Driver Module Framework (DMF)

Sam Tertzakian and Rajesh Gururaj
Principal/Senior Software Engineers

Microsoft Devices Team

Introduction and Agenda

Session Goals

Introduction to DMF and Explanation of Core DMF Concepts

Session Agenda:

Part 1: Summary of WinHEC talk, Overview of GitHub site, Clone and Compile DMF and Samples.

Part 2: Modules and Libraries, Steps to Create a Module and Library from scratch, Protocol/Transport Feature

Part 3: Review of Modules in Library and View of Surface Library

Part 4: Reviews of Surface Drivers

Part 1

Section Introduction

What is DMF and what are its goals?

Section Agenda:

Traditional driver diagram and discussion

Introduce DMFMODULE and DMF Core

DMF driver diagram and discussion

Modules discussion

How to make a DMF driver?

Look at sample source code

DMF Resources and Overview of GitHub site

Clone Repository and Build DMF and Samples

What is DMF and what are its goals?

Framework that makes it easier to write better drivers

"DMF Drivers" are "WDF Drivers". Programmers use WDF and DMF together.

Goals

Make it easier and more intuitive to write **modular**, **layered code** inside drivers.

Make it possible to **directly reuse** (by linking) driver code without using "copy/paste/modify".

Make it easier to **properly architect** drivers by eliminating improper dependencies and code paths.

Make it easier for driver writers to think using **high-level constructs**.

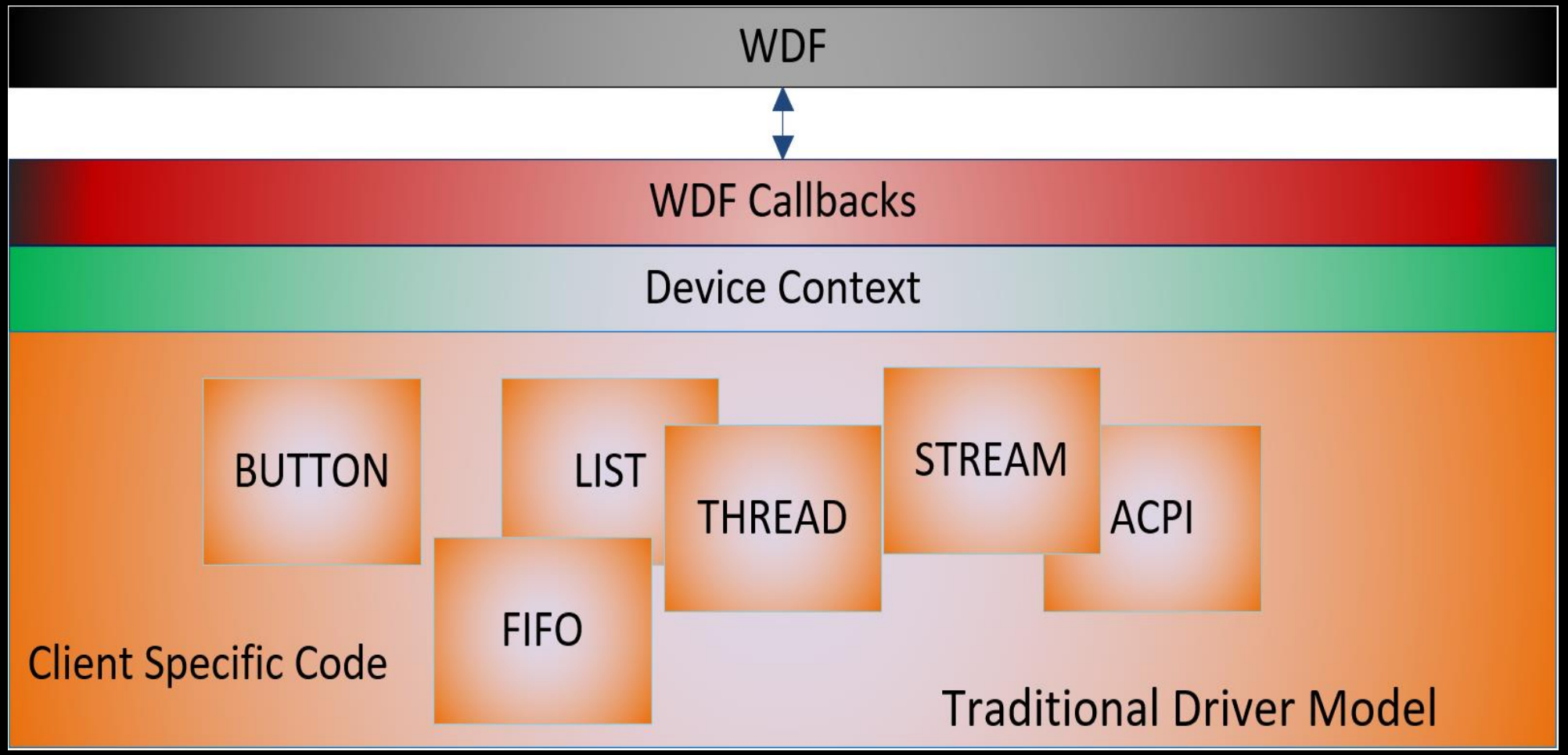
Make it easier for driver writers to create their own high-level constructs and let **others reuse** them.

Make driver programming **easier**, **faster**, **cheaper** and more **satisfying**.

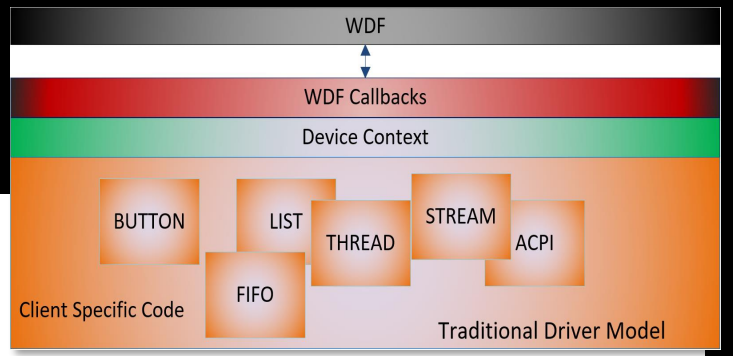
Result when the above goals are met:

Programmers spend more time thinking, writing and debugging **new code that accomplishes their specific requirements** and less time writing code that has been written many times before.

Traditional Driver Diagram

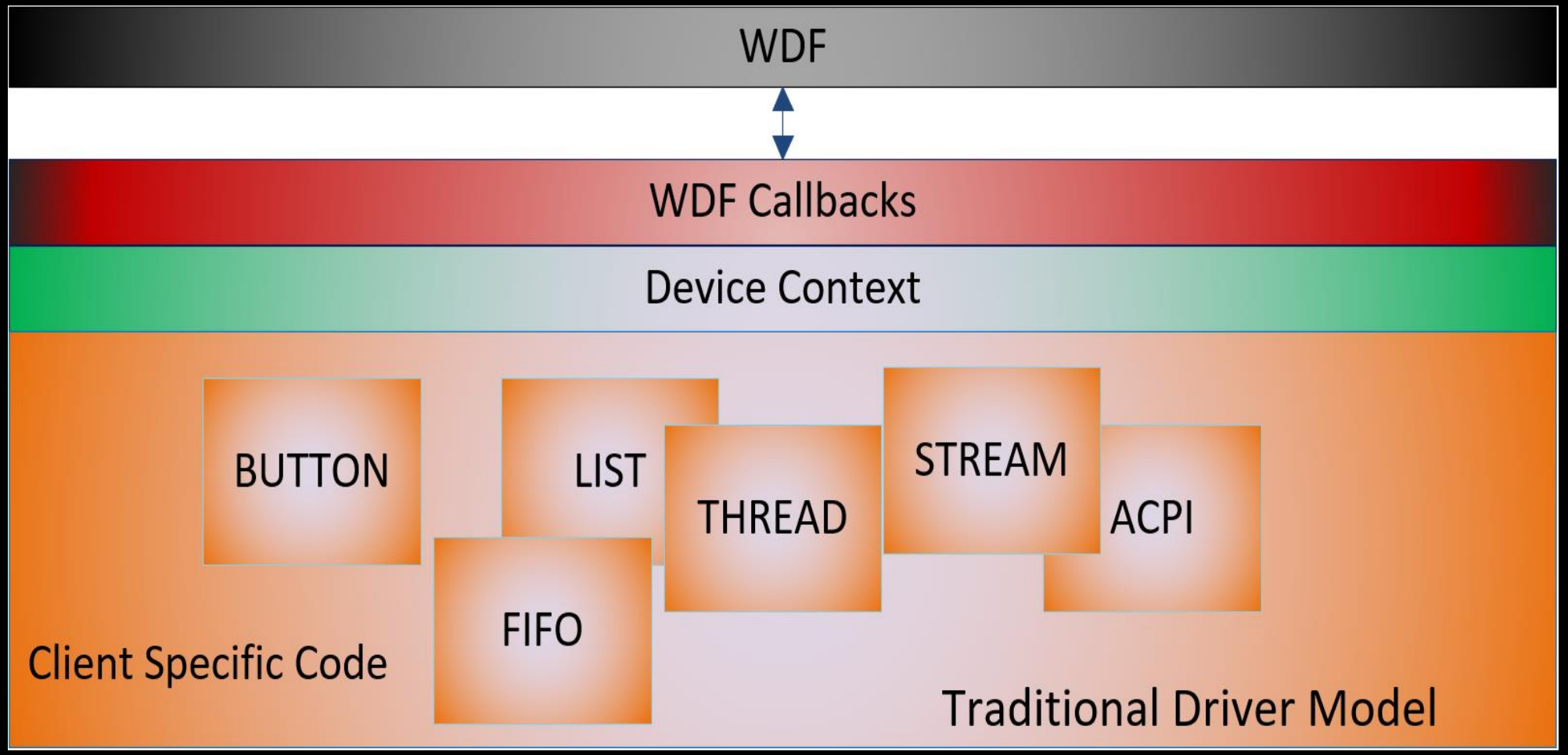


C++ helps…but still needs “glue”

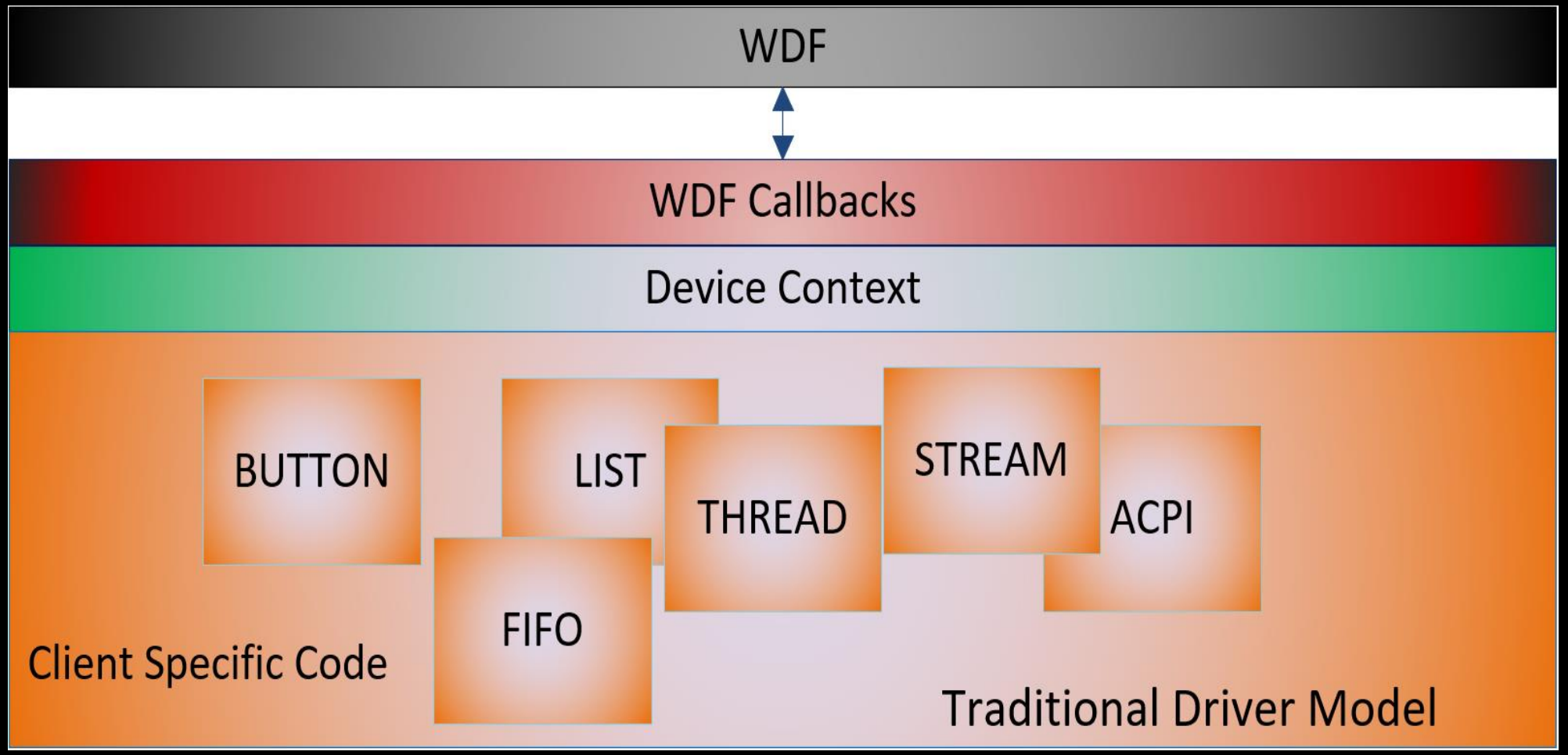



```
NTSTATUS MyDriver_D0Entry(…) {
DEVICE_CONTEXT*
deviceContext=DeviceContextGet(Device);


  deviceContext->Button.D0Entry(deviceContext);
  deviceContext->Thread.D0Entry(deviceContext);
  /*… some other code */
  deviceContext-> Stream.D0Entry(deviceContext);
}
```

Traditional Driver Diagram



Traditional Driver Diagram

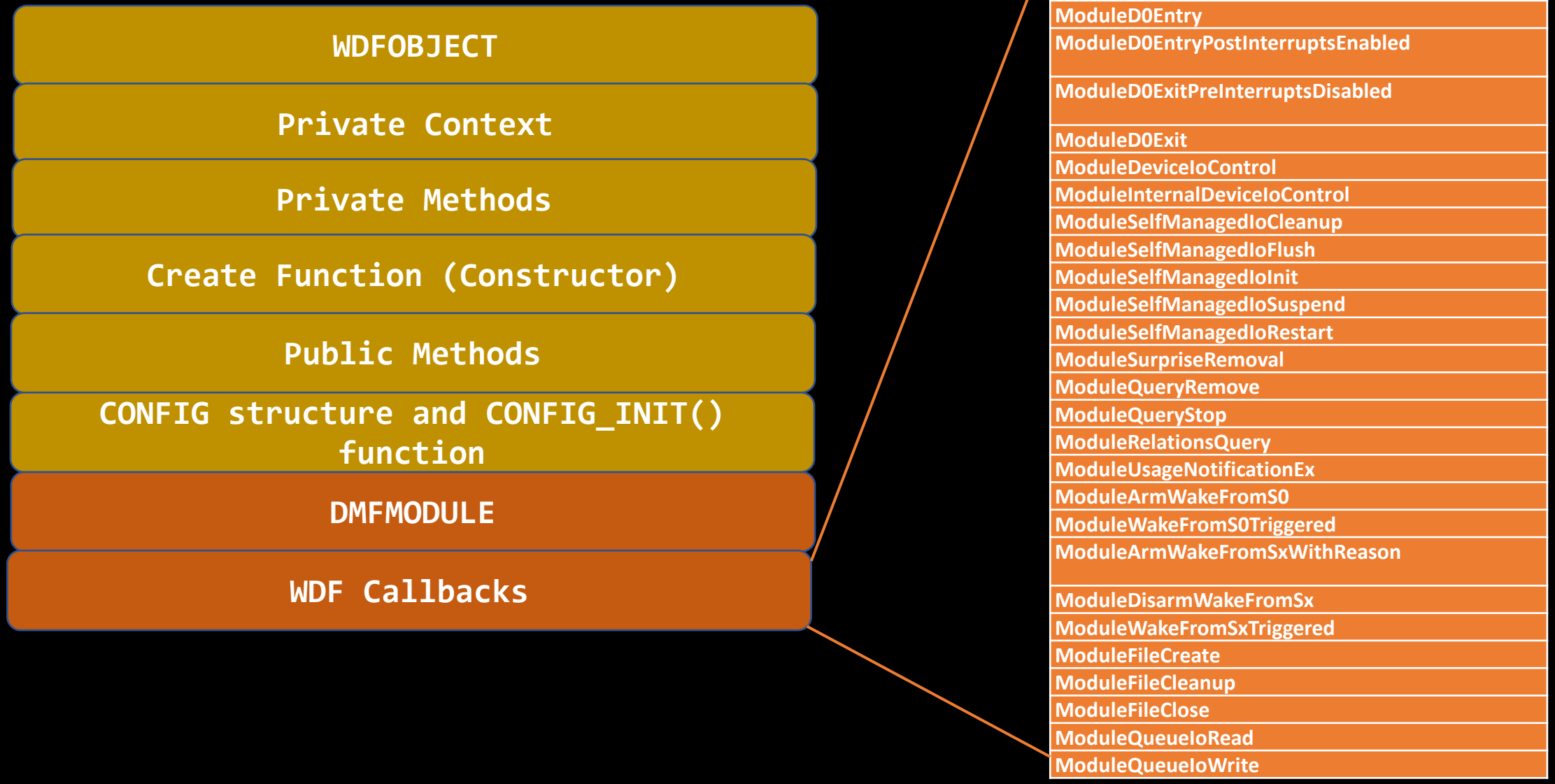


Introduction to DMF

DMFMODULE

DMF Core

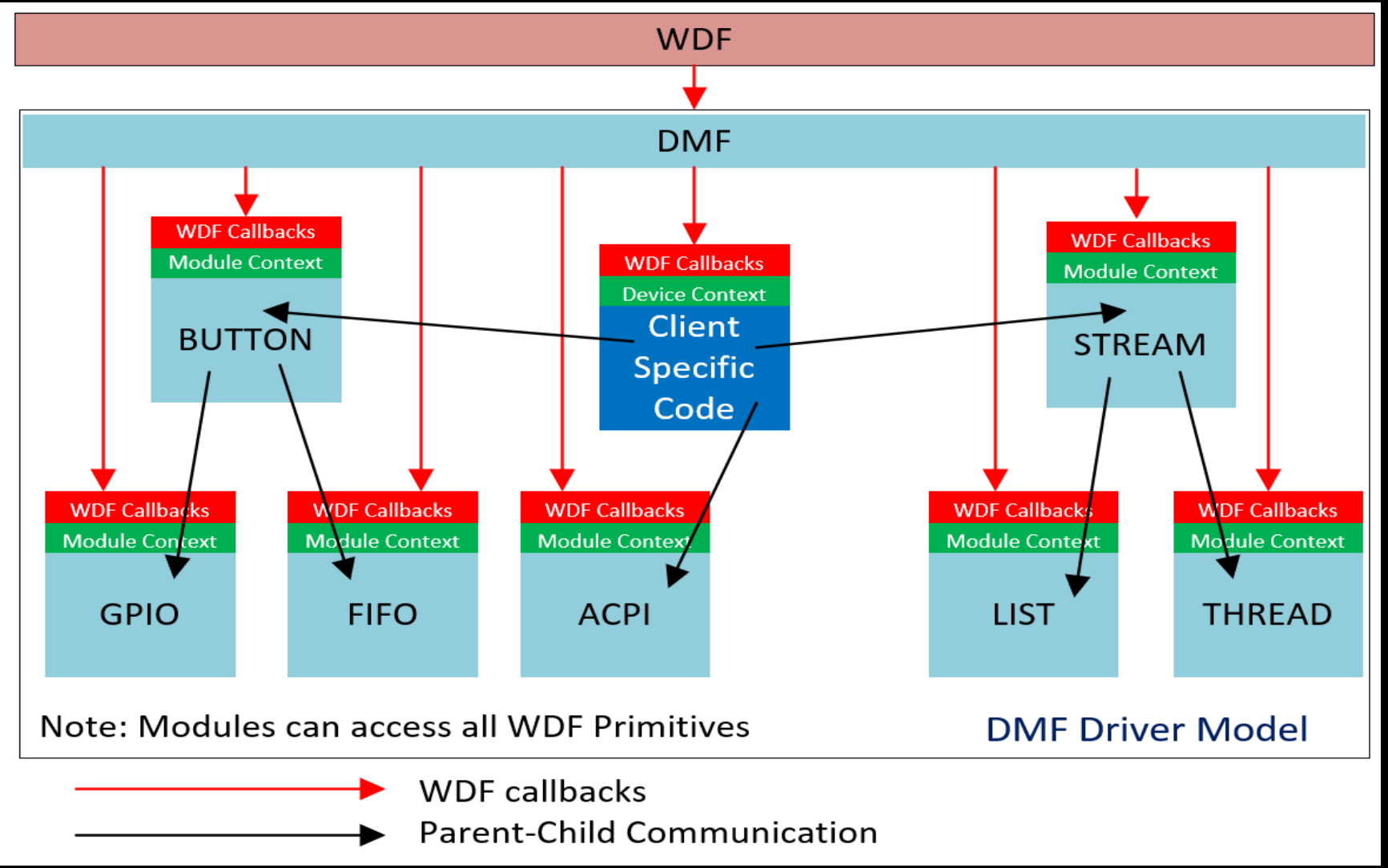
DMF Module (DMFMODULE)



DMF Core

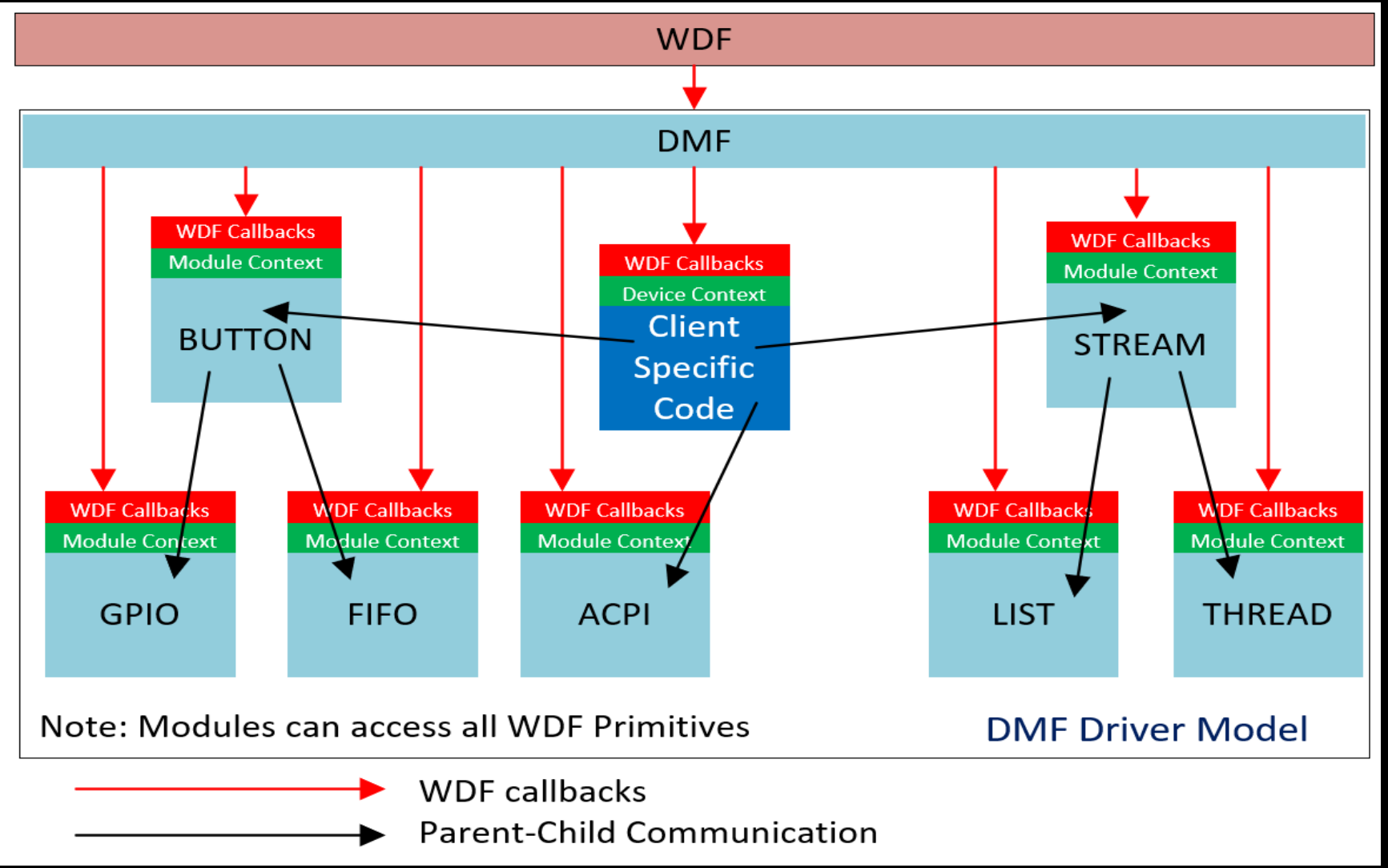
- The DMF Core is a framework is statically linked to the Client driver.
- It resides between the Client driver and WDF.
- The Client driver gives DMF Core a list of each DMFMODULE it wants to use.
- **When WDF calls into the Client driver via a callback, DMF dispatches the callback to every DMFMODULE it has created on behalf of the Client.**

DMF Driver Diagram



Goal: Make it easier to **properly architect** drivers by eliminating improper dependencies and code paths.

DMF Driver Diagram



Goal: Make it easier to **properly architect** drivers by eliminating improper dependencies and code paths.

Other Properties of a Module (1)

- Any Module can instantiate **as many instances of other Modules as Child Modules as it wants**. There is no limit to the number of descendants or siblings a Module can have (except for memory).
- Programmers can use Modules and create new Modules.
- When a programmer creates a new `DMFMODULE`, DMF automatically creates the `DMFMODULE` using an underlying `WDFOBJECT`.
- A Module's functionality is always accessed using a handle that is created by DMF and given to the Client.
- Internally, the Module uses its handle to retrieve its private context. This is similar to how a driver uses its WDFDEVICE to access its device context.

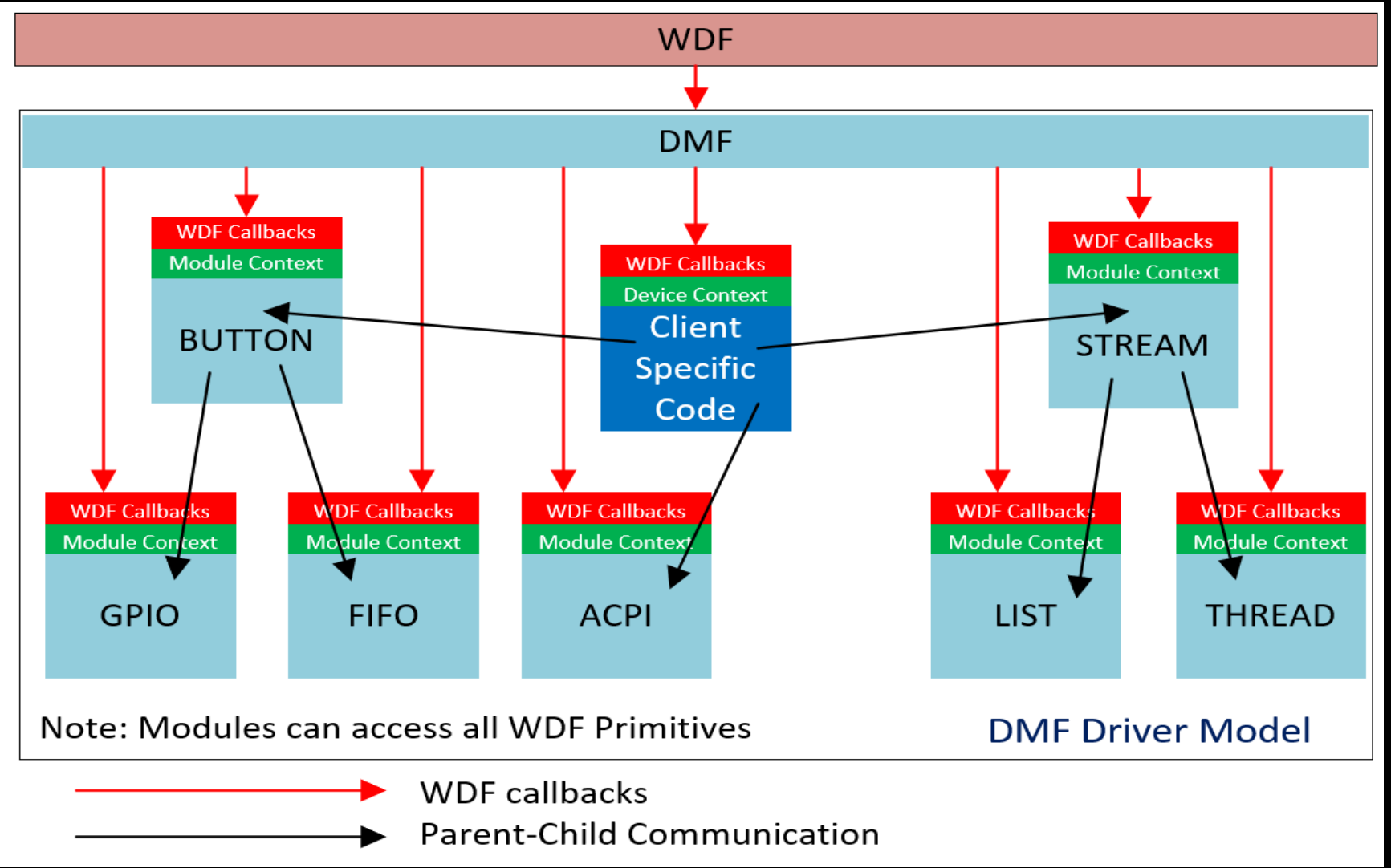
Other Properties of a Module (2)

- As with any object-oriented programming paradigm, Modules are agnostic about their parent which can be either the Client Driver or another Module.

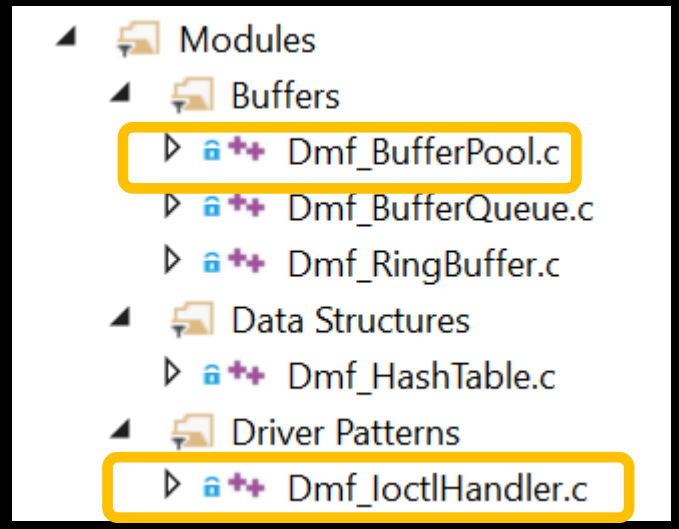
- DMF Modules can abstract any kind of code, from simple object like a list of buffers to an entire algorithm, data structure, programming pattern and even a full device driver.

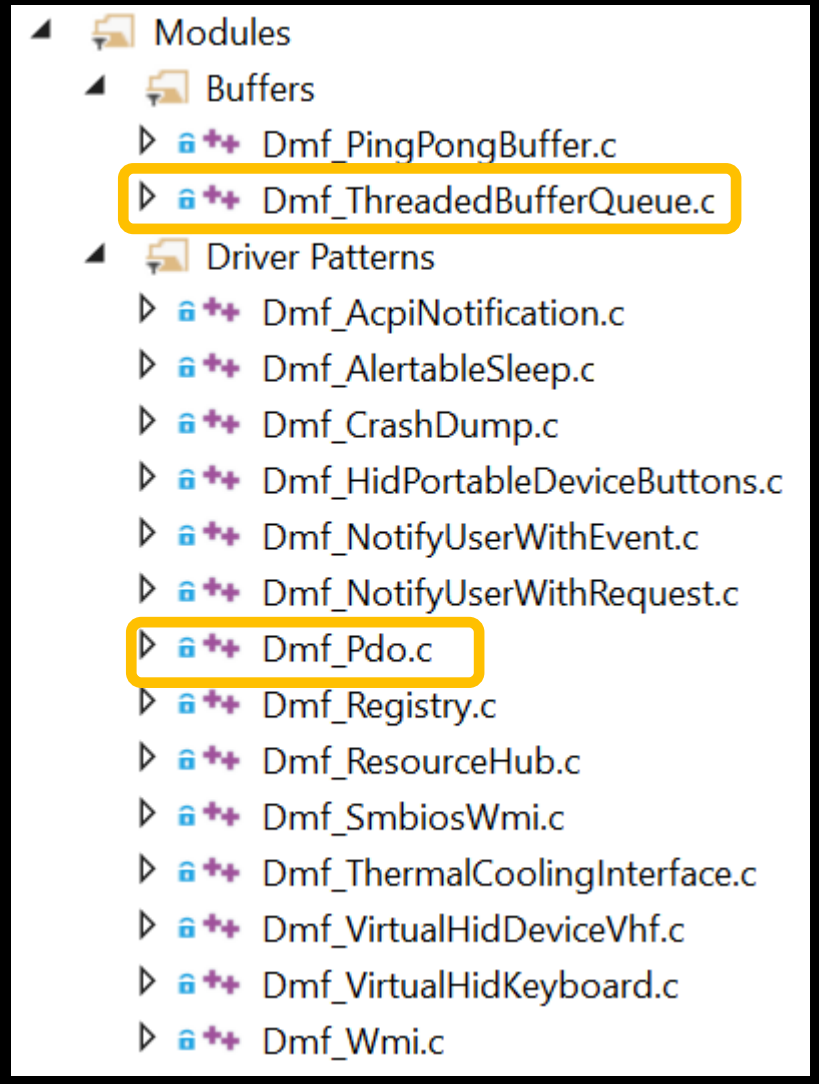
- Important: Like WDF, DMF's interfaces to the Client driver are in C. Like WDF, DMF can also be used in C++ drivers.

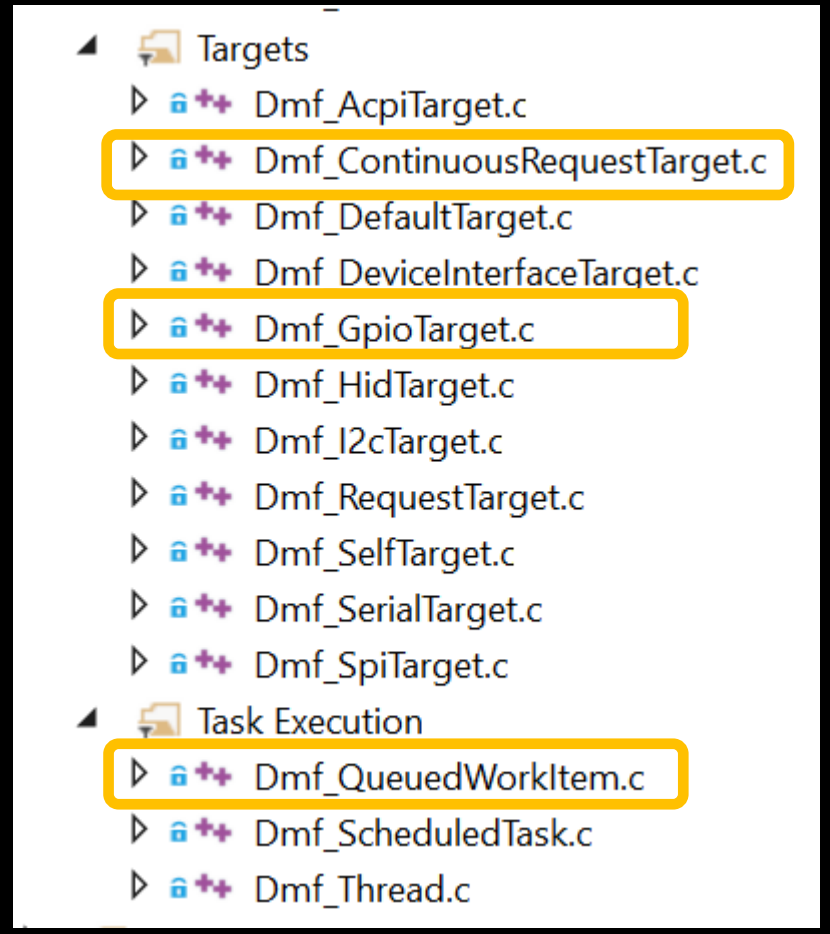
DMF Driver Diagram



Modules in Library





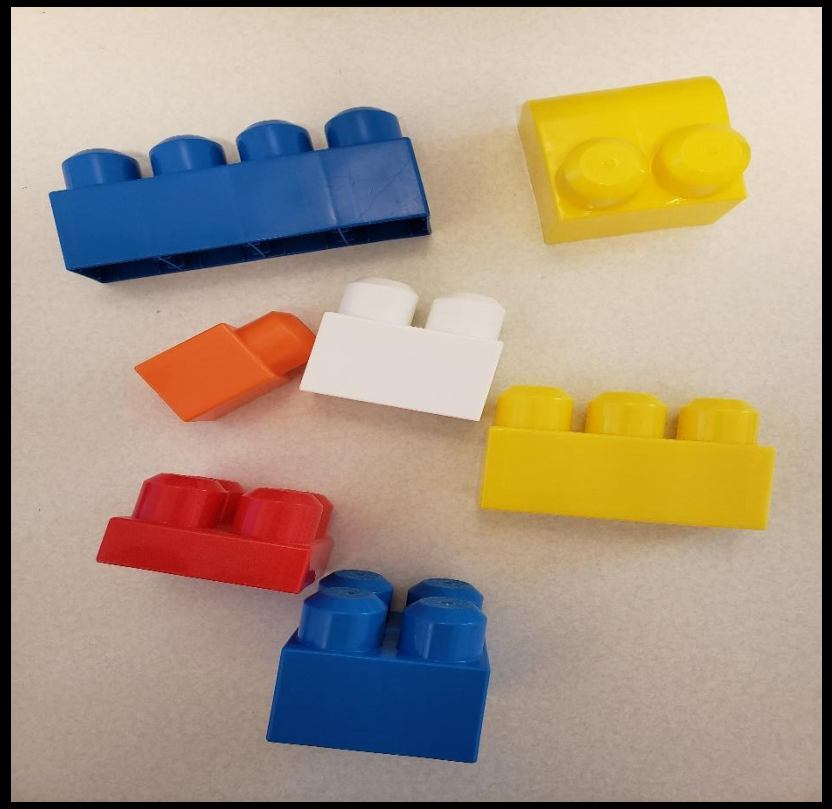


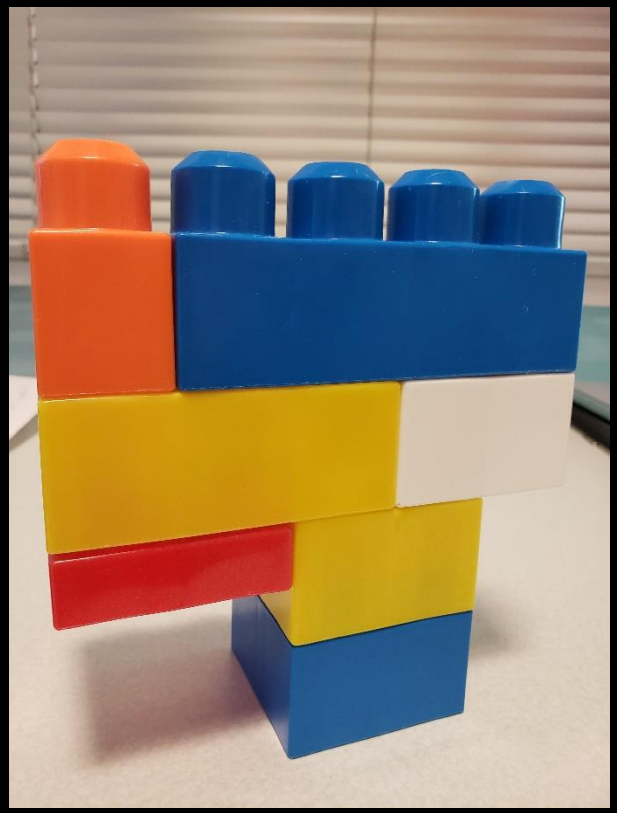
Goal: Make it easier for driver writers to think using **high-level constructs**.

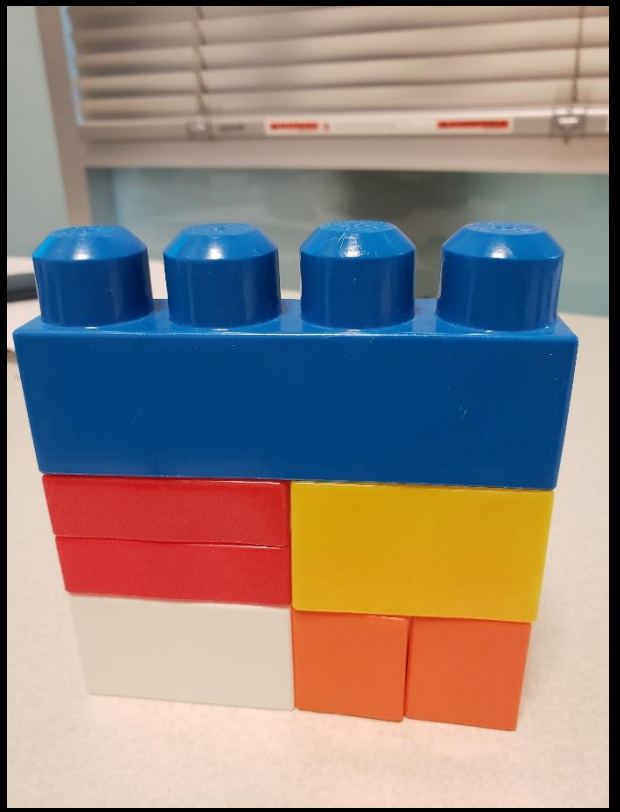
Modules have a Common Interface

Module's Create Function is called by DMF

Create() function is agnostic regardless of Module type so any Module can use any other Module.

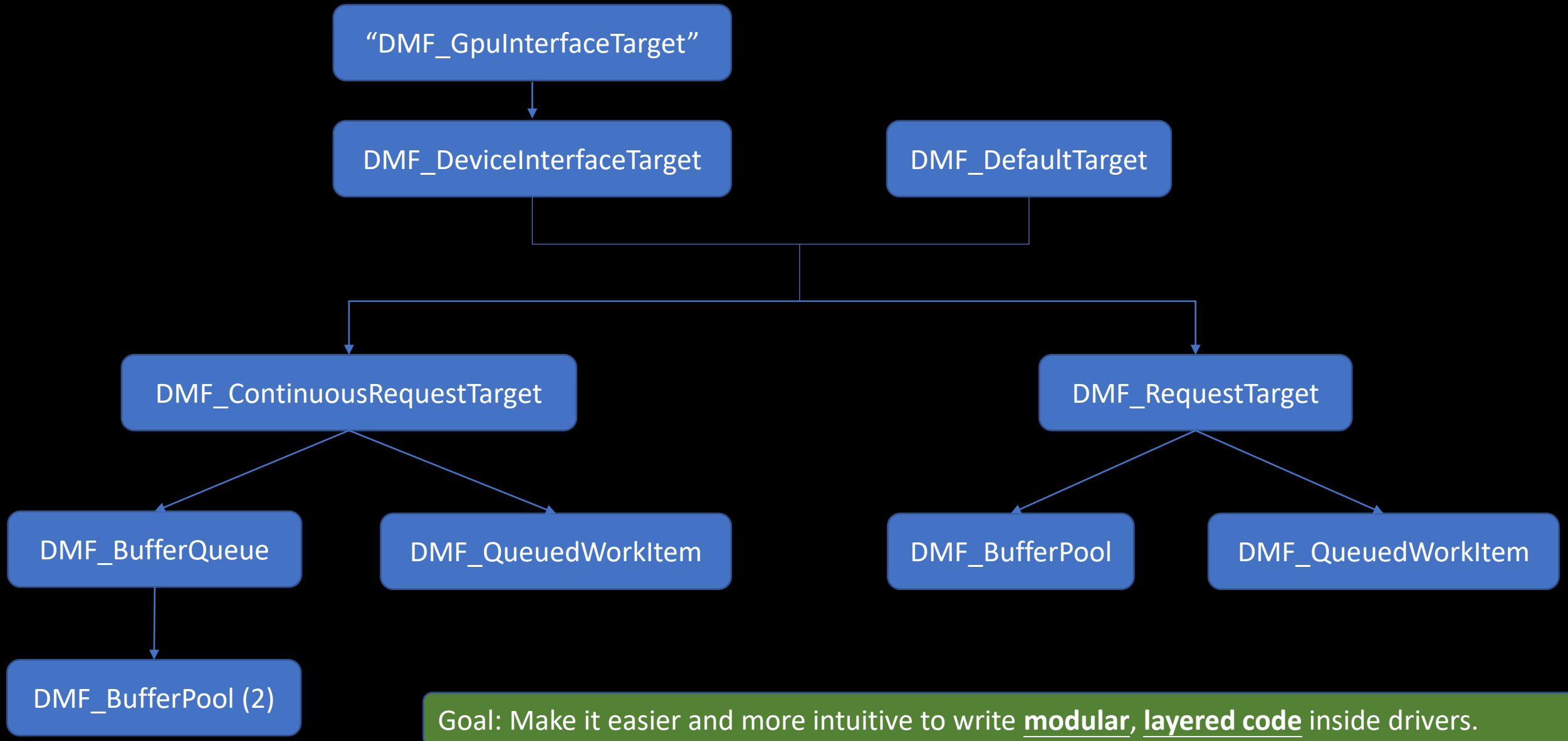






Goal: Make it easier for driver writers to create their own high-level constructs and let **others reuse** them.

Module Hierarchy Example



Goal: Make it easier and more intuitive to write **modular**, **layered code** inside drivers.

How to make a DMF Driver?

What makes a normal WDF driver a DMF driver?

1. A WDF Client driver must call APIs to hook DMF into the Client driver:
   - `DMF_DmfDeviceInitAllocate()`
   - `DMF_DmfDeviceInitHookPnpPowerEventCallbacks()`
   - `DMF_DmfDeviceInitHookFileObjectConfig()`
   - `DMF_DmfDeviceInitHookPowerPolicyEventCallbacks()`
   - `DMF_DmfDeviceInitHookQueueConfig()` (optional)
   - DMF_DmfFdoSetFilter(); (optional)

2. A WDF Client driver must call this API to initialize DMF.
   - `DMF_ModulesCreate()`

3. The Client driver receives one additional callback where the list of Modules the driver uses is given to DMF.

How to use a Module

- You need two things:
  - The Module's CONFIG structure.
  - The Module's Methods.
- Every Module has a .md File
  - This file contains documentation about the purpose of the Module, its CONFIG, structures, enumerations, callbacks and Methods.

DMF Sample Code Tour

Enough theory…Let’s look at code!

There are several samples on Github. Many of the samples use the OSR USB FX2 board because it is accessible and well known.

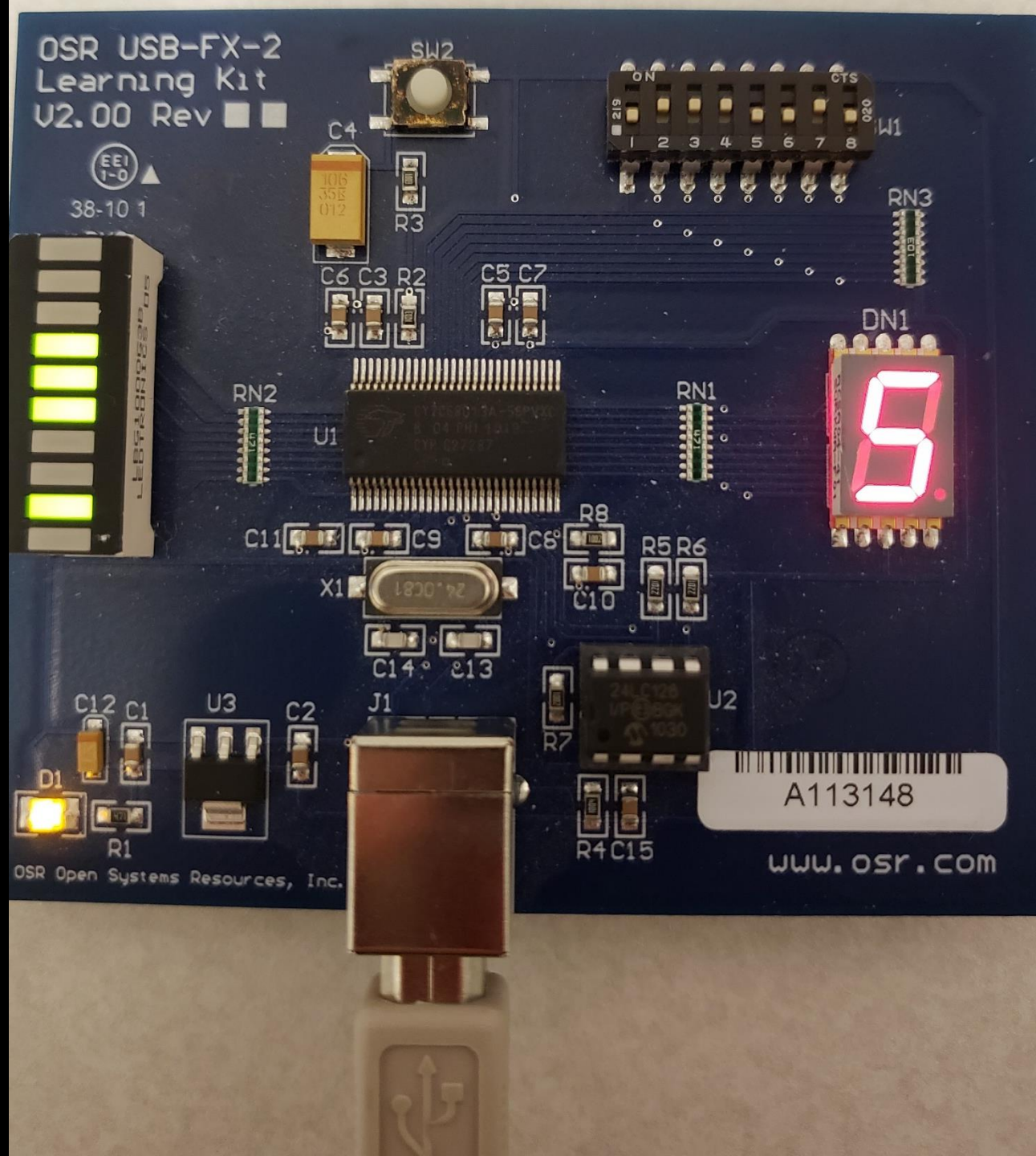
(Detailed information about the OSR USB FX2 board is available in the MSDN device driver samples repository.)

The board has a bank of 8 switches, a bank of lights, a button and an LED segment display.

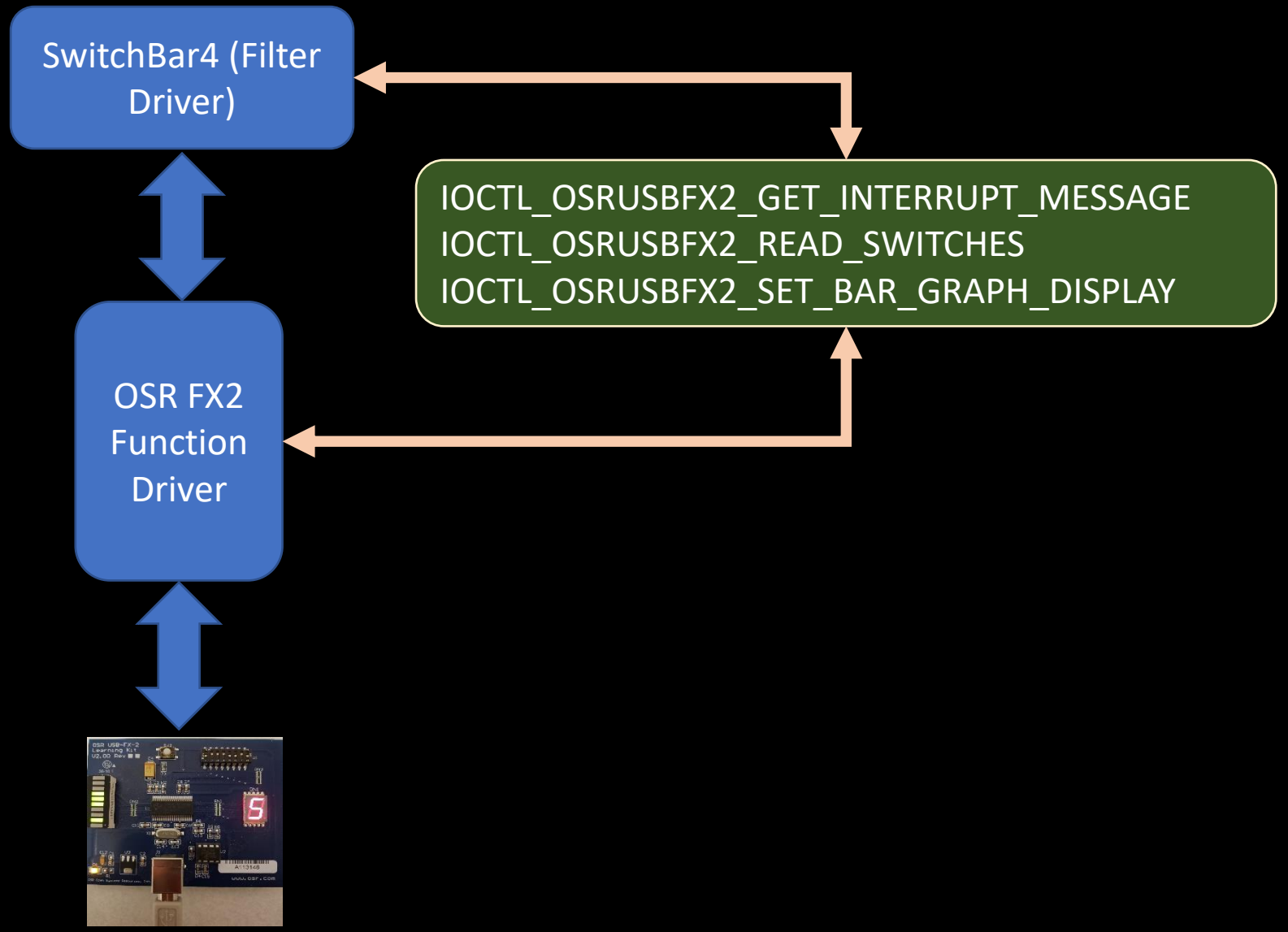
A sample function driver for the board is provided on MSDN.

The driver exposes a device interface and IOCTL codes that allow an application to control the lights on the lightbar as well as get notification that the switches have changed and what the switches are set to (on/off).

Using the application a user can read the switches and set the lights.



SwitchBar4 (Filter Driver)



SwitchBar4 Sample

- High Level Tasks for New Driver
  - When D0Entry() occurs read the state of switches and set lights to match.
  - Send IOCTL requests that are completed every time the switches are changed.
  - Every time switches are changed, the driver should read the state, convert the bit mask and set the lights to match.

Initializing DMF in a Client Driver (1)

```
NTSTATUS MyDriver_DeviceAdd(PDEVICEINIT DeviceInit){
    NTSTATUS ntStatus; WDFDEVICE device; PDMFDEVICE_INIT dmfDeviceInit;
    DMF_EVENT_CALLBACKS dmfCallbacks;
    WDF_OBJECT_ATTRIBUTES objectAttributes; WDF_PNPPOWER_EVENT_CALLBACKS pnpPowerCallbacks;

    dmfDeviceInit = DMF_DmfDeviceInitAllocate(DeviceInit);

    WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);
    pnpPowerCallbacks.EvtDeviceD0Entry = SwitchBarEvtDeviceD0Entry;
    DMF_DmfDeviceInitHookPnpPowerEventCallbacks(dmfDeviceInit, &pnpPowerCallbacks);
    WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);

    DMF_DmfDeviceInitHookFileObjectConfig(dmfDeviceInit, NULL);
    DMF_DmfDeviceInitHookPowerPolicyEventCallbacks(dmfDeviceInit, NULL);

    WdfDeviceInitSetDeviceType(DeviceInit, FILE_DEVICE_UNKNOWN);
    WdfDeviceInitSetExclusive(DeviceInit, FALSE);
    WdfFdoInitSetFilter(DeviceInit);
    DMF_DmfFdoSetFilter(dmfDeviceInit);
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&objectAttributes, DEVICE_CONTEXT);
    ntStatus = WdfDeviceCreate(DeviceInit, &ObjectAttributes, &device); { if (!NT_SUCCESS(ntStatus) goto Exit; }
    dmfCallbacks.EvtDmfDeviceModulesAdd = DmfDeviceModulesAdd;
    DMF_DmfDeviceInitSetEventCallbacks(dmfDeviceInit, &dmfCallbacks);
    ntStatus = DMF_ModulesCreate(device, &dmfDeviceInit);
Exit:
    if (dmfDeviceInit != NULL) DMF_DmfDeviceInitFree(&dmfDeviceInit);
    return ntStatus; }
```

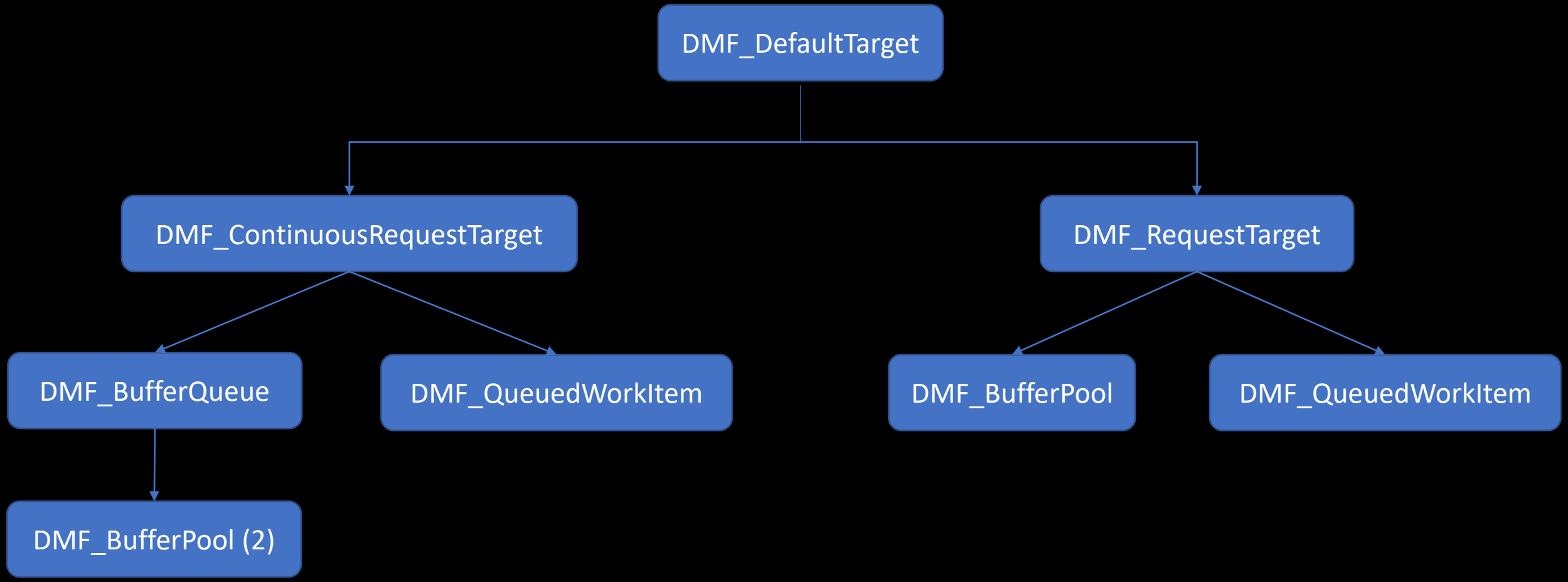
Initializing DMF in a Client Driver (2)

```
NTSTATUS MyDriver_DeviceAdd(PDEVICEINIT DeviceInit){
    NTSTATUS ntStatus; WDFDEVICE device; PDMFDEVICE_INIT dmfDeviceInit;
    DMF_EVENT_CALLBACKS dmfCallbacks;
    WDF_OBJECT_ATTRIBUTES objectAttributes; WDF_PNPPOWER_EVENT_CALLBACKS pnpPowerCallbacks;

    dmfDeviceInit = DMF_DmfDeviceInitAllocate(DeviceInit);

    WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);
    pnpPowerCallbacks.EvtDeviceD0Entry = SwitchBarEvtDeviceD0Entry;
    DMF_DmfDeviceInitHookPnpPowerEventCallbacks(dmfDeviceInit, &pnpPowerCallbacks);
    WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);

    DMF_DmfDeviceInitHookFileObjectConfig(dmfDeviceInit, NULL);
    DMF_DmfDeviceInitHookPowerPolicyEventCallbacks(dmfDeviceInit, NULL);

    WdfDeviceInitSetDeviceType(DeviceInit, FILE_DEVICE_UNKNOWN);
    WdfDeviceInitSetExclusive(DeviceInit, FALSE);
    WdfFdoInitSetFilter(DeviceInit);
    DMF_DmfFdoSetFilter(dmfDeviceInit);
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&objectAttributes, DEVICE_CONTEXT);
    ntStatus = WdfDeviceCreate(DeviceInit, &ObjectAttributes, &device); { if (!NT_SUCCESS(ntStatus) goto Exit; }
    dmfCallbacks.EvtDmfDeviceModulesAdd = DmfDeviceModulesAdd;
    DMF_DmfDeviceInitSetEventCallbacks(dmfDeviceInit, &dmfCallbacks);
    ntStatus = DMF_ModulesCreate(device, &dmfDeviceInit);
Exit:
    if (dmfDeviceInit != NULL) DMF_DmfDeviceInitFree(&dmfDeviceInit);
    return ntStatus; }
```

Tell DMF the List of Modules to Use

```
VOID DmfDeviceModulesAdd(WDFDEVICE Device, PDMFDEVICEINIT DmfDeviceInit) {
    DEVICE_CONTEXT* deviceContext = DeviceContextGet(Device);
    DMF_MODULE_ATTRIBUTES moduleAttributes;
    DMF_CONFIG_DefaultTarget moduleConfigDefaultTarget;

    DMF_CONFIG_DefaultTarget_AND_ATTRIBUTES_INIT(&moduleConfigDefaultTarget, &moduleAttributes);
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.BufferCountOutput = 1;
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.BufferOutputSize = sizeof(SWITCH_STATE);
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.ContinuousRequestCount = 1;
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.PoolTypeOutput = NonPagedPoolNx;
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.PurgeAndStartTargetInD0Callbacks = FALSE;
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.ContinuousRequestTargetIoctl =
IOCTL_OSRUSBFX2_GET_INTERRUPT_MESSAGE;
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.EvtContinuousRequestTargetBufferOutput =
SwitchBarSwitchChangedCallback;
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.RequestType = ContinuousRequestTarget_RequestType_Ioctl;
    moduleConfigDefaultTarget.ContinuousRequestTargetModuleConfig.ContinuousRequestTargetMode =
ContinuousRequestTarget_Mode_Automatic;
    moduleAttributes.PassiveLevel = TRUE;
    DMF_DmfModuleAdd(DmfModuleInit, &moduleAttributes, WDF_NO_OBJECT_ATTRIBUTES, &deviceContext->DmfModuleDefaultTarget);

    // Add more Modules as needed.
    //
}
```

SwtichBar4 Module Tree



DMF calling the Client Driver's D0Entry

```
NTSTATUS SwitchBarEvtDeviceD0Entry(WDFDEVICE Device, WDF_POWER_DEVICE_STATE PreviousState) {
    DEVICE_CONTEXT* deviceContext = DeviceContextGet(Device);
    NTSTATUS ntStatus;

    // Read the state of switches and initialize lightbar.
    //
    ntStatus = SwitchBarReadSwitchesAndUpdateLightBar(deviceContext->DmfModuleDefaultTarget);

    return ntStatus;
}
```

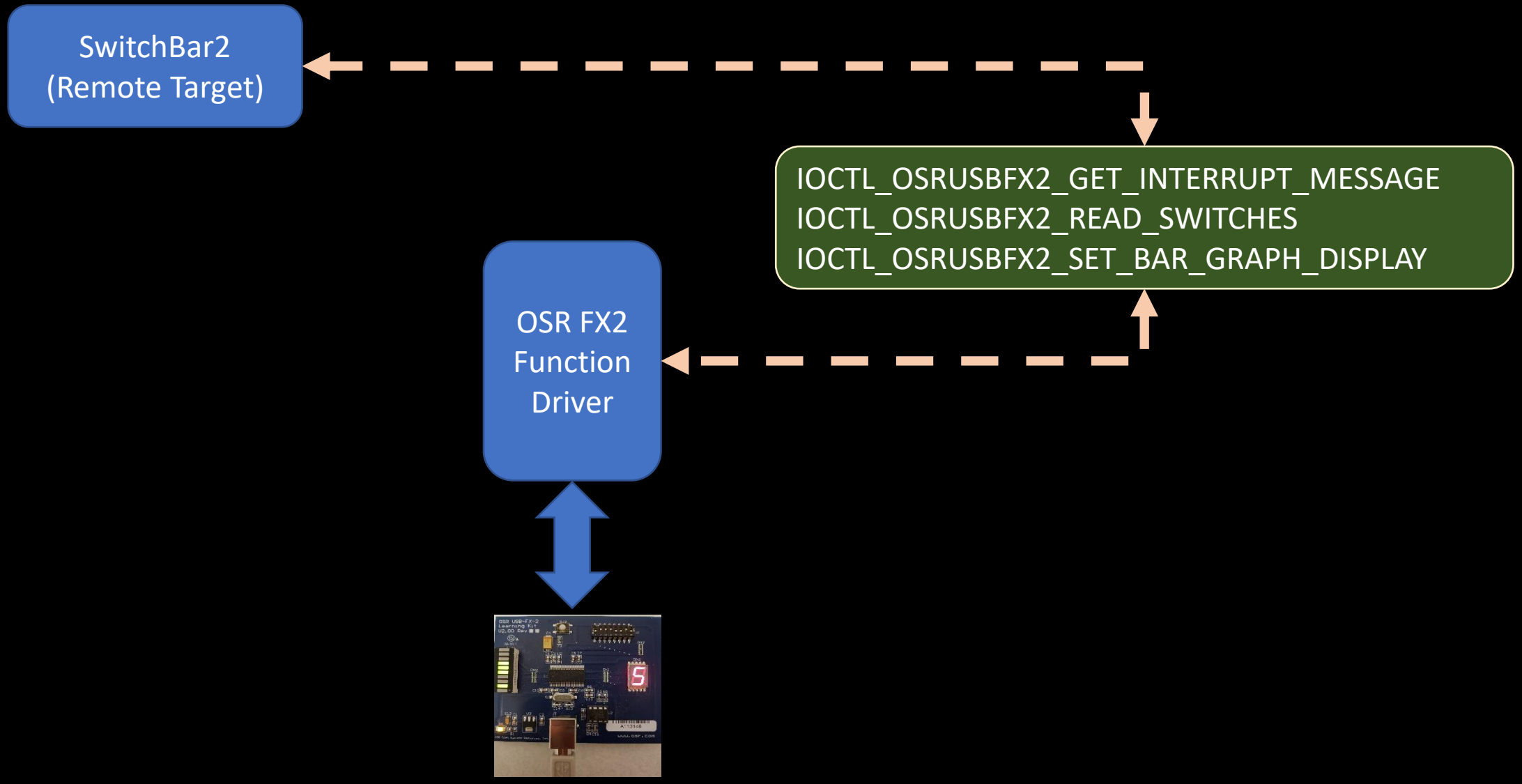
DMF calls Client Driver when Switch is Changed

```
ContinuousRequestTarget_BufferDisposition
SwitchBarSwitchChangedCallback(
    _In_ DMFMODULE DmfModuleDefaultTarget,
    _In_reads_(OutputBufferSize) VOID* OutputBuffer,
    _In_ size_t OutputBufferSize,
    _In_ VOID* ClientBufferContextOutput,
    _In_ NTSTATUS CompletionStatus
    )
{
    ContinuousRequestTarget_BufferDisposition returnValue;

    if (!NT_SUCCESS(CompletionStatus)) {
        returnValue = ContinuousRequestTarget_BufferDisposition_ContinuousRequestTargetAndStopStreaming;
        goto Exit;
    }

    SwitchBarReadSwitchesAndUpdateLightBar(DmfModuleDefaultTarget);

    returnValue = ContinuousRequestTarget_BufferDisposition_ContinuousRequestTargetAndContinueStreaming;
Exit:
    return returnValue;
}
```

Last function in driver. Does the work!

```
NTSTATUS SwitchBarReadSwitchesAndUpdateLightBar(_In_ DMFMODULE DmfModuleDefaultTarget) {
    NTSTATUS ntStatus;
    SWITCH_STATE switchData;
    ntStatus = DMF_DefaultTarget_SendSynchronously(DmfModuleDefaultTarget,
                                                   NULL,
                                                   0,
                                                   (VOID*)&switchData,
                                                   sizeof(SWITCH_STATE),
                                                   ContinuousRequestTarget_RequestType_Ioctl,
                                                   IOCTL_OSRUSBFX2_READ_SWITCHES,
                                                   0,
                                                   NULL);
    if (! NT_SUCCESS(ntStatus)) goto Exit;
    ntStatus = DMF_DefaultTarget_ SendSynchronously(DmfModuleDefaultTarget,
                                                    &switchData.SwitchesAsUChar,
                                                    sizeof(UCHAR),
                                                    NULL,
                                                    0,
                                                    ContinuousRequestTarget_RequestType_Ioctl,
                                                    IOCTL_OSRUSBFX2_SET_BAR_GRAPH_DISPLAY,
                                                    0,
                                                    NULL);
Exit:
    return ntStatus;}
```

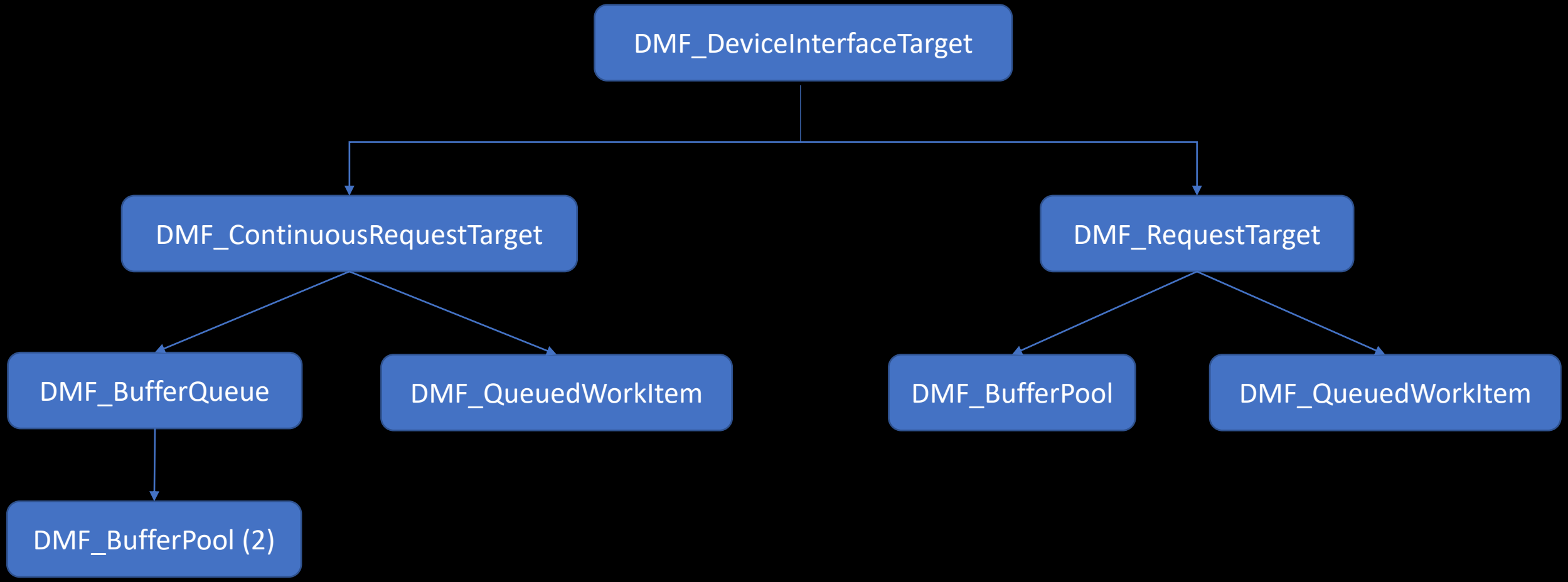
SwitchBar2 (Remote Target)



Tell DMF the List of Modules to Use

```
VOID DmfDeviceModulesAdd(_In_ WDFDEVICE Device, _In_ PDMFMODULE_INIT DmfModuleInit) {
    DMF_MODULE_ATTRIBUTES moduleAttributes;
    DMF_CONFIG_DeviceInterfaceTarget moduleConfigDeviceInterfaceTarget;
    DMF_MODULE_EVENT_CALLBACKS moduleEventCallbacks;

    DMF_CONFIG_DeviceInterfaceTarget_AND_ATTRIBUTES_INIT(&moduleConfigDeviceInterfaceTarget,
                                                         &moduleAttributes);
    moduleConfigDeviceInterfaceTarget.DeviceInterfaceTargetGuid = GUID_DEVINTERFACE_OSRUSBFX2;
    moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.BufferCountOutput = 1;
    moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.BufferOutputSize = sizeof(SWITCH_STATE);
    moduleConfigDeviceInterfaceTarget.ContinuousRequestTargetModuleConfig.ContinuousRequestCount = 1;
    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;
moduleAttributes.PassiveLevel = TRUE;

    DMF_MODULE_ATTRIBUTES_EVENT_CALLBACKS_INIT(&moduleAttributes, &moduleEventCallbacks);
    moduleEventCallbacks.EvtModuleOnDeviceNotificationPostOpen = SwitchBar_OnDeviceArrivalNotification;

    DMF_DmfModuleAdd(DmfModuleInit,
                     &moduleAttributes,
                     WDF_NO_OBJECT_ATTRIBUTES,
                     NULL);
}
```

SwtichBar4 Module Tree



DMF informs Client Remote Target Appears

```
VOID
SwitchBar_OnDeviceArrivalNotification(
    _In_ DMFMODULE DmfModule
    )
{
    SwitchBarReadSwitchesAndUpdateLightBar(DmfModule);
}
```

OSR FX2 DMF Samples

- Sample 3
  - Instantiates the OSR FX2 Module to make a driver that does same as normal WDF driver sample.
  - In interrupt pipe callback, this driver simply completes the IOCTL_OSRUSBFX2_GET_INTERRUPT_MESSAGE request.
- Sample 4
  - Instantiates the OSR FX2 Module but disables enter/exit Idle and IOCTL handling.
  - In interrupt pipe callback, this driver reads the state of the switches and creates PDOs that match the number of switches that are on. The PDOs are visible in Device Manager.
  - This sample replicates “kmdf_enumswitches” sample.

Goal: Make it possible to **directly reuse** (by linking) driver code without using “copy/paste/modify”.

Tell DMF the List of Modules to Use

```
VOID OsrDmfModulesAdd(_In_ WDFDEVICE Device, _In_ PDMFMODULE_INIT DmfModuleInit) {
    DMF_MODULE_ATTRIBUTES moduleAttributes;
    DMF_CONFIG_OsrFx2 moduleConfigOsrFx2;
    DMF_CONFIG_Pdo moduleConfigPdo;
    DMF_CONFIG_QueuedWorkItem moduleConfigQueuedWorkitem;
    DEVICE_CONTEXT* pDevContext = GetDeviceContext(Device);

    DMF_CONFIG_OsrFx2_AND_ATTRIBUTES_INIT(&moduleConfigOsrFx2, &moduleAttributes);
    moduleConfigOsrFx2.InterruptPipeCallback = OsrFx2InterruptPipeCallback;
    moduleConfigOsrFx2.EventWriteCallback = OsrFx2_EventWriteCallback;
    moduleConfigOsrFx2.Settings = (OsrFx2_Settings_NoEnterIdle | OsrFx2_Settings_NoDeviceInterface);
    DMF_DmfModuleAdd(DmfModuleInit, &moduleAttributes, WDF_NO_OBJECT_ATTRIBUTES, &pDevContext->DmfModuleOsrFx2);

    DMF_CONFIG_Pdo_AND_ATTRIBUTES_INIT(&moduleConfigPdo, &moduleAttributes);
    moduleConfigPdo.InstanceIdFormatString = L"SwitchBit=%d";
    DMF_DmfModuleAdd(DmfModuleInit, &moduleAttributes, WDF_NO_OBJECT_ATTRIBUTES, &pDevContext->DmfModulePdo);

    DMF_CONFIG_QueuedWorkItem_AND_ATTRIBUTES_INIT(&moduleConfigQueuedWorkitem, &moduleAttributes);
    moduleConfigQueuedWorkitem.BufferQueueConfig.SourceSettings.BufferCount = 4;
    moduleConfigQueuedWorkitem.BufferQueueConfig.SourceSettings.BufferSize = sizeof(UCHAR);
    moduleConfigQueuedWorkitem.BufferQueueConfig.SourceSettings.PoolType = NonPagedPoolNx;
    moduleConfigQueuedWorkitem.EvtQueuedWorkitemFunction = OsrFx2QueuedWorkitem;
    DMF_DmfModuleAdd(DmfModuleInit, &moduleAttributes, WDF_NO_OBJECT_ATTRIBUTES, &pDevContext-
>DmfModuleQueuedWorkitem);
}
```

Updates PDOs when Switch is Changed 1

```
VOID
OsrFx2InterruptPipeCallback(
    _In_ DMFMODULE DmfModule,
    _In_ UCHAR SwitchState,
    _In_ NTSTATUS NtStatus
    )
{
    WDFDEVICE device;
    DEVICE_CONTEXT* pDevContext

    device = DMF_ParentDeviceGet(DmfModule);
    pDevContext = GetDeviceContext(device);

    if (NT_SUCCESS(NtStatus))
    {
        // Create the PDOs in PASSIVE_LEVEL in a synchronized manner.
        //
        DMF_QueuedWorkItem_Enqueue(pDevContext->DmfModuleQueuedWorkitem,
                                   &SwitchState,
                                   sizeof(UCHAR));
    }
}
```

Updates PDOs when Switch is Changed 2

```
ScheduledTask_Result_Type OsrFx2QueuedWorkitem(_In_ DMFMODULE DmfModule, _In_ VOID* ClientBuffer, _In_ VOID* ClientBufferContext ) {
    DEVICE_CONTEXT* pDevContext;
    WDFDEVICE device;
    UCHAR* switchState;
    NTSTATUS ntStatus;
    WCHAR* hwIds[] = { L"{3030527A-2C4D-4B80-80ED-05B215E23023}\\OSRFX2DMFPDO"};

    switchState = (UCHAR*)ClientBuffer;

    device = DMF_ParentDeviceGet(DmfModule);
    pDevContext = GetDeviceContext(device);

    for (ULONG bitIndex = 1; bitIndex <= 0x80; bitIndex <<= 1)
    {
        if ((*switchState) & bitIndex)
        {
            // The bit is on. Create the PDO.
            //
            ntStatus = DMF_Pdo_DevicePlug(pDevContext->DmfModulePdo, hwIds, 1, NULL, 0, L"OsrFx2DmfPdo", bitIndex, NULL);
        }
        else
        {
            // The bit is off. Destroy the PDO.
            //
            ntStatus = DMF_Pdo_DeviceUnplugUsingSerialNumber(pDevContext->DmfModulePdo, bitIndex);
        }
    }

    return ScheduledTask_WorkResult_Success;
}
```

Compare Device Context (non-DMF)

```
typedef struct _BUTTON_INFO
{
    BYTE         Id;
    GPIO_LEVEL   GPIOLevel;
}BUTTON_INFO, *PBUTTON_INFO;

typedef struct _RING_BUFF{
    PLONG_A32  Round;
    ULONG  Size;
    PBUTTON_INFO pButtonInfo;
    ULONG  RdPtr;
    ULONG  WrPtr;
    WDFWAITLOCK  RBReadLock;
    WDFWAITLOCK  RBWriteLock;
}RING_BUFF, *PRING_BUFF;

typedef struct _INTERRUPT_CONTEXT {
    BYTE            ButtonIndex;
    PDEVICE_CONTEXT DeviceContext;
} INTERRUPT_CONTEXT, *PINTERRUPT_CONTEXT;

typedef struct _WORKITEM_CONTEXT {
    BYTE        ButtonIndex;
    WDFREQUEST  Request;
    UINT8*      auBuff;
    WDFMEMORY   ioctlMemory;
    DWORD       btnMsgcount[GPIO_SLATE_BUTTON_COUNT];

} WORKITEM_CONTEXT, *PWORKITEM_CONTEXT;

typedef struct _WORKITEM_FIRSTSAMPLEOFGPIO_CONTEXT {
    PRING_BUFF                  rbuf;
    WDFWAITLOCK                 WaitLockFirstSample;
    __declspec(align(4)) UINT8  uData1;
}WORKITEM_FIRSTSAMPLEOFGPIO_CONTEXT, *PWORKITEM_FIRSTSAMPLEOFGPIO_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(WORKITEM_FIRSTSAMPLEOFGPIO_CONTEXT,
GetWorkItemFirstSampleOfButtonGPIO);

typedef struct _WORKITEM_TIMERSAMPLEOFGPIO_CONTEXT {
    ULONG                       TimerElapsed[GPIO_SLATE_BUTTON_COUNT];
    WDFWAITLOCK                 WaitLockTimerSample;
    __declspec(align(4)) UINT8  uData2;
}WORKITEM_TIMERSAMPLEOFGPIO_CONTEXT, *PWORKITEM_TIMERSAMPLEOFGPIO_CONTEXT;
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(WORKITEM_TIMERSAMPLEOFGPIO_CONTEXT,
GetWorkItemTimerSampleOfButtonGPIO);
```

```
typedef struct _DEVICE_CONTEXT
{
    WDFDEVICE                       wdsBtnDevice;

    CM_PARTIAL_RESOURCE_DESCRIPTOR  SpbConnection;
    LARGE_INTEGER                   SpbPeripheralId;
    LARGE_INTEGER                   GpioIoPeripheralId[GPIO_SLATE_BUTTON_COUNT];
    WDFIOTARGET                     wdsGpioIoButton[GPIO_SLATE_BUTTON_COUNT];
    WDF_INTERRUPT_POLARITY          InterruptPolarity[GPIO_SLATE_BUTTON_COUNT];
    WDFWORKITEM                     WorkItemInjectButton;
    WDFWAITLOCK                     WorkItemWaitLock;

    WDFWORKITEM                     WorkItemFirstSampleOfButtonGPIO;
    WDFWORKITEM                     WorkItemTimerSampleOfButtonGPIO;

    WDFTIMER                        SampleButtonGpioWdfTimer[GPIO_SLATE_BUTTON_COUNT];

    WDFIOTARGET                     msgpioWin32;
    PVOID                           msgpioDevExtn;
    PFILE_OBJECT                    msgpioFileObject;
    PVOID                           NotificationHandle; // Interface notification handle
    BOOLEAN                         fMsgpioWriteIoAvailable;
    WDFIOTARGET                     IoTargetToOurself;
    UINT8                           buttonState[GPIO_SLATE_BUTTON_COUNT];
    PRING_BUFF                      rb;

    WDFKEY keyObject;
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;
```

Compare Device Context (DMF)

```
typedef struct
{
                UCHAR ButtonOverrideKeys[BUTTON_OVERRIDE_DATA_SIZE];
                ULONG ButtonOverrideKeysLength;
} BUTTON_OVERRIDE;
```

```
typedef struct _DEVICE_CONTEXT
{
        WDFDEVICE WdfDevice;

        WDFSPINLOCK ButtonStateLock;

        ButtonStateType ButtonState[ButtonIdMaximum];

        BOOLEAN ThrowAwayPowerButtonUp;
        ULONG PowerDownAlreadySent;
        ULONG TouchLockEnabled;
        BOOLEAN DebounceLongEnoughForValidButtonPress;
        BOOLEAN WaitingForPowerDoubleClick;
        BOOLEAN DoubleClickDeferredPowerUp;
        WDFTIMER PowerButtonDebounceTimer;
        WDFTIMER PowerDoubleClickTimer;
        ULONG PowerButtonDebounceTimeoutMs;
        ULONG PowerButtonDoubleClickTimeoutMs;
        ULONG SurfacePlatformId;
        ULONG FeatureFlags;
        LONG InConnectedStandby;
        PVOID PowerSettingHandleLowPowerEpoch;
        BUTTON_OVERRIDE ButtonOverride[ButtonIdMaximum];
        BOOLEAN AtLeastOneButtonIsOverriden;

        DMFMODULE DmfModuleNotificationAcpi;
        DMFMODULE DmfModuleButtonFifo;
        DMFMODULE DmfModuleButtonsViaVhf;
        DMFMODULE DmfModuleSlateLaptopInjector;
        DMFMODULE DmfModuleGpioSlConn;
        DMFMODULE DmfModuleGpioButton[NUMBER_OF_GPIO_BUTTONS];
        DMFMODULE DmfModuleNotifierViaAcpi;
        DMFMODULE DmfModuleHid;
        DMFMODULE DmfModuleSamCommunicationViaSsh;
        DMFMODULE DmfModuleSamNotificationViaSshKeypad;
        DMFMODULE DmfModuleQueuedWorkitem;
        DMFMODULE DmfModuleCrashDump;
        DMFMODULE DmfModuleKernelUserEvent;
        DMFMODULE DmfModuleRegistry;
        DMFMODULE DmfModuleVirtualHidKeyboard;
        DMFMODULE DmfModuleBranchTrack;
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;
```

Other Notes about DMF

- DMF supports:
  - Function, Filter, Bus drivers and non-PnP drivers as well as NDIS Class Extensions.
  - Kernel-mode and User-mode drivers.
  - C and C++ drivers.
- DMF Modules always expose a C interface, but internally can use C++. Some of our User-mode drivers use COM.
- DMF developers should **always work with DEBUG build** as that is DMF's "Verifier" which uses ASSERT() heavily.
- Microsoft Surface Team has used DMF for over three years.
- Some of our drivers have as much as 99% code reuse (based on lines of code in Client driver). These drivers consist of 2 or 3 callback functions and the ModulesAdd() function.
- Driver with least amount of code reuse still reuses 69%.

- Remember: The true power of DMF is that it allows you to build your own Modules!

DMF Resources

Links

Github repository : https://github.com/Microsoft/DMF

Blog post is here : http://aka.ms/DMF

Documentation is here : https://github.com/Microsoft/DMF/tree/master/Dmf/Documentation

Email contacts : dmf-feedback@microsoft.com

Component Firmware Update : http://aka.ms/CFU

I will be available throughout WinHEC for any questions, feedback or more in depth discussion.

Demo

Steps:

1. Clone DMF Repository
2. Build DMF
3. Build Samples
4. Overview of Repository to show documentation

Note: There are many ways to build DMF, however in the repository only contains .vcxproj files. Soon, we will add two other ways to build:

- sources file using legacy DDK.
- makefiles (even more legacy)

Part 1 Summary

Goal: Make driver programming **easier**, **faster**, **cheaper** and more **satisfying**.

Goals of DMF

Spend time writing and debugging new code to **solve new problems**!

Call to Action

- Download the latest release of DMF and build it.
- Read the list of available Modules.
- Read the samples.
- Read the documentation.
- Try to use a Module in a driver.
- Try to write a Module you need and use it in a driver.
- Try to write a whole driver using Modules.
- Send us questions and feedback!

Part 2

Section Introduction

Modules and Module Libraries

Section Agenda:

Module Libraries

Modules

Module Callbacks

Steps to Create a Module from scratch

Steps to Create a Library from scratch

Static Modules/Dynamic Modules

Protocol/Transport Feature

What is a Module Library?

A Module Library is a collection of Modules

Usually this collection has a common characteristic. Examples include:

- They are components of a specific feature.
- They are distributed separately to others.

Important: Client drivers link to a Module Library, not individual Modules themselves.

DMF contains a library of Modules

- These Modules expose simple, root data structures that are common to all types of drivers.
- Module authors can instantiate Modules from that Library to create more complex Modules or different versions of those Modules.
- These Modules are used by DMF itself. (Parts of DMF are written using Modules that are automatically loaded as needed by the framework).
  - This allows us to add features to all Modules automatically in the future without Clients having to specifically instantiate those new Modules to access the new features.
- New Module Libraries are usually supersets of the Module Library distributed with DMF.
- Example: Surface Team has a Module Library named, “Modules.Surface”, a superset of “Modules.Library”.

Modules

In this section, we discuss how to write Modules.

- Modules work like Lego blocks
- Regardless of their purpose, Modules have a common interface and a common structure.
- Later, we will see many Modules that perform many different functions. Yet, the interface to all them from creation to usage is the same.

Module Files

Modules are composed of 3 files…

- DMF_[ModuleName].h
- DMF_[ModuleName].c
- DMF_[ModuleName].md

…and an optional public header file

- DMF_[ModuleName]_Public.h

Module Files

The Module DMF_[ModuleName].h File

This file contains **only** definitions/declarations that Clients use to interact with the Module:

- Enumerations and Structure/Type definitions referenced by CONFIG.
- Client callback definitions referenced by CONFIG or Methods.
- Module CONFIG.
- Declaration of Module Macro that automatically defines functions Clients use to instantiate the Module.
- Method Prototypes used by Clients.

Important: **Clients do not include this file directly. Instead, Clients include the Library's include file which includes this file and its dependencies.**

Module Files

The Module DMF_[ModuleName].c File

This file contains the implementation of the Module (the Module's code):

- This file has several sections that are always in the same order and have specific types of code.
- All sections must be present even if there is no code in that section.
- (To simplify, information about each section is now listed on a separate slide.)

Important: **Clients do not link to this file directly. Instead, Clients link to the Library's .lib file which includes this file and its dependencies.**

Module Files

The Module DMF_[ModuleName].c File Sections (1):

File header:

- Summarizes the Module.

```
/*++

    Copyright (c) Microsoft Corporation. All rights reserved.
    Licensed under the MIT license.

Module Name:

    Dmf_ContinuousRequestTarget.c

Abstract:

    Creates a stream of asynchronous requests to a specific IO Target. Also, there is support
    for sending synchronous requests to the same IO Target.

Environment:

    Kernel-mode Driver Framework
    User-mode Driver Framework

--*/
```

Module Files

The Module DMF_[ModuleName].c File Sections (2):

DMF Includes:

- DMF Core APIs, specific to Modules.
- Library include file
- WPP tracing definitions

```
// DMF and this Module's Library specific definitions.
//
#include "DmfModule.h"
#include "DmfModules.Library.h"
#include "DmfModules.Library.Trace.h"
#include "Dmf_ContinuousRequestTarget.tmh"
```

Module Files

The Module DMF_[ModuleName].c File Sections (3):

Definitions and include files used by Module's (Private) Context:

```
////////////////////////////////////////////////////////////////////////////////
// Module Private Enumerations and Structures
////////////////////////////////////////////////////////////////////////////////
//
```

Module Files

The Module DMF_[ModuleName].c File Sections (4):

Module's Context and Macros:

- Module Context (similar to a WDF Driver's Device Context)
  - Private to this Module
  - Contains data structures used by the Module.
  - Contains handles to Child Modules as needed.
  - Must be locked in cases where multiple asynchronous calls are possible.
- Module Context and CONFIG macros.
- Module's Memory Tag.

```
//////////////////////////////////////////////////////////////////////////////
// Module Private Context
//////////////////////////////////////////////////////////////////////////////
//
```

Module Files

The Module DMF_[ModuleName].c File Sections (5):

Module Support Code

- Contains any include files not already included by DMF that are needed by this Module.
- Contains all the Module's private Code. These are all static functions.

```
//////////////////////////////////////////////////////////////////////////////
// DMF Module Support Code
//////////////////////////////////////////////////////////////////////////////
//
```

Module Files

The Module DMF_[ModuleName].c File Sections (6):

Module's WDF Callbacks

- Contains all the WDF callbacks the Module supports.
- These are called automatically, as needed, by DMF.
- Note: Function signatures are similar, but not identical, to the corresponding WDF callbacks because the first parameter is DMFMODULE.

```
//////////////////////////////////////////////////////////////////////////////
// WDF Module Callbacks
//////////////////////////////////////////////////////////////////////////////
//
```

Module Files

The Module DMF_[ModuleName].c File Sections (7):

Module's DMF Callbacks

- Contains all the DMF callbacks the Module supports.
- These are called automatically, as needed, by DMF.
- Separate slides will discuss each of them.

```
////////////////////////////////////////////////////////////////////////////////
// DMF Module Callbacks
////////////////////////////////////////////////////////////////////////////////
//
```

Module Files

The Module DMF_[ModuleName].c File Sections (8):

Module's Descriptor declarations.

- These are global buffers for descriptors are used by the Module.
- They are initialized and used by Module's Create callback as well as Module Methods.

```
////////////////////////////////////////////////////////////////////////////////
// DMF Module Descriptor
////////////////////////////////////////////////////////////////////////////////
//
static DMF_MODULE_DESCRIPTOR DmfModuleDescriptor_ContinuousRequestTarget;
static DMF_CALLBACKS_WDF DmfCallbacksWdf_ContinuousRequestTarget;
static DMF_CALLBACKS_DMF DmfCallbacksDmf_ContinuousRequestTarget;
```

Module Files

The Module DMF_[ModuleName].c File Sections (9):

Module's publicly accessible functions:

- Contains Module's Create function. (Discussed in a separate slide.)
- Contains Module's Methods. (These have declarations in the Module's .h file).

```
//////////////////////////////////////////////////////////////////////////////
// Public Calls by Client
//////////////////////////////////////////////////////////////////////////////
//
```

Module Files

The Module DMF_[ModuleName]_Public.h File

Contains definitions that User-mode applications and/or other drivers use to communicate with the Module

Examples:

- Device Interface GUID
- IOCTL definitions
- Enumerations and Structures used in IOCTL calls

Important: **It must be possible to compile this file using only the Win32 SDK or DDK. It must not contain any references to DMF or DMFMODULE or any DMF construct.**

Module Files

The Module DMF_[ModuleName].md File

Contains detailed documentation for the Module.

- This file has a specific format and is divided into sections similar to a Module's .c file.
- Use this file to understand the purpose of the Module and how to instantiate it and use its Methods.
- This file encourages the Module's programmer to write documentation. That documentation never needs to be written again by a user of the Module.

DMF Module Callbacks

There are 3 kinds of callbacks

1. DMF/WDF callbacks
   - Called by DMF either for Module's lifetime management or because WDF has called into the driver.

2. Direct Callbacks to Module's Parent
   - Called directly by the underlying Modules.

3. Indirect Callbacks from Child Modules (*)
   - Called from Child Module that is not an immediate child. (Grand-child).

DMF Module Callbacks

Module Callbacks called by DMF.

- All DMF/WDF callbacks receive DMFMODULE (this Module's handle).
- WDF Callbacks also receive additional parameters passed by WDF.
- Use DMF_CONFIG_GET(DmfModule) to retrieve the Module's Config data.
- Use DMF_CONTEXT_GET(DmfModule) to retrieve the Module's Context.
- Use DMF_ModuleLock(DmfModule) and DMF_ModuleUnlock(DmfModule) to lock and unlock the Module's Context when necessary.

DMF Module Callbacks

Module Callbacks called by Modules (Direct)

- These callbacks receive the **caller's** DMFMODULE when called by a Module.
  (Similar to how WDFTIMER's Timer callback works.)
- Use DMF_ParentModuleGet(DmfModule) to get its Parent Module or DMF_ParentDeviceGet(DmfModule) to get its Parent Device.
- If callback is part of a Parent Module,
  - use DMF_CONFIG_GET(ParentDmfModule) to retrieve the passed DMFMODULE's Config data.
  - use DMF_CONTEXT_GET(ParentDmfModule) to retrieve the passed DMFMODULE's Context.
  - Use DMF_ModuleLock(ParentDmfModule) and DMF_ModuleUnlock(ParentDmfModule) to lock and unlock the Module's Context when necessary.

DMF Module Callbacks

Module Callbacks called by Modules (Indirect) *

- These callbacks receive a WDFDEVICE when called by a Module as wells as an VOID* context.
- If Callback is part of a Parent Module,
  - Use ParentModule = DMF_VOIDTOHANDLE(Context) to get its Parent Module.
  - Then, use DMF_CONFIG_GET(ParentDmfModule) to retrieve the passed DMFMODULE's Config data.
  - Also, use DMF_CONTEXT_GET(ParentDmfModule) to retrieve the passed DMFMODULE's Context.
  - Use DMF_ModuleLock(ParentDmfModule) and DMF_ModuleUnlock(ParentDmfModule) to lock and unlock the Module's Context when necessary.

DMF Module Callbacks

These callbacks are DMF specific

- DMF_[ModuleName]_Create()
- DMF_[ModuleName]_ChildModulesAdd()
- DMF_[ModuleName]_ResourcesAssign()
- DMF_[ModuleName]_Open()
- DMF_[ModuleName]_Close()
- DMF_[ModuleName]_NotificationRegister()
- DMF_[ModuleName]_NotificationUnregister()

DMF Module Callbacks

DMF_[ModuleName]_Create()

DMF calls this callback when it needs to create an instance of a Module.

- Definition of Create: *Create the structures that DMF uses to manage the lifetime and dispatching of messages to the Module.*
- Module author must populate several descriptors:
  1. WDF callbacks descriptor which tell DMF what WDF callbacks are supported.
  2. DMF callbacks descriptor which tell DMF what DDF callbacks are supported.
  3. Descriptor which defines the Module's Context and Module's Config and Module's synchronization mode.
- Call DMF_ModuleCreate() using the above descriptors.

DMF Module Callbacks

DMF_[ModuleName]_ChildModulesAdd()

DMF calls this callback when wants the list of a Parent Module's Child Modules.

- The Module author performs two tasks:
  1. Declare an instance of DMF_MODULE_ATTRIBUTES.
  2. Initializes the unique Config structure and calls DMF_DmfModuleAdd() for every Module that will be a Child Module of the Parent Module. All these Configs are added to a list.
- The Modules are not created here. After this callback returns, DMF will call the DMF_[ModuleName]_Create() function of every Module in that list.
- In addition, DMF places the Modules in a tree structure for lifetime management and dispatching of WDF callbacks.
- There is no limit to the number of Child Modules, nor the depth of the tree except for system memory and stack.

DMF Module Callbacks

DMF_[ModuleName]_ResourcesAssign()

DMF calls this callback so that the Module can find and configure its resources.

- In this callback, Modules loop through the available resources and extract configuration information.
- That information is used later to configure constructs such as GPIO lines and interrupts.
- NOTE: DMF_GpioTarget and DMF_InterruptResource do this automatically for GPIO and Interrupts. Instead of doing that work for those resources, it is better to simply instantiate those Modules.
- For example, one might use this callback to get information about the PCI BAR.

DMF Module Callbacks

DMF_[ModuleName]_Open()

DMF calls this callback to allow the Module to initialize its Module Context (which has been automatically allocated).

- Here the Module Author may perform tasks such as:
  - Read parameters from the Module's Config.
  - Allocate memory.
  - Allocate WDF handles to WDF objects such as WDFTIMER.
  - Create C++ objects and/or other non-C++ structures.
  - Initialize flags and the above structures for further use.

Important: **DMF guarantees that its Child Modules have been opened when this callback executes.**

DMF Module Callbacks

DMF_[ModuleName]_Close()

DMF calls this callback to allow the Module to perform the inverse of what it did in DMF_[ModuleName]_Open().

- Free all resources allocated in DMF_[ModuleName]_Open().
- Important: **DMF guarantees that the Parent Module has already been closed, but not its Child Modules, when this callback executes.**

DMF Module Callbacks

DMF_[ModuleName]_NotificationRegister()

DMF calls this callback to allow the Module to register for an asynchronous event that must happen before the Module is ready to “open”.

- Often, this callback calls IoRegisterPlugPlayNotification().
- But there are other uses and ways in which notifications can arrive.
- Eventually, usually from the notification callback passed to IoRegisterPlugPlayNotification(), DMF_ModuleOpen() is called.

DMF Module Callbacks

DMF_[ModuleName]_NotificationUnregister()

DMF calls this callback to allow the Module to do the inverse of any actions done in DMF_[ModuleName]_NotificationRegister().

- For example, this callback often calls IoUnregisterPlugPlayNotification().
- Any resources or handles allocated in DMF_[ModuleName]_NotificationRegister() must be released.

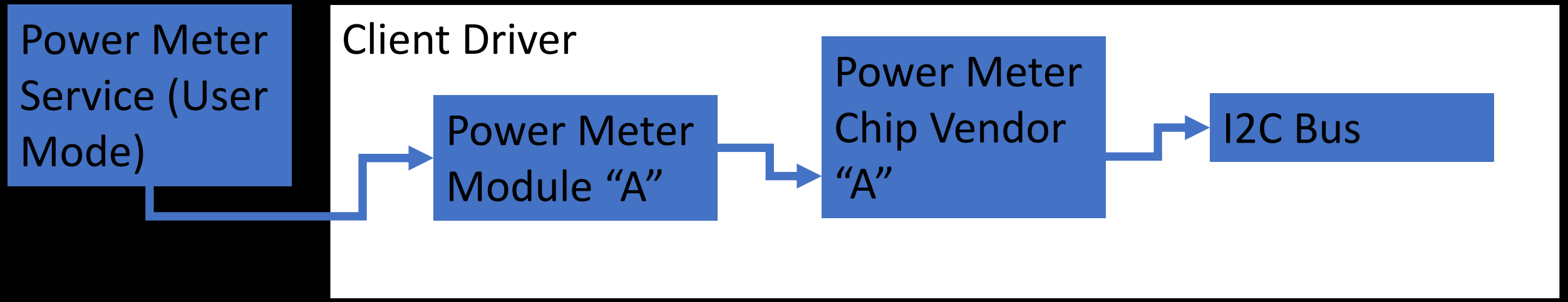
Protocol-Transport Modules

Allows selection of Child and Number of Children at Runtime

- Ordinary Modules choose their Child Modules at compile time.
- Choosing Modules (and the number of Modules) at run-time provides the ability perform even more code modularization and reusability.
- Consider two scenarios in which this mode is helpful:
  1. Power Meter driver.
  2. Load balancing driver.

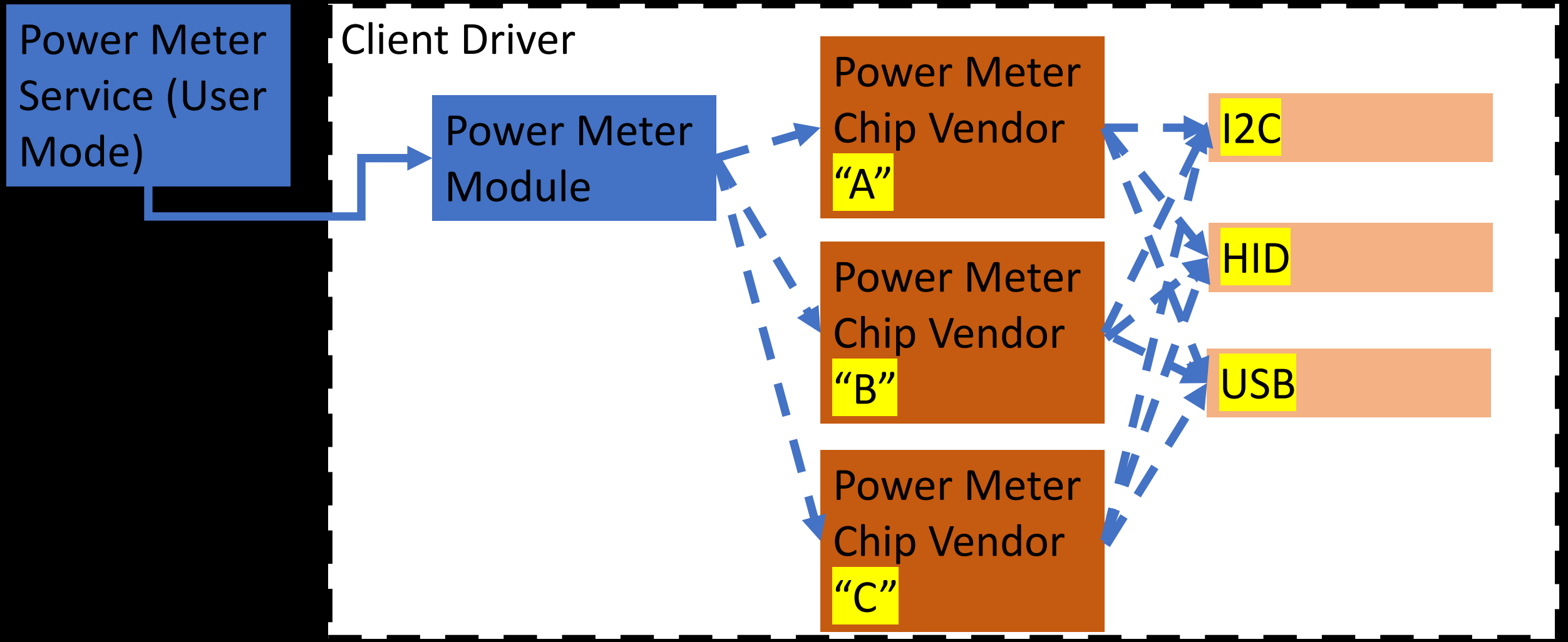
Protocol-Transport Modules

Power Meter driver using non-Protocol-Transport Modules



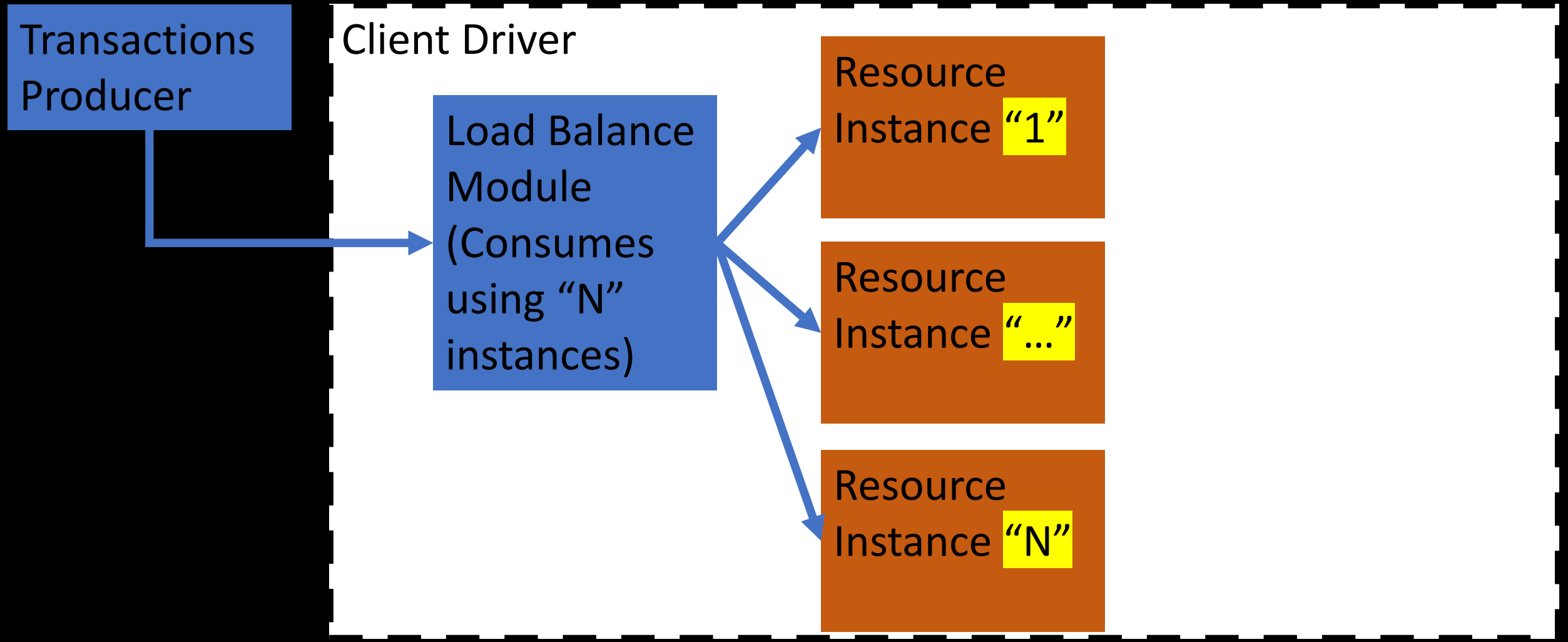
Protocol-Transport Modules

Power Meter driver using Protocol-Transport Modules



Protocol-Transport Modules

Load Balance driver using Protocol-Transport Modules



Steps to Create A Module

1. Choose either the DMF_Template Module or an existing Module that is similar to the new Module that is to be created. (This is the Reference Module for purposes of this slide.)
2. Open all the files associated with the Reference Module: .c, .h., _Public.h, .md. (Note: These should be the only open files, since step 4 does search and replace all open files)
3. Select each of the open files and “Save as…” the name of the New Module in the directory of the target Module Library.
4. Search and replace the Reference Module name with the name of the new Module.
5. Include the target Module’s Library’s .h file at the top of the new Module’s .c file (it is probably set that way already).
6. Add the new Module’s .h file to the list of Modules in the Module Library’s .h file.
7. Add the new Module’s files (.c, .h, _Public.h and .md file) to the Module Library’s project file.
8. (At this point, intellisense should work and it is possible to successfully compile the Module Library with the new Module.)
9. Modify the Module’s “Support Code” as needed.
10. Add/delete WDF/DMF callbacks as needed (including adding its necessary Child Modules).

Steps to Create A Module Library

1. Copy the Solution\DmfKModules.Template directory as the name of the new Library (e.g., DmfKModules.NewLibrary).
2. Switch to the new directory.
3. Rename DmfKModules.Template.vcxproj as DmfKModules.NewLibrary.vcxproj.
4. Rename DmfKModules.Template.vcxproj as DmfKModules.NewLibrary.vcxproj.filters.
5. Copy the Dmf\Modules.Template directory as the name of the new Library (e.g., Modules.NewLibrary).
6. Delete all the “Dmf_*.*” files. Keep the DmfModules.Template.* files.
7. Rename the DmfModules.Template.* files as DmfModules.NewLibrary.*.
8. Do steps 1 to 4 but use “DmfU” instead of “DmfK” (unless you don’t need a User Mode version of the Library).
9. Open the DMF Solution in Visual Studio.
10. Add the DmfKModules.NewLibrary.vcxproj and DmfKModules.NewLibrary.vcxproj.filters to the solution. Do the same for the “DmfU…” files.
11. Remove all the files from the projects.
12. Add the DmfModules.NewLibrary.h, Modules.NewLibrary.Public.h and DmfModules.NewLibrary.Trace.h files to both projects.
13. Open DmfModules.NewLibrary.h and delete the names of all the Module .h files referenced in that file.
14. Do the same for DmfModules.NewLIbrary.Public.h.
15. Update the include and link paths to indicate the location the new Library.
16. Now it is possible to add new Modules to this Library and successfully compile the new Library.

Demo

Steps:

1. Create a new Module Library
2. Create a new Module

Part 2 Summary

Sharing Driver Code is Simple in DMF because many mundane technical questions are already answered for you! All you need to do is write the code that is to be shared and debug it. The following questions already have answers:

1. What directory does the new shared code and header files go?
2. **How should the new shared code be instantiated and used by a Client?**
3. How should the different parts of a new shared code be named?
4. Where is the new shared code's private data and private methods be defined?
5. Where is the new shared code's public data and public methods be defined?
6. Where do common Kernel and User-mode definitions go?
7. Where does the new shared code's documentation go? What is the format of that documentation?
8. How to ensure that the new shared code is compatible with external users?
9. How to make a shared library of shared code?

Part 3

Section Introduction

Review of some Modules distributed with DMF

Section Agenda:

DMF_BufferPool

DMF_BufferQueue

DMF_QueuedWorkitem

DMF_Thread

DMF_ThreadedBufferQueue

DMF_RequestTarget

DMF_ContinuousRequestTarget

DMF_IoctlHandler

DMF_BufferPool

A simple list of pre-allocated buffers of a certain size

- Number of buffers can be finite or from a lookaside list.
- Each buffer can have its own context.
- Module performs bounds checking to check for buffer overrun when buffers are used in Methods. (Debug build only.)
- Each buffer has an optional timer which can be used for tasks such as stale data detection or hardware operation timeout.
- Makes it easy to uses lists of buffers instead of other simpler but less optimal data structures.

DMF_BufferQueue

Producer-Consumer Buffer Lists

1. Composed of two instances of DMF_BufferPool.
2. Producer: It is a list (finite or infinite) of pre-allocated buffers.
3. Consumer: It is an empty list.
4. When Client has work to do or some data that needs to be held for a while, the Client “fetches” a buffer from DMF_BufferQueue. (Fetch = Get from Producer list.)
5. Client writes to the buffer.
6. Client “enqueues” the buffer in DMF_BufferQueue. (Enqueue = put in Consumer list.)
7. Later, Client “dequeues” the buffer. (Dequeue = get from Consumer list.)
8. Client reads data from that buffer and uses it.
9. Client then “reuses” the buffer. (Reuses = put buffer back into Producer list.)
10. Because these are DMF_BufferPool buffers, bounds checking comes for free.

DMF_QueuedWorkitem

WDFWORKITEM with call specific context

1. Allows caller to enqueue a workitem with a call (enqueue) specific context.
2. WDFWORKITEM has a single context attached to the WDFOBJECT.
3. But, DMF_QueuedWorkitem's callback function passes an additional (optional) context that is passed when the workitem is enqueued.
4. The Module maintains a count of how many pending enqueues remain in case the workitem is enqueued while it is already enqueued.
5. Thus, the workitem's callback will execute exactly the number of times the enqueue Method is called. Thus, the Client's callback function need not worry about knowing how much work needs to be done.
6. This Module uses DMF_BufferQueue.

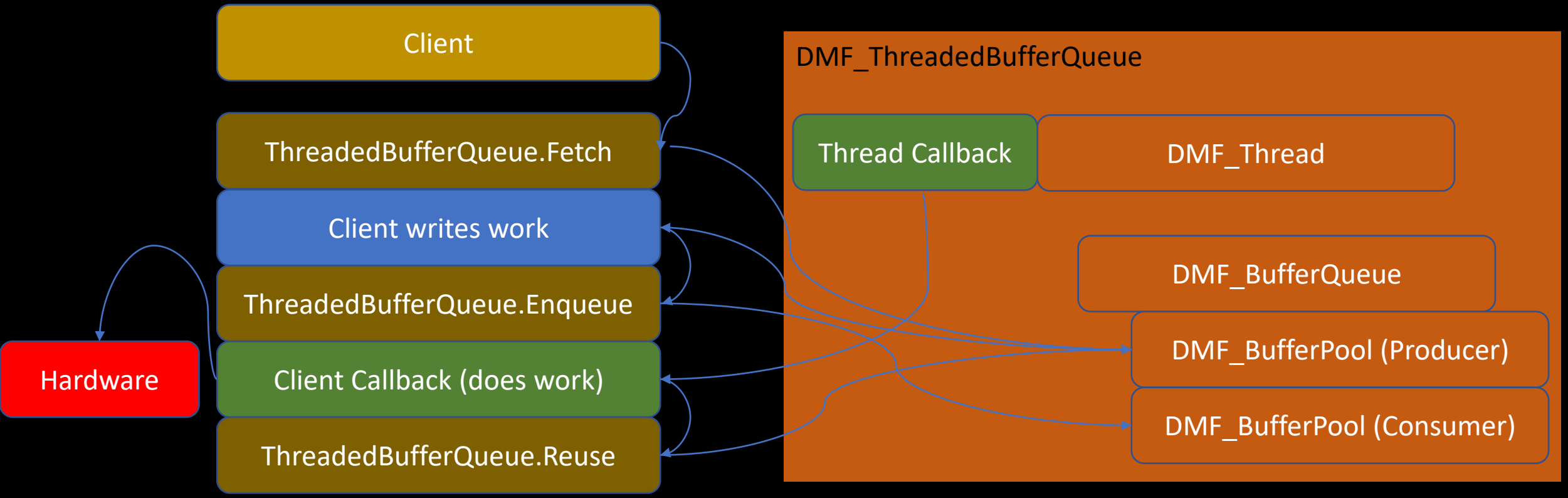
DMF_Thread

Creates a thread and manages its operation

1. This Module creates a thread, a “thread-stop” event and a “work-ready” event.
2. It contains a loop that waits on the two events.
3. When the “work-ready” event is set, a Client callback is called where the Client can perform work.
4. When the “thread-stop” event is set the thread loop ends and the thread stops executing. Module, of course, waits for thread to end properly.

DMF_ThreadedBufferQueue

Allows multiple asynchronous callers to enqueue work that is performed synchronously.



DMF_RequestTarget

Allows Client to send/receive data to underlying WDFIOTARGET by just sending a buffer instead of creating a WDFREQUEST.

- Synchronous/Asynchronous calls are supported.
- IOCTL/Read/Write calls are supported.
- Target WDFIOTARGET is set by the Client.
- This Module is used by other high-level Modules that expose the same functionality for their Clients.

DMF_ContinuousRequestTarget

Allows Client to automatically create, send and receive multiple buffers to a WDFIOTARGET.

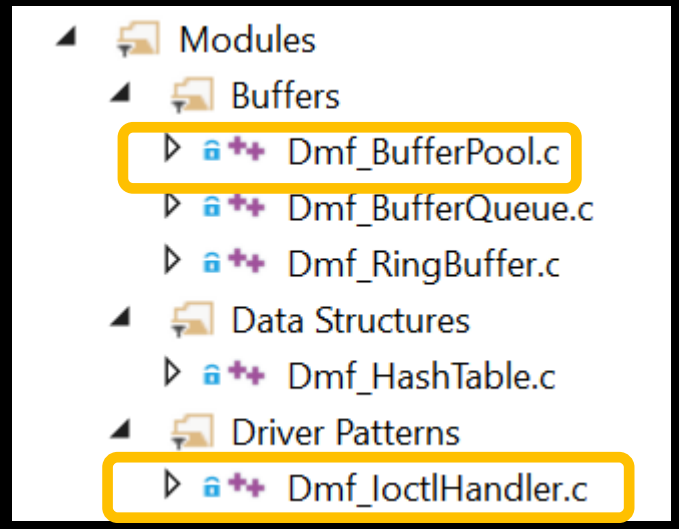
1. This Module is similar to WdfUsbContinuousReader except that this Module works for any WDFIOTARGET.
2. Client specifies number of buffers and size/type of each buffer. Module automatically allocates memory and WDFREQUESTS and sends them to the WDFIOTARGET specified by the Client.
3. Client specifies Input Buffer Callback so that the Client can populate buffers prior to being sent.
4. Client specifies Output Buffer Callback so that the Client can extract data from buffers after the WDFIOTARGET completes them. Afterward, Module automatically sends them back to the WDFIOTARGET.
5. The Module implements proper rundown logic when it is time to stop sending buffers.
6. This Module is also used by high-level Modules that expose this functionality for their Clients.

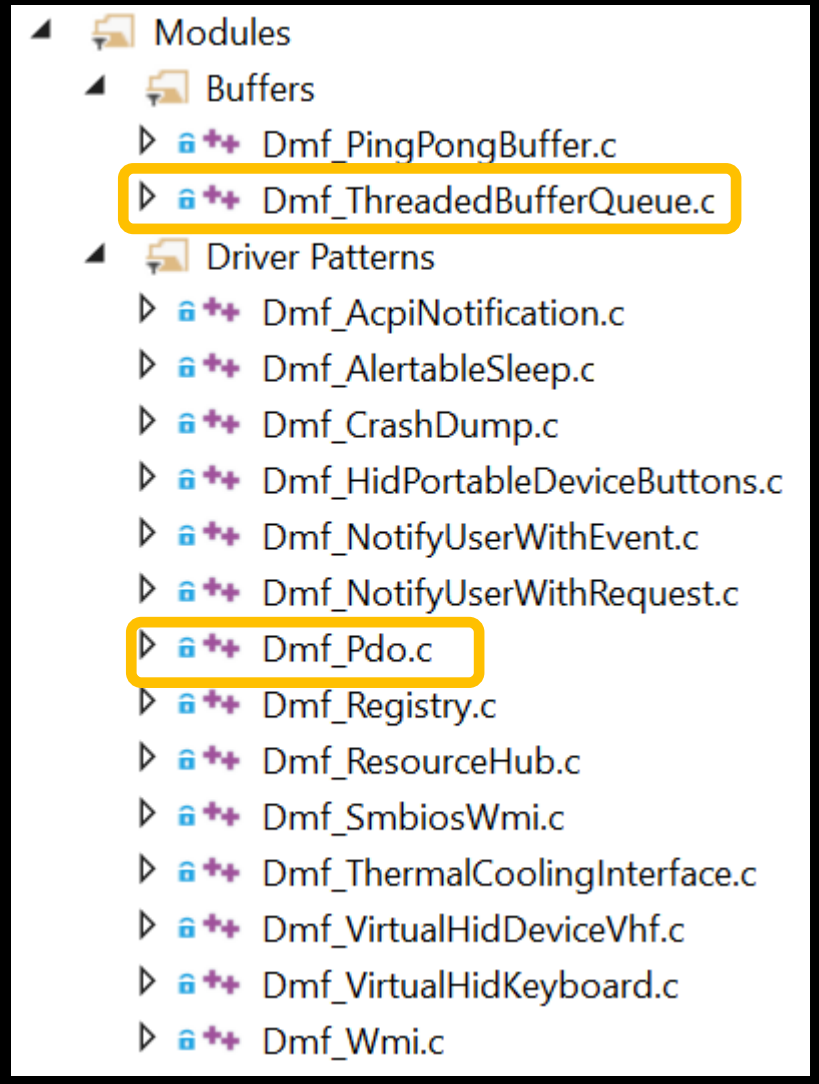
DMF_IoctlHandler

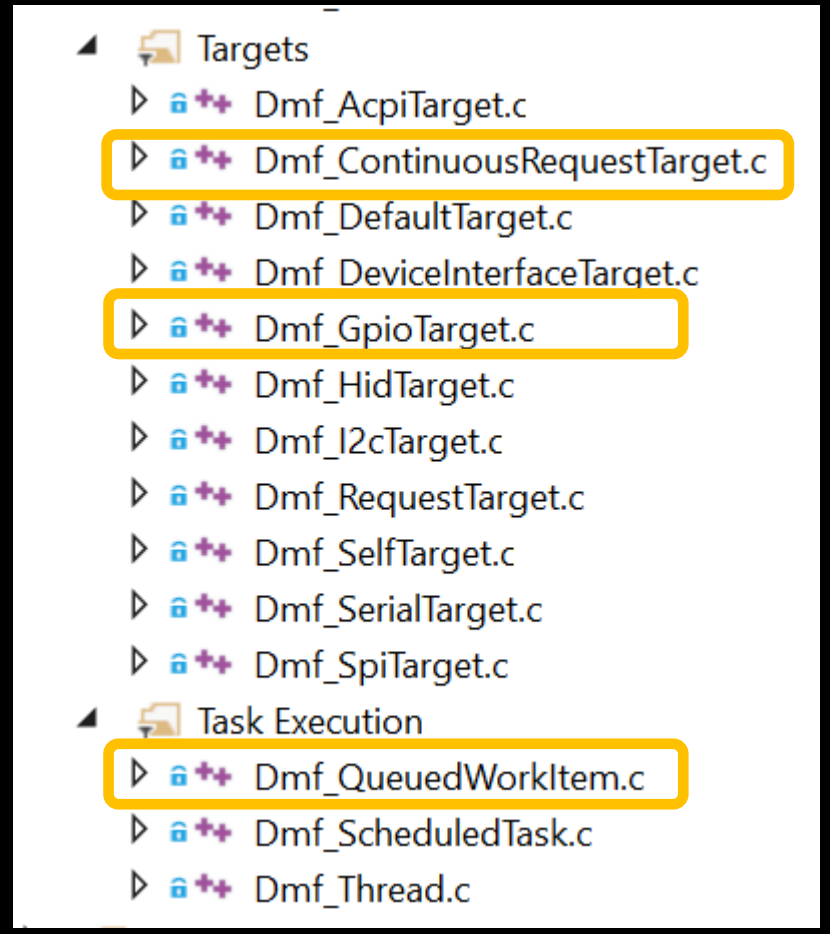
Given a table of each supported IOCTL along with the appropriate minimum input/output buffer sizes, this Module automatically validates IOCTLs that arrive.

1. There is an option to also validate if the caller is running “As Administrator”.
2. Also, there is an option for Client control of access based on file handle.
3. After the Module validates the IOCTL, its input/output buffers are extracted and presented to the Client in a callback function.
4. Client just needs to use the buffers as needed.
5. Client returns STATUS_PENDING to retain the buffers (and WDFREQUEST). Otherwise, the Module will automatically complete the WDFREQUEST on behalf of the Client using the NTSTATUS the Client returns from the callback.

Modules in Library







Goal: Make it easier for driver writers to think using **high-level constructs**.

Part 3 Summary

This section has shown how small, simple Modules are used to build bigger, more complex Modules. In turn, those Modules can be used to build even bigger, more complex Modules, and so on and so on…

1. One can see how each Module is its own self-contained layer. It only worries about its own tasks and code.
2. As stated before, Modules are like Lego pieces.
3. It is up to you and your imagination how to combine Modules.
4. It is up to you and your imagination what Module you write.
5. A Module can be an entire driver. Even that Module can become a Child of another Module.
6. If you do not like how a Module is implemented, you can write your own version.
7. If you need to add a Method to a Module, you can do so.

Part 4

Section Introduction

Review of Surface Modules and drivers that use DMF

Section Agenda:

Surface Library Modules Overview

SurfaceButton

SurfaceAcpiNotify

SurfaceIntegrationDriver

SurfaceSarManager

Part 4 Summary

Surface Team uses DMF for all new drivers…

1. We have written 95 Modules, 39 of which are publicly distributed with DMF.
2. Our Modules and drivers that use them work in various environments.
3. User-mode Modules even use COM and Lambdas as well as very high-level APIs.
4. Sound engineering practices coupled with tools, guard against regressions and build breaks due to code sharing.
5. DMF is the go to framework for all new Driver development tasks.

   We believe the same will happen in your teams. ☺
6. Regardless, we continue to improve DMF based on feedback as well as our usage needs.

Plans for going forward

DMF is Open Source

- Want to be a DMF Contributor? Simply create a pull request on Github.
- Found an issue in DMF? Open a bug in the Git repository's bug database.

- We have been updating the code base frequently and will continue to do so.
- We are adding more samples, especially more complex samples.
- We are adding more Modules to the Library.

DMF is not officially supported by Microsoft as a product. DMF is not supported by Customer Support Services.

Thank you!

Thank you for hosting us and listening to us!

Happy Coding. Looking forward to your contributions to DMF.

If you have questions, please contact us: dmf-feedback@microsoft.com

We are making improvements rapidly based on your valuable feedback.