UEFI Development Exploration 40 – Building Your Own Package

原创 luobing4365 Posted on 2019-12-08 19:53:07 Read 1.6k Collection 14 Likes 1

Category columns: UEFI Development Article Tags: UEFI Programming UEFI Package UEFI BIOS Low-level programming EDKII

(Please keep it-> Author: Luo Bing https://blog.csdn.net/luobing4365)

Some time ago, when developing UEFI programs under Linux , I found that the 32-bit programs of AppPkg I wrote before could not be compiled, and I could not test the execution program in a simulated environment.

I wanted to get rid of AppPkg and build my own Package. Of course, StdLib libraries could not be used, and the main() function could not be used as the entry point. I thought this was not a big deal, after all, the Option ROM code that is normally built could not use these.

Just do it, and review the knowledge points of various types of files.

1 Compile framework

The compilation system of EDKII is built based on Python and C code, which can be compiled across platforms. It can also run on different CPU architectures, such as X86 and ARM. Recently, we are working on software for Loongson MIPS, and we are happy to see the expansion of UEFI development.

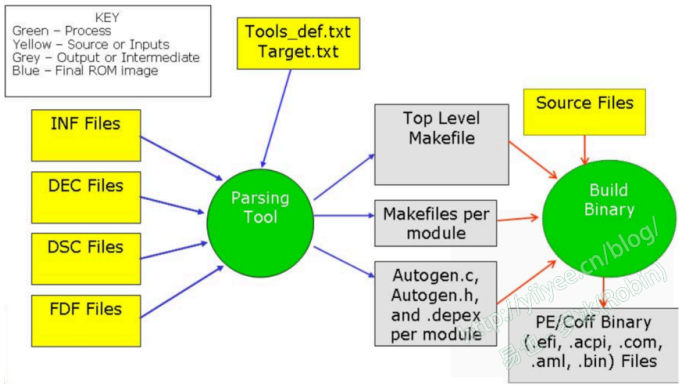

Figure 1 EDKII workflow

Figure 1 shows the workflow of EDKII. In summary, the compilation tool will analyze metadata files (DSC, DEC, INF, etc.) and generate a top-level Makefile, as well as Makefiles and AutoGen.c/Autogen.h for each module.

In the Autogen file, EDKII's compilation tool will generate the guids, protocols, ppis, PCDs, etc. required by the module. Finally, it will be compiled into the specified binary files, including efi, acpi, aml, etc.

Carefully observe the normal compilation process. Different platforms use different compilers (for example, VS Stduio is used to compile under Windows , and GCC is used for Linux). The final executable file that can be run in the UEFI environment is generated using the GenFw tool (GenFV is used to generate the image file).

Most of the tools are located in the /BaseTools directory, which is generated when we build the environment. Of course, you can also use the compiled tools directly. For example, in the current development under Loongson, the manufacturer does not provide source code, so you can only use the compiled tools.

In normal development, UEFI Driver and UEFI Application are most commonly generated. Option ROM is essentially a UEFI Driver, and some adjustments can be made during compilation. The specific process has been described in the previous blog.

The above is the basic compilation framework, which gives us a general idea of how the files are compiled. Next, we will learn about the details of each metadata file.

2.dsc file

The .dsc file is used to compile a Package, which contains several necessary parts such as [Defines], [LibraryClasses], [Components], and optional parts such as [PCD], [BuildOptions], and [SkuIds].

In addition, EDKII metadata files are basically in EBNF format (Extended Backus Naur Form). For specific descriptions, please refer to the DSC specification file "edk-ii-dsc-specification.pdf".

What is useful to me is that when using Vs code, I can specify the dsc and dec files as BNF specifications, and the highlighted code is easier to read:

Figure 2 Highlighting the dsc file

[Defines] block

Used to define variables, which can be used in the compilation step and can be referenced by other modules in the package. It must be the first part of the .dsc file, and the format of each section is Name = Value.

In addition, there are two other formats: DEFINE MACRO = Value and EDK_GLOBAL MACRO = Value. Macros defined in these two ways can be used in .dsc and .fdf files through the reference method of $(MACRO).

The macro variables that must be defined in this block include: DSC_SPECIFICATION, PLATFORM_GUID, PLATFORM_VERSION, PLATFORM_NAME, SKUID_IDENTIFIER, SUPPORTED_ARCHITECTURES, and BUILD_TARGETS. Other macro variables are optional, please refer to the .dsc specification file for details.

[LibraryClasses] Block

Defines the library name and the path to the library .inf file. These libraries can be referenced by modules in the [Components] block (if the module you are writing requires a new library, define it here).

It is mainly used to provide library interfaces for Modules. If the dsc file does not have EDKII Modules, it is an optional block. Of course, our goal is to compile Modules, so it is still considered a must.

Common formats are: [LibraryClasses], [LibraryClasses.IA32], [LibraryClasses.X64], [LibraryClasses.IPF], [LibraryClasses.EBC], [LibraryClasses.common]. Following is the library entry:

LibraryClassName|Path/To/LibInstanceName.inf
LibraryClassName1|Path/To/LibInstanceName1.inf

LibraryClassName cannot be NULL, it is the keyword of the library name and must be unique. All INF files will use the entry of this library when linking other libraries and modules.

The general syntax is as follows:

[LibararyClasses.$(Arch).$(MODULE_TYPE)]
LibraryName | path/LibraryName.inf

You can also use the structure

[LibararyClasses.$(Arch1).$(MODULE_TYPE1),LibararyClasses.$(Arch1).$(MODULE_TYPE1)]
LibraryName | path/LibraryName.inf

$Arch and $MODULE_TYPE are optional. Leave them blank for universal use.

$Arch indicates the architecture and can be one of the following values: IA32, X64, IPF, EBC, ARM, common. Common means it is valid for all architectures.

*$MODULE_TYPE indicates the category of the module. The libraries listed in the block can only provide module connections of the $(MODULE_TYPE) category. It can be the following values: SEC, PEI_CORE, PEIM, DEX_CORE, DEX_SAL_DRIVER, BASE, DXE_SMM_DRIVER, DXE_DRIVER, *

DXE_RUNTIME_DRIVER, UEFI_DRIVER, UEFI_APPLICATION, USER_DEFINED.

[Components] Block

One or more [Components] sections contain a list of EDKII Moudles. Common formats include: [Components], [Components.IA32], [Components.X64], [Components.IPF], [Components.EBC], [Components.common].

The path to the inf file is usually specified in this format: /Path/and/Filename.inf. Or you can use a nested format,

Path\Exectuables.inf{ < LibraryClasses> # Nested LibraryName | Path/LibraryName.inf < BuildOptions> # Nested blocks # The block can also contain < Pcds> }*

Note that Path uses a relative path relative to the EDK2 root directory.

[BuildOption] Block

For Modules, if you do not want to use the compilation parameters specified in the toolchain, you can specify your own unique compilation parameters here.

Its format is:

```
[BuildOptions.$(Arch).$(CodeBase)]
    [Compiler]:[$(Target)]_[Tool]_[$(Arch)]_[CC|DLINK]_FLAGS=
```

The compiler macro name is FAMILY, which is defined in Conf/tools_def.txt and can be MSFT, INTEL or GCC, etc. Other parameters can be found in the dsc specification file and will not be discussed in detail here.

Previously, AppPkg could not compile for IA32 under Linux because this block defined files that could only be compiled under the Microsoft compiler.

3.dec files

.dec files support the compilation of EDKII Modules. They are used to define information shared between modules. There are eight types of blocks that can be defined in .dec files, namely Defines, Includes, LibraryClasses, Guids, Protocols, Ppis, PCD and UserExtensions.

In other words, the .dec file is a collection of shared data and interfaces of various modules.

[Defines] block

This is a required block. A typical example is as follows:

```
[Defines]
DEC_SPECIFICATION = 0x0001001B
PACKAGE_NAME = MdePkg
PACKAGE_GUID = 1E73767F-8F52-4603-AEB4-F29B510B6766
PACKAGE_VERSION = 1.02
PACKAGE_UNI_FILE = MdePkg.uni
```

PACKAGE_UNI_FILE is used to specify the file name for storing Unicode strings. I personally think it is mainly convenient for supporting multiple languages.

[Includes] Block

This is an optional block. Lists the directories where the header files provided by this package are located, which can be specified for different architectures.

[LibraryClasses] Block

This is an optional block. Package can provide libraries to the outside world through .dec files. Each library must have a header file placed in the Include\Library directory. This block is used to clarify the correspondence between libraries and header files. Format:

```
[LibraryClasses.$(Arch)]
LibraryName | Path/LibraryHeader.h
```

[Guids] Block

Optional block, used to define the Guid value of the Guid C name. The identifier name Private in this block is used to prevent modules outside this package from accessing it.

Commonly used definition names: [Guids], [Guids.common], [Guids.common.Private], [Guids.IA32], [Guids.IA32.Private], [Guids.X64], [Guids.X64.Private], [Guids.IPF], [Guids.IPF.Private], [Guids.EBC], [Guids.EBC.Private]. These definition names: names can also be combined, for example: [Guids.IA32, Guids.X64].

[Protocols] Block

Optional block. It is similar to the Guids block. Commonly used definition names are as follows: [Protocols], [Protocols.common], [Protocols.common.Private], [Protocols.IA32], [Protocols.IA32.Private], [Protocols.X64], [Protocols.X64.Private], [Protocols.IPF], [Protocols.IPF.Private], [Protocols.EBC], [Protocols.EBC.Private].

These definition names can be combined, such as [Protocols.IA32, Protocols.X64].

The general format is as follows:

[Protocol.$(Arch)]
ProtocolCName = {C Format Guid Value} # Comment

The other blocks are not listed in detail, please refer to the specification of the dec file.

4. Build Package

The contents of other files are not relevant to building the package, such as .fdf files, .uni files, etc. You can find the corresponding specification files. Basically, they can be found on this page:

https://github.com/tianocore/tianocore.github.io/wiki/EDK-II-Specifications

The purpose of building this package is to make the code compileable under both Linux and Windows, without hanging under other packages (if you have time, change ftol2.obj to an assembly version that can be compiled under Linux, then the main() function can also be used), which is convenient for debugging.

I used the .dsc and .dec files of AppPkg as a template and modified them to the files I needed.

There are not many places to modify, mainly:

1) Change the value of PLATFORM_NAME in the .dsc file to RobinPkg;
2) Change the value of PLATFORM_GUID in the .dsc file to any number to ensure it is unique; (Of course, you can also use Microsoft's tools to generate one)
3) Change the OUTPUT_DIRECTORY of the .dsc file to Build/RobinPkg;
4) Comment out the last StdLib and Sockets in the .dsc file;
5) Change Name and GUID in the .dec file;
6) Comment out all blocks except [Defines] in the .dec file;

The modified package is provided at the end of the blog. After modification, just compile it. The built package is as follows:

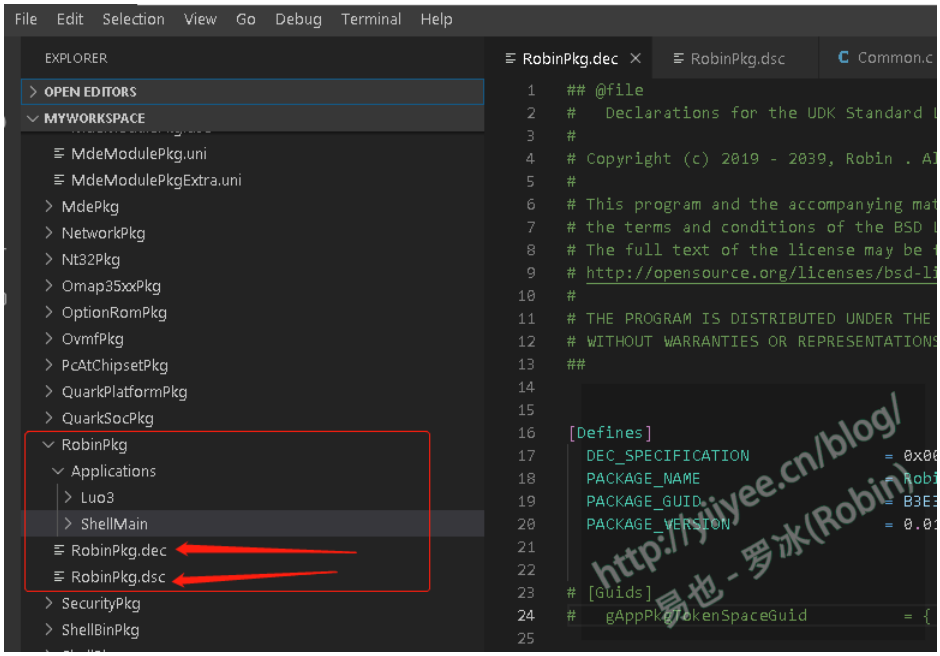

Figure 3 The built RobinPkg

I used the two framework examples in the previous blog that were compiled in Ubuntu 16.04 for the experiment. I found a small problem: files stored in UTF-8 in Linux could not be compiled in Windows.

I suspect that it is caused by Chinese characters. There are many Chinese comments in my code. I changed the file that prompted the error to store in GB 2312, and the compilation passed. I changed it on several platforms and editing tools, and I don't know which character caused the compiler to be dissatisfied. I will try to use English comments in the future.

(First published address: http://yiiyee.cn/blog/author/luobing/)

Gitee address: https://gitee.com/luobing4365/uefi-explorer
Project code is located at: / 25 RobinPkg