

# KMDF

*How to develop framework drivers*

Kumar Rajeev  
SDET  
Microsoft Corporation

# KMDF And HID Minidriver

KMDF does not support HID minidrivers natively due to conflicting KMDF and HID architecture requirements

HID architecture requires that HIDclass driver own the driver dispatch table, while KMDF requires that it own the dispatch table of the minidriver

Solution is to use a driver stack that consists of a minimum WDM pass-through driver and a complete KMDF driver

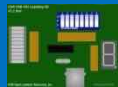
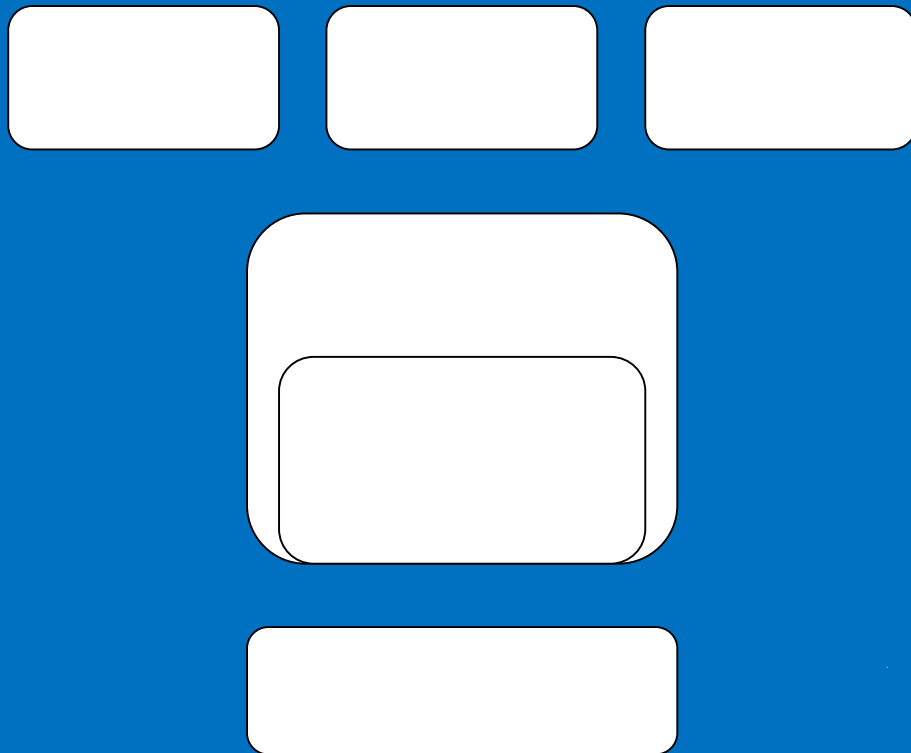
Pass-through driver registers with HIDclass as HID minidriver and forwards all requests to KMDF driver

KMDF driver processes all the requests

The sample HIDUSBFX2 demonstrates this solution

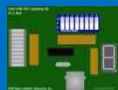
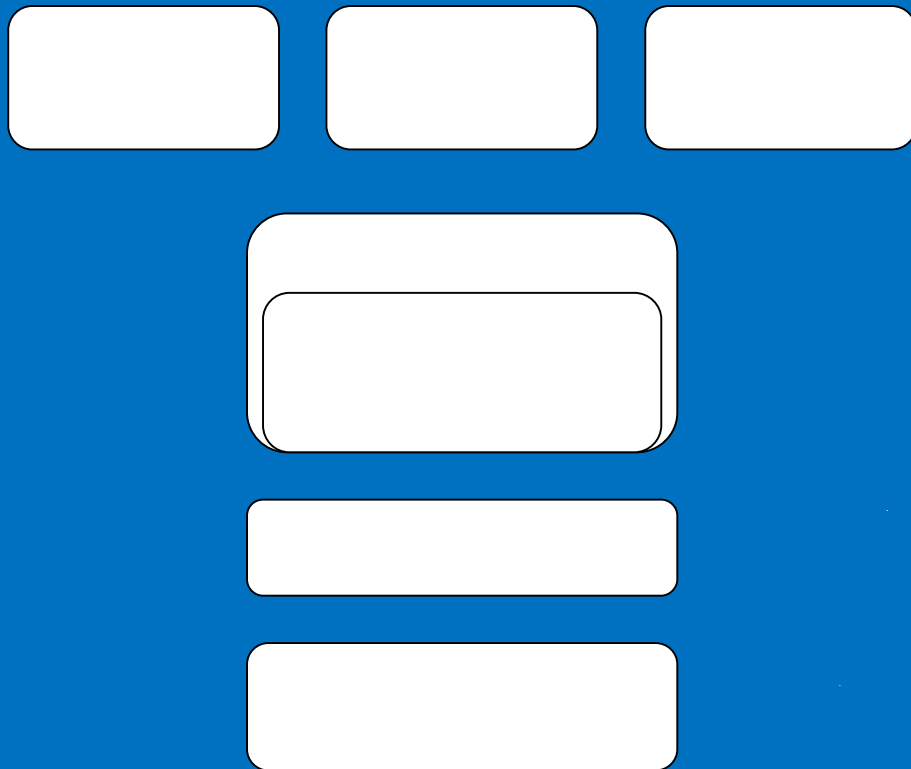
# Driver Stack

## WDM



# Driver Stack

## KMDF



# Purpose Of The Sample

Encourage use of KMDF for writing custom HID minidrivers

Demonstrate KMDF features suitable for HID minidrivers

Some of the reasons one may end up writing HID minidriver are

- Easier to provide complex HID logic in s/w than in firmware

- Making a change in s/w may be cheaper than in firmware when device is already in market

- Need to make an existing non-HID USB device appear as HID device w/o updating firmware

- When there is no inbox support for the device

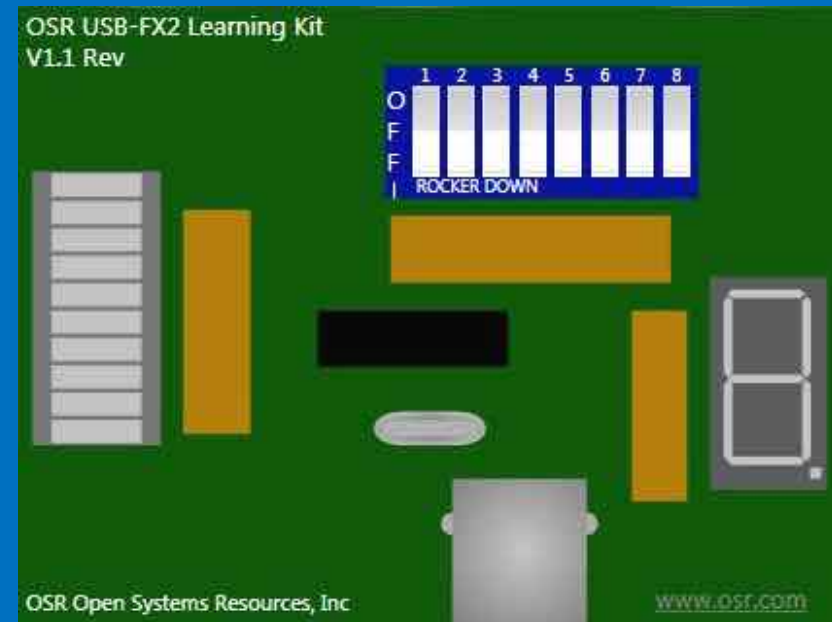
- When sideband communication with minidriver is needed, since HIDclass driver doesn't allow user IOCTLs/WMI. With KMDF you can easily enumerate PDOs and use them for sideband communication

# Sample HIDUSBFX2

For OSR USB-FX2 device  
(non-HID device)

Maps USB-FX2 device's  
switch pack to HID  
controls for keyboard  
hot-keys: Browser, Mail,  
Sleep etc

Maps 7-segment display  
and bar graph display as  
HID features

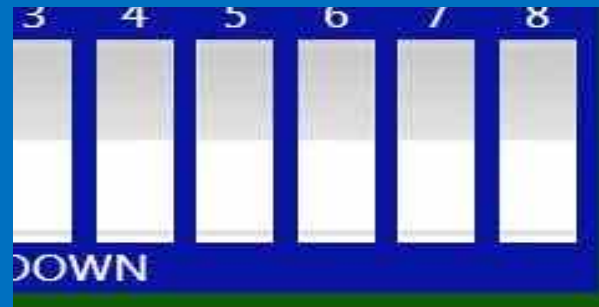


# Switch Pack Mapping

Mapped as HID “Input Report”

Lower 7 bits represent usages from Consumer Control Collection for keyboard hot-keys

One highest bit is mapped to “Sleep” usage in System Control Collection



# 7-Segment Display

Mapped as HID “Feature”

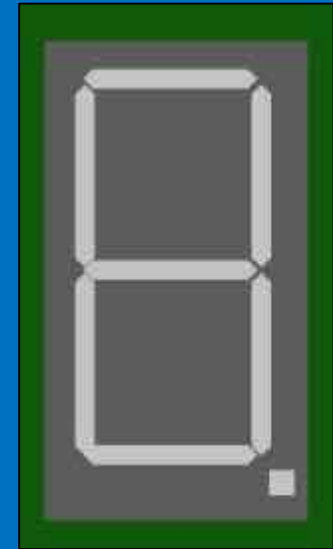
HID clients can send SetFeature request to display numbers 1 thru 8 on the segment display

Vendor Defined Usage Page (0xff00)

Vendor Defined Usages

Usage 0x1 thru 0x8

Each usage corresponds to numbers 1 through 8 displayed on segment display. For example, sending usage 0x7 causes the display to show number 7





# Bar Graph Display

Mapped as HID “Feature”

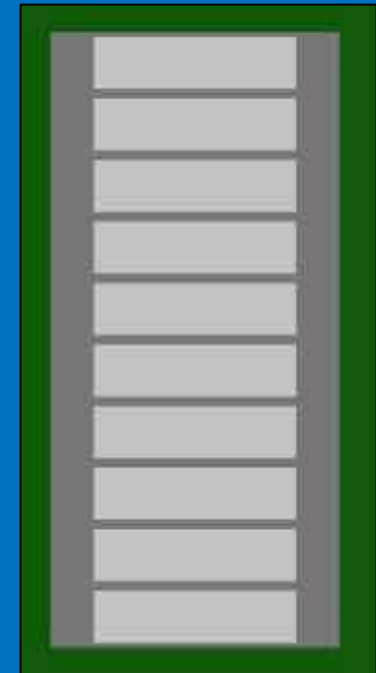
HID clients can send SetFeature request to light up LEDs on the bar display

Vendor Defined Usage Page (0xff00)

Vendor Defined Usages

Usage 9 through 18

E.g. Sending usage 9 causes the display to turn on LED 1



# Sample Details

Sample has a default parallel queue and a manual queue

HID data is generally provided by USB interrupt endpoints. KMDF provides “Continuous Reader” mechanism to read such data

Minidriver relinquishes power policy ownership since HIDclass driver owns power policy

# Request Cancellation In A Framework Driver

Eliyas Yakub  
Development Lead  
Microsoft Corporation

# *Request Cancellation*

Long term request should be held in a cancelable state for better user experience

Difficult to get it right in WDM

Framework provides following options to deal with the complexity

- Manual queues to hold request

- Polling for canceled state of request

- Using cancel-routine

# Using Queues

Request waiting for hardware event should be placed in a manual queue

When the request is canceled, framework will complete the request on driver's behalf

If you want to be notified before the request is completed, you can register for

`EvtIoCanceledOnQueue` callback on Queue

Can defer completion if the request is consumed by the hardware

Callback is subject to synchronization scope and execution level of the queue

`EvtContextCleanup` callback on Request object

# Sample Code

## Smartcard

```
NTSTATUS CBCardTracking(PSMARTCARD_EXTENSION SmartcardExtension)
{
    request = GET_WDFREQUEST_FROM_IRP(
        SmartcardExtension->OsData->NotificationIrp);

    status = WdfRequestForwardToIoQueue(request,
        DeviceExtension->NotificationQueue);
    if (!NT_SUCCESS(status)) {
        InterlockedExchangePointer(
            &(SmartcardExtension->OsData->NotificationIrp), NULL);
        WdfRequestComplete(request, status);
    }
    return status;
}

VOID PscrEvtIoCanceledOnQueue(WDFQUEUE Queue, WDFREQUEST Request)
{
    InterlockedExchangePointer(
        &(SmartcardExtension->OsData->NotificationIrp), NULL);
    WdfRequestComplete(Request, STATUS_CANCELLED);
}
```

# Polling For Cancellation

Framework provides `WdfRequestIsCanceled` to check the state of the IRP

Useful if you are staging single I/O into multiple transactions or actively pooling the hardware before initiating the I/O

Check the canceled state before initiating the next transfer

# Using CancelRoutine

Use this approach when you cannot keep long-term requests in queue

This is by far the most difficult approach

Complexity level of this approach is equivalent to the WDM model

Call `WdfRequestMarkCancelable` to set `EvtRequestCancel` on a request

Call `WdfRequestUnMarkCancelable` to clear the cancel routine before completing the request



# Using CancelRoutine (2)

Request must be a valid uncompleted request when you call `WdfRequestUnmarkCancelable`

That means you have to manage the race between cancel routine and another asynchronous routine that tries to complete the request

# Using CancelRoutine (3)

Framework enables you to manage the complexity

By using framework provided synchronization scope

By tracking state in the request context area using your own lock

# Using Synchronization

```
NTSTATUS EVtDeviceAdd() {
    attributes.SynchronizationScope = wdfsynchronizationScopeDevice;
    status = wdfDeviceCreate(&DeviceInit, &attributes, &device);
    ...
}
VOID EchoEvtIoRead(Queue, Request, Length) {
    wdfRequestMarkCancelable(Request, EchoEvtRequestCancel);
    queueContext->CurrentRequest = Request;
}
VOID EchoEvtRequestCancel(Request) {
    wdfRequestComplete(Request, STATUS_CANCELLED);
    queueContext->CurrentRequest = NULL;
}
VOID EchoEvtTimerFunc(WDFTIMER Timer) {
    Request = queueContext->CurrentRequest;
    if( Request != NULL ) {
        status = wdfRequestUnmarkCancelable(Request);
        if(status != STATUS_CANCELLED) {
            queueContext->CurrentRequest = NULL;
            wdfRequestComplete(Request, status);
        }
    }
}
```

# Using Driver Lock

Track the cancel state in the context area of the request

```
typedef struct _REQUEST_CONTEXT {
    BOOLEAN IsCancelled;
    BOOLEAN IsTerminateFailed;
    KSPIN_LOCK Lock;
} REQUEST_CONTEXT, *PREQUEST_CONTEXT;

EvtDriverDeviceAdd(Driver, DeviceInit) {
    WDF_OBJECT_ATTRIBUTES attributes;
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, REQUEST_CONTEXT);
    WdfDeviceInitSetRequestAttributes(DeviceInit, &attributes);
}

EvtIoDispatch(Queue, Request) {
    reqContext->IsCancelled = FALSE;
    reqContext->IsTerminateFailed = FALSE;
    KeInitializeSpinLock(&reqContext->Lock);
    WdfObjectReference(Request);
    WdfRequestMarkCancelable(Request, EvtRequestCancelRoutine);
}
```

# Using Driver Lock (2)

```
EvtRequestCancelRoutine(Request)
{
    KeAcquireSpinlock(&reqContext->Lock,
&oldIrql);

    reqContext->IsCancelled = TRUE;

    if (TerminateIO() == TRUE) {
        WdfObjectDereference(Request);
        completeRequest = TRUE;
    }
    else {
        reqContext->IsTerminateFailed = TRUE;
        completeRequest = FALSE;
    }
    KeReleaseSpinlock(&reqContext->Lock, oldIrql);

    if (completeRequest) {
        WdfRequestComplete(Request,
STATUS_CANCELLED);
    };
}
```

```
EvtDpcForIsr(Interrupt)
{
    completeRequest = TRUE;
    KeAcquireSpinlock(&reqContext->Lock,
&oldIrql);

    if (reqContext->IsCancelled == FALSE) {
        status =
WdfRequestUnmarkCancelable(Request);
        if (status == STATUS_CANCELLED) {
            completeRequest = FALSE;
        }
        status = STATUS_SUCCESS;
    } else {
        if (reqContext->IsTerminateFailed {
            status = STATUS_CANCELLED;
        } else {
            completeRequest = FALSE
        }
    }
    KeReleaseSpinlock(&reqContext->Lock,
oldIrql);

    WdfObjectDereference(Request);
    if (completeRequest) {
        WdfRequestComplete(Request, status);
    };
}
```

# **Microsoft®**

*Your potential. Our passion.™*

© 2007 Microsoft Corporation. All rights reserved. Microsoft, Windows, Windows Vista and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries.

The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.

MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.