
# EDK II Meta-Data Expression Syntax Specification

# TABLE OF CONTENTS

# EDK II Meta-Data Expression Syntax Specification

**Revision 1.20**

**04/30/2025 09:38:18**

## Acknowledgements



## Revision History

| Revision | Revision History | Date |
|---|---|---|
| 1.0 | Initial Release | September 2014 |
| 1.1 | | September 2014 |
| 1.2 | Convert to Gitbook | April 2017 |

# 1 INTRODUCTION

This document describes the syntax of expression statements for EDK II Meta-data files used in data fields, feature flag expressions and conditional directive statements.

# 1.1 Overview

This document describes, using EBNF, the syntax of expressions used in EDK II meta-data documents. This syntax can be used to create a parser for expression evaluations. Whether a value derived from an expression, is valid for any given field in the EDK II meta-data document is outside the scope of this document.

# 1.2 Related Information

The following publications and sources of information may be useful to you or are referred to by this specification.

- ANSI C99 Specification, ISO/IEC 9899:TC2
- Unified Extensible Firmware Interface Specification, Version 2.3.1, Unified EFI, Inc, 2011, http://www.uefi.org
- UEFI Platform Initialization Specification, Version 1.2, Unified EFI, Inc., 2010, http://www.uefi.org
- UEFI Platform Initialization Distribution Package Specification, Version 1.0 with Errata B, Unified EFI, Inc., 2014, http://www.uefi.org
- The following specifications are available from http://www.tianocore.org
  - EDK II User Manual, Intel, 2010.
  - UEFI Driver Writer Guide, Version 1.00, Intel, 2012.
  - EDK II C Coding Standard, Intel, 2014.
  - EDK II DSC Specification, Intel, 2014.
  - EDK II INF File Specification, Intel, 2014.
  - EDK II FDF Specification, Intel, 2014.
  - EDK II Build Specification, Intel, 2014.
  - EDK II UNI Unicode File Specification, Intel, 2014.
  - VFR Programming Language, Intel, 2012.
- INI file, Wikipedia, http://en.wikipedia.org/wiki/INI_file
- Augmented BNF for Syntax Specifications: ABNF, RFC5234, Network Working Group, 2008, http://tools.ietf.org/html/rfc5234

# 1.3 Terms

The following terms are used throughout this document to describe varying aspects of input localization:

**BaseTools**

The BaseTools are the tools required for an EDK II build.

**BDS**

Boot Devices Selection phase.

**BNF**

BNF is an acronym for "Backus Naur Form". John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language.

**Component**

An executable image. Components defined in this specification support one of the defined module types.

**DEC**

EDK II Package Declaration File. This file declares information about what is provided in the package. An EDK II package is a collection of like content.

**DEPEX**

Module dependency expressions that describe runtime process restrictions.

**Dist**

This refers to a distribution package that conforms to the UEFI Platform Initialization Distribution Package Specification.

**DSC**

EDK II Platform Description File. This file describes what and how modules, libraries and components are to be built, as well as defining library instances which will be used when linking EDK II modules.

**DXE**

Driver Execution Environment.

**DXE SAL**

A special class of DXE modules that provides SAL Runtime Services. DXE SAL modules differ from DXE Runtime modules in that the DXE Runtime modules support Virtual mode OS calls at OS runtime and DXE SAL modules support intermixing Virtual or Physical mode OS calls.

**DXE SMM**

A special class of DXE modules that are loaded into the System Management Mode memory.

**DXE Runtime**

A special class of DXE modules that provide Runtime Services.

**EBNF**

Extended "Backus Naur Form" meta-syntax notation with the following additional constructs: square brackets "[...]" surround optional items, suffix "*" for a sequence of zero or more of an item, suffix "+" for one or more of an items, suffix "?" for zero or one of an item, curly braches "{...}" enclosing a choice of a list of alternatives and superscripts indicating between n and m occurrences.

**EDK**

Extensible Firmware Interface Development Kit, the original implementation of the Intel(R) Platform Innovation Framework for EFI Specifications developed in 2007.

**EDK II**

EFI Development Kit, version II that provides updated firmware module layouts and custom tools, superseding the original EDK.

**EDK Compatibility Package (ECP)**

The EDK Compatibility Package (ECP) provides libraries that will permit using most existing EDK drivers with the EDK II build environment and EDK II platforms.

**EFI**

Generic term that refers to one of the version of the EFI or UEFI specifications.

**FDF**

EDK II Flash Definition File. This file is used to define the content and binary image layouts for firmware images, update capsules and PCI option ROMs.

**FLASH**

This term is used throughout this document to describe one of the following:

- An image that is loaded into a hardware device on a platform - traditional ROM image.
- An image that is loaded into an Option ROM device on an add-in ard
- A bootable image that is installed on removable, bootable media, such as a Floppy, CD-ROM or USB storage device.
- A UEFI application that can be accessed during but (at an EFI Shell prompt), prior to hand-off to the OS loader.

**Foundation**

The set of code and interfaces that holds implementations of EFI together.

**Framework**

Intel(R) Platform Innovation Framework for EFI consist of the Foundation, plus other modular components that characterize the portability surface for modular components designed to work on any implementation of the Tiano architecture.

**GUID**

Globally Unique Identifier. A 128-bit value used to name entities uniquely. A unique GUID can be generated by an individual without the help of a centralized authority. This allows the generation of names that will never conflict, even among multiple, unrelated parties. GUID values can be registry format, 8-4-4-4-12, or C data structure format.

GUID also refers to an API named by a GUID.

**HII**

Human Interface Infrastructure. This generally refers to the database that contains string, font and IFR information along with other pieces that use one of the database components.

**HOB**

Hand-off blocks are key architectural mechanisms that are used to hand off system information in the early pre-boot phase.

**IFR**

Internal Forms Representation. This is the binary encoding that is used for the representation of user interface pages.

**INF**

EDK II Module Information File. This file describes how the module is coded.

**Library Class**

**A library class defines the API or interface set for a library. The consumer of the library is coded to the library class definition. Library classes are defined via a library class .h file that is published by a package.**

**Library Instance**

An implementation of one or more library classes.

**Module**

A module is either an executable image or a library instance. For a list of module types supported by a package, see Module Type.

**Module Type**

All libraries and components belong to one of the following module types: `BASE`, `SEC`, `PEI_CORE`, `PEIM`, `DXE_CORE`, `DXE_DRIVER`, `DXE_RUNTIME_DRIVER`, `DXE_SMM_DRIVER`, `DXE_SAL_DRIVER`, `UEFI_DRIVER` or `UEFI_APPLICATION`. These definitions provide a framework that is consistent with a similar set of requirements. A module that is of type module type `BASE`, depends only on header and libraries provided in the MdePkg, while a module that is of module type `DXE_DRIVER` depends on common DXE components. An additional module type, `USER_DEFINED`, is allowed for extensibility. The EDK II build system also permits modules of type `USER_DEFINED`. These modules will not be processed by the EDK II Build system.

**Numeric Values**

Numeric values in the EDK II meta-data file expressions are either unsigned integer values (base 10) or hexadecimal values (base 16). No other numeric data types are permitted.

**Package**

A package is a container. It can hold a collection of files for any given set of modules. Packages may be described as containing zero or more of any of the following:

- Source modules, containing all source files and descriptions of a module.
- Binary modules, containing EFI sections for a Framework File System and a description file specific to linking and binary editing of features and attributes specified in a Platform Configuration Database (PCD).
- Mixed modules, with both binary and source modules.

Multiple modules can be combined into a package and multiple packages can be combined into a single package.

**PCD**

Platform Configuration Database.

**PEI**

Pre-EFI Initialization Phase

**PEIM**

An API named by GUID.

**PI**

UEFI Platform Initialization Specification.

**PPI**

A PEIM-to-PEIM interface that is named by a GUID.

**Protocol**

An API named by GUID.

**Runtime Services**

Interfaces that provide access to underlying platform-specific hardware that might be useful during OS runtime, such as time and date services.

These services become active during the boot process but also persist after the OS loader terminates boot services.

**SAL**

System Abstraction Layer. A firmware interface specification used on Intel(R) Itanium(R) Processor based systems.

**SEC**

Security Phase is the code in the Framework that contains the processor reset vector and launches PEI. This phase is separate from PEI because some security schemes require ownership of the reset vector.

**SKU**

Stock Keeping Unit.

**SMM**

System Management Mode. A generic term for the execution mode entered when a CPU detect an SMI. The firmware, in response to the interrupt type, will gain control in physical mode. For this document, "SMM" describes the operational regime for IA32 and x64 processors that share the OS-transparent characteristics.

**UEFI**

Unified Extensible Firmware Interface

**UEFI Application**

An application that follows the UEFI Specification. The only difference between a UEFI application and a UEFI driver is that an application is unloaded from memory when it exits regardless of return status, while a driver that returns a successful return status is not unloaded when its entry point exits.

**UEFI Driver**

A driver that follows the UEFI Specification.

**UEFI Driver**

A driver that follows the UEFI specification.

**UEFI Specification Version 2.4**

Current UEFI version.

**UEFI Platform Initialization Distribution Package Specification 1.0**

The current version of this specification includes Errata B.

**UEFI Platform Initialization Specification 1.3**

Current version of the UEFI PI Specification.

**Unified EFI Forum**

A non-profit collaborative trade organization formed to promote and manage the UEFI standard. For more information, see http://www.uefi.org

**VFR**

Visual Forms Representation.

**VPD**

Vital Product Data that is read-only binary configuration data, typically located within a region of a flash part. This data would typically be updated as part of a firmware build, post firmware build (via patching tools), through automation on a manufacturing line as the 'FLASH' parts are programmed or through special tools.

## 1.4 Target Audience

Those performing UEFI development and support for platforms, distributable modules and tool development.

# 1.5 Conventions Used in this Document

This document uses typographic and illustrative conventions described below.

**Table 1 Font Conventions**

| Typographic Convention | Typographic convention description |
|---|---|
| Plain text | The normal text typeface is used for the vast majority of the descriptive text in a specification. |
| Plain text (blue) | Any plain text that is underlined and in blue indicates an active link to the crossreference. Click on the word to follow the hyperlink. |
| **Bold** | In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph. |
| Italic | In text, an Italic typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name. |
| `BOLD Monospace` | Computer code, example code segments, and all prototype code segments use a `BOLD Monospace` typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph. |
| `Bold Monospace` | Words in a `Bold Monospace` typeface that is underlined and in blue indicate an active hyper link to the code definition for that function or type definition. Click on the word to follow the hyper link. |
| `$(VAR)` | This symbol VAR defined by the utility or input files. |
| **Italic Bold** | In code or in text, words in **Italic Bold** indicate placeholder names for variable information that must be supplied (i.e., arguments). |

**Note:** Due to management and file size considerations, only the first occurrence of the reference on each page is an active link. Subsequent references on the same page will not be actively linked to the definition and will use the standard, non-underlined `BOLD Monospace` typeface. Find the first instance of the name (in the underlined `Bold Monospace` typeface) on the page and click on the word to jump to the function or type definition.

The following typographic conventions are used in this document to illustrate the Extended Backus-Naur Form.

**Table 2 EBNF Conventions**

| Typographic Convention | Typographic convention description |
|---|---|
| [item] | Square brackets denote the enclosed item is optional. |
| `{item}` | Curly braces denote a choice or selection item, only one of which may occur on a given line. |
| `<item>` | Angle brackets denote a name for an item. |
| `(range-range)` | Parenthesis with characters and dash characters denote ranges of values, for example, (a-zA-Z0-9) indicates a single alphanumeric character, while (0-9) indicates a single digit. |
| | Characters within quotation marks are the exact content of an item, as they must |

| | |
|---|---|
| | appear in the output text file. |
| `?` | The question mark denotes zero or one occurrences of an item. |
| `*` | The star character denotes zero or more occurrences of an item. |
| `+` | The plus character denotes one or more occurrences of an item. |
| `item{n}` | A superscript number, n, is the number occurrences of the item that must be used. Example: (0-9)8 indicates that there must be exactly eight digits, so 01234567 is valid, while 1234567 is not valid. |
| `item{n,}` | A superscript number, n, within curly braces followed by a comma "," indicates the minimum number of occurrences of the item, with no maximum number of occurrences. |
| `item{,n}` | A superscript number, n, within curly brackets, preceded by a comma ","indicates a maximum number of occurrences of the item. |
| `item{n,m}` | A super script number, n, followed by a comma "," and a number, m, indicates that the number of occurrences can be from n to m occurrences of the item, inclusive. |

# 2 EXPRESSION OVERVIEW

The expression syntax follows closely, the expression syntax and precedence of C as defined in the ISO/IEC 9899:TC2 document. Since the EDK II meta-data files are not parsed as C code, but rather as script files, some of the items listed in the C expression section (6.5) do not apply. Also, some of the items have been extended to use scripting syntax for logical comparison operators such as "EQ" as a synonym for "==".

The Conditional directive section is loosely based on section 6.10, Preprocessing directives. Again, some of the items listed in the Preprocessing directives section do not apply.

## 2.1 Constraints and Semantics

1. Prior to evaluation, all macro values, PCD values and defined literals (such as TRUE or FALSE) used in the expression must be resolved. The only exception to this rule is when a macro value assignment contains an expression. In this case, an expression on the right side of the assignment operator ("=") must have all values resolved in the expression's operands prior to evaluation and subsequent assignment. In this instance, in order to have the operand values resolved, the tools may be required to perform more than one pass over a file to obtain values for the operands.
2. Floating point values are not supported under PCD datum types.
3. When used in an expression, a PCD's value is used during evaluation.
4. An expression is a sequence of operators and operands that specifies computation of a value.
5. Between the previous and next sequence, an object shall have its stored value modified at most once by the evaluation of an expression.
6. Some operators (the unary operator, ~, and the binary operators, `<<`, `>>` , &, | and ^ collectively described as bitwise operators) are required to have operands that are integer (base10) or hexadecimal (base16) types.
7. The unary arithmetic operators, +, - and ~ must have an operand that is an integer (base10) or hexadecimal (base16) type, while the scalar unary operators, ! and NOT, must have operands that are of type scalar.
8. The Boolean values, False, FALSE and false have a numerical value of zero. The Boolean values, True, TRUE and true have a numerical value of one.
9. Tools may be required to translate the expression sequence into an expression format that is comprehended by the tool's native language. For example, a tool written in Python may require changing the exact syntax of an expression to a syntax that can be processed by Python's eval() function.
10. Multiplicative and additive operators require both operands to be arithmetic in type.
11. Relational and equality operators require both operands to be of the same type. Relational comparison on string literals and byte arrays must be performed comparing the byte values, left to right. The first pair of bytes (or a single extra byte) that is not equal will determine the result of the comparison. The following are examples of string comparisons:

    ```
    Foo = "zero", Bar = "three";
    ```

    `Foo < Bar` will evaluate to zero (AKA, FALSE), as "z" is greater than "t".

```
Foo = "thirty", Bar = "thirty1";`
```

`Foo < Bar` will evaluate to one (AKA, TRUE), as Bar has an extra character. ```

12. Logical operators require operands that are type scalar.
13. For the Conditional Operator, the first operand must be scalar, while the second and third operands must have the same type (i.e., both being scalar, both being integers, or both being string literals).

# 3 EXPRESSION FORMAT

The following EBNF describes the syntax and precedence of EDK II meta-data expression notation.

1. An Expression is a sequence of operators and operands that specifies a computational value, or that designates an object (or a function), or that performs a combination thereof.
2. Statements in nested parenthesis are evaluated inside to outside.
3. Statements in parenthesis are evaluated prior to evaluating statements outside of parenthesis.
4. Operators within a grouping (as designated by the name preceding the "::=" character sequence) are evaluated left to right.
5. The syntax is listed in precedence order (high to low).
6. The syntax does not define acceptable values of a data field; those are defined in the EDK II meta-data documents that define the data fields.

## 3.1 Data Field Expression

The following expression syntax may be used in a data field in EDK II Meta-data documents. All text inclusive of a semi-colon character through the end of the line must be viewed as a comment to the content preceding the semi-colon character. The use of a semi-colon character as a comment delimiter is for this specification only; other specifications may use different characters as comment delimiters.

### Restrictions

`<Function>`

FUNCTIONS SHOULD ONLY BE USED IF ALL TOOLS THAT PROCESS THE ENTRY IN THE META-DATA FILE COMPREHEND THE FUNCTION SYNTAX AND THE FUNCTION ELEMENT HAS BEEN DEFINED IN A HIGH LEVEL DESIGN DOCUMENT OR SOFTWARE ARCHITECTURE SPECIFICATION.

IF A TOOL DOES NOT COMPREHEND THE FUNCTION FORMAT, THE TOOL SHOULD FAIL GRACEFULLY.

### Prototype

```
<PrimaryExpression>     ::= {<Identifier>} {<Constant>} {<StringLiteral>} {<Function>}
                            {"(" <Expression> ")"} {<Expression>}
<Identifier>            ::= {<CName>} {<MACROVAL>}
<MACROVAL>              ::= "$(" <MACRO> ")"
<MACRO>                 ::= (A-Z) [(A-Z0-9_)]*
<Function>              ::= <CName> "(" <Argument> [<CSP> <Argument>]* ")"
<Argument>              ::= <PrimaryExpression>
<CName>                 ::= (a-zA-Z_) [(a-zA-Z0-9_)]*
<Constant>              ::= {<TrueFalse>} {<Number>} {<GuidValue>} {<Array>}
<TrueFalse>             ::= {<True>} {<False>}
<True>                  ::= {"TRUE"} {"True"} {"true"}
<False>                 ::= {"FALSE"} {"False"} {"false"}
<Number>                ::= {<Integer>} {<HexNumber>}
<Integer>               ::= <Base10>
<Base10>                ::= {(0-9)} {(1-9) [(0-9)]*}
<HexNumber>             ::= <Base16>
<Base16>                ::= <HexPrefix> [<HexDigit>]+
<HexDigit>              ::= (a-fA-F0-9)
<HexPrefix>             ::= {"0x"} {"0X"}
<GuidValue>             ::= {<RformatGuid>} {<CformatGuid>}
Rhex2                   ::= [<HexDigit>] <HexDigit>
Rhex4                   ::= [<HexDigit>] [<HexDigit>] Rhex2
```

```
Rhex8                   ::= [<HexDigit>] [<HexDigit>] [<HexDigit>] [<HexDigit>] Rhex4
<RformatGuid>           ::= Rghex8 "-" Rghex4 "-" Rghex4 "-" Rghex4 "-" Rghex12
Rghex2                  ::= <HexDigit> <HexDigit>
Rghex4                  ::= <HexDigit> <HexDigit> Rghex2
Rghex8                  ::= <HexDigit> <HexDigit> <HexDigit> <HexDigit> Rghex4
Rghex12                 ::= <HexDigit> <HexDigit> <HexDigit> <HexDigit> Rghex8
<Byte>                  ::= <HexPrefix> Rhex2
<Hex16>                 ::= <HexPrefix> Rhex4
<Hex32>                 ::= <HexPrefix> Rhex8
<CSP>                   ::= 0x2c [<TSP>]*
<TSP>                   ::= {0x20} {0x09}
<CformatGuid>           ::= "{" [<TSP>]* <Hex32> <CSP> <Hex16> <CSP> <Part2>
<Part2>                 ::= <Hex16> <CSP> "{" [<TSP>]* <Byte> <CSP> <Part3>
<Part3>                 ::= <Byte> <CSP> <Byte> <CSP> <Byte> <CSP> <Part4>
<Part4>                 ::= <Byte> <CSP> <Byte> <CSP> <Byte> <CSP> <Part5>
<Part5>                 ::= <Byte> [<TSP>]* "}" [<TSP>]* "}"
<Array>                 ::= {<EmptyArray>} {<Array>}
<EmptyArray>            ::= "{" <TSP>* "}"
<ByteArray>             ::= "{" <TSP>* <Byte> [<CSP> <Byte>]* "}"
<StringLiteral>         ::= {<QuotedString>} {"L" <QuotedString>}
<DblQuote>              ::= 0x22
<QuotedString>          ::= <DblQuote> [<CCHAR>]* <DblQuote>
<CCHAR>                 ::= {<SingleChars>} {<EscapeCharSeq>}
<SingleChars>           ::= {0x20} {0x21} {(0x23 - 0x5B)} {(0x5D - 0x7E)}
<EscapeCharSeq>         ::= "\" {"n"} {"r"} {"t"} {"f"} {"b"} {"0"} {"\"} {<DblQuote>}
<PostFixExpression>     ::= {<PrimaryExpression>} {<PcdName>}
<PcdName>               ::= <CName> "." <CName>
<UnaryExpression>       ::= {<PostFixExpression>} {<UnaryOp> <UnaryExpression>}
<UnaryOp>               ::= {<IntegerOps>} {<ScalarOps>}
<IntegerOps>            ::= {"+"} {"-"} {"~"}
<ScalarOps>             ::= {"NOT"} {"not"} {"!"}
<MultiplicativeExpress> ::= {<UnaryExpression>}
                            {<MultiplicativeExpress> <TSP>* "*" <TSP>* <UnaryExpression>}
                            {<MultiplicativeExpress> <TSP>* "/" <TSP>* <UnaryExpression>}
                            {<MultiplicativeExpress> <TSP>* "%" <TSP>* <UnaryExpression>}
<AdditiveExpress>       ::= {<MultiplicativeExpress>}
                            {<AdditiveExpress> <TSP>* "+" <TSP>* <MultiplicativeExpress>}
                            {<AdditiveExpress> <TSP>* "-" <TSP>* <MultiplicativeExpress>}
<ShiftExpression>       ::= {<AdditiveExpress>}
                            {<ShiftExpression> <TSP>* "<<" <TSP>* <AdditiveExpress>}
                            {<ShiftExpression> <TSP>* ">>" <TSP>* <AdditiveExpress>}
<RelationalExpress>     ::= {<ShiftExpression>}
                            {<RelationalExpress>} <TSP>* "<" <TSP>* <ShiftExpress>}
                            {<RelationalExpress>} <TSP>* "LT" <TSP>* <ShiftExpress>}
                            {<RelationalExpress>} <TSP>* ">" <TSP>* <ShiftExpress>}
                            {<RelationalExpress>} <TSP>* "GT" <TSP>* <ShiftExpress>}
                            {<RelationalExpress>} <TSP>* "<=" <TSP>* <ShiftExpress>}
                            {<RelationalExpress>} <TSP>* "LE" <TSP>* <ShiftExpress>}
                            {<RelationalExpress>} <TSP>* ">=" <TSP>* <ShiftExpress>}
                            {<RelationalExpress>} <TSP>* "GE" <TSP>* <ShiftExpress>}
<EqualityExpression>    ::= {<RelationalExpress>}
                            {<EqualityExpression> <TSP>* "==" <TSP>* <RelationalExpress>}
                            {<EqualityExpression> <TSP>* "EQ" <TSP>* <RelationalExpress>}
                            {<EqualityExpression> <TSP>* "!=" <TSP>* <RelationalExpress>}
                            {<EqualityExpression> <TSP>* "NE" <TSP>* <RelationalExpress>}
<BitwiseAndExpression>  ::= {<EqualityExpression>}
                            {<BitwiseAndExpression> <TSP>* "&" <TSP>* <EqualityExpression>}
<BitwiseXorExpress>     ::= {<BitwiseAndExpression>}
                            {<BitwiseXorExpress> <TSP>* "^" <TSP>* <BitwiseAndExpression>}
<BitwiseOrExpress>      ::= {<BitwiseXorExpress>}
                            {"(" <BitwiseOrExpress> <TSP>* "|" <TSP>* <BitwiseXorExpress> ")"}
<LogicalAndExpress>     ::= {<BitwiseOrExpress>}
                            {<LogicalAndExpress> <TSP>* "&&" <TSP>* <BitwiseOrExpress>}
                            {<LogicalAndExpress> <TSP>* "AND" <TSP>* <BitwiseOrExpress>}
                            {<LogicalAndExpress> <TSP>* "and" <TSP>* <BitwiseOrExpress>}
<LogicalXorExpress>     ::= {<LogicalAndExpress>}
                            {<LogicalAndExpress> <TSP>* "XOR" <TSP>* <LogicalXorExpress>}
                            {<LogicalAndExpress> <TSP>* "xor" <TSP>* <LogicalXorExpress>}
<LogicalOrExpress>      ::= {<LogicalXorExpress>}
                            {"(" <LogicalXorExpress> <TSP>* "||" <TSP>* <LogicalOrExpress> ")"}
                            {<LogicalXorExpress> <TSP>* "OR" <TSP>* <LogicalOrExpress>}
                            {<LogicalXorExpress> <TSP>* "or" <TSP>* <LogicalOrExpress>}
```

```
<CondExpress>          ::= {<LogicalOrExpress>}
                           {<LogicalOrExpress> <TSP>* "?" <IsTrue> ":" <TSP>* <CondExpress>
<IsTrue>               ::= <TSP>* <Expression> <TSP>*
<Expression>           ::= {<CondExpress>} {<Expression>}
```

## Example

```
gUefiCpuPkgTokenSpaceGuid.PcdCpuLocalApicBaseAddress | !gCrownBayTokenSpaceGuid.PcdProgrammableLocalApic
```

## Notes

**MACROVAL**

This is the value of the MACRO assigned in a DEFINE statement.

**Expressions**

If the "|" character is used in an expression, the expression must be encapsulated by parenthesis.

# 3.2 Conditional Directive Expressions

Conditional directive statements are defined in the EDK II Platform Description (DSC) File and Flash Definition (FDF) File. The following EBNF describes the format for expressions used in conditional directives. The format is based on section 6.10 of the ANSI C-99 Specification.

## Restrictions

**ConstantExpression**

The ConstantExpression in the !if statement must evaluate to either True (1) or False (0).

## Prototype

```
<EOL>                ::= 0x0D 0x0A
<TSP>                ::= {0x20} {0x09}
<Group>              ::= {<GroupPart>} {<Group> <GroupPart>}
<GroupPart>          ::= {<IfSection>} {<TextLine>}
<IfSection>          ::= <IfGroup> [<ElifGroups>] [<ElseGroup>] <EndIfLine>
<IfGroup>            ::= {<TSP>* "!if" <TSP>+ <ConstantExpression> <EOL> [<Group>]
                           {<TSP>* "!ifdef" <TSP>+ <MACROorMACROVAL> <EOL> [<Group>]
                           {<TSP>* "!ifndef" <TSP>+ <MACROorMACROVAL> <EOL> [<Group>]
<MACROorMACROVAL>    ::= {<MACRO>} {"$(" <MACRO> ")"}
<ElifGroup>          ::= {<ElifGroup>} {<ElifGroups> <ElifGroup>}
<ElifGroup>          ::= <TSP>* "!elif" <TSP>* <ConstantExpression> <EOL> [<Group>]
<ElseGroup>          ::= <TSP>* "!else" <TSP>* <EOL> [<Group>]
<EndIfLine>          ::= <TSP>* "!endif" <EOL>
<TextLine>           ::= Content specific to the meta-data file
<ConstantExpression> ::= <CondExpress> ; see Data Field Expression definitions
```

## Example

```
!if $(LOGGING)
  DebugLib|IntelFrameworkModulePkg/Library/PeiDxeDebugLibReportStatusCode/PeiDxeDebugLibReportStatusCode.inf
  DebugPrintErrorLevelLib|MdePkg/Library/BaseDebugPrintErrorLevelLib/BaseDebugPrintErrorLevelLib.inf
!else
  DebugLib|MdePkg/Library/BaseDebugLibNull/BaseDebugLibNull.inf
!endif

!if $(LOGGING) == FALSE
```

```
  gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask|0x0
  gEfiMdePkgTokenSpaceGuid.PcdReportStatusCodePropertyMask|0x3
!endif
```

# APPENDIX A ABNF SYNTAX

This section provides the Augmented Backus Naur Form of the Expression Format.

## A.1 Data Field Expression ABNF

```
PrimaryExpression     ::= Identifier / Constant / StringLiteral / "(" Expression ")" / Function
Identifier            ::= CName / MacroValue
MacroValue            ::= "$(" MACRO ")"
MACRO                 ::= (A-Z) *(A-Z0-9_)
Function              ::= CName "(" Argument *(CSP Argument) ")"
Argument              ::= PrimaryExpression
CName                 ::= (a-zA-Z_) *(a-zA-Z0-9_)
Constant              ::= TrueFalse / Number / GuidValue / Array
TrueFalse             ::= True / False
True                  ::= "TRUE" / "True" / "true"
False                 ::= "FALSE" / "False" / "false"
Number                ::= Integer / HexNumber
Integer               ::= Base10
Base10                ::= (0-9) / ((1-9) *(0-9))
HexNumber             ::= Base16
Base16                ::= HexPrefix *(HexDigit) HexDigit
HexDigit              ::= (a-fA-F0-9)
HexPrefix             ::= "0x" / "0X"
GuidValue             ::= RformatGuid / CformatGuid
Rhex2                 ::= [HexDigit] HexDigit
Rhex4                 ::= [HexDigit] [HexDigit] Rhex2
Rhex8                 ::= [HexDigit] [HexDigit] [HexDigit] [HexDigit] Rhex4
Rghex2                ::= HexDigit HexDigit
Rghex4                ::= HexDigit HexDigit Rghex2
Rghex8                ::= HexDigit HexDigit HexDigit HexDigit Rghex4
Rghex12               ::= HexDigit HexDigit HexDigit HexDigit Rghex8
RformatGuid           ::= Rghex8 "-" Rghex4 "-" Rghex4 "-" Rghex4 "-" Rghex12
Byte                  ::= HexPrefix Rhex2
Hex16                 ::= HexPrefix Rhex4
Hex32                 ::= HexPrefix Rhex8
CSP                   ::= %x2c *(TSP)
TSP                   ::= %x20 / %x09
CformatGuid           ::= "{" *(TSP) Hex32 CSP Hex16 CSP Hex16 CSP Part2
Part2                 ::= "{" *(TSP) Byte CSP Byte CSP Byte CSP Byte CSP Part3
Part3                 ::= Byte CSP Byte CSP Byte CSP Byte *(TSP) "}" *(TSP) "}"
Array                 ::= EmptyArray / Array
EmptyArray            ::= "{" *(TSP) "}"
ByteArray             ::= "{" *(TSP) Byte *(CSP Byte) "}"
StringLiteral         ::= QuotedString / "L" QuotedString
DblQuote              ::= %x22
QuotedString          ::= DblQuote *(CCHAR) DblQuote
CCHAR                 ::= SingleChars / EscapeCharSeq
SingleChars           ::= %x20 / %x21 / %x23-5B / %x5D-7E
EscapeCharSeq         ::= "\" ("n" / "r" / "t" / "\" / "f" / "b" / "0" / DblQuote)
PostFixExpression     ::= PrimaryExpression / PcdName
PcdName               ::= CName "." CName
UnaryExpression       ::= PostFixExpression / UnaryOp UnaryExpression
UnaryOp               ::= IntegerOps / ScalarOps
IntegerOps            ::= "+" / "-" / "~"
ScalarOps             ::= "NOT" / "not" / "!"
MultiplicativeExpress ::= UnaryExpression /
                          ( MultiplicativeExpress *(TSP) "*" *(TSP) UnaryExpression ) /
                          ( MultiplicativeExpress *(TSP) "/" *(TSP) UnaryExpression ) /
                          ( MultiplicativeExpress *(TSP) "%" *(TSP) UnaryExpression )
AdditiveExpress       ::= MultiplicativeExpress /
                          ( AdditiveExpress *(TSP) "+" *(TSP) MultiplicativeExpress ) /
                          ( AdditiveExpress *(TSP) "-" *(TSP) MultiplicativeExpress )
ShiftExpression       ::= AdditiveExpress /
                          ( ShiftExpression *(TSP) "<<" *(TSP) AdditiveExpress )
```

```
                        ( ShiftExpression *(TSP) ">>" *(TSP) AdditiveExpress )
RelationalExpress     ::= ShiftExpression /
                        ( RelationalExpress *(TSP) "<" *(TSP) ShiftExpress ) /
                        ( RelationalExpress *(TSP) ">" *(TSP) ShiftExpress ) /
                        ( RelationalExpress *(TSP) "<=" *(TSP) ShiftExpress ) /
                        ( RelationalExpress *(TSP) ">=" *(TSP) ShiftExpress)
EqualityExpression    ::= RelationalExpress /
                        ( EqualityExpression *(TSP) "==" *(TSP) RelationalExpress ) /
                        ( EqualityExpression *(TSP) "EQ" *(TSP) RelationalExpress ) /
                        ( EqualityExpression *(TSP) "!=" *(TSP) RelationalExpress ) /
                        ( EqualityExpression *(TSP) "NE" *(TSP) RelationalExpress ) /
BitwiseAndExpression  ::= EqualityExpression /
                        ( BitwiseAndExpression *(TSP) "&" *(TSP) EqualityExpression )
BitwiseXorExpress     ::= BitwiseAndExpression /
                        ( BitwiseXorExpress *(TSP) "^" *(TSP) BitwiseAndExpression )
BitwiseOrExpress      ::= BitwiseXorExpress /
                        ( "(" BitwiseOrExpress *(TSP) "|" *(TSP) BitwiseXorExpress ")" )
LogicalAndExpress     ::= BitwiseOrExpress /
                        ( LogicalAndExpress *(TSP) "&&" *(TSP) BitwiseOrExpress ) /
                        ( LogicalAndExpress *(TSP) "AND" *(TSP) BitwiseOrExpress /
                        ( LogicalAndExpress *(TSP) "and" *(TSP) BitwiseOrExpress )
LogicalXorExpress     ::= LogicalAndExpress /
                        ( LogicalAndExpress *(TSP) "XOR" *(TSP) LogicalXorExpress ) /
                        ( LogicalAndExpress *(TSP) "xor" *(TSP) LogicalXorExpress )
LogicalOrExpress      ::= LogicalXorExpress /
                        ( "(" LogicalXorExpress *(TSP) "||" *(TSP) LogicalOrExpress ")" ) /
                        ( LogicalXorExpress *(TSP) "OR" *(TSP) LogicalOrExpress ) /
                        ( LogicalXorExpress *(TSP) "or" *(TSP) LogicalOrExpress )
CondExpress           ::= LogicalOrExpress /
                        ( LogicalOrExpress *(TSP) "?" IsTrue ":" *(TSP) CondExpress )
IsTrue                ::= *(TSP) Expression *(TSP)
Expression            ::= CondExpress / Expression
```

## A.2 Conditional Directive Expression ABNF

```
EOL                ::= %x0D.0A
TSP                ::= %x20 / %x09
Group              ::= GroupPart / ( Group GroupPart )
GroupPart          ::= IfSection / TextLine
IfSection          ::= IfGroup [ElifGroups] [ElseGroup] EndIfLine
IfGroup            ::= ( *(TSP) "!if" 1*(TSP) ConstantExpression EOL [Group] ) /
                       ( *(TSP) "!ifdef" 1*(TSP) MACROorMACVAL EOL [Group] ) /
                       ( *(TSP) "!ifndef" 1*(TSP) MACROorMACVAL EOL [Group]
MACROorMACVAL      ::= MACRO / MacroValue
ElifGroups         ::= ElifGroup / ( ElifGroups ElifGroup )
ElifGroup          ::= *(TSP) "!elif" 1*(TSP) ConstantExpression EOL [Group]
ElseGroup          ::= *(TSP) "!else" *(TSP) EOL [Group]
EndIfLine          ::= *(TSP) "!endif" EOL
TextLine           ::= Content specific to the meta-data file
ConstantExpression ::= CondExpress ; see Data Field Expression definitions
```