UEFI Development

UEFI Driver Model, Protocols and Apps

By Elvis M. Teixeira [elvismtt@gmail.com]

UEFI Images

- UEFI applications and drivers are compiled into images
- An UEFI image is executable (PE/COFF) code
- Images can be loaded into memory and unloaded from there (removed)
- A loaded image can be started (The entry point is called)

Drivers VS Applications

Applications

- An application is executed from the beginning of its entry point to its end
- Possibly with side effects (I/O, etc)

Drivers

- A driver exposes a service to be used asynchronously by others.
- ‘Others’ may be apps, drivers or timer events

Protocols

- Protocols are data structures that contain function pointers
- They can also have data members (e.g. version numbers)
- These pointers should point to the implementation provided by some driver

Example 1: EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

```
typedef struct
_EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
    UINT64          Revision;
    EFI_VOLUME_OPEN OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;


typedef
EFI_STATUS
(EFIAPI *EFI_VOLUME_OPEN) (
    IN EFI_SIMPLE_FILE_SYSTEM_PROTOCOL    * This,
    OUT EFI_FILE                          **Root
);
```

Handles

- The handle database is the most important data structure in the DXE phase
- In each handle there may be any number of protocols and images installed
- A GUID uniquely identifies a resource within a handle
- In a given handle there can be only one resource with a given GUID

The Boot Services Table

Is a set of functions that is globally accessible.

They can be used to:

- Find resources in the handle database
- Load, start and unload images
- Create and start timers
- Many other things

Header UefiBootServicesTableLib.h declares a global pointer `gBS` to this table

Example 2: Using the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

```
EFI_HANDLE Handle = NULL;
EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *FSProtocol = NULL;
EFI_FILE_PROTOCOL *RootDir = NULL;
EFI_FILE_PROTOCOL *File = NULL;

EFI_STATUS Status = gBS->LocateHandle (
    AllHandles,
    &gEfiSimpleFileSystemProtocol,
    NULL,
    &BufferSize,
    &Handle
    );

Status = gBS->OpenProtocol (
    Handle,
    &gEfiSimpleFileSystemProtocol,
    (VOID **) &FSProtocol,
    ImageHandle,
    NULL,
    EFI_OPEN_PROTOCOL_GET_PROTOCOL
    );
```

```
Status = FSProtocol->OpenVolume (
    FSProtocol,
    &RootDir
    );

Status = RootDir->Open (
    RootDir,
    &File,
    L”FileName.txt”,
    EFI_FILE_MODE_READ,
    EFI_FILE_VALID_ATTR
    );

Status = File->Read (
    File,
    &BufferSize,
    Buffer
    );
```

Driver development

A driver that follows the “UEFI driver model” exposes an entry point,

an unload function (optional but recommended) and installs at least:

- The EFI_DRIVER_BINDING_PROTOCOL
- The EFI_SUPPORTED_EFI_VERSION_PROTOCOL
- The EFI_COMPONENT_NAME_PROTOCOL
- The EFI_COMPONENT_NAME2_PROTOCOL

Installing the protocols

The driver’s entry point:

```
EFI_STATUS
EFIAPI
MyDriverEntry (
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status = gBS->InstallMultipleProtocolInterfaces (
        &ImageHandle,
        &gEfiDriverSupportedEfiVersionProtocolGuid,
        &gMyDriverSupportedEfiVersion,
        &gEfiDriverBindingProtocolGuid,
        &gMyDriverDriverBinding,
        &gEfiComponentNameProtocolGuid,
        &gMyDriverComponentName,
        &gEfiComponentName2ProtocolGuid,
        &gMyDriverComponentName2,
        NULL
    );

    return Status;
}
```

The EFI_DRIVER_BINDING_PROTOCOL

Contains 3 functions:

Supported():

- Should check if the a handle provides access to a supported device

Start():

- Should install the protocols that make the driver's services available

Stop():

- Should undo everything Start() does

Finding supported devices

Supported(): returns EFI_SUCCESS if ControllerHandle has a reference to a device the driver can manage. Otherwise it returns EFI_UNSUPPORTED.

Supported() is called for each HANDLE in the handle database on driver initialization, and when new devices are attached.

```
EFI_STATUS
EFIAPI
MyDriverSupported (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                   ControllerHandler,
    IN EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath
)
{
    EFI_STATUS Status          = EFI_SUCCESS;
    BOOLEAN    ThereIsADevice  = FALSE;

    Status = DoesHandleContainsAnyMyDevice (
        ControllerHandle,
        &ThereIsADevice
        );

    if (EFI_ERROR (Status)) {
        MaybeHandleError (Status, ControllerHandle);
    }

    if (ThereIsADevice) {
        return EFI_SUCCESS;
    }

    return EFI_UNSUPPORTED;
}
```

Registering driver services

Starting drivers often include installing IO protocols through which users can access the driver’s services.

These protocols may be abstractions on top of other IO protocols.

In addition to IO protocols timer events sometimes appear.

```
EFI_STATUS
EFIAPI
MyDriverStart (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                   ControllerHandler,
    IN EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath
)
{
    EFI_STATUS      Status          = EFI_SUCCESS;
    MY_IO_PROTOCOL *MyIOProtocol    = NULL;

    Status = InitializeMyIOProtocol (
        &MyIOProtocol
        );

    Status = gBS->InstallMultipleProtocolInterfaces (
        ControllerHandle,
        &gMyIOProtocolGuid
        MyIOProtocol,
        NULL
        );

    return EFI_SUCCESS;
}
```

The build system

The EDK2 source tree includes a custom build system which helps with:

- Providing different implementations for the same interface (library classes)
- Generating code for common tasks and data objects e. g. GUIDS
- Creating a dependency tree for each package

The build system - platform file (.dsc)

- Each package has one
- Defines overall compilation context and lists apps and drivers in the package
- Many apps and drivers can be built by pointing the build utility to a .dsc
    - `build -p SomePkg.dsc -b X64`
- Maps library classes to a particular implementation

The build system - platform file (.dsc) EXAMPLE

```
[Defines]
  PLATFORM_NAME                  = Shell
  PLATFORM_GUID                  = E1DC9BF8-7013-4c99-9437-795DAA45F3BD
  PLATFORM_VERSION               = 1.01
  DSC_SPECIFICATION              = 0x00010006
  OUTPUT_DIRECTORY               = Build/Shell
  SUPPORTED_ARCHITECTURES        = IA32|IPF|X64|EBC|ARM|AARCH64
  BUILD_TARGETS                  = DEBUG|RELEASE|NOOPT
  SKUID_IDENTIFIER               = DEFAULT

[LibraryClasses.common]
  UefiApplicationEntryPoint|MdePkg/Library/UefiApplicationEntryPoint/UefiApplicationEntryPoint.inf
  !if $(TARGET) == RELEASE
    DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf
  !else
    DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf
  !endif

[Components]
  ShellPkg/Library/UefiShellLib/UefiShellLib.inf
```

The build system - “dec” file (.dec)

- Each package has one or more of these too
- Lists include directories
- Lists package GUIDS
- Lists PCD values fixed at build

The build system - “dec” file (.dec) EXAMPLE

```
[Defines]
  DEC_SPECIFICATION              = 0x00010005
  PACKAGE_NAME                   = ShellPkg
  PACKAGE_GUID                   = C1014BB7-4092-43D4-984F-0738EB424DBF
  PACKAGE_VERSION                = 1.01

[Includes]
  Include

[Guids]
  gEfiShellPkgTokenSpaceGuid = {0x171e9188, 0x31d3, 0x40f5, {0xb1, 0x0c, 0x53, 0x9b, 0x2d, 0xb9, ...
  gShellVariableGuid         = {0x158def5a, 0xf656, 0x419c, {0xb0, 0x27, 0x7a, 0x31, 0x92, 0xc0, ...


[PcdsFixedAtBuild]
  gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask|0xFF
  gEfiShellPkgTokenSpaceGuid.PcdShellLibAutoInitialize|FALSE
```

The build system - “inf” file (.inf)

- One for each app or driver
- Lists the source files that make up the driver/app
- Lists the library classes needed by the driver/app

The build system - “inf” file (.inf) EXAMPLE

```
[Defines]
  INF_VERSION                    = 0x00010006
  BASE_NAME                      = Hello
  FILE_GUID                      = a912f198-7f0e-4803-b908-b757b806ec83
  MODULE_TYPE                    = UEFI_APPLICATION
  VERSION_STRING                 = 0.1
  ENTRY_POINT                    = ShellCEntryLib

[Sources]
  Hello.c

[Packages]
  MdePkg/MdePkg.dec
  ShellPkg/ShellPkg.dec

[LibraryClasses]
  UefiLib
  ShellCEntryLib
```

Review notes

- Decide whether you need an app or a driver (what the entry point does)
- Create a new .inf file with a new GUID generated by a proper tool
- Insert the app/driver’s .inf into the list of Components of a platform file
- In case you have a driver follow the driver model
    - At entry point install ComponentName, DriverBinding and SupportedEfiVersion
    - Think carefully what your Start() and Stop() functions should do
    - If it makes any sense provide an Unload() function
    - If there is a version ‘2’ of a protocol, you should implement both :(