Lab 2: A Bus Driver

# Overview

A bus driver is responsible for creating the Physical Device Object.

In this lab, we’ll create a fictional bus – The RoboBus, and we’ll convert SmplDevice to RoboDevice on top of the RoboBus. We’ll create drivers for everything.

Since RoboBus isn’t a real hardware bus, we’ll “detect” a new device from a user mode application, sending IO control

## Goal

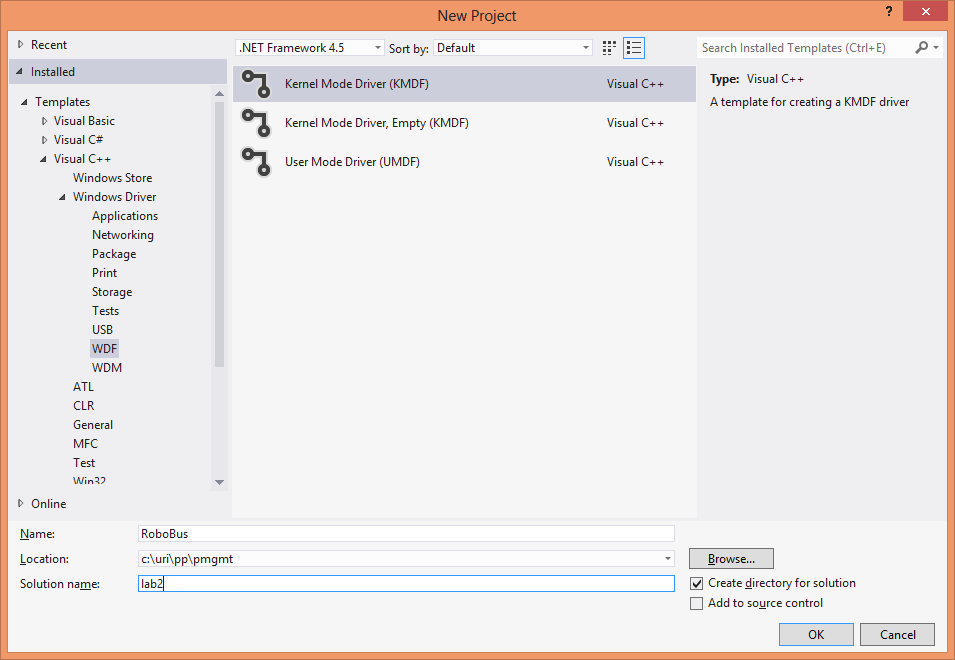
* Build a bus driver that will create physical devices.

# Create the skeleton for the RoboBus, RoboDevice and RoboMngr

We’ll create a new solution, with 3 new projects: **RoboBus**, **RoboDevice** and **RoboMngr**. As we’ll create the first device, Visual Studio will also create package project for us.

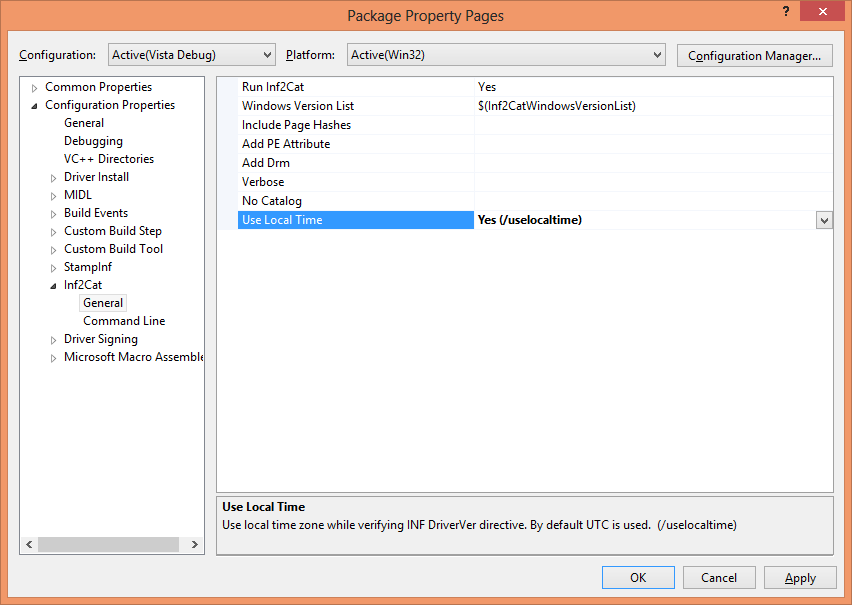
## Create a skeleton for RoboBus

* + 1. Open Visual Studio 2012 and create a new project
    2. Create a new KMDF project. The project name will be RoboBus. The location will be c:\projects, and the solution name will be c:\projects\lab2



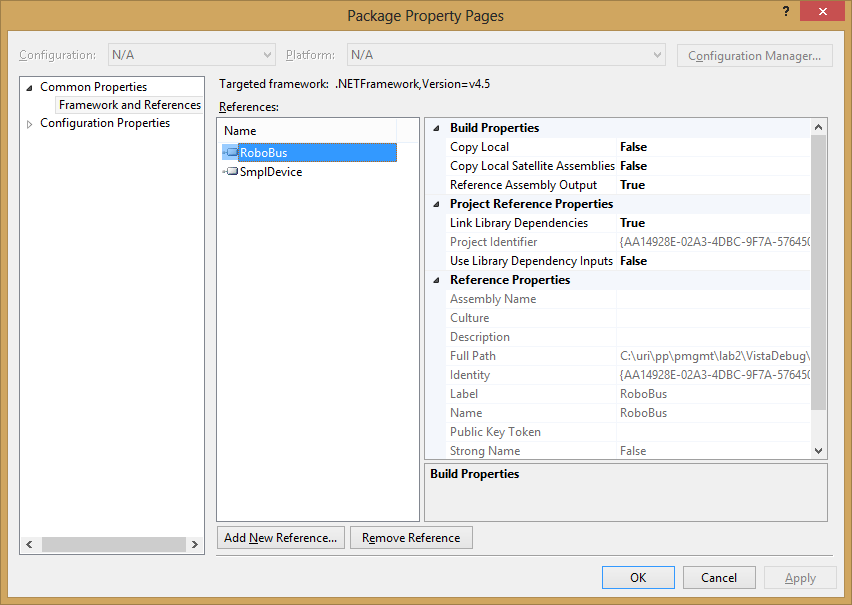
### To rename ‘RoboBus Package’ to Package (easier to type)

* + 1. On solution explorer, Right click on ‘RoboBus Package’ project and rename to Package (will make typing path easier).
    2. Save and exit Visual Studio
    3. With a text editor open lab2.sln and rename all strings ‘RoboBus Package’ to ‘Package’ (there should be 3 strings).
    4. Rename the folder ‘RoboBus Package’ to ‘Package’. Within that folder rename the \*.vcxproj and \*.vcxproj.filters files appropriately as well.
    5. Reopen the solution file
    6. Rename RoboBus.inf to RoboBus.inx
    7. Delete Readme.txt
    8. Right click on Package -> Properties -> Inf2Cat -> General, and set ‘Use Local Time’ to Yes.



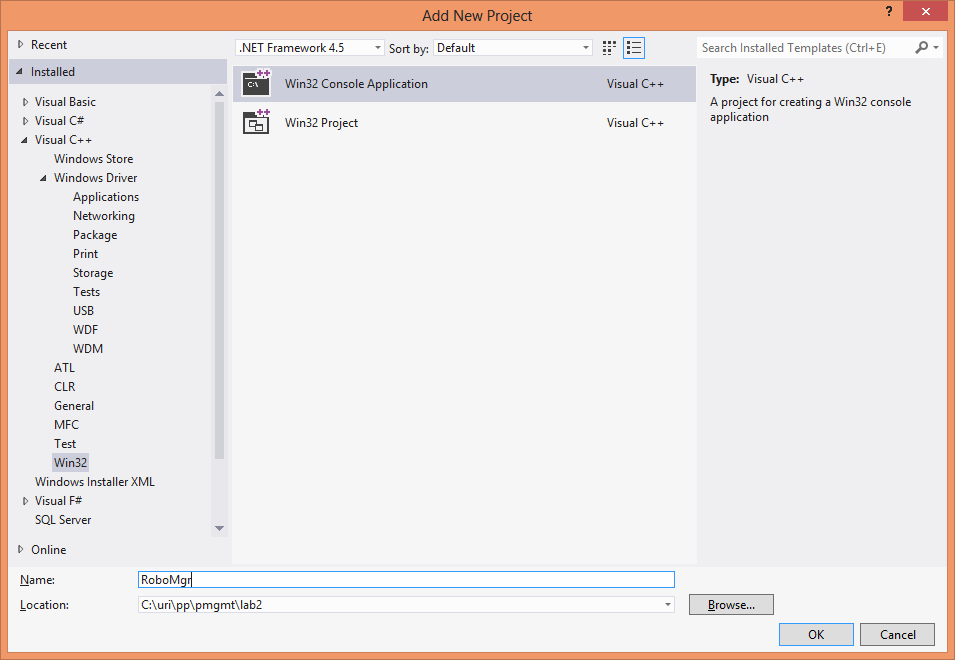
## Create a skeleton bus driver named RoboBus

* + 1. Right click on lab2, and select Add -> Add New Project
    2. Choose Kernel Mode Driver (KMDF). Location will be c:\projects\lab2, and name it RoboDevice
    3. In the Pakcage project, right click on Properties, and add RoboBus to list of Framework and References.



## Add a user mode project

Add a user mode C++ Win32 Console application to the project, and name it RoboMgr. User default settings



## Modification to INX files, Build and test

### Fix RoboBux.inx

* + 1. Open RoboBus.inx
    2. For the class, we will reuse the System class. Find out the GUID of the System class in the registry (HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Class).
    3. Change the *Class* from Sample to System
    4. Change the *ClassGuid* to appropriate value.
    5. We’ll have a common catalog file for all our devices. Rename *CatalogFile* value to Robot.cat
    6. Set the value for DriverVer to the appropriate format. Version should be 1.0.0.0
    7. Remove the ClassInstall32 section entirely (we aren’t installing a class), as well as SampleClassReg (a sub section)
    8. Change hardware id (in %RoboBus.DeviceDesc%) to root\robobus (optional. Just a personal preference of mine).
    9. Add the following two sections, after the CopyFiles line:

[ToasterStatBus\_Device.NT.HW]

AddReg=ToasterStatBus\_Device.NT.AddReg

[ToasterStatBus\_Device.NT.AddReg]

HKR,,DeviceCharacteristics,0x10001,0x0100

;Use same security checks on relative opens

HKR,,Security,,"D:P(A;;GA;;;BA)(A;;GA;;;SY)"

;Allow generic-all access to administrators and Local system

* + 1. Change *ManufacturerName* in the Strings section
    2. Delete ClassName
    3. Change RoboBus.DeviceDesc to: ‘RoboBus Enumerator’
    4. Change RoboBus.SVCDESC to: ‘RoboBus Enumerator Service’.

### Fix RoboDevice.inx

* + 1. Rename Class to ‘Robot’
    2. Generate a good with the format DEFINE\_GUID. Save it for later.
    3. Copy the guid comment, and change the ClassGuid
    4. Change CatalogFile to Robot.cat
    5. Change DriverVer appropriately.
    6. Within ClassInstall32, change AddReg to RobotClassReg
    7. Change section name SampleClassReg to RobotClassReg
    8. Add the following two lines to RobotClassReg section:

HKR,,DeviceCharacteristics,0x10001,0x100

;Use same security checks on relative opens

HKR,,Security,,"D:P(A;;GA;;;SY)(A;;GA;;;BA)(A;;GA;;;LS)"

;Allow generic all access to system, admin, locals

### Build and test

* + 1. Change target to Win8Debug and build everything
    2. Install on RoboBus on target machine
    3. Verify you can Robot bus under the System device class.
    4. Switch to the ‘By Connection’ view and verify you can see the Robot bus device under the root.

# RoboMgr: A user mode application that accesses RoboBus

RoboMgr is a user mode console application implemented in C++. It controls the RoboBus enumerator. Because we don’t have actual hardware, we simulate plugin/plugout/remove with a control device object.

## Accessing the RoboBus device

### Header files

* + 1. Within stdafx.h add include to SetupAPI.h
    2. We want to have the guid for the RoboBus interface (defined in ../RoboBus/public.h). Include this file from RoboFile.cpp. Don’t forget to include first <initguid.h>, to actually define it.

### Parse arguments

* + 1. Within main, parse the command line. The usage should be:

Usage: RoboMgr [-p SerialNo] Plugs in a device

[-u SerialNo] Plugs out a device. 0 for all

[-e SerialNo] Eject a device. 0 for all

SerialNo is a number greater than 0.

* + 1. In main, define a HDEVINFO variable and call it hdevinfo.
    2. Within main, we need to call SetupDiGetClassDevs to get hdevinfo. Use the following line:

hdevinfo = SetupDiGetClassDevs( &GUID\_DEVINTERFACE\_RoboBus, NULL, NULL,

DIGCF\_PRESENT | DIGCF\_DEVICEINTERFACE );

if( hdevinfo == INVALID\_HANDLE\_VALUE ) {

throw std::exception( "SetupDiGetClassDevs failed", GetLastError() );

}

* + 1. Next, still within main, you need to get the device interface data. Use this API: SetupDiEnumDeviceInterfaces
    2. Create a new function and call it openBusInterface, with the following prototype:

HFILE openBusInterface( HDEVINFO hdevinfo, SP\_DEVICE\_INTERFACE\_DATA\* pdevIntData );

* + 1. You need to have the interface details. Use the following function: SetupDiGetDeviceInterfaceDetail. You will call this function twice: First to get the required length of the structure, and a second time to get the actual data. Don’t forget to set up the cbSize of the structure.
    2. CreateFile to get a handle to the device. Return this value.

hfile = CreateFile( pdevIntDetailData->DevicePath, GENERIC\_READ, 0,

NULL, OPEN\_EXISTING, 0, NULL );

* + 1. For now that’s it. We’ll get back to the manager once we implement the IO controls. In main, close the hfile.

# Implement IO Control for RoboBus

We need to define data structure for the IO controls that will communicate between the user mode application and the control device of RoboBus.

### Add structure to IO control

* + 1. In public.h, add the following 3 structures.

//

// Data structure used in PlugIn and UnPlug ioctls

//

typedef struct \_BUSENUM\_PLUGIN\_HARDWARE

{

IN ULONG Size;

IN ULONG SerialNo;

#pragma warning(push)

#pragma warning(disable:4200)

IN wchar\_t HardwareIDs[];

#pragma pop(warning)

} BUSENUM\_PLUGIN\_HARDWARE;

typedef struct \_BUSENUM\_UNPLUG\_HARDWARE

{

IN ULONG Size;

ULONG SerialNo;

ULONG Reserved[2];

} BUSENUM\_UNPLUG\_HARDWARE;

typedef struct \_BUSENUM\_EJECT\_HARDWARE

{

IN ULONG Size;

ULONG SerialNo;

ULONG Reserved[2];

} BUSENUM\_EJECT\_HARDWARE;

* + 1. You also need to define 3 custom IoCtl codes:

#define BUSENUM\_IOCTL(\_index\_) \

CTL\_CODE (FILE\_DEVICE\_BUSENUM, \_index\_, METHOD\_BUFFERED, FILE\_READ\_DATA)

#define IOCTL\_BUSENUM\_PLUGIN\_HARDWARE BUSENUM\_IOCTL (0x0)

#define IOCTL\_BUSENUM\_UNPLUG\_HARDWARE BUSENUM\_IOCTL (0x1)

#define IOCTL\_BUSENUM\_EJECT\_HARDWARE BUSENUM\_IOCTL (0x2)

#define IOCTL\_TOASTER\_DONT\_DISPLAY\_IN\_UI\_DEVICE BUSENUM\_IOCTL (0x3)

* + 1. Also in public.h add the GUID you saved earlier of the Class of Robot. Also create a static string which will be the hardware ID for the Robot PDO.

DEFINE\_GUID( GUID\_DEVCLASS\_ROBOT,

0x4a1b20a7, 0x3232, 0x4932, 0xb4, 0x5c, 0x8a, 0xa6, 0x41, 0x27, 0xf5, 0x25);

// {4A1B20A7-3232-4932-B45C-8AA64127F525}

#define BUS\_HARDWARE\_IDS L"{4A1B20A7-3232-4932-B45C-8AA64127F525}\\Robot\0"

### Modify Queue.c

* + 1. Within queue.c, prototype 3 functions:

NTSTATUS RoboBusPlugInDevice( WDFDEVICE Device, wchar\_t\*\* HardwareIds, ULONG SerialNo )

{

return STATUS\_SUCCESS;

}

For now, leave it empty

* + 1. In queue.c, we need to implement RoboBusEvtIoDeviceControl. Start by getting the device from the queue WdfIoQueueGetDevice, and then switch on the IoControlCode parameter.
    2. Also, define 3 pointers: pplugin, punplug, and peject for the 3 Io Control structures.
    3. In case of IOCTL\_BUSENUM\_PLUGIN\_HARDWARE use the following line to get the pointer to the input buffer:

status = WdfRequestRetrieveInputBuffer(

Request,

sizeof(BUSENUM\_PLUGIN\_HARDWARE) + sizeof(UNICODE\_NULL)\*2,

(void\*\*)&pplugin, &length );

* + 1. Verify that HardwareIDs is ended with two NULLs, and that Size is sizeof(BUSENUM\_PLUGIN\_HARDWARE).
    2. Call RoboBusPlugin

### Back to RoboMgr

After you have hfile to the device we need to call the IO control: Use the following code:

* + 1. Allocate a buffer that is the size of BUSENUM\_PLUGIN\_HARDWARE and the size of the hardware IDs.
    2. Alias that butter to BASEENUM\_PLUGIN\_HARDWARE
    3. Set the Size signature.
    4. Put the SerialNo
    5. Copy the BUS\_HARDWARE\_IDS
    6. Call the DeviceIoControl

if( fPlugin ) {

ULONG bytes = sizeof(BUSENUM\_PLUGIN\_HARDWARE) + sizeof(BUS\_HARDWARE\_IDS);

std::unique\_ptr<BYTE> buffer( new BYTE[bytes] );

BUSENUM\_PLUGIN\_HARDWARE\* pplugin = (BUSENUM\_PLUGIN\_HARDWARE\*)buffer.get();

pplugin->Size = sizeof(BUSENUM\_PLUGIN\_HARDWARE);

pplugin->SerialNo = SerialNo;

memcpy( pplugin->HardwareIDs, BUS\_HARDWARE\_IDS, sizeof(BUS\_HARDWARE\_IDS) );

bresult = DeviceIoControl( hfile, IOCTL\_BUSENUM\_PLUGIN\_HARDWARE, pplugin, bytes, NULL, 0, &bytes, NULL );

if( !bresult ) {

throw std::exception( "Failed DeviceIoControl", GetLastError() );

}

}

### Test

* + 1. Build everything and deploy
    2. By now, you should be able to run RoboMgr with ‘-p 1’ switch and have a breakpoint on RoboBusEvtIoControl. You’ll end up at the empty RoboBusPluginDevice.

### Add skeleton for unplug and eject

* + 1. Similar to Plugin, add the cases for unplug and eject operations. Do it both in the user mode application as well as for the RoboBus driver. You should create two additional empty function RoboBusUnplugDevice and RoboBusEjectDevice, which will be called from RoboBusEvtIoControl on appropriate Io Control code. Those cases will be easier to implement because the buffer if at fixed size.

# Adding a PDO

## Device Contexts

We now going to have two types of devices: the RoboBus enumerator and then PDO for each physical device we are going to create. In order to accommodate that, let’s start by having two device contexts.

* + 1. Rename DEVICE\_CONTEXT to FDO\_DEVICE\_CONTEXT
    2. Create a new context PDO\_DEVICE\_CONTEXT
    3. Using the macro: WDF\_DECLARE\_CONTEXT\_TYPE\_WITH\_NAME, define the functions getFdoContext and getPdoContext appropriately.
    4. In PDO\_DEVICE\_CONTEXT we’ll only have ULONG SerialNo.
    5. In FDO\_DEVICE\_CONTEXT we’ll have WDFWAITLOCK ChildLock (to be used later).
    6. Fix all compilation errors.

## Implementing RoboBusPlugInDevice

This is the function that creates the PDO (simulation of a hardware plug-in).

* + 1. Get pointer to context of the FDO (it has the lock). Call it fdoContext
    2. Call these two locks:

WdfWaitLockAcquire( fdoContext->ChildLock, NULL );

WdfFdoLockStaticChildListForIteration( device );

* + 1. We need to iterate through all PDO’s and verify that their SerialNo is different than the new SerialNo. Do this with:

while( (hChild = WdfFdoRetrieveNextStaticChild(device, hChild, WdfRetrieveAddedChildren))

!= NULL )

* + 1. Compare SerialNo within the fdoContext with the SerialNo passed as a parameter. If any of the child device is equal, set a local flag unique to false, and status to STATUS\_INVALID\_PARAMETER.
    2. If still unique, call a function RoboBusCreatePdo (to be implemented below)

## Implementation of RoboBusCreatePdo

* + 1. Create a new file and name it pdo.c
    2. Create a function RoboBusCreatePdo with the following prototype:

NTSTATUS RoboBusCreatePdo( WDFDEVICE device, wchar\_t\* HardwareIds, ULONG SerialNo )

* + 1. Allocate device initialization structure of type WDFDEVICE\_INIT\* with WdfPdoInitAllocate. Call it pDeviceInit.
    2. Add the following initializations (Don’t forget to add error handling code), and then create the PDO device:

WdfDeviceInitSetDeviceType( pDeviceInit, FILE\_DEVICE\_BUS\_EXTENDER );

RtlInitUnicodeString( &deviceId, HardwareIds );

status = WdfPdoInitAssignDeviceID( pDeviceInit, &deviceId );

status = WdfPdoInitAddHardwareID( pDeviceInit, &deviceId );

status = RtlUnicodeStringPrintf( &buffer, L"%02d", SerialNo );

status = WdfPdoInitAssignInstanceID( pDeviceInit, &buffer );

status = RtlUnicodeStringPrintf( &buffer, L"Uri\_London\_Robot\_%02d", SerialNo );

status = WdfPdoInitAddDeviceText( pDeviceInit, &buffer, &deviceLocation, 0x409 );

WdfPdoInitSetDefaultLocale( pDeviceInit, 0x409 );

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE( &attributes, PDO\_DEVICE\_CONTEXT );

status = WdfDeviceCreate( &pDeviceInit, &attributes, &hchild );

* + 1. Then, get the context of the PDO, and set the SerialNo to the context
    2. Using the API WdfFdoAddStaticChild, add hchild to list of child device of the enumerator.
    3. Add error handling. If we had a failure, clean up the pDeviceInit structure and delete hchild if they were created.

### Some tweaks creating the FDO device: RoboBusCreateDevice

* + 1. Before creating the device, set two attributes to the DeviceInit:

WdfDeviceInitSetDeviceType(DeviceInit, FILE\_DEVICE\_BUS\_EXTENDER);

WdfDeviceInitSetExclusive(DeviceInit, TRUE);

* + 1. After the FDO was created, you need to initialize the ChildLock (part of the FDO context).

deviceContext = getFdoContext(device);

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

attributes.ParentObject = device;

status = WdfWaitLockCreate( &attributes, &deviceContext->ChildLock );

* + 1. Add a PNP\_BUS\_INFORMATION busInfo variable to the function
    2. Add the following code to the bottom of the function. This will tell the framework what kind of devices the PDO is handling.

busInfo.BusTypeGuid = GUID\_DEVCLASS\_ROBOT;

busInfo.LegacyBusType = PNPBus;

busInfo.BusNumber = 0;

WdfDeviceSetBusInformationForChildren(device, &busInfo);