallbacksWdf\_ResourceHub;

DmfModuleDescriptor\_ResourceHub.ModuleConfigSize = sizeof(DMF\_CONFIG\_ResourceHub);

// Step 5: Tell DMF to create the Module.

//

ntStatus = DMF\_ModuleCreate(Device,

DmfModuleAttributes,

ObjectAttributes,

&DmfModuleDescriptor\_ResourceHub,

DmfModule);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_ResourceHub, "DMF\_ModuleCreate fails: ntStatus=%!STATUS!", ntStatus);

}

FuncExit(DMF\_TRACE\_ResourceHub, "ntStatus=%!STATUS!", ntStatus);

return(ntStatus);

}

#pragma code\_seg()

## How to Create One or More Child Modules

Child Modules are optionally created by a Module (the Parent Module) so that the Parent Module can use the Child Module’s functionality or even create a superset of the Child Module’s functionality. There are numerous examples of Modules that create Child Modules. For example, Dmf\_ThreadedBufferQueue has two Child Modules, DMF\_Thread and DMF\_BufferQueue. DMF\_BufferQueue, in turn, has two Child Modules of type DMF\_BufferPool.

To review, here are some general notes about Child Modules:

* There is no limit to the number of Child Modules a Module can have. Child Modules, in turn, can have any number of Child Modules.
* Any Module an instantiate any other Module. Any Module can be instantiated by any other Module. (Of course, in practice there should be a logical and functional reason for doing so.)
* Of course, a Module may not instantiate itself as a Child Module.
* Child Modules are agnostic about their Parent Modules or whether their Parents are the Client Driver or another Module.
* The code needed to add Child Modules is very similar to the code a Client Driver uses to instantiate a Modules.

### Contents of a Module’s ChildModulesAdd Callback

This callback is very similar to the callback used by Client drivers to instantiate Modules. This callback performs the following steps:

1. In the Parent Module’s DMF\_CALLBACKS\_DMF structure set the ChildModulesAdd callback.
2. Define the callback function in the section of the file where DMF callback functions are defined.
3. DMF will call this callback when it is ready to get the list of Child Modules to add to the Parent.
4. Inside the callback, declare a DMF\_MODULE\_ATTRIBUTES structure. **Important: Only declare a single instance of this structure. You must reuse this same instance for every Child Module. Keep passing the same instance to the functions in step 5. This structure maintains a list of all the Child Modules added in this callback.**
5. For each Child Module to be added, declare and initialize that Module’s corresponding Config structure and the DMF\_MODULE\_ATTRIBUTES using the Module’s corresponding initialization macro. If the Module has no Config structure, initialize the DMF\_MODULE\_ATTRIBUTES using the Module’s corresponding initialization macro.
6. If necessary, declare and initialize a DMF\_MODULE\_EVENT\_CALLBACKS structure using DMF\_MODULE\_EVENT\_CALLBACKS\_INIT(). This allows the Parent Module to know when the Child Module opens and closes.
7. Finally, call DMF\_DmfModuleAdd() using the above structures.
8. Perform the above steps for every Child Module that is to be added to the Parent Module.
9. After the callback returns, DMF will create and attach the Child Modules to the Parent Module. DMF will then dispatch callbacks to all the Modules in the tree as needed.

### Annotated ChildModulesAdd Callback

This example shows the callback referenced above. This callback function adds a single Child Module to its Parent Module. There are many examples of Modules that have Child Modules in the Library.

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

VOID

DMF\_ResourceHub\_ChildModulesAdd(

\_In\_ DMFMODULE DmfModule,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* DmfParentModuleAttributes,

\_In\_ PDMFMODULE\_INIT DmfModuleInit

)

/\*++

Routine Description:

Configure and add the required Child Modules to the given Parent Module.

Arguments:

DmfModule - The given Parent Module.

DmfParentModuleAttributes - Pointer to the parent DMF\_MODULE\_ATTRIBUTES structure.

DmfModuleInit - Opaque structure to be passed to DMF\_DmfModuleAdd.

Return Value:

None

--\*/

{

// Steps 1 to 3 happen in the Module’s create function.

//

// Step 4: Declare an instance of DMF\_MODULE\_ATTRIBUTES.

//

DMF\_MODULE\_ATTRIBUTES moduleAttributes;

DMF\_CONFIG\_IoctlHandler ioctlHandlerModuleConfig;

UNREFERENCED\_PARAMETER(DmfParentModuleAttributes);

UNREFERENCED\_PARAMETER(DmfModule);

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_ResourceHub);

// Steps 5: Inititialize the DMF\_MODULE\_ATTRIBUTES and the Module’s CONFIG structure.

//

// IoctlHandler

// ------------

//

DMF\_CONFIG\_IoctlHandler\_AND\_ATTRIBUTES\_INIT(&ioctlHandlerModuleConfig,

&moduleAttributes);

// NOTE: No GUID is necessary because device interface is not created.

//

ioctlHandlerModuleConfig.AccessModeFilter = IoctlHandler\_AccessModeDefault;

ioctlHandlerModuleConfig.EvtIoctlHandlerAccessModeFilter = NULL;

ioctlHandlerModuleConfig.IoctlRecordCount = ARRAYSIZE(ResourceHub\_IoctlSpecification);

ioctlHandlerModuleConfig.IoctlRecords = ResourceHub\_IoctlSpecification;

// Step 6: Set callbacks to Parent. (Ommited as it is not necessary in this example.)

//

// Step 7: Tell DMF to add this Module as a Child Module of the Parent Module.

//

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

NULL);

FuncExitVoid(DMF\_TRACE\_ResourceHub);

}

#pragma code\_seg()

# DMF Library Include File

DMF is designed so that Modules can be easily packaged and, optionally, distributed to external teams. DMF itself contains the core framework files, as well as a Library that contains Modules that are useful for many driver developers.

Generally, speaking, a team that uses DMF will create their own Library of Modules that contain code that is specific for the device drivers they work on. These Modules, of course, can be built using preexisting Modules in DMF or, even Modules distributed by 3rd parties.

It is a best practice for each team to build their own Library of Modules. To do so is easy as a template Library exists. That template Library has a project file as well as a Library Include File that are copied to make the initial project for the new Library.

## Library Include File

Every Library has a Library Include File. This is a crucial file that contains the following:

1. DMF framework definitions.
2. Definitions needed to use Modules that come with DMF.
3. Definitions needed to use Modules in the (new) Library.
4. Includes for every Module in the (new) Library.

It is important to not that items one and two above are already included in DmfModules.Library.h. So, a Library Include File generally includes that file to satisfy requirements one and two above.

### Using the Library Include File

Any Client that wants to use a Module in a specific Library needs to include that Library’s Library Include File. This is true for both Module authors writing Modules for that Library as well as Client drivers that want to use Modules in the specific Library. When any Client includes the Library Include File, all of DMF as well as the Modules that come with DMF are accessible.

### Sample Library Include File

It is critical that new Library Include Files contain exactly the same structure as other Library Include Files in order to ensure that Modules in the new Library work seamlessly with various Clients, including both C and C++ Clients.

Here is an example of a hypothetical Library which depends on the DMF Library and contains 3 Modules:

/\*++

Copyright (c) Microsoft Corporation. All rights reserved.

Module Name:

DmfModules.NewLibrary.h

Abstract:

Definitions specific for the "NewLibrary" DMF Library.

Environment:

Kernel/User mode

--\*/

#pragma once

// NOTE: The definitions in this file must be surrounded by this annotation to ensure

// that both C and C++ Clients can easily compile and link with Modules in this Library.

//

#if defined(\_\_cplusplus)

extern "C"

{

#endif

// Include DMF Framework and Public Modules.

//

#include "..\Modules.Library\DmfModules.Library.h"

// Other library specific includes.

//

// All the Modules in this Library.

//

#include "Dmf\_NewModule0.h"

#include "Dmf\_NewModule1.h"

#include "Dmf\_NewModule2.h"

#if defined(\_\_cplusplus)

}

#endif

// eof: DmfModules.NewLibrary.h

//

## Module Dependencies

Note that there may be dependencies between different Modules in a Library. In that case, the order of the includes for each Module is set by the author to resolve those dependencies so that Clients do not need to so. In the example above, since Dmf\_NewModule2.h needs definitions in Dmf\_NewModule1.h, Dmf\_NewModule1.h is listed before Dmf\_NewModule2.h. This is often the case when, for example, Dmf\_NewModule2 instantiates a Child Module of type Dmf\_NewModule1 and the Config for Dmf\_NewModule2 contains definitions in Dmf\_NewModule1.

# DMF Client Driver API Reference

This section lists and explains all the structures and functions that DMF Client Drivers use to work with DMF.

DMF Client Drivers that have a Device Context need to perform the following tasks:

1. Hook DMF into WDF.
2. Create the DMF Device.
3. Tell DMF what Modules to use.

DMF Client Drivers that do not have a Device Context only need to perform step 3 above.

After that, the Client Driver is free to use the Modules via the Module Methods.

The APIs in this section allow a Client Driver to perform these steps.

## Client Driver DMF Structures

This section lists and explains all the structures that DMF Client Drivers use to work with DMF.

### PDMFDEVICE\_INIT

A data structure allocated using DMF\_DmfDeviceInitAllocate(). This structure is used by other DMF APIs in order to initialize DMF in a Client Driver. This is an opaque structure that the Client does not need to set directly. DMF will set the elements of this structure as its APIs are called.

### DMF\_EVENT\_CALLBACKS

This structure tells DMF how to initialize. Specifically, it tells DMF the name of the function that DMF will call when it is ready to get the list of Modules to instantiate.

Use DMF\_EVENT\_CALLBACKS\_INIT() to initialize this structure. Next, set the EvtDmfDeviceModulesAdd member to the function that the Client Driver uses to add Modules that will be instantiated. Then, pass that structure to DMF\_DmfDeviceInitSetEventCallbacks.

|  |  |
| --- | --- |
| **Member** | **Description** |
| Size | Size of the structure initialized by DMF\_EVENT\_CALLBACKS\_INIT(). |
| PFN\_DMF\_DEVICE\_MODULES\_ADD EvtDmfDeviceModulesAdd | Set the EvtDmfDeviceModulesAdd member to the function that the Client Driver uses to add Modules that will be instantiated. |

## Client Driver DMF Initialization Macros

This section lists and explains all the macros that DMF Clients use to **initialize** DMF. Client Drivers **only** need to use these macros if the Client Driver **does not** have its own AddDevice callback.

### DMF\_DEFAULT\_DRIVERENTRY

DMF\_DEFAULT\_DRIVERENTRY(DmfDriverEntry, DmfDriverContextCleanup, DmfEvtDeviceAdd)

This macro is used by the Client Driver to tell DMF to use its own DriverEntry function.

#### Parameters

|  |  |
| --- | --- |
| DmfDriverEntry | This parameter **must** be set to “DriverEntry”. |
| DmfDriverContextCleanup | This parameter can be any name of the Client Driver’s choosing, but the same name must be used in DMF\_DEFAULT\_DRIVERCLEANUP. |
| DmfEvtDeviceAdd | This parameter can be any name of the Client Driver’s choosing, but the same name must be used in DMF\_DEFAULT\_DEVICEADD. |

#### Returns

None

#### Remarks

* Client Drivers use this macro to use DMF’s DriverEntry so that the Client Driver does not need to implement its own DriverEntry.
* This function will initialize WPP tracing.

### DMF\_DEFAULT\_DEVICEADD

DMF\_DEFAULT\_DEVICEADD(DmfEvtDeviceAdd, DmfDeviceModuleAdd)

This macro is used by the Client Driver to tell DMF to use its DMF’s AddDevice function.

#### Parameters

|  |  |
| --- | --- |
| DmfEvtDeviceAdd | This parameter can be any name of the Client Driver’s choosing |
| DmfDeviceModuleAdd | This parameter can be any name of the Client Driver’s choosing. The Client Driver must have a function that uses the same name with the proper signature. DMF uses this name to tell DMF the name of the function that the Client Driver uses to tell DMF the list of Modules to instantiate. |

#### Returns

None

#### Remarks

* Client Drivers use this macro to use DMF’s AddDevice so that the Client Driver does not need to implement its own AddDevice.

### DMF\_DEFAULT\_DRIVERCLEANUP

DMF\_DEFAULT\_DRIVERCLEANUP(DmfDriverContextCleanup)

This macro is used by the Client Driver to tell DMF to use its own DriverContextCleanup function.

#### Parameters

|  |  |
| --- | --- |
| DmfDriverContextCleanup | This parameter can be any name of the Client Driver’s choosing |

#### Returns

None

#### Remarks

* Client Drivers use this macro to use DMF’s DriverContextCleanup so that the Client Driver does not need to implement its own DriverContextCleanup.
* This function will uninitialize WPP tracing.

## Client Driver DMF Initialization Functions

This section lists and explains all the functions that DMF Clients use to **initialize** DMF. Client Drivers only need to use these functions if the Client Driver **has** its own AddDevice callback.

### DMF\_DmfDeviceInitAllocate

PDMFDEVICE\_INIT

DMF\_DmfDeviceInitAllocate(

\_In\_opt\_ PWDFDEVICE\_INIT DeviceInit

)

This function is called by the Client Driver to create an instance of PDMFDEVICE\_INIT.

#### Parameters

|  |  |
| --- | --- |
| PWDFDEVICE\_INIT DeviceInit | The structure passed by WDF to the Client Driver’s AddDevice function. |

#### Returns

|  |  |
| --- | --- |
| PWDFDEVICE\_INIT | An data structure passed to other DMF initialization functions. |

#### Remarks

* Use this function only if the Client Driver has a Client Driver specific AddDevice callback.

#### Example

WDF\_PNPPOWER\_EVENT\_CALLBACKS pnpPowerCallbacks;

WDF\_OBJECT\_ATTRIBUTES deviceAttributes;

PDEVICE\_CONTEXT deviceContext;

WDFDEVICE device;

WDF\_DEVICE\_STATE deviceState;

NTSTATUS ntStatus;

WDF\_TIMER\_CONFIG timerConfig;

WDF\_OBJECT\_ATTRIBUTES timerAttributes;

DMF\_EVENT\_CALLBACKS dmfCallbacks;

DMF\_CONFIG\_BranchTrack branchTrackModuleConfig;

PDMFDEVICE\_INIT dmfDeviceInit;

PAGED\_CODE();

FuncEntry(TRACE\_DEVICE);

dmfDeviceInit = DMF\_DmfDeviceInitAllocate(DeviceInit);

DMF\_DmfDeviceInitHookFileObjectConfig(dmfDeviceInit,

NULL);

DMF\_DmfDeviceInitHookPowerPolicyEventCallbacks(dmfDeviceInit,

NULL);

WDF\_PNPPOWER\_EVENT\_CALLBACKS\_INIT(&pnpPowerCallbacks);

pnpPowerCallbacks.EvtDeviceD0Entry = DmfSampleEvtDeviceD0Entry;

pnpPowerCallbacks.EvtDeviceD0Exit = DmfSampleEvtDeviceD0Exit;

pnpPowerCallbacks.EvtDevicePrepareHardware = DmfSampleEvtDevicePrepareHardware;

DMF\_DmfDeviceInitHookPnpPowerEventCallbacks(dmfDeviceInit,

&pnpPowerCallbacks);

WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit,

&pnpPowerCallbacks);

WdfDeviceInitSetDeviceType(DeviceInit,

FILE\_DEVICE\_UNKNOWN);

WdfDeviceInitSetIoType(DeviceInit,

WdfDeviceIoBuffered);

WdfDeviceInitSetExclusive(DeviceInit,

FALSE);

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&deviceAttributes,

DEVICE\_CONTEXT);

deviceAttributes.EvtCleanupCallback = DmfSampleEvtDeviceContextCleanup;

ntStatus = WdfDeviceCreate(&DeviceInit,

&deviceAttributes,

&device);

if (!NT\_SUCCESS(ntStatus))

{

goto Exit;

}

deviceContext = DeviceContextGet(device);

RtlSecureZeroMemory(deviceContext,

sizeof(DEVICE\_CONTEXT));

deviceContext->WdfDevice = device;

DMF\_EVENT\_CALLBACKS\_INIT(&dmfCallbacks);

dmfCallbacks.EvtDmfDeviceModulesAdd = DmfDeviceModulesAdd;

DMF\_DmfDeviceInitSetEventCallbacks(dmfDeviceInit,

&dmfCallbacks);

ntStatus = DMF\_ModulesCreate(device,

&dmfDeviceInit,

deviceContext);

### DMF\_DmfDeviceInitFree

PDMFDEVICE\_INIT

DMF\_DmfDeviceInitFree(

\_In\_ PWDFDEVICE\_INIT DeviceInit

)

This function frees the PDMFDEVICE\_INIT structure that was previously allocated using DMF\_DmfDeviceInitAllocate.

#### Parameters

|  |  |
| --- | --- |
| PDMFDEVICE\_INIT DmfDeviceInit | The data structure created using DMF\_DmfDeviceInitAllocate. |

#### Returns

None

#### Remarks

Use this function if an error was encountered after using DMF\_DmfDeviceInitAllocate and the structure will not be returned.

### DMF\_DmfDeviceInitHookPnpPowerEventCallbacks

VOID

DMF\_DmfDeviceInitHookPnpPowerEventCallbacks(

\_In\_ PDMFDEVICE\_INIT DmfDeviceInit,

\_Inout\_opt\_ PWDF\_PNPPOWER\_EVENT\_CALLBACKS PnpPowerEventCallbacks

)

This function allows DMF to route all the Client Driver’s Pnp Power callbacks to itself (and the instantiated Modules) before calling the Client Driver’s corresponding callbacks.

#### Parameters

|  |  |
| --- | --- |
| PDMFDEVICE\_INIT DmfDeviceInit | The data structure created using DMF\_DmfDeviceInitAllocate(). |
| PWDF\_PNPPOWER\_EVENT\_CALLBACKS PnpPowerEventCallbacks | The Client Driver passes an initialized instance of this structure (after the Client Driver has set its own callback functions) **and before** calling WdfDeviceInitSetPnpPowerEventCallbacks(). |

#### Returns

None

#### Remarks

* Use this function only if the Client Driver has a Client Driver specific AddDevice callback.
* If the Client Driver has an AddDevice callback, the Client Driver must always call this function even if the Client Driver does not register for Pnp Power callbacks. If the Client Driver does not register for Power callbacks, pass NULL via PnpPowerEventCallbacks.

#### Example

* See DMF\_DmfDeviceInitAllocate.

### DMF\_DmfDeviceInitHookPowerPolicyEventCallbacks

VOID

DMF\_DmfDeviceInitHookPowerPolicyEventCallbacks(

\_In\_ PDMFDEVICE\_INIT DmfDeviceInit,

\_Inout\_opt\_ PWDF\_POWER\_POLICY\_EVENT\_CALLBACKS PowerPolicyEventCallbacks

)

This function allows DMF to route all the Client Driver’s Pnp Power Policy callbacks to itself (and the instantiated Modules) before calling the Client Driver’s corresponding callbacks.

#### Parameters

|  |  |
| --- | --- |
| PDMFDEVICE\_INIT DmfDeviceInit | The data structure created using DMF\_DmfDeviceInitAllocate(). |
| PWDF\_POWER\_POLICY\_EVENT\_CALLBACKS PowerPolicyEventCallbacks | The Client Driver passes an initialized instance of this structure (after the Client Driver has set its own callback functions) **and before** calling WdfDeviceInitSetPnpPowerPolicyEventCallbacks(). |

#### Returns

None

#### Remarks

* Use this function only if the Client Driver has a Client Driver specific AddDevice callback.
* If the Client Driver has an AddDevice callback, the he Client Driver must always call this function even if the Client Driver does not register for Pnp Power Policy callbacks. If the Client Driver does not register for Power Policy callbacks, pass NULL via PowerPolicyEventCallbacks.

#### Example

* See DMF\_DmfDeviceInitAllocate.

### DMF\_DmfDeviceInitHookFileObjectConfig

VOID

DMF\_DmfDeviceInitHookFileObjectConfig(

\_In\_ PDMFDEVICE\_INIT DmfDeviceInit,

\_Inout\_opt\_ PWDF\_FILEOBJECT\_Config FileObjectConfig

)

This function allows DMF to route all the Client Driver’s File Object callbacks to itself (and the instantiated Modules) before calling the Client Driver’s corresponding callbacks.

#### Parameters

|  |  |
| --- | --- |
| PDMFDEVICE\_INIT DmfDeviceInit | The data structure created using DMF\_DmfDeviceInitAllocate(). |
| PWDF\_FILEOBJECT\_Config FileObjectConfig | The Client Driver passes an initialized instance of this structure (after the Client Driver has set its own callback functions) **and before** calling WDF\_FILEOBJECT\_Config\_INIT(). |

#### Returns

None

#### Remarks

* Use this function only if the Client Driver has a Client Driver specific AddDevice callback.
* The Client Driver must always call this function even if the Client Driver does not register for File Object callbacks. If the Client Driver does not register for File Object callbacks, pass NULL via FileObjectConfig.

#### Example

* See DMF\_DmfDeviceInitAllocate.

### DMF\_DmfDeviceInitHookQueueConfig

VOID

DMF\_DmfDeviceInitHookQueueConfig(

\_In\_ PDMFDEVICE\_INIT DmfDeviceInit,

\_Inout\_ PWDF\_IO\_QUEUE\_Config QueueConfig

)

This function allows DMF to route all the Client Driver’s Default WDF Queue Object callbacks to itself (and the instantiated Modules) before calling the Client Driver’s corresponding callbacks.

#### Parameters

|  |  |
| --- | --- |
| PDMFDEVICE\_INIT DmfDeviceInit | The data structure created using DMF\_DmfDeviceInitAllocate(). |
| PWDF\_IO\_QUEUE\_Config QueueConfig | The Client Driver passes an initialized instance of this structure (after the Client Driver has set its own callback functions) **and before** calling WdfIoQueueCreate(). |

#### Returns

None

#### Remarks

* Use this function only if the Client Driver has a Client Driver specific AddDevice callback and the Client Driver creates a default WDF queue.
* If the Client Driver does not create a default queue, it is not necessary for the Client Driver to call this function because DMF will do so.

### DMF\_DmfDeviceInitSetEventCallbacks

VOID

DMF\_DmfDeviceInitSetEventCallbacks(

\_In\_ PDMFDEVICE\_INIT DmfDeviceInit,

\_In\_ PDMF\_EVENT\_CALLBACKS DmfEventCallbacks

)

This function allows the Client Driver to initialize a structure that DMF uses to initialize DMF. It has the name of the function in the Client Driver that tells DMF the list of all the Modules that DMF should instantiate on behalf of the Client Driver.

#### Parameters

|  |  |
| --- | --- |
| PDMFDEVICE\_INIT DmfDeviceInit | The data structure created using DMF\_DmfDeviceInitAllocate(). |
| PDMF\_EVENT\_CALLBACKS DmfEventCallbacks | The Client Driver passes an initialized instance of this structure. Use DMF\_EVENT\_CALLBACKS\_INIT() to initialize this structure. Then, prior to calling this API, set the name of the Client Driver’s function (EvtDmfDeviceModulesAdd) that DMF will call to get the list of Modules to instantiate. |

#### Returns

None

#### Remarks

* Use this function only if the Client Driver has a Client Driver specific AddDevice callback and the Client Driver creates a default queue.
* In rare cases, the Client Driver may not instantiate any Modules. If so, it is not necessary to initialize EvtDmfDeviceModulesAdd.

#### Example

* See DMF\_DmfDeviceInitAllocate.

### DMF\_ModulesCreate

NTSTATUS

DMF\_ModulesCreate(

\_In\_ WDFDEVICE Device,

\_In\_ PDMFDEVICE\_INIT\* DmfDeviceInitPointer,

\_In\_opt\_ PVOID ClientDriverContext

)

The Client Driver uses this function to tell DMF to instantiate the Modules that the Client Driver will use. After this function is called, DMF will call the Client Driver’s function that tells DMF the list of Modules to instantiate. After that function returns, the Client Driver’s Modules will be instantiated and they will start receiving callbacks that the Modules support.

Furthermore, after this function returns, the Client Driver may start using the instantiated Module’s Methods.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The Client Driver’s WDFDEVICE. |
| PDMFDEVICE\_INIT\* DmfDeviceInitPointer | The address of an allocated and initialized instance of PDMFDEVICE\_INIT. |
| PVOID ClientDriverContext | A ClientDriver specific context. This is optional and generally not used. |

#### Returns

NTSTATUS. If an error is returned, the Client Driver must return this status to WDF and the driver cannot load.

#### Remarks

* After this function is called, DMF will call the Client Driver’s function that tells DMF the list of Modules to instantiate. After that function returns, the Client Driver’s Modules will be instantiated and they will start receiving callbacks that the Modules support.
* Furthermore, after this function returns, the Client Driver may start using the instantiated Modules.

#### Example

* See DMF\_DmfDeviceInitAllocate.

## Client Driver DMF Callbacks

This section explains the callback functions that DMF uses to call back into the Client Driver. Currently, there is only a single callback function.

### EVT\_DMF\_DEVICE\_MODULES\_ADD

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

VOID

EVT\_DMF\_DEVICE\_MODULES\_ADD(

\_In\_ WDFDEVICE Device,

\_In\_ PDMFMODULE\_INIT DmfModuleInit

);

DMF calls this callback function in the Client Driver when it is ready to accept the list of Modules that the Client Driver wants to instantiate.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The Client Driver’s WDFDEVICE (which corresponds to its FDO). |
| PDMFMODULE\_INIT DmfModuleInit | An opaque structure that is passed from DMF. It is passed to DMF\_DmfModuleAdd(). |

#### Returns

None

#### Remarks

* After this function returns, DMF will instantiate all the Modules specified by the Client Driver using this callback.

# DMF Client API Reference

A DMF Client is either a Client Driver (a driver that uses DMF) or a DMF Module that instantiates a Child Module.

The API in this section used by both types of Clients:

* Client Drivers uses these APIs to instantiate a Module.
* Modules use these APIs to create Child Modules.

## DMF Client Structures

### DMF\_CONFIG\_[ModuleName]

Most Modules have a Config structure that is specific to each Module. This Config structure contains parameters that tell the Module how it will be used by the Client. For example, a Module that exposes a FIFO usually has a Config that contains, at minimum, the number of entries in the FIFO as well as the size of each entry in the FIFO.

The Client initializes the Module’s Config using DMF\_CONFIG\_[ModuleName]\_AND\_ATTRIBUTES\_INIT(). Afterward, the Client overwrites elements of the structure to suit the Client’s needs.

### DMF\_MODULE\_ATTRIBUTES

Clients use this structure when they create Modules. This structure must be initialized using one of two functions:

* DMF\_CONFIG\_[ModuleName]\_AND\_ATTRIBUTES\_INIT

Clients use this function when they instantiate Modules that have a Config structure.

* DMF\_[ModuleName]\_ATTRIBUTES\_INIT

Client use this function when they instantiate Modules that do not have a Config structure.

After the structure is initialized using one of the two above functions, Clients must not modify the contents of the structure. There is one exception: If the Module requires a Transport, the TransportsCreator member must be set to a callback function. This callback function is called after the Module is created and allows the Client to create the Transport Module that is required.

This structure has an element called PassiveLevel which is not commonly used. The Client must set PassiveLevel = TRUE in cases where the following two conditions are met:

1. The Module options are set to DMF\_MODULE\_OPTIONS\_DISPATCH\_MAXIMUM.
2. The Client wishes the Module to create PASSIVE\_LEVEL locks because the Module will allocate Paged Pool on behalf of the Client.

### DMF\_MODULE\_EVENT\_CALLBACKS

Clients use this structure when they create Modules that support the DMF\_[ModuleName]\_NotificationRegister callbacks. These types of Modules are opened asynchronously: The Client sets this callbacks in this structure to know when the Module has actually been opened and is ready for use. Module’s that have dependencies on resources that asynchronously appear/disappear (e.g. HID devices) use this method.

**This structure is used commonly because many Modules instantiate Child Modules that support the DMF\_[ModuleName]\_NotificationRegister callback.**

Use DMF\_MODULE\_EVENT\_CALLBACKS\_INIT() to initialize this structure. Then, set the members in the following table as needed:

|  |  |
| --- | --- |
| **Member** | **Description** |
| EVT\_DMF\_MODULE\_OnDeviceNotificationOpen\* EvtModuleOnDeviceNotificationOpen | When the Module’s notification function has detected that the underlying resource it needs is available and the Module is ready to be opened, this callback is called. The Client has an opportunity to fail the open operation. |
| EVT\_DMF\_MODULE\_OnDeviceNotificationPostOpen\* EvtModuleOnDeviceNotificationPostOpen | After the Module has been opened, this callback is called. This tells the Client that the Module is ready for use (meaning that its Module Methods may be called). |
| EVT\_DMF\_MODULE\_OnDeviceNotificationPreClose\* EvtModuleOnDeviceNotificationPreClose | When the Module’s notification function has detected that the underlying resource it needs is no longer available and the Module will be closed, this callback is called. This tells the Client the Module will close. |
| EVT\_DMF\_MODULE\_OnDeviceNotificationClose\* EvtModuleOnDeviceNotificationClose | After the Module has been closed, this callback is called. This tells the Client that the Module is no longer ready for use (meaning that its Module Methods may not be called). |

## DMF Client API

**These functions are used by Clients to create the structures that tell Modules how to instantiate.**

### DMF\_CONFIG\_[ModuleName]\_AND\_ATTRIBUTES\_INIT

VOID

DMF\_CONFIG\_[ModuleName]\_AND\_ATTRIBUTES\_INIT(

\_Out\_ DMF\_CONFIG\_[ModuleName]\* ModuleConfig,

\_Out\_ DMF\_MODULE\_ATTRIBUTES\* ModuleAttributes

);

This Client uses this function to initialize a Module’s Config and DMF\_MODULE\_ATTRIBUTES of a Module that is to be instantiated.

#### Parameters

|  |  |
| --- | --- |
| DMF\_CONFIG\_[ModuleName]\* | The address of a locally allocated Config structure of a Module that is to be instantiated. This function will initialize the structure. Afterward, the Client may initialize members of this structure as needed. Note: These members are Module specific. |
| DMF\_MODULE\_ATTRIBUTES\* | The address of a locally allocated DMF\_MODULE\_ATTRIBUTES structure that is to be initialized by this function. NOTE: Use the same structure for all Modules that are allocated. The Client must **not** modify the contents of the structure after this function is called except for the case where the Module that is instantiated is requires a Transport Module. In that case the TransportCreator member must be set. |

#### Returns

None.

Remarks

* The prototype for this function is defined in every Module’s .h file using a DMF Macro.
* After calling this function, the Client may (and usually does) override elements of the Module’s Config to suit the purposes of the Client.

### DMF\_[ModuleName]\_AND\_ATTRIBUTES\_INIT

VOID

DMF\_[ModuleName]\_ATTRIBUTES\_INIT(

\_Out\_ DMF\_MODULE\_ATTRIBUTES\* ModuleAttributes

);

This Client uses this function to initialize the DMF\_MODULE\_ATTRIBUTES of a Module that is to be instantiated.

#### Parameters

|  |  |
| --- | --- |
| DMF\_MODULE\_ATTRIBUTES\* | The address of a locally allocated DMF\_MODULE\_ATTRIBUTES structure that is to be initialized by this function. NOTE: Use the same structure for all Modules that are allocated. The Client must **not** modify the contents of the structure after this function is called except for the case where the Module that is instantiated is requires a Transport Module. In that case the TransportCreator member must be set. |

#### Returns

None.

Remarks

* The prototype for this function is defined in every Module’s .h file using a DMF Macro.

### DMF\_MODULE\_EVENT\_CALLBACKS\_INIT

VOID

DMF\_MODULE\_EVENT\_CALLBACKS\_INIT(

\_Out\_ DMF\_MODULE\_EVENT\_CALLBACKS\* ModuleEventCallbacks

);

This Client uses this function to initialize the DMF\_MODULE\_EVENT\_CALLBACKS structure of a Module that is to be instantiated.

#### Parameters

|  |  |
| --- | --- |
| DMF\_MODULE\_EVENT\_CALLBACKS\* | The address of a locally allocated DMF\_MODULE\_EVENT\_CALLBACKS structure that is to be initialized by this function. |

#### Returns

None.

Remarks

* After calling this function, the Client sets the callbacks as needed in the initialized structure.

### DMF\_DmfModuleAdd

VOID

DMF\_DmfModuleAdd(

\_Inout\_ PDMFMODULE\_INIT DmfModuleInit,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* ModuleAttributes,

\_In\_opt\_ WDF\_OBJECT\_ATTRIBUTES\* ObjectAttributes,

\_In\_opt\_ DMFMODULE\* ResultantDmfModule

)

**Client Drivers** call this function from its DmfModulesAdd callback one time for each instance of each Module it wants to use while its corresponding WDFDEVICE is active. (Modules are “added” to the instance of the WDFDEVICE.) Prior to calling this function, the Module specific Config must be properly initialized.

**Client Modules (Parent Modules)** call this function from its ChildModulesAdd callback one time for each instance of each Child Module it wants to use while the Parent Module is instantiated. (Modules are “added” to the Parent Module’s list of Child Modules.) Prior to calling this function, the Child Module specific Config must be properly initialized.

#### Parameters

|  |  |
| --- | --- |
| PDMFMODULE\_INIT DmfModuleInit | A structure passed from DMF to the Client Drivers Module Add function. |
| DMF\_MODULE\_ATTRIBUTES\* ModuleAttributes | The structure that contains information about the Module to add. These are Module specific attributes. |
| WDF\_OBJECT\_ATTRIBUTES\* ObjectAttributes | A structure that contains information about the Module to add. These are DMF specific attributes. |
| DMFMODULE\* ResultantDmfModule | An address where the Client Driver can receive the handle of the instantiated Module. This is optional and only necessary if the Client Driver will call the Module’s Methods. |

#### Returns

None

#### Remarks

* This function adds the information passed to a list but does not actually instantiate the Module. After this function returns, all the DMF instantiates all the Module’s in the list.
* Each Module’s Config is unique to each Module. The author must consult the Module’s .h file or .txt file to understand how to initialize that structure. Every Module has a Module specific macro that initializes the Config structure.
* This function is called by all DMF drivers, those that do and do not have a DEVICE\_CONTEXT.
* For more information and examples, please see the section, “Instantiating Modules”.

#### Example

#pragma code\_seg("PAGED")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

VOID

DmfDeviceModulesAdd(

\_In\_ WDFDEVICE Device,

\_In\_ PDMFMODULE\_INIT DmfModuleInit

)

/\*++

Routine Description:

Add all the Dmf Modules used by this driver.

Arguments:

Device - WDFDEVICE handle.

DmfModuleInit - Opaque structure to be passed to DMF\_DmfModuleAdd.

Return Value:

NTSTATUS

--\*/

{

DMF\_MODULE\_ATTRIBUTES moduleAttributes;

DMF\_CONFIG\_DeviceInterfaceTarget moduleConfigDeviceInterfaceTarget;

DMF\_MODULE\_EVENT\_CALLBACKS moduleEventCallbacks;

UNREFERENCED\_PARAMETER(Device);

PAGED\_CODE();

// DeviceInterfaceTarget

// ---------------------

//

DMF\_CONFIG\_DeviceInterfaceTarget\_AND\_ATTRIBUTES\_INIT(&moduleConfigDeviceInterfaceTarget,

&moduleAttributes);

moduleConfigDeviceInterfaceTarget.DeviceInterfaceTargetGuid = GUID\_DEVINTERFACE\_OSRUSBFX2;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.BufferCountOutput = 4;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.BufferOutputSize = sizeof(SWITCH\_STATE);

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.ContinuousRequestCount = 4;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.PoolTypeOutput = NonPagedPoolNx;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.PurgeAndStartTargetInD0Callbacks = FALSE;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.ContinuousRequestTargetIoctl = IOCTL\_OSRUSBFX2\_GET\_INTERRUPT\_MESSAGE;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.EvtContinuousRequestTargetBufferOutput = SwitchBarSwitchChangedCallback;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.RequestType = ContinuousRequestTarget\_RequestType\_Ioctl;

moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.ContinuousRequestTargetMode = ContinuousRequestTarget\_Mode\_Automatic;

// These callbacks tell us when the underlying target is available.

//

DMF\_MODULE\_ATTRIBUTES\_EVENT\_CALLBACKS\_INIT(&moduleAttributes,

&moduleEventCallbacks);

moduleEventCallbacks.EvtModuleOnDeviceNotificationPostOpen = SwitchBar\_OnDeviceArrivalNotification;

DMF\_DmfModuleAdd(DmfModuleInit,

&moduleAttributes,

WDF\_NO\_OBJECT\_ATTRIBUTES,

NULL);

### DMF\_ModuleDestroy

VOID

DMF\_Module\_Destroy(

\_In\_ DMFMODULE DmfModule

);

Given an instance of a Module, tells DMF to destroy the Module.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

#### Returns

None

Remarks

* This function is used by Clients that instantiate Modules that are dynamically created. Otherwise, DMF calls this API automatically as needed.
* This function is used by Modules when they support the DMF\_[ModuleName]\_Destroy callback. **It is rare that this is the case, however.**

# DMF Module API Reference

DMF provides an API that Module authors use to create Modules. This API is not visible to Client Drivers. The Module API allows Module code to look and act like its own small driver. This API provides a way for the Module to set callbacks that are called by DMF in much the same way that WDF calls a WDF driver’s callbacks.

## Module Enumerations

This section lists all the enumeration types that are used by the DMF Module API.

### DmfModuleOpenOption

This enumeration tells DMF when to open/close the Module. Specifically, it tells DMF when to call the Module’s Open and Close callbacks.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| DMF\_MODULE\_OPEN\_OPTION\_OPEN\_PrepareHardware | Tells DMF that the Module’s Open callback should be called when the Client Driver receives an EvtDevicePrepareHardware callback. The Module’s Close callback will be called when the Client Driver receives an EvtDeviceReleaseHardware callback. |
| DMF\_MODULE\_OPEN\_OPTION\_NOTIFY\_PrepareHardware | Tells DMF that the Module’s NotificationRegister callback should be called when the Client Driver receives an EvtDevicePrepareHardware callback. In this case, the Module will decide when the Open/Close callbacks are called (usually when the underlying interface has appeared/disappeared). The Module’s NotificationUnregister callback will be called when the Client Driver receives an EvtDeviceReleaseHardware callback. |
| DMF\_MODULE\_OPEN\_OPTION\_OPEN\_D0Entry | Tells DMF that the Module’s Open callback should be called when the Client Driver receives an EvtDeviceD0Entry callback. The Module’s Close callback will be called when the Client Driver receives an EvtDeviceD0Exit callback. |
| DMF\_MODULE\_OPEN\_OPTION\_NOTIFY\_D0Entry | Tells DMF that the Module’s notification register callback should be called when the Client Driver receives an EvtDeviceD0Entry callback. In this case, the Module will decide when the Open/Close callbacks are called (usually when the underlying interface has appeared/disappeared). The Module’s NotificationUnregister callback will be called when the Client Driver receives an EvtDeviceD0Exit callback. |
| DMF\_MODULE\_OPEN\_OPTION\_OPEN\_Create | Tells DMF that the Module should be opened/closed when the Module is created/destroyed. This is common for Modules that do not interact with hardware and, instead, expose support for data structures that just require memory resources. DMF will call the Module’s Open callback soon after the Module is created. DMF will call the Module’s Close callback right before the Module is destroyed. |
| DMF\_MODULE\_OPEN\_OPTION\_NOTIFY\_Create | Tells DMF that the Module’s Notification register/unregister callbacks should called when the Module is create/destroyed. DMF will call the Module’s NotificationRegister callback soon after the Module is created. DMF will call the Module’s NotificationUnregister callback right before the Module is destroyed. |

## Module Structures

This section discusses structures that the DMF Module API uses.

### DMF\_MODULE\_DESCRIPTOR

This structure holds the information about a Module that tells DMF how to create and open the Module and how to use it while it is instantiated. DMF uses this structure to allocate resources for the Module as well as to know what callbacks the Module supports.

Always use this function to initialize the structure before updating it specifically for the Module:

DMF\_MODULE\_DESCRIPTOR\_INIT()

|  |  |
| --- | --- |
| **Member** | **Description** |
| Size | Indicates the size of this structure. It is initialized by DMF\_MODULE\_DESCRIPTOR\_INIT(). |
| Module Name | The name of the Module. It should match the name indicated by the Module’s file name. This field is used for debugging purposes to make it easy to identify the Module. |
| ModuleOptions | Indicates whether the Module’s locks are wait locks or spin locks. It is initialized by DMF\_MODULE\_DESCRIPTOR\_INIT(). |
| OpenOption | Indicates when DMF should open the Module. See DmfModuleOpenOption. |
| ModuleConfigSize | Indicates the size of the Module’s Config structure (if any). |
| EntrypointsDmf | Tells DMF what DMF specific callbacks this Module supports. |
| EntrypointsWdf | Tells DMF what WDF specific callbacks this Module supports. |
| BranchTrackInitialize | The name of the function that initializes BranchTrack for this Module. See the separate document that describes BranchTrack for more information. |
| NumberOfAuxiliaryLocks | The number of additional locks that should be created for this Module in addition to its default lock. |
| TransportMethod | Indicates the Module’s Transport Method. When this member is set, the Module may be instantiated by a Client as a Transport Module. |

### DMF\_ENTRYPOINTS\_DMF

This structure contains all the DMF specific callbacks the Module supports.

Always use this function to initialize the structure before updating it specifically for the Module:

DMF\_ENTRYPOINTS\_DMF\_INIT()

|  |  |
| --- | --- |
| **Member** | **Description** |
| Size | Indicates the size of this structure. It is initialized by DMF\_ENTRYPOINTS\_DMF\_INIT(). |
| ModuleInstanceDestroy | The callback function that is called when the Module is destroyed. Generally, it is not necessary for a Module to support this callback. In some rare cases, if an allocation or some resource is acquired in the Module’s Create function, then this callback can be used to release that allocation or resource. |
| DeviceResourcesAssign | DMF calls this callback to allow the Module to retrieve the Client’s Driver’s resources. |
| DeviceNotificationRegister | DMF calls this callback to allow the Module to register for notification of an event that will tell the Module its required dependencies are available or not available. |
| DeviceNotificationUnregister | DMF calls this callback to allow the Module to unregister for notification of an event that will tell the Module its required dependencies are available or not available. |
| DeviceOpen | DMF calls this callback to open the Module. Generally speaking, the Module uses this callback to prepare its Private Context in preparation for later calls to other callbacks or for calls to its Module Methods by the Client. |
| DeviceClose | DMF calls this callback to close the Module. Generally speaking, the Module uses this callback to do the inverse of what it did in the DeviceOpen callback. |
| ChildModulesAdd | DMF calls this callback so that the Module can tell DMF about the Child Module(s) it needs to create. |

### DMF\_ENTRYPOINTS\_WDF

This structure contains all the WDF specific callbacks the Module supports.

Always use this function to initialize the structure before updating it specifically for the Module:

DMF\_ENTRYPOINTS\_WDF\_INIT()

|  |  |
| --- | --- |
| **Member** | **Description** |
| Size | Indicates the size of this structure. It is initialized by DMF\_ENTRYPOINTS\_WDF\_INIT(). |
| ModulePrepareHardware | Routes EvtDevicePrepareHardware to the Module. (WDFDEVICE) |
| ModuleReleaseHardware | Routes EvtDeviceReleaseHardware to the Module. (WDFDEVICE) |
| ModuleD0Entry | Routes EvtDeviceD0Entry to the Module. (WDFDEVICE) |
| ModuleD0EntryPostInterruptsEnabled | Routes EvtDeviceD0EntryPostInterruptsEnabled to the Module. (WDFDEVICE) |
| ModuleD0ExitPreInterruptsDisabled | Routes EvtDeviceD0ExitPreInterruptsDisabled to the Module. (WDFDEVICE) |
| ModuleD0Exit | Routes EvtDeviceD0Exit to the Module. (WDFDEVICE) |
| ModuleDeviceIoControl | Routes EvtQueueDeviceIoControl to the Module. (WDFQUEUE) |
| ModuleInternalDeviceIoControl | Routes EvtQueueInternalDeviceIoControl to the Module. (WDFQUEUE) |
| ModuleSelfManagedIoCleanup | Routes EvtDeviceSelfManagedIoCleanup to the Module. (WDFDEVICE) |
| ModuleSelfManagedIoFlush | Routes EvtDeviceSelfManagedIoFlush to the Module. (WDFDEVICE) |
| ModuleSelfManagedIoInit | Routes EvtDeviceSelfManagedIoInit to the Module. (WDFDEVICE) |
| ModuleSelfManagedIoSuspend | Routes EvtDeviceSelfManagedIoSuspend to the Module. (WDFDEVICE) |
| ModuleSelfManagedIoRestart | Routes EvtDeviceSelfManagedIoRestart to the Module. (WDFDEVICE) |
| ModuleSurpriseRemoval | Routes EvtDeviceSurpriseRemoval to the Module. (WDFDEVICE) |
| ModuleQueryRemove | Routes EvtDeviceQueryRemove to the Module. (WDFDEVICE) |
| ModuleQueryStop | Routes EvtDeviceQueryStop to the Module. (WDFDEVICE) |
| ModuleRelationsQuery | Routes EvtDeviceRelationsQuery to the Module. (WDFDEVICE) |
| ModuleUsageNotificationEx | Routes EvtDeviceUsageNotificationEx to the Module. (WDFDEVICE) |
| ModuleArmWakeFromS0 | Routes EvtDeviceModuleArmWakeFromS0 to the Module. (WDFDEVICE) |
| ModuleWakeFromS0Triggered | Routes EvtDeviceWakeFromS0Triggered to the Module. (WDFDEVICE) |
| ModuleArmWakeFromSxWithReason | Routes EvtDeviceArmWakeFromSxWithReason to the Module. (WDFDEVICE) |
| ModuleDisarmWakeFromSx | Routes EvtDeviceDisarmWakeFromSx to the Module. (WDFDEVICE) |
| ModuleWakeFromSxTriggered | Routes EvtDeviceWakeFromSxTriggered to the Module. (WDFDEVICE) |
| ModuleFileCreate | Routes EvtFileCreate to the Module. (WDFDEVICE) |
| ModuleFileCleanup | Routes EvtFileCleanup to the Module. (WDFDEVICE) |
| ModuleFileClose | Routes EvtFileClose to the Module. (WDFDEVICE) |
| ModuleQueueIoRead | Routes EvtQueueIoRead to the Module. (WDFQUEUE) |
| ModuleQueueIoWrite | Routes EvtQueueIeWrite to the Module. (WDFQUEUE) |

## Module WDF Callbacks

This section discusses the WDF callbacks that Modules can support. Recall that when DMF is properly initialized in the Client Driver, DMF will receive all WDF callbacks from the Client and route the callbacks to each instantiated Module/Child Module.

When a Module receives these callbacks, it also receives all the parameters that are sent to the Client Driver. However, there is always one additional parameter that is first in the parameter list. That parameter is the Module handle for that Module.

Using that Module handle, the Module’s callback function will retrieve the Module’s Private Context and/or the Module’s Config data. The Client’s WDFDEVICE is also accessible via the Module handle using DMF\_ParentDeviceGet(). Then, the Module can handle the callback as it needs to.

Note the following points:

* When a Module’s WDF callback is called, the Module has no information about what the Client Driver or any other Module that will receive that same callback will do. Therefore, it should assume nothing and only act upon the information contained in the parameters passed to the callback as well as the Module’s Private Context
* **Using these callbacks, it is possible to write an entire device driver in a Module since all the WDF callbacks are supported.** Doing so is recommended since it makes it possible to easily put the driver inside of another driver or be easily reused by another driver in the future. It also eliminates a lot of code that needs to be written if the code is in in a standalone driver.

### DMF\_[ModuleName]\_ModulePrepareHardware

static

NTSTATUS

DMF\_[ModuleName]\_ModulePrepareHardware(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFCMRESLIST ResourcesRaw,

\_In\_ WDFCMRESLIST ResourcesTranslated)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDevicePrepareHardware callback if the Module supports this callback. **Generally speaking, Modules do not use this callback.**

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **This callback is rarely used by Modules. Instead of using this callback, use DMF\_MODULE\_OPEN\_OPTION\_OPEN\_PrepareHardware and define DMF\_[ModuleName]\_ResourcesAssign**.
* If the Module supports this callback, then the Module must call its DMF\_[ModuleName]\_Open() callback.
* Even if this function returns an error, DMF\_[ModuleName]\_ModuleReleaseHardware will be called.

### DMF\_[ModuleName]\_ModuleReleaseHardware

static

NTSTATUS

DMF\_[ModuleName]\_ModuleReleaseHardware(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFCMRESLIST ResourcesTranslated)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceReleaseHardware callback if the Module supports this callback. **Generally speaking, this Modules do not use this callback.**

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **This callback is rarely used by Modules. Instead of using this callback, use DMF\_MODULE\_OPEN\_OPTION\_OPEN\_PrepareHardware**.
* If the Module supports this callback, then the Module must call its DMF\_[ModuleName]\_Close() callback.

### DMF\_[ModuleName]\_ModuleD0Entry

static

NTSTATUS

DMF\_[ModuleName]\_ModuleD0Entry(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDF\_POWER\_DEVICE\_STATE PreviousState)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceD0Entry callback if the Module supports this callback. The Module supports this callback when it needs to perform operations when the underlying device this Module supports is powered up.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* If this function returns an error, DMF\_[ModuleName]\_D0Exit will not be called.

### DMF\_[ModuleName]\_ModuleD0EntryPostInterruptsEnabled

static

NTSTATUS

DMF\_[ModuleName]\_ModuleD0EntryPostInterruptsEnabled(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDF\_POWER\_DEVICE\_STATE PreviousState)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceD0EntryPostInterruptsEnabled callback if the Module supports this callback. The Module supports this callback when it needs to perform operations after interrupts have been enabled.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleD0Exit

static

NTSTATUS

DMF\_[ModuleName]\_ModuleD0Exit(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDF\_POWER\_DEVICE\_STATE TargetState)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceD0Exit callback if the Module supports this callback. The Module supports this callback when it needs to perform operations when the underlying device this Module supports is powered down.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleD0ExitPreInterruptsDisabled

static

NTSTATUS

DMF\_[ModuleName]\_ModuleD0ExitPreInterruptsDisabled(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDF\_POWER\_DEVICE\_STATE TargetState)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceD0EntryPostInterruptsEnabled callback if the Module supports this callback. The Module supports this callback when it needs to perform operations before interrupts are disabled.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleDeviceIoControl

static

BOOLEAN

DMF\_[ModuleName]\_ModuleDeviceIoControl(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFQUEUE Queue,

\_In\_ WDFREQUEST Request,

\_In\_ size\_t OutputBufferLength,

\_In\_ size\_t InputBufferLength,

\_In\_ ULONG IoControlCode)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtQueueDeviceIoControl callback if the Module supports this callback.

**NOTE: Modules should not use this callback. Instead, use DMF\_IoctlHandler because that Module simplifies the non-trivial handling of the return value and automatically performs validations of the input/output buffer sizes.**

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

TRUE if this Module has handled the call. FALSE if not. When TRUE is returned, DMF will not dispatch this callback to any other instantiated Module. **Note: This return value is DMF specific and varies from the WDF implementation.**

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Only one instantiated Module may handle this callback per IOCTL. Once a Module indicates it has handled the call, DMF will not dispatch this callback to any other instantiated Module.**
* Please see DMF\_IoctlHandler.

### DMF\_[ModuleName]\_ModuleInternalDeviceIoControl

static

BOOLEAN

DMF\_[ModuleName]\_ModuleInternalDeviceIoControl(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFQUEUE Queue,

\_In\_ WDFREQUEST Request,

\_In\_ size\_t OutputBufferLength,

\_In\_ size\_t InputBufferLength,

\_In\_ ULONG IoControlCode)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtQueueInternalDeviceIoControl callback if the Module supports this callback.

**NOTE: Modules should not use this callback. Instead, use DMF\_IoctlHandler because that Module simplifies the non-trivial handling of the return value and automatically performs validations of the input/output buffer sizes.**

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

TRUE if this Module has handled the call. FALSE if not. When TRUE is returned, DMF will not dispatch this callback to any other instantiated Module. **Note: This return value is DMF specific and varies from the WDF implementation.**

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Only one instantiated Module may handle this callback per IOCTL. Once a Module indicates it has handled the call, DMF will not dispatch this callback to any other instantiated Module.**
* Please see DMF\_IoctlHandler.

### DMF\_[ModuleName]\_ModuleSelfManagedIoCleanup

static

VOID

DMF\_[ModuleName]\_ModuleSelfManagedIoCleanup(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceSelfManagedIoCleanup callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

None.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleSelfManagedIoFlush

static

VOID

DMF\_[ModuleName]\_ModuleSelfManagedIoFlush(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceSelfManagedIoFlush callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

None.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleSelfManagedIoInit

static

NTSTATUS

DMF\_[ModuleName]\_ModuleSelfManagedIoInit(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceSelfManagedIoInit callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleSelfManagedIoSuspend

static

NTSTATUS

DMF\_[ModuleName]\_ModuleSelfManagedIoSuspend(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceSelfManagedIoSuspend callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleSelfManagedIoRestart

static

NTSTATUS

DMF\_[ModuleName]\_ModuleSelfManagedIoRestart(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceSelfManagedIoRestart callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleSurpriseRemoval

static

VOID

DMF\_[ModuleName]\_ModuleSurpriseRemoval(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceSurpriseRemoval callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

None

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* Use this callback to prevent the Module from talking to underlying hardware after the Client Driver detects that its hardware has been surprise removed.
* Usually this callback sets a flag in the Module’s Private Context to so that other callbacks and Methods know the underlying device is no longer present.

### DMF\_[ModuleName]\_ModuleQueryRemove

static

NTSTATUS

DMF\_[ModuleName]\_ModuleQueryRemove(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceQueryRemove callback if the Module supports this callback.

**Generally speaking, Module’s should use DMF\_NotificationStream for resources/hardware that can be gracefully removed at runtime.**

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* Please see DMF\_NotificationStream.

### DMF\_[ModuleName]\_ModuleQueryStop

static

NTSTATUS

DMF\_[ModuleName]\_ModuleQueryStop(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceQueryStop callback if the Module supports this callback.

**Generally speaking, Module’s should use DMF\_Notification for resources/hardware that can be gracefully stopped at runtime.**

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* Please see DMF\_NotificationStream.

### DMF\_[ModuleName]\_ModuleRelationsQuery

static

VOID

DMF\_[ModuleName]\_ModuleRelationsQuery(

\_In\_ DMFMODULE DmfModule,

\_In\_ DEVICE\_RELATION\_TYPE RelationType)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceRelationsQuery callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

None.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleUsageNotificationEx

static

NTSTATUS

DMF\_[ModuleName]\_ModuleUsageNotificationEx(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDF\_SPECIAL\_FILE\_TYPE NotificationType,

\_In\_ BOOLEAN IsInNotificationPath)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceUsageNotificationEx callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleArmWakeFromS0

static

NTSTATUS

DMF\_[ModuleName]\_ModuleArmWakeFromS0(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceArmWakeFromS0 callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleDisarmWakeFromS0

static

VOID

DMF\_[ModuleName]\_ModuleDisarmWakeFromS0(

\_In\_ DMFMODULE DmfModule)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtDeviceDisarmWakeFromS0 callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleWakeFromS0Triggered

static

VOID

DMF\_[ModuleName]\_ModuleWakeFromS0Triggered(

\_In\_ DMFMODULE DmfModule)

EvtDeviceWakeFromS0Triggered callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

None.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleArmWakeFromSxWithReason

static

NTSTATUS

DMF\_[ModuleName]\_ModuleArmWakeFromSxWithReason(

\_In\_ DMFMODULE DmfModule,

\_In\_ BOOLEAN DeviceWakeEnabled,

\_In\_ BOOLEAN ChildrenArmedForWake)

EvtDeviceArmWakeFromSxWithReason callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned. See the official WDF documentation for more information.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleDisarmWakeFromSx

static

VOID

DMF\_[ModuleName]\_ModuleDisarmWakeFromSx(

\_In\_ DMFMODULE DmfModule)

EvtDeviceDisarmWakeFromSx callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

None.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleWakeFromSxTriggered

static

VOID

DMF\_[ModuleName]\_ModuleWakeFromSxTriggered(

\_In\_ DMFMODULE DmfModule)

EvtDeviceWakeFromSxTriggered callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

None.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

### DMF\_[ModuleName]\_ModuleFileCreate

static

BOOLEAN

DMF\_[ModuleName]\_ModuleFileCreate(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFDEVICE Device,

\_In\_ WDFREQUEST Request,

\_In\_ WDFFILEOBJECT FileObject)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtFileOpen callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

TRUE if this Module has handled the call. FALSE if not. When TRUE is returned, DMF will not dispatch this callback to any other instantiated Module. **Note: This return value is DMF specific and varies from the WDF implementation.**

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Only one instantiated Module may handle this callback. Once a Module indicates it has handled the call, DMF will not dispatch this callback to any other instantiated Module.**

### DMF\_[ModuleName]\_ModuleFileCleanup

static

BOOLEAN

DMF\_[ModuleName]\_ModuleFileCleanup(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFFILEOBJECT FileObject)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtFileCleanup callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

TRUE if this Module has handled the call. FALSE if not. When TRUE is returned, DMF will not dispatch this callback to any other instantiated Module. **Note: This return value is DMF specific and varies from the WDF implementation.**

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Only one instantiated Module may handle this callback. Once a Module indicates it has handled the call, DMF will not dispatch this callback to any other instantiated Module.**

### DMF\_[ModuleName]\_ModuleFileClose

static

BOOLEAN

DMF\_[ModuleName]\_ModuleFileClose(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFFILEOBJECT FileObject)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtFileClose callback if the Module supports this callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

TRUE if this Module has handled the call. FALSE if not. When TRUE is returned, DMF will not dispatch this callback to any other instantiated Module. **Note: This return value is DMF specific and varies from the WDF implementation.**

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Only one instantiated Module may handle this callback. Once a Module indicates it has handled the call, DMF will not dispatch this callback to any other instantiated Module.**

### DMF\_[ModuleName]\_ModuleQueueIoRead

static

BOOLEAN

DMF\_[ModuleName]\_ModuleQueueIoRead(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFQUEUE Queue,

\_In\_ WDFREQUEST Request,

\_In\_ size\_t Length)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtQueueIoRead callback if the Module supports this callback. The Module supports this callback when it needs to provide support for a Read operation.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

TRUE if this Module has handled the call. FALSE if not. When TRUE is returned, DMF will not dispatch this callback to any other instantiated Module. **Note: This return value is DMF specific and varies from the WDF implementation.**

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Only one instantiated Module may handle this callback. Once a Module indicates it has handled the call, DMF will not dispatch this callback to any other instantiated Module.**

### DMF\_[ModuleName]\_ModuleQueueIoWrite

static

BOOLEAN

DMF\_[ModuleName]\_ModuleQueueIoWrite(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFQUEUE Queue,

\_In\_ WDFREQUEST Request,

\_In\_ size\_t Length)

DMF calls this callback for every instantiated Module when the Client Driver receives the EvtQueueIoRead callback if the Module supports this callback. The Module supports this callback when it needs to provide support for a Write operation.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

*Consult the official WDF documentation to understand the other parameters. The meanings of the parameters are the same. To ensure accuracy and eliminate duplication, the information about those parameters is not listed in this document.*

#### Returns

TRUE if this Module has handled the call. FALSE if not. When TRUE is returned, DMF will not dispatch this callback to any other instantiated Module. **Note: This return value is DMF specific and varies from the WDF implementation.**

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Only one instantiated Module may handle this callback. Once a Module indicates it has handled the call, DMF will not dispatch this callback to any other instantiated Module.**

## Module DMF Callbacks

This section enumerates and explains the DMF specific callbacks that Modules can register for and receive.

This is a list of all DMF callbacks:

* DMF\_[ModuleName]\_ChildModulesAdd
* DMF\_[ModuleName]\_ResourcesAssign
* DMF\_[ModuleName]\_Open
* DMF\_[ModuleName]\_Close
* DMF\_[ModuleName]\_NotificationRegister
* DMF\_[ModuleName]\_NotificationUnregister
* DMF\_[ModuleName]\_Destroy

### DMF\_[ModuleName]\_ChildModulesAdd

static

VOID

DMF\_ChildModulesAdd(

\_In\_ DMFMODULE DmfModule,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* DmfParentModuleAttributes,

\_In\_ PDMFMODULE\_INIT DmfModuleInit)

Modules support this callback when they need to create Child Modules. This callback adds one ore more Child Modules to a given Parent Module.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The given Parent Module. |
| DMF\_MODULE\_ATTRIBUTES\* DmfParentModuleAttributes | The Parent Module’s attributes. In some cases it is necessary for the Child to know the Parent’s attributes. (See DMF\_BufferQueue.) |
| PDMFMODULE\_INIT DmfModuleInit | An opaque structure created by DMF that must be passed to DMF\_DmfModuleAdd(). |

#### Returns

None

#### Remarks

* After this function returns, DMF will instantiate all the Modules specified by the Client Module and maintain those Modules in a tree structure. These Child Modules will receive DMF/WDF callbacks as needed.
* This function is very similar to the callback used by the Client driver to instantiate Modules.

### DMF\_[ModuleName]\_ResourcesAssign

static

NTSTATUS

DMF\_[ModuleName]\_ResourcesAssign(

\_In\_ DMFMODULE DmfModule,

\_In\_ WDFCMRESLIST ResourcesRaw,

\_In\_ WDFCMRESLIST ResourcesTranslated)

Modules that do not support EvtDevicePrepareHardware can support this callback in order to get the Client Driver’s resources. The Module will enumerate the resources and choose the resources it needs.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |
| WDFCMRESLIST ResourcesRaw | Raw resource information passed to Client Driver’s EvtDevicePrepareHardware callback. |
| WDFCMRESLIST ResourcesTranslated | Translated resource information passed to Client Driver’s EvtDevicePrepareHardware callback. |

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* Resources acquired by this call should be released in the Module’s Close callback.
* This callback is usually supported by Modules that need resources but do not use the PrepareHardware callback because DMF opens the Module automatically during EvtDevicePrepareHardware or EvtDeviceD0Entry. *Note: Modules that support* EvtDevicePrepareHardware *must call* DMF\_[ModuleName]\_Open() *from inside that callback, so using this callback is easier than supporting* EvtDevicePrepareHardware*.*

### DMF\_[ModuleName]\_Open

static

NTSTATUS

DMF\_[ModuleName]\_Open(

\_In\_ DMFMODULE DmfModule)

If the Module supports this callback, DMF calls this callback after the Module is created. Depending on the Module’s Open Flags, this callback may be called immediately after the Module is created, during EvtDevicePrepareHardware or during EvtDeviceD0Entry. It is also possible that this callback is called on demand by the Client if DMF\_MODULE\_OPEN\_OPTION\_Generic is set.

The purpose of this callback is to allow the Module to perform actions one time before further callbacks happen and before the Module’s Methods are called. Frequently, this callback is used to allocate resources such as memory, timer and handles to other WDF primitives that the Module will use later.

In some cases, all the work the Module does is in the DMF\_[ModuleName]\_Open() callback.

This callback is used to initialize the Module’s Private Context (which is analogous to the Device Context of a Client Driver).

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* All resources acquired by this callback must be release in the Module’s Close callback.
* If the Module instantiates Child Modules, DMF automatically calls DMF\_[ModuleName]\_Open for each Child Module, **before** calling the Parent Module’s DMF\_[ModuleName]\_Open callback. This ensures that all Child Modules are ready for use by the Parent during DMF\_[ModuleName]\_Open.

#### Example

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

\_Must\_inspect\_result\_

static

NTSTATUS

DMF\_I2cTarget\_Open(

\_In\_ DMFMODULE DmfModule

)

/\*++

Routine Description:

Initialize an instance of a DMF Module of type I2cTarget.

Arguments:

DmfModule - This Module's handle.

Return Value:

NTSTATUS

--\*/

{

NTSTATUS ntStatus;

UNICODE\_STRING resourcePathString;

WCHAR resourcePathBuffer[RESOURCE\_HUB\_PATH\_SIZE];

WDFDEVICE device;

WDF\_OBJECT\_ATTRIBUTES objectAttributes;

WDF\_IO\_TARGET\_OPEN\_PARAMS openParams;

DMF\_CONTEXT\_I2cTarget\* moduleContext;

DMF\_CONFIG\_I2cTarget\* moduleConfig;

PAGED\_CODE();

moduleContext = DMF\_CONTEXT\_GET(DmfModule);

moduleConfig = DMF\_CONFIG\_GET(DmfModule);

if (! moduleContext->I2cConnectionAssigned)

{

// In some cases, the minimum number of resources is zero because the same driver

// is used on different platforms. In that case, this Module still loads and opens

// but it does nothing.

//

TraceEvents(TRACE\_LEVEL\_VERBOSE, DMF\_TRACE\_I2cTarget, "No I2C Resources Found");

ntStatus = STATUS\_SUCCESS;

goto Exit;

}

device = DMF\_AttachedDeviceGet(DmfModule);

RtlInitEmptyUnicodeString(&resourcePathString,

resourcePathBuffer,

sizeof(resourcePathBuffer));

ntStatus = RESOURCE\_HUB\_CREATE\_PATH\_FROM\_ID(&resourcePathString,

moduleContext->I2cConnection.u.Connection.IdLowPart,

moduleContext->I2cConnection.u.Connection.IdHighPart);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_I2cTarget, "RESOURCE\_HUB\_CREATE\_PATH\_FROM\_ID fails: ntStatus=%!STATUS!", ntStatus);

goto Exit;

}

WDF\_OBJECT\_ATTRIBUTES\_INIT(&objectAttributes);

objectAttributes.ParentObject = DmfModule;

ntStatus = WdfIoTargetCreate(device,

&objectAttributes,

&moduleContext->I2cTarget);

if (! NT\_SUCCESS(ntStatus))

{

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_I2cTarget, "RESOURCE\_HUB\_CREATE\_PATH\_FROM\_ID fails: ntStatus=%!STATUS!", ntStatus);

goto Exit;

}

WDF\_IO\_TARGET\_OPEN\_PARAMS\_INIT\_OPEN\_BY\_NAME(&openParams,

&resourcePathString,

FILE\_GENERIC\_READ | FILE\_GENERIC\_WRITE);

// Open the IoTarget for I/O operation.

//

ntStatus = WdfIoTargetOpen(moduleContext->I2cTarget,

&openParams);

if (! NT\_SUCCESS(ntStatus))

{

ASSERT(NT\_SUCCESS(ntStatus));

TraceEvents(TRACE\_LEVEL\_ERROR, DMF\_TRACE\_I2cTarget, "WdfIoTargetOpen fails: ntStatus=%!STATUS!", ntStatus);

goto Exit;

}

Exit:

return ntStatus;

}

#pragma code\_seg()

### DMF\_[ModuleName]\_Close

static

VOID

DMF\_[ModuleName]\_Close(

\_In\_ DMFMODULE DmfModule)

If the Module supports this callback, DMF calls this callback before the Module is destroyed. Depending on the Module’s Open Flags, this callback may be called immediately before the Module is destroyed, during EvtDeviceReleaseHardware or during EvtDeviceD0Exit. It is also possible that this callback is called on demand by the Client if DMF\_MODULE\_OPEN\_OPTION\_Generic is set.

The purpose of this callback is to allow the Module to release all resources allocated in the Module’s Open callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

None

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* Resources released in this callback are generally acquired in the Module’s Open callback.
* If the Module instantiates Child Modules, DMF automatically calls DMF\_[ModuleName]\_Close for each Child Module, after to calling the Parent Module’s DMF\_[ModuleName]\_Close callback. This ensures that all Child Modules are still ready for use by the Parent during DMF\_[ModuleName]\_Close.

#### Example

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

static

VOID

DMF\_I2cTarget\_Close(

\_In\_ DMFMODULE DmfModule

)

/\*++

Routine Description:

Uninitialize an instance of a DMF Module of type I2cTarget.

Arguments:

DmfModule - This Module's handle.

Return Value:

None

--\*/

{

DMF\_CONTEXT\_I2cTarget\* moduleContext;

PAGED\_CODE();

moduleContext = DMF\_CONTEXT\_GET(DmfModule);

if (moduleContext->I2cTarget != NULL)

{

WdfIoTargetClose(moduleContext->I2cTarget);

WdfObjectDelete(moduleContext->I2cTarget);

moduleContext->I2cTarget = NULL;

}

}

#pragma code\_seg()

### DMF\_[ModuleName]\_NotificationRegister

static

NTSTATUS

DMF\_[ModuleName]\_NotificationRegister(

\_In\_ DMFMODULE DmfModule)

If the Module supports this callback, DMF calls this callback after the Module is created. Depending on the Module’s Open Flags, this callback may be called immediately after the Module is created, during EvtDevicePrepareHardware or during EvtDeviceD0Entry. It is also possible that this callback is called on demand by the Client if DMF\_MODULE\_OPEN\_OPTION\_Generic is set.

The purpose of this callback is to allow the Module to register for an asynchronous notification that a resource needed by the Module’s Open callback has appeared in the system. When the registered callback happens, the Module must call DMF\_ModuleNotificationOpen() to tell DMF to call the Module’s Open callback.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned.

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).
* **Do not call** DMF\_[ModuleName]\_Open() **directly from the notification routine when the resource appears**. **Use** DMF\_ModuleNotificationOpen() **instead**. DMF\_ModuleNotificationOpen() allows DMF to automatically synchronize the arrival/removal of the resource with Methods and will call DMF\_[ModuleName]\_Open() itself.
* **Do not call** DMF\_[ModuleName]\_Close() **directly from the notification routine when the resource disappears. Use DMF\_ModuleNotificationClose() instead.** DMF\_ModuleNotificationClose() allows DMF to automatically synchronize the arrival/removal of the resource with Methods and will call DMF\_[ModuleName]\_Close() itself.

#### Example

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

\_Must\_inspect\_result\_

static

NTSTATUS

DMF\_DeviceInterfaceTarget\_NotificationRegister(

\_In\_ DMFMODULE DmfModule

)

/\*++

Routine Description:

This callback is called when the Module Open Flags indicate that the this Module

is opened after an asynchronous notification has happened.

(DMF\_MODULE\_OPEN\_OPTION\_NOTIFY\_PrepareHardware or DMF\_MODULE\_OPEN\_OPTION\_NOTIFY\_D0Entry)

This callback registers the notification.

Arguments:

DmfModule - The given DMF Module.

Return Value:

STATUS\_SUCCESS

--\*/

{

NTSTATUS ntStatus;

WDFDEVICE parentDevice;

PDEVICE\_OBJECT deviceObject;

PDRIVER\_OBJECT driverObject;

DMF\_CONTEXT\_DeviceInterfaceTarget\* moduleContext;

DMF\_CONFIG\_DeviceInterfaceTarget\* moduleConfig;

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_DeviceInterfaceTarget);

moduleContext = DMF\_CONTEXT\_GET(DmfModule);

moduleConfig = DMF\_CONFIG\_GET(DmfModule);

// This function should not be not called twice.

//

ASSERT(NULL == moduleContext->DeviceInterfaceNotification);

parentDevice = DMF\_AttachedDeviceGet(DmfModule);

ASSERT(parentDevice != NULL);

deviceObject = WdfDeviceWdmGetDeviceObject(parentDevice);

ASSERT(deviceObject != NULL);

driverObject = deviceObject->DriverObject;

ntStatus = IoRegisterPlugPlayNotification(EventCategoryDeviceInterfaceChange,

PNPNOTIFY\_DEVICE\_INTERFACE\_INCLUDE\_EXISTING\_INTERFACES,

(void\*)&moduleConfig->DeviceInterfaceTargetGuid,

driverObject,

(PDRIVER\_NOTIFICATION\_CALLBACK\_ROUTINE)DeviceInterfaceTarget\_InterfaceArrivalCallback,

(VOID\*)DmfModule,

&(moduleContext->DeviceInterfaceNotification));

FuncExit(DMF\_TRACE\_DeviceInterfaceTarget, "ntStatus=%!STATUS!", ntStatus);

return ntStatus;

}

#pragma code\_seg()

### DMF\_[ModuleName]\_NotificationUnregister

static

VOID

DMF\_[ModuleName]\_NotificationUnregister(

\_In\_ DMFMODULE DmfModule)

If the Module supports this callback, DMF calls this callback before the Module is destroyed. Depending on the Module’s Open Flags, this callback may be called immediately before the Module is destroyed, during EvtDeviceReleaseHardware or during EvtDeviceD0Exit. It is also possible that this callback is called on demand by the Client if DMF\_MODULE\_OPEN\_OPTION\_Generic is set.

The purpose of this callback is to allow the Module to unregister for any notifications that were registered in DMF\_[ModuleName]\_NotificationOpen.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

None

#### Remarks

* This callback is always called directly by DMF. This function is never called directly (nor is it accessible to Clients or other Modules).

#### Example

#pragma code\_seg("PAGE")

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

static

VOID

DMF\_DeviceInterfaceTarget\_NotificationUnregister(

\_In\_ DMFMODULE DmfModule

)

/\*++

Routine Description:

This callback is called when the Module Open Flags indicate that the this Module

is opened after an asynchronous notification has happened.

(DMF\_MODULE\_OPEN\_OPTION\_NOTIFY\_PrepareHardware or DMF\_MODULE\_OPEN\_OPTION\_NOTIFY\_D0Entry)

This callback unregisters the notification that was previously registered.

Arguments:

DmfModule - The given DMF Module.

Return Value:

None

--\*/

{

NTSTATUS ntStatus;

DMF\_CONTEXT\_DeviceInterfaceTarget\* moduleContext;

PAGED\_CODE();

FuncEntry(DMF\_TRACE\_DeviceInterfaceTarget);

ntStatus = STATUS\_SUCCESS;

moduleContext = DMF\_CONTEXT\_GET(DmfModule);

// The notification routine could be called after the IoUnregisterPlugPlayNotification method

// has returned which was undesirable. IoUnregisterPlugPlayNotificationEx prevents the

// notification routine from being called after IoUnregisterPlugPlayNotificationEx returns.

//

if (moduleContext->DeviceInterfaceNotification != NULL)

{

ntStatus = IoUnregisterPlugPlayNotificationEx(moduleContext->DeviceInterfaceNotification);

if (! NT\_SUCCESS(ntStatus))

{

ASSERT(FALSE);

TraceEvents(TRACE\_LEVEL\_VERBOSE,

DMF\_TRACE\_DeviceInterfaceTarget,

"IoUnregisterPlugPlayNotificationEx fails: ntStatus=%!STATUS!",

ntStatus);

goto Exit;

}

moduleContext->DeviceInterfaceNotification = NULL;

DeviceInterfaceTarget\_ModuleCloseAndTargetDestroyAsNeeded(DmfModule);

}

else

{

// Allow caller to unregister notification even if it has not been registered.

//

}

Exit:

FuncExit(DMF\_TRACE\_DeviceInterfaceTarget, "ntStatus=%!STATUS!", ntStatus);

}

#pragma code\_seg()

### DMF\_[ModuleName]\_Destroy

static

VOID

DMF\_[ModuleName]\_Destroy(

\_In\_ DMFMODULE DmfModule)

If the Module supports this callback, DMF calls this callback when Module is destroyed. It is almost never necessary for a Module to support this callback. DMF’s Generic version of this callback destroys the Module.

This callback only needs to be defined in the rare case where resources or actions are taken during the Module’s Create function that must be accounted for prior to Module destruction.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

None

#### Remarks

* Module authors should avoid supporting this callback because Module Create functions should not allocate resources that DMF Framework does not know about.
* **If a Module supports this callback, the callback must call** DMF\_ModuleDestroy().
* After this callback returns, the Module nor the Module’s Methods may be used because the Module’s data structures will have been destroyed. Generally, this is not an issue because of the lifetime of Modules is managed by DMF.

## Module API

This section discusses the DMF API available to Module authors. These APIs are not used by DMF Client Drivers.

### The Module Create Function

Every Module has a publicly accessible Create function. This function is always named using this convention:

NTSTATUS

DMF\_[ModuleName]\_Create(

\_In\_ WDFDEVICE Device,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* DmfModuleAttributes,

\_In\_ WDF\_OBJECT\_ATTRIBUTES\* ObjectAttributes,

\_Out\_ DMFMODULE\* DmfModule)

Use DECLARE\_DMF\_MODULE() to declare this function in the Module’s .h file.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The Client Driver’s WDFDEVICE. |
| DMF\_MODULE\_ATTRIBUTES\* DmfModuleAttributes | Attributes that tell DMF how to create the function. |
| WDF\_OBJECT\_ATTRIBUTES\* ObjectAttributes | Attributes that tell DMF about the parent of the Module that is to be created. |
| DMFMODULE\* DmfModule | After the Module has been created (by this function), the resultant Module handle is returned here. |

#### Returns

STATUS\_SUCCESS if the Module (and its optional children) are created successfully. Otherwise, the error code that tells the reason why the function failed.

#### Remarks

* This function is usually called by the DMF Framework after the DmfModulesAdd callback returns.
* This function is also called by Modules to create Child Module’s. (A Module Create function can call any other Module’s Create Function, but not its own as that will cause infinite recursion.)
* This function is also called by Clients to create a Dynamic Module. See the section *Dynamic Modules* for more information.
* Every Module must implement this function.
* This function should only initialize and create the Module and its Child Modules. It should not allocate resources or talk to hardware.

### DECLARE\_DMF\_MODULE

DECLARE\_DMF\_MODULE(ModuleName)

This macro declares the Module’s publicly available functions and macros. Always use this macro in the Module’s .h file. This macro is used for Module that have a Config.

#### Parameters

|  |  |
| --- | --- |
| ModuleName | The name of the Module. |

#### Returns

None

#### Remarks

* If a Module has no Config, use DECLARE\_DMF\_MODULE\_NO\_CONFIG instead.

#### Example

// This macro declares the following functions:

// DMF\_OsrFx2\_ATTRIBUTES\_INIT()

// DMF\_CONFIG\_OsrFx2\_AND\_ATTRIBUTES\_INIT()

// DMF\_OsrFx2\_Create()

//

DECLARE\_DMF\_MODULE(OsrFx2)

### DECLARE\_DMF\_MODULE\_NO\_CONFIG

DECLARE\_DMF\_MODULE\_NO\_CONFIG(ModuleName)

This macro declares the Module’s publicly available functions and macros. Always use this macro in the Module’s .h file. This macro is used for Module that do not have a Config.

#### Parameters

|  |  |
| --- | --- |
| ModuleName | The name of the Module. |

#### Returns

None

#### Remarks

* If a Module has a Config, use DECLARE\_DMF\_MODULE() instead.

### DMF\_ENTRYPOINTS\_DMF\_INIT

VOID

DMF\_ENTRYPOINTS\_DMF\_INIT(

\_Out\_ PDMF\_ENTRYPOINTS\_DMF EntryPointsDmf)

This function initializes a DMF\_ENTRYPOINTS\_DMF structure.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_ENTRYPOINTS\_DMF EntryPointsDmf | The structure buffer to initialize. |

#### Returns

None

#### Remarks

* Call this function before setting the structure’s Module specific settings.
* After initializing this structure, set the DMF callbacks the Module supports.

### DMF\_ENTRYPOINTS\_WDF\_INIT

VOID

DMF\_ENTRYPOINTS\_WDF\_INIT(

\_Out\_ PDMF\_ENTRYPOINTS\_WDF EntryPointsWdf)

This function initializes a DMF\_ENTRYPOINTS\_WDF structure.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_ENTRYPOINTS\_WDF EntryPointsWdf | The structure buffer to initialize. |

#### Returns

None

#### Remarks

* Call this function before setting the structure’s Module specific settings.
* After initializing this structure, set the WDF callbacks the Module supports.

### DMF\_MODULE\_DESCRIPTOR\_INIT

VOID

DMF\_MODULE\_DESCRIPTOR\_INIT(

\_Inout\_ PDMF\_MODULE\_DESCRIPTOR ModuleDescriptor,

\_In\_ PSTR ModuleName,

\_In\_ ULONG ModuleOptions,

\_In\_ DmfModuleOpenOption OpenOption)

This function initializes a DMF\_MODULE\_DESCRIPTOR structure.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_MODULE\_DESCRIPTOR ModuleDescriptor | The structure buffer to initialize. |
| PSTR ModuleName | The name of the Module. It should match the Module’s file name. This name is useful when debugging so that it is easy to know what Module the Module’s handle refers to. |
| ULONG ModuleOptions | Flags that indicate attributes about the Module. Currently only these flags are supported:  DMF\_MODULE\_OPTIONS\_PASSIVE: Indicates that the Module uses wait locks because the Module is only used at PASSIVE\_LEVEL.  DMF\_MODULE\_OPTIONS\_DISPATCH: Indicates that the Module uses spin locks because the Module is used at DISPATCH\_LEVEL.  DMF\_MODULE\_OPTIONS\_DISPATCH\_MAXIMUM: Indicates that the Module uses spin locks because the Module is used at DISPATCH\_LEVEL by default. However, the Client may instantiate the Module using PASSIVE\_LEVEL locks. (If cases where the Module allocates from the memory pool, the locks need to be PASSIVE\_LEVEL locks if the Client chooses to allocate Paged Pool.)  DMF\_MODULE\_OPTIONS\_TRANSPORT\_REQUIRED: Indicates that the Module requires that the Client instantiate a Transport Module. |
| DmfModuleOpenOption OpenOption | See DmfModuleOpenOption. |

#### Returns

None

#### Remarks

* Call this function before setting the structure’s Module specific settings.
* After using this function to initialize the structure, set the Module’s DMF and WDF callbacks in this structure as needed.
* After using this function to initialize the structure, set the size of the Module’s Config structure if it is defined.

### DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT

DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT(Local, Formal)

This function initializes the Module’ Object Attributes. Do not use this function if the Module has a Private Context.

#### Parameters

|  |  |
| --- | --- |
| Local | A locally allocated WDF\_OBJECT\_ATTRIBUTES structure. |
| Formal | WDF\_OBJECT\_ATTRIBUTES\*. This value Is passed into the Module’s Create function. |

#### Returns

None

#### Remarks

* If the Module has a Private Context, use DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE instead.

### DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE

DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(Local, Formal, Context)

This function initializes the Module’ Object Attributes and sets the type of the Module’s Private Context.

#### Parameters

|  |  |
| --- | --- |
| Local | A locally allocated WDF\_OBJECT\_ATTRIBUTES structure. |
| Formal | WDF\_OBJECT\_ATTRIBUTES\*. This value Is passed into the Module’s Create function. |
| Context | The name of the Module’s Private Context. |

#### Returns

None

#### Remarks

* If the Module has no Private Context, use DMF\_WDF\_OBJECT\_ATTRIBUTES\_INIT instead.

### DMF\_CONFIG\_GET

DMF\_CONFIG\_[Modulename]\*

DMF\_CONFIG\_GET(

\_In\_ DMFMODULE DmfModule)

Given an instance of a Module, this function returns the given Module’s Config.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The given Module’s DMF Module handle. |

#### Returns

The given Module’s Config.

Remarks

* There is no need to check the return value. If the Module is properly instantiated and has a Context defined, this return value is always correct.
* Do not use this function if the Module has no Config.
* This function is private to each Module. This function is defined automatically by DMF using DMF\_MODULE\_DECLARE\_CONFIG().
* Do not return this pointer to a Client.
* Although the Client originally set the Config structure, the buffer returned by this function is a copy of that structure. The Module may write to this buffer, but the Client will not see that change. Generally speaking, this buffer should not be written to. Use the Module’s Context instead.

### DMF\_CONTEXT\_GET

DMF\_CONTEXT\_[Modulename]\*

DMF\_CONTEXT\_GET(

\_In\_ DMFMODULE DmfModule)

Given an instance of a Module, this function returns the given Module’s Private Context.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The given Module’s DMF Module handle. |

#### Returns

The given Module’s Context.

Remarks

* There is no need to check the return value. If the Module is properly instantiated and has a Context defined, this return value is always correct.
* Do not use this function the if Module has no Context.
* This function is private to each Module. This function is defined automatically by DMF using DMF\_MODULE\_DECLARE\_CONTEXT().
* Do not return this pointer to a Client.

### DMF\_ModuleCreate

NTSTATUS

DMF\_ModuleCreate(

\_In\_ WDFDEVICE Device,

\_In\_ DMF\_MODULE\_ATTRIBUTES\* DmfModuleAttributes,

\_In\_ PWDF\_OBJECT\_ATTRIBUTES DmfModuleObjectAttributes,

\_In\_ PDMF\_MODULE\_DESCRIPTOR ModuleDescriptor,

\_Out\_ DMFMODULE\* DmfModule)

This function crates and instance of a Module.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The Client Driver’s WDFDEVICE. This value is passed into the Module’s Create function. |
| DMF\_MODULE\_ATTRIBUTES\* DmfModuleAttributes | Contains information DMF uses to create the Module. This value Is passed into the Module’s Create function. |
| PWDF\_OBJECT\_ATTRIBUTES DmfModuleObjectAttributes | Holds the Module’s parent information. |
| PDMF\_MODULE\_DESCRIPTOR ModuleDescriptor | The descriptor which describes all the Module specific information such as the callbacks it supports and the size of its Config structure. |
| DMFMODULE\* DmfModule | This value holds the handle of the created Module when this function returns success. It must be passed back to the caller of the Module’s Create function. |

#### Returns

STATUS\_SUCCESS if the Module is successfully created. If any aspect of Module creation fails, an error code is returned. This return value must be checked.

#### Remarks

* See section titled *The Module’s Create Function* for detailed information about how to use this function.

### DMF\_ModuleOpen

NTSTATUS

DMF\_ModuleOpen(

\_In\_ DMFMODULE DmfModule)

Given an instance of a Module, this function “opens” the Module. It means that DMF will call the Module’s DMF Open callback and the Module will be ready for use. After using this call, the Client may call the Module’s Methods.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned.

Remarks

* Module’s only use this call from their DMF\_Module\_NotificationRegister() callbacks; otherwise, DMF opens the Module automatically based on the Module’s Open Option.
* This callback is where the Module prepares its Module Context and acquires any resources needed for further use by the Client.
* This callback is where the Module can start doing any processing it needs to perform it function without any interaction from its parent. (For example, if a Module has no Methods because it does all its work on its own, this callback is where such work can start.)
* Clients will not call the Module’s Methods before this callback happens or while this callback is happening.

### DMF\_ModuleClose

VOID

DMF\_ModuleClose(

\_In\_ DMFMODULE DmfModule)

Given an instance of a Module, this function “closes” the Module. It means that DMF will call the Module’s DMF Close callback. The Client may not call the Module’s Methods after this call.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. Also, the Client Driver’s WDFDEVICE is accessible via this parameter. |

#### Returns

STATUS\_SUCCESS if no error is encountered in the callback. Otherwise, an error code corresponding to the error is returned.

Remarks

* Module’s only use this call from their DMF\_Module\_NotificationUnregister() callbacks; otherwise, DMF closes the Module automatically based on the Module’s Open Option.
* This callback is where the Module releases any resources acquired when it opened.
* This callback is where the Module can stop doing any processing it started during its Open callback.
* Clients will not call the Module’s Methods after this callback happens or while this callback is happening.

### DMF\_ModuleAcquire

NTSTATUS

DMF\_ModuleAcquire(

\_In\_ DMFMODULE DmfModule)

Module methods that use DMF\_[ModuleName]\_NotificationOpen() or the DMF\_Notification Module call this function at the beginning of the Method’s code prior to accessing the Module’s Private Context or calling any Module support functions. This function tells DMF to prevent the underlying resource’s handle from being closed while the Method is executing. If the underlying resource’s handle has already been closed, then this function returns an error and the Method should immediately exit.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

None

#### Remarks

* If this call succeeds, the Method must always call DMF\_ModuleRelease() before exiting to release the reference count acquired by this function.
* This function is only applicable if DMF\_ModuleNotificationOpen() is used by the Module from the Module’s notification callback.

### DMF\_ModuleRelease

NTSTATUS

DMF\_ModuleRelease(

\_In\_ DMFMODULE DmfModule)

Module methods that use DMF\_[ModuleName]\_NotificationOpen() or the DMF\_Notification Module call this function at the end of the Method’s code if the prior call to DMF\_ModuleAcquire() succeeded.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

None

#### Remarks

* This function is only applicable if DMF\_ModuleNotificationClose() is used by the Module from the Module’s notification callback.

### DMF\_ModuleDestroy

VOID

DMF\_ModuleDestroy(

\_In\_ DMFMODULE DmfModule)

This function destroys a Module. It is the opposite of DMF\_ModuleCreate. Child Module’s are automatically recursively destroyed.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

None

#### Remarks

* Generally speaking, Modules do not need to call this function because DMF’s default Destroy handler calls this function.
* This function, if it is used, should only be called from the Module’s Destroy callback.
* After this function is called, the Module’s Private Context and Config are destroyed and inaccessible.

### DMF\_[ModuleName]\_TransportMethod

NTSTATUS

DMF\_[ModuleName]\_TransportMethod(

\_In\_ DMFMODULE DmfModule,

\_In\_ ULONG Message,

\_In\_reads\_(InputBufferSize) PVOID InputBuffer,

\_In\_ size\_t InputBufferSize,

\_Out\_writes\_(OutputBufferSize) PVOID OutputBuffer,

\_In\_ size\_t OutputBufferSize)

This Method is implemented by Transport Modules. Modules that require a Transport Child Module call this Method to perform Transport specific functions.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. |
| Message | Indicates the reason for the call. Also, indicates the format of the Input and Output buffers. |
| InputBuffer | Specifies data sent to the Method. Message indicates the contents of this buffer. |
| InputBufferSize | Indicates the size of InputBuffer in bytes. |
| OutputBuffer | Specifies data returned by the Method. Message indicates the contents of this buffer. |
| OutputBufferSize | Indicates the size of OutputBuffer in bytes. |

#### Returns

STATUS\_SUCCESS indicates the Method succeeded. Otherwise, an error code is returned.

#### Remarks

* The formal parameters passed must conform to a specific interface that is defined by the Parent Module that requires the Transport.
* It is up to the implementor of this Method to make sure that this Method accepts and returns data understood by the Parent Module.
* This Method is similar to a Device IO Control handler in that the caller and callee must use a predefined interface.

# Feature Module Access API

DMF automatically instantiates Modules that provide commonly used functionality in device drivers. In this way, Clients can easily use that functionality without even instantiating the Modules that expose this support directly. Clients simply need to access the previously instantiated instance of that Module and use that Module’s Methods.

This table lists the Features available:

|  |  |
| --- | --- |
| **Feature Identifier** | **Purpose** |
| DmfFeature\_BranchTrack | Gives access to the BranchTrack Module. |
| DmfFeature\_LiveKernelDump | *Gives access to the LiveKernelDump Module.* |
| DmfFeature\_Performance | *Gives access to the Performance Module. (Coming soon.)* |

Note: Each Feature Module is documented in a separate document.

Clients access Feature Modules using these APIs:

### DMF\_FeatureModuleGetFromDevice

DMFMODULE

DMF\_FeatureModuleGetFromModule(

\_In\_ DMFMODULE DmfModule,

\_In\_ DmfFeatureType DmfFeature);

Given a WDF Device, retrieve a Feature Module handle given its identifier.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The given WDFDEVICE. |
| DmfFeatureType DmfFeature | Indicates which Feature Module to retrieve. |

#### Returns

The DMFMODULE handle for the given Feature identifier.

Remarks

* This function is used only by Client Drivers. Module’s use a similar but different function.

### DMF\_FeatureModuleGetFromModule

DMFMODULE

DMF\_FeatureModuleGetFromModule(

\_In\_ DMFMODULE DmfModule,

\_In\_ DmfFeatureType DmfFeature);

Given a DMF Module, retrieve a Feature Module handle given its identifier.

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The given DMFMODULE handle. |
| DmfFeatureType DmfFeature | Indicates which Feature Module to retrieve. |

#### Returns

The DMFMODULE handle for the given Feature identifier.

Remarks

* This function is used only by Modules. Client Drivers use a similar but different function.

# General DMF API

In addition to the APIs listed above, DMF provides other APIs that are useful in various situations.

## Module Parent Device and Client Context

These two functions allow the Client to access the Client Driver’s WDFEVICE and Client Context.

### DMF\_ParentDeviceGet

WDFDEVICE

DMF\_ParentDeviceGet(

\_In\_ DMFMODULE DmfModule)

Given a Module, this function retrieves the Client Driver’s WDFDEVICE (corresponding to its FDO).

#### Parameters

|  |  |
| --- | --- |
| DMFMODULE DmfModule | The Module’s DMF Module handle. Use this handle to retrieve the Module’s Private Context and Config. |

#### Returns

Client Driver’s WDFDEVICE that was set in DMF\_ModulesCreate().

#### Remarks

* WDFDEVICE is often needed when calling WDF APIs.

## Portable API

These functions allow code to be written so that it compiles and functions in both Kernel and User Mode. DMF is built upon WDF, however, there are some primitives that differ between Kernel and User Mode. These functions abstract those differences so that Modules and Client Drivers that use them do not have to write two versions using different primitives.

### Portable API Structures and Enumerations

### DMF\_Portable\_EventCreate

VOID

DMF\_Portable\_EventCreate(

\_Inout\_ PDMF\_PORTABLE\_EVENT EventPointer,

\_In\_ EVENT\_TYPE EventType,

\_In\_ BOOLEAN State

);

Creates an event that is unified across Kernel and User-modes.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_PORTABLE\_EVENT EventPointer | The address where the event is created. Use this address with the APIs that use this event. |
| EVENT\_TYPE EventType | NotificationEvent or  SynchronizationEvent. See Win32 API or NTAPI for the exact meaning and use cases. |
| BOOLEAN State | Initial state of the event when this function returns. |

#### Returns

None.

#### Remarks

* Events in User Mode are of type HANDLE. Events in Kernel-mode are KEVENT. The APIs for working with both are different. Use this function to create and work with an event in a unified manner.

### DMF\_Portable\_EventSet

VOID

DMF\_Portable\_EventSet(

\_In\_ PDMF\_PORTABLE\_EVENT EventPointer

);

Set a given portable event. Any threads that are waiting for that event to be set will continue executing.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_PORTABLE\_EVENT EventPointer | The address of the given event to set. |

#### Returns

None.

#### Remarks

* Any threads that are waiting for that event to be set will continue executing.

### DMF\_Portable\_EventReset

VOID

DMF\_Portable\_EventReset(

\_In\_ PDMF\_PORTABLE\_EVENT EventPointer

);

Reset a given portable event.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_PORTABLE\_EVENT EventPointer | The address of the given event to reset (set it not a non-set state). |

#### Returns

None.

#### Remarks

* Use this API with Notification events. Synchronization events, auto reset after the first thread that is waiting starts.
* Use this API with caution to avoid race conditions.

### DMF\_Portable\_EventWaitForSingle

DWORD

DMF\_Portable\_EventWaitForSingle(

\_In\_ PDMF\_PORTABLE\_EVENT EventPointer,

\_In\_ BOOLEAN Alertable,

\_In\_opt\_ PLARGE\_INTEGER TimeoutPointer

);

Cause the current thread to wait for an event to be set.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_PORTABLE\_EVENT EventPointer | The address of the given event to wait for. |
| BOOLEAN Alertable |  |
| PLARGE\_INTEGER TimeoutPointer | Use NULL to indicate infinite wait. Otherwise, it is the address of a LARGE\_INTEGER holding the number of 100-ns to wait. |

#### Returns

STATUS\_WAIT\_OBJECT\_0 to indicate the event was set. STATUS\_TIMEOUT to indicate the event was not set during the timeout period.

#### Remarks

* There is not no option to wait for multiple events at this time.

### DMF\_Portable\_EventClose

VOID

DMF\_Portable\_EventClose(

\_In\_ PDMF\_PORTABLE\_EVENT EventPointer

);

Cause a portable event to be unusable (closed).

#### Parameters

|  |  |
| --- | --- |
| PDMF\_PORTABLE\_EVENT EventPointer | The address of the given event to close. |

#### Returns

None.

#### Remarks

* In Kernel-mode, this is a NOP. For portability purposes, this use this call before the driver unloads.

*.*

### DMF\_Portable\_LookasideListCreate

NTSTATUS

DMF\_Portable\_LookasideListCreate(

\_In\_ PWDF\_OBJECT\_ATTRIBUTES LookasideAttributes,

\_In\_ size\_t BufferSize,

\_In\_ POOL\_TYPE PoolType,

\_In\_ PWDF\_OBJECT\_ATTRIBUTES MemoryAttributes,

\_In\_ ULONG PoolTag,

\_Out\_ PDMF\_PORTABLE\_LOOKASIDELIST LookasidePointer

);

Creates a lookaside list that is usable in both Kernel and User-mode.

#### Parameters

|  |  |
| --- | --- |
| PWDF\_OBJECT\_ATTRIBUTES LookasideAttributes |  |
| size\_t BufferSize | Size of each buffer that is allocated from the lookaside list. |
| POOL\_TYPE PoolType | PagedPool or NonPagedPoolNx. In User-mode all memory is pageable. |
| PWDF\_OBJECT\_ATTRIBUTES MemoryAttributes |  |
| ULONG PoolTag | Pool tag so for tracking memory leaks. |
| PDMF\_PORTABLE\_LOOKASIDELIST LookasidePointer | The address where the created lookaside list handle is created. Use this handle with APIs that use the lookaside list. |

#### Returns

STATUS\_SUCCESS if the lookaside list is created and ready for use. Otherwise an error code is returned.

#### Remarks

* There is no support for lookaside list in UMDF. A Module that uses a lookaside list should use this portable API so that the Module works both in Kernel and User-modes.

### DMF\_Portable\_LookasideListCreateMemory

NTSTATUS

DMF\_Portable\_LookasideListCreateMemory(

\_In\_ PDMF\_PORTABLE\_LOOKASIDELIST LookasidePointer,

\_Out\_ WDFMEMORY\* Memory

);

Allocated a buffer from a given lookaside list.

#### Parameters

|  |  |
| --- | --- |
| PDMF\_PORTABLE\_LOOKASIDELIST LookasidePointer | The address of the given lookaside list handle from which to allocate a buffer. |
| WDFMEMORY\* Memory | Address of the WDF Memory Handle associated with the allocated memory. |

#### Returns

STATUS\_SUCCESS if the allocation succeeded. Otherwise, an error code is returned.

#### Remarks

* Be sure to use WdfObjectDelete() to free the memory allocated by this function.

## Utility API

DMF provides several utility functions that perform common tasks in device drivers.

In this family of DMF APIs there are several functions that allow a Client Driver and/or Modules to write to the Event Log. These functions help in two important ways:

1. They work in both Kernel and User-mode so that the code to write to the Event Log only needs to be written one time in Modules that support both User and Kernel Modes.
2. They make it easier to write to the Event Log because the entire operation happens in a single line (in addition to any type declarations).

### DMF\_Utility\_AclPropagateInDeviceStack

NTSTATUS

DMF\_Utility\_AclPropagateInDeviceStack(

\_In\_ WDFDEVICE Device

);

Given a WDFDEVICE, set propagate its ACL down its device stack.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The given WDFDEVICE. |

#### Returns

STATUS\_SUCCESS if the allocation succeeded. Otherwise, an error code is returned.

#### Remarks

* This function is not available if DMF\_USER\_MODE is defined.

### DMF\_Utility\_ActivityIdFromDevice

GUID

DMF\_Utility\_ActivityIdFromDevice(

\_In\_ WDFDEVICE Device);

Given a WDFDEVICE, retrieve an associated Activity Id.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The given WDFDEVICE. |

#### Returns

The associated Activity Id of the given WDFDEVICE.

#### Remarks

* This function returns the given WDFDEVICE handle.
* See the DMF OSR FX-2 sample driver.

### DMF\_Utility\_ActivityIdFromRequest

GUID

DMF\_Utility\_RequestToActivityId(

\_In\_ WDFREQUEST Request);

Given a WDFREQUEST, retrieve an associated Activity Id.

#### Parameters

|  |  |
| --- | --- |
| WDFREQUEST Request | The given WDFREQUEST. |

#### Returns

The associated Activity Id of the given WDFREQUEST.

#### Remarks

* This function uses the WDM function IoGetActivityIdIrp if possible. If not, it returns the given WDFREQUEST handle.
* See the DMF OSR FX-2 sample driver.

### DMF\_Utility\_DelayMilliseconds

VOID

DMF\_Utility\_DelayMilliseconds(

\_In\_ ULONG Milliseconds

);

Cause the current running thread to sleep for a given number of milliseconds.

#### Parameters

|  |  |
| --- | --- |
| ULONG Milliseconds | The given number of milliseconds that the current thread should sleep for. |

#### Returns

None.

#### Remarks

* This function is better than the version provided in the DDK because it is portable between Kernel and User-mode.

### DMF\_Utility\_EventLogEntryWriteDriverObject

VOID

DMF\_Utility\_EventLogEntryWriteDriverObject(

\_In\_ PDRIVER\_OBJECT DriverObject,

\_In\_ NTSTATUS ErrorCode,

\_In\_ NTSTATUS FinalNtStatus,

\_In\_ ULONG UniqueId,

\_In\_ ULONG TextLength,

\_In\_opt\_ PCWSTR Text,

\_In\_ INT NumberOfFormatStrings,

\_In\_opt\_ PWCHAR\* FormatStrings,

\_In\_ INT NumberOfInsertionStrings,

...

);

Given a WDM DRIVER\_OBJECT, write an event to the Event Log using a variable number of parameters.

#### Parameters

|  |  |
| --- | --- |
| PDRIVER\_OBJECT DriverObject | The address of the given lookaside list handle from which to allocate a buffer. |
| NTSTATUS ErrorCode | ErrorCode from header generated by mc compiler. |
| NTSTATUS FinalNtStatus | The final NTSTATUS for the error being logged. |
| ULONG UniqueId | Unique long word that identifies the specific call to this function. |
| ULONG TextLength | The length in bytes (including the terminating NULL) of the Text string. |
| PCWSTR Text | Additional data to add to be included in the error log. |
| INT NumberOfFormatStrings | Number of format strings. |
| PWCHAR\* FormatStrings | An array of format specifiers for each argument passed below. |
| INT NumberOfInsertionStrings | Number of insertion strings. |
| … | Variable list of insertion strings. |

#### Returns

None.

#### Remarks

* This call is designed to be used from the Client Driver’s DriverEntry function (before DMF is initialized).
* This function performs non-trivial parsing and manipulation of the given parameters to output a proper entry in the Event Log.

### DMF\_Utility\_EventLogEntryWriteDriver

VOID

DMF\_Utility\_EventLogEntryWriteDriver(

\_In\_ WDFDRIVER Driver,

\_In\_ NTSTATUS ErrorCode,

\_In\_ NTSTATUS Status,

\_In\_ ULONG UniqueId,

\_In\_ ULONG TextLength,

\_In\_opt\_ PCWSTR Text,

\_In\_ INT NumberOfFormatStrings,

\_In\_opt\_ PWCHAR\* FormatStrings,

\_In\_ INT NumberOfInsertionStrings,

...

);

Given a WDF WDFDRIVER object, write an event to the Event Log using a variable number of parameters.

#### Parameters

|  |  |
| --- | --- |
| WDFDRIVER Driver | The given WDFDRIVER object. |
| NTSTATUS ErrorCode | ErrorCode from header generated by mc compiler. |
| NTSTATUS FinalNtStatus | The final NTSTATUS for the error being logged. |
| ULONG UniqueId | Unique long word that identifies the specific call to this function. |
| ULONG TextLength | The length in bytes (including the terminating NULL) of the Text string. |
| PCWSTR Text | Additional data to add to be included in the error log. |
| INT NumberOfFormatStrings | Number of format strings. |
| PWCHAR\* FormatStrings | An array of format specifiers for each argument passed below. |
| INT NumberOfInsertionStrings | Number of insertion strings. |
| … | Variable list of insertion strings. |

#### Returns

None.

#### Remarks

* This call is designed to be used from the Client Driver’s DriverEntry function (before DMF is initialized).
* This function performs non-trivial parsing and manipulation of the given parameters to output a proper entry in the Event Log.

### DMF\_Utility\_EventLogEntryWriteDevice

VOID

DMF\_Utility\_EventLogEntryWriteDevice(

\_In\_ WDFDEVICE Device,

\_In\_ NTSTATUS ErrorCode,

\_In\_ NTSTATUS Status,

\_In\_ ULONG UniqueId,

\_In\_ ULONG TextLength,

\_In\_opt\_ PCWSTR Text,

\_In\_ INT NumberOfFormatStrings,

\_In\_opt\_ PWCHAR\* FormatStrings,

\_In\_ INT NumberOfInsertionStrings,

...

);

Given a WDF WDFDEVICE object, write an event to the Event Log using a variable number of parameters.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The given WDFDEVICE object. |
| NTSTATUS ErrorCode | ErrorCode from header generated by mc compiler. |
| NTSTATUS FinalNtStatus | The final NTSTATUS for the error being logged. |
| ULONG UniqueId | Unique long word that identifies the specific call to this function. |
| ULONG TextLength | The length in bytes (including the terminating NULL) of the Text string. |
| PCWSTR Text | Additional data to add to be included in the error log. |
| INT NumberOfFormatStrings | Number of format strings. |
| PWCHAR\* FormatStrings | An array of format specifiers for each argument passed below. |
| INT NumberOfInsertionStrings | Number of insertion strings. |
| … | Variable list of insertion strings. |

#### Returns

None.

#### Remarks

* This call is designed to be used from the Client Driver’s AddDevice function (before DMF is initialized).
* This function performs non-trivial parsing and manipulation of the given parameters to output a proper entry in the Event Log.

### DMF\_Utility\_EventLogEntryWriteDmfModule

VOID

DMF\_Utility\_EventLogEntryWriteDmfModule(

\_In\_ DMFMODULE DmfModule,

\_In\_ NTSTATUS ErrorCode,

\_In\_ NTSTATUS Status,

\_In\_ ULONG UniqueId,

\_In\_ ULONG TextLength,

\_In\_opt\_ PCWSTR Text,

\_In\_ INT NumberOfFormatStrings,

\_In\_opt\_ PWCHAR\* FormatStrings,

\_In\_ INT NumberOfInsertionStrings,

...

);

Given a DMFMODULE object, write an event to the Event Log using a variable number of parameters.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The given DMFMODULE object. |
| NTSTATUS ErrorCode | ErrorCode from header generated by mc compiler. |
| NTSTATUS FinalNtStatus | The final NTSTATUS for the error being logged. |
| ULONG UniqueId | Unique long word that identifies the specific call to this function. |
| ULONG TextLength | The length in bytes (including the terminating NULL) of the Text string. |
| PCWSTR Text | Additional data to add to be included in the error log. |
| INT NumberOfFormatStrings | Number of format strings. |
| PWCHAR\* FormatStrings | An array of format specifiers for each argument passed below. |
| INT NumberOfInsertionStrings | Number of insertion strings. |
| … | Variable list of insertion strings. |

#### Returns

None.

#### Remarks

* This call is designed to be used from Modules or a Client Driver function that has access to an instantiated Module.
* This function performs non-trivial parsing and manipulation of the given parameters to output a proper entry in the Event Log.

### DMF\_Utility\_EventLogEntryWriteUserMode

\_IRQL\_requires\_max\_(PASSIVE\_LEVEL)

VOID

DMF\_Utility\_EventLogEntryWriteUserMode(

\_In\_ PWSTR Provider,

\_In\_ WORD EventType,

\_In\_ DWORD EventID,

\_In\_ INT NumberOfFormatStrings,

\_In\_opt\_ PWCHAR\* FormatStrings,

\_In\_ INT NumberOfInsertionStrings,

...

);

Write an Event Log entry specifically in User-mode.

#### Parameters

|  |  |
| --- | --- |
| PWSTR Provider | Provider of the event. |
| WORD EventType | Type of the event: EVENTLOG\_SUCCESS/EVENTLOG\_ERROR\_TYPE/  EVENTLOG\_INFORMATION\_TYPE/  EVENTLOG\_WARNING\_TYPE |
| DWORD EventID | EventId from header generated by mc compiler. |
| INT NumberOfFormatStrings | Number of format strings. |
| PWCHAR\* FormatStrings | An array of format specifiers for each argument passed below. |
| INT NumberOfInsertionStrings | Number of insertion strings. |
| … | Variable list of insertion strings. |

#### Returns

None.

#### Remarks

* This function is only available if DMF\_USER\_MODE is defined.
* This function performs non-trivial parsing and manipulation of the given parameters to output a proper entry in the Event Log.

### DMF\_Utility\_IsEqualGUID

BOOLEAN

DMF\_Utility\_IsEqualGUID(

\_In\_ GUID\* Guid1,

\_In\_ GUID\* Guid2

);

Compares two GUIDs.

#### Parameters

|  |  |
| --- | --- |
| GUID\* Guid1 | The first GUID. |
| GUID\* Guid2 | The second GUID. |

#### Returns

TRUE if the two GUIDs are the same. FALSE, otherwise.

#### Remarks

This function is better than the version provided in the DDK because it is portable between Kernel and User-mode.

### DMF\_Utility\_UserModeAccessCreate

NTSTATUS

DMF\_Utility\_UserModeAccessCreate(

\_In\_ WDFDEVICE Device,

\_In\_opt\_ const GUID\* DeviceInterfaceGuid,

\_In\_opt\_ WCHAR\* SymbolicLinkName

);

Given a WDFDEVICE, create an associated device interface and/or symbolic link.

#### Parameters

|  |  |
| --- | --- |
| WDFDEVICE Device | The given WDFDEVICE. |
| GUID\* DeviceInterfaceGuid | The GUID of the device interface to create. |
| WCHAR\* SymbolicLinkName | The name of the symbolic link to create. |

#### Returns

STATUS\_SUCCESS if the allocation succeeded. Otherwise, an error code is returned.

#### Remarks

* **Do not use this function. Use DMF\_IoctlHandler instead.**
* This function is present for legacy reasons.

# DMF Coding Conventions

DMF Module source code is designed to be read and updated by many people for many years. For this reason, strict coding guidelines are enforced that are particularly useful for such a code base.

Note: Client Drivers are free to use their own coding conventions. The coding conventions listed here apply only to DMF Modules.

## Conventions

### Do:

* When creating a Module, follow the directions in this document so that all the sections of code are in the same order as all other Modules.
* Functions that have multiple parameters must place the parameters vertically instead of the same line. This makes code reviews easier to read.
* Names of private functions in Modules begin with [ModuleName]\_.
* Names of DMF callbacks in Modules begin with DMF\_[ModuleName]\_.
* Names of WDF callbacks in Modules also begin with DMF\_[ModuleName]\_.
* Follow the conventions for naming files that contain Module code. These conventions are listed above in this document.
* Use function headers to document every function using the format in any DMF Module (or the Template Module).
* All comments should be grammatically correct English sentences that are properly capitalized and end with a period. (DMF source code is read by many people who have various exposure to English. Proper grammar increases the chance that more people can clearly understand the comments and code easily.)
* Acronyms are always Pascal case (first letter is the only letter capitalize). Use “Usb” not “USB”. This makes it easier and clear when they are used in long names, such as “UsbHubRead”.
* Formal parameters are Pascal case always starting with a capital letter.
* Local variables are camel case always starting with a lower-case letter.

### Do Not:

* **Do not use abbreviations.** Abbreviations are arbitrary and make searching for code difficult. They are also easy to misinterpret. Using full words to name variables makes it easier for many people to read and search the code base for many years to come. It reduces the chances of misunderstanding arbitrary abbreviations. It makes searching for names easier. It makes intellisense work better in the compiler’s editor.
* Do not use Hungarian Notation such as “pVariable”, “nCount”.
* Do not use underscores in variables.
* Do not use underscores in Dmf Module names.
* Do not use underscores in functions unless there is a clear need for indicating a sub group of functions.
* Do not typedef pointers to structures. Do not use typedef pointers to types such as PUCHAR, PULONG, etc. Use UCHAR\* and ULONG\* instead.

# Additional Information

## Git Repository

DMF is available in a public GIT repository here:

<https://github.com/Microsoft/DMF>

## Contributors

Several people in the Microsoft Surface Team have contributed in many ways to DMF. These include:

* Sweta Ananth
* Charlie Arcuri
* Scott Durland
* Pankaj Gupta
* Vivek Gupta
* Rajesh Gururaj
* Anoop Kurungod
* Sergii Liashenko
* Fizal Peermohamed
* Mika Rintamaeki
* Sam Tertzakian
* Raja Venkatachalam
* Eliyas Yakub
* Rob Young