EDK II Multi-String .UNI File Format Specification

TABLE OF CONTENTS

EDK II Multi-String .UNI File Format Specification

DRAFT FOR REVIEW

12/01/2020 05:50:21

Acknowledgements

Revision History

Revision	Description	Date
1.0	Initial Release.	February 2014
1.1	Updated EBNF to follow syntax specified in EBNF by the ANTLR project.	August 2014
	Added content related to EDK II Meta-Data Unicode files.	
	Restructured document.	
	Removed security and C format GUID definitions, not required for HII or other UNI files.	
	Removed invalid escape code sequences.	
1.2	Added optional font formatting	September 2014
1.2 Errata A	Correct misspelling of: `STR_PROPERTIES_MODULE_NAME`	April 2015
1.3	Added: Syntax for non-ascii characters inside quoted strings.	March 2016

1.4	Convert to GitBook format	April 2017
	#506 UNI Spec: Clean up Related Information section	
	#507 UNI Spec: Clarify that .uni files maybe UTF-8 without a BOM	

1 INTRODUCTION

This document describes file format for Unicode string files. This file format supports multiple layouts and formats in the Unicode file. This versatility allows strings to be grouped either by language or by string identifier.

1.1 Related Information

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

- Unified Extensible Firmware Interface Specifications, http://uefi.org/specifications.
- http://www.tianocore.org/docs/EDK_II_Documents.html
  - EDK II Build Specification, Intel, 2016.
  - EDK II DEC File Specification, Intel, 2016.
  - EDK II INF File Specification, Intel, 2016.
  - EDK II DSC File Specification, Intel, 2016.
  - EDK II FDF File Specification, Intel, 2016.
  - EDK II Expression Syntax Specification, Intel, 2015.
  - EDK II C Coding Standards Specification, Intel, 2015.

1.2 Terms

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

BDS

Framework Boot Device 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.

DXE SAL

Framework Driver Execution Environment phase. A special class of DXE module that produces 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 module that is loaded into the System Management Mode memory.

DXE Runtime

Special class of DXE module that provides Runtime Services

EFI

Generic term that refers to one of the versions of the EFI specification: EFI 1.02, EFI 1.10, or UEFI 2.0.

EFI 1.10 Specification

Intel Corporation published the Extensible Firmware Interface Specification. Intel donated the EFI specification to the Unified EFI Forum, and the UEFI now owns future updates of the EFI specification. See UEFI Specifications.

Foundation

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

Framework

Intel(R) Platform Innovation Framework for EFI consists 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. An individual without the help of a centralized authority can generate a unique GUID. This allows the generation of names that will never conflict, even among multiple, unrelated parties.

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.

IFR

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

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. See the EDK 2.0 Module Development Environment Library Specification for a list of libraries defined in this package.

Library Instance

An implementation of one or more library classes. See the EDK 2.0 Module Development Environment Library Specification for a list of library defined in this package.

Module

A module is either an executable image or a library instance. For a list of module types supported by this 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 module type `BASE`, depends only on headers and libraries provided in the MDE, while a module that is of module type `DXE_DRIVER` depends on common DXE components. For a definition of the various module types, see module type.

Module Surface Area (MSA)

The MSA is an XML description of how the module is coded. The MSA contains information about the different construction options for the module. After the module is constructed the MSA can describe the interoperability requirements of a module.

Package

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

- Source modules, containing all source files and descriptions of a module
- Binary modules, containing EFI Sections or 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.

Protocol

An API named by a GUID as defined by the EFI specification.

PCD

Platform Configuration Database.

PEI

Pre-EFI Initialization Phase.

PPI

A PEIM-to-PEIM Interface that is named by a GUID as defined by the PEI CIS.

SAL

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

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.

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.

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 Specification Version 2.0

Current version of the EFI specification released by the Unified EFI Forum. This specification builds on the EFI 1.10 specification and transfers ownership of the EFI specification from Intel to a non-profit, industry trade organization.

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

1.3 Conventions used in this document

This document uses the typographic and illustrative conventions described below.

1.3.1 Pseudo-code conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a list is an unordered collection of homogeneous objects. A queue is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the Extensible Firmware Interface Specification.

1.3.2 Typographic conventions

This document uses the typographic and illustrative conventions described below:

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 cross-reference. Click on the word to follow the hyperlink. USE ONLY IF YOU MAKE AN ACTUAL CROSS-REFERENCE LINK.
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. or as a definition heading (GlossTerm)
Italic	In text, an Italic typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`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.
`Monospace (blue)`	Words in a `Monospace` typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. USE ONLY IF YOU MAKE AN ACTUAL HYPERLINK.
Italic Bold	In code or in text, words in Italic Monospace indicate placeholder names for variable information that must be supplied (i.e., arguments).

2 UNICODE STRINGS FILE FORMAT

EDK II Unicode files are used for mapping token names to localized strings that are identified by an RFC4646 language code. The format for storing EDK II Unicode files on disk is UTF-8 (without a BOM character) or UTF-16LE (with a BOM character). The character content must be UCS-2.

Strings ends are determined by the first of the following items found:

- a control character
- a comment
- the end of the file
- a blank line

Comments may appear anywhere within the string file.

All UTF-16LE files must begin with a Unicode BOM character. All UTF-8 files must not begin with a Unicode BOM character.

NOTE: Please make sure you select an editor that supports UCS-2 characters that can be stored in either a UTF-8 (without a BOM character) or a UTF-16LE file (with a BOM character).

2.1 Common EBNF

The following EBNF uses quoted (double quotes) encapsulated characters to represent UCS-2 string literals. In the following definitions, the semi-colon is used to denote a comment.

```
<US>           ::= " "
<Letter>       ::= {(\u0041-\u005A)} ; Characters A - Z
                   {(\u0061-\u007A)} ; Characters a - z
<Digit>        ::= (\u0030-\u0039)   ; Characters 0 - 9
<MS>           ::= <US>+
<ME>           ::= {<MS>} {<EOL>}
<CommentLine>  ::= "//" <US>* <PCHars> <EOL>
<BlankLine>    ::= <EOL>
<Chars>        ::= (\u0001-\uF6FF)
<PChars>       ::= {(\u0020-\uF6FF)} {<OpChar>}
<OpChars>      ::= "\x" [{<Letter>} {<Digit>}]{4} "\"
<VChars>       ::= (\u0021-\uF6FF)
<UnicodeLines> ::= <Token> <ME>
                   [<Ldef> [<String> <ME>]+]+
<Ldef>         ::= <CtrlChar> "language" <MS> <LangCode> <ME>
<HexDigit>     ::= {<Digit>}
                   {(\u0041-\u0046)} ; Characters A - F
                   {(\u0061-\u0066)} ; Characters a - f
<CtrlChar>     ::= <US>* "#"
<Token>        ::= <CtrlChar> "string" <MS> <Identifier>
<Identifier>   ::= <Letter> [{<Letter>} {<Digit>} {<UN>}]*
<LangCode>     ::= <RFC4646>
<RFC4646>      ::= <Letter>{2,8} [<ShortExt> <LongExt>*]
<ShortExt>     ::= "-" [{<Letter>} {<Digit>}]{1,8}
<LongExt>      ::= "-" [{<Letter>} {<Digit>}]{1,}
<UDblQuote>    ::= \u0022  ; Double Quote Character, "
<String>       ::= <UDblQuote> <SContent>* <UDblQuote>
<SContent>     ::= {<PChars>} {<Attributes>}
<Attributes>   ::= "\" {"narrow"} {"wide"} {<UDblQuote>}
                   {"n"} {"r"} {"t"} {"nbr"} {"\"} {"'"}
```

2.1.1 Definitions

LanguageCodes

The language code must be a valid RFC4646 language code.

EscChar

In order to include some standard characters, such as the "\" back-slash character within a string, the character must be prefixed with the escape character. Characters that may require a prefixed escape character include the following, back slash "\" character, single-quote "'" character, double-quote '"' character and the forward slash "/" character. The back slash always requires the escape character.

Token

The token (strong identifier) may only contain numbers, upper and lower case letters, underscore character, and dash character.

Include

An include line is used to parse another file, also compliant with this specification, as if it was in the file. The tokens should not overlap between the file for the same language.

3 HII STRING PACKS

Unicode files used for creating HII String Packs have the following format:

```
<StringFileFormat> ::= <CommentLine>*
                       <LanguageDefs>
                       <Content>+
```

The following EBNF describes content is specific to the Unicode files used for generating HII String Packs.

```
<Content> ::= {<CommentLine>} {<BlankLine>}
              {<UnicodeLines>} {<ControlRefactor>}
              {<LanguageDefs>} {<SecurityLines>}
              {<IncludeLines>}
```

Additional Definitions used for Unicode files used to create HII String Packs.

```
<LanguageDefs>    ::= <CtrlChar> "langdef" <MS> <LangCode> <MS>
                      <LangDesc> <EOL>
<LangDesc>        ::= <UDblQuote> <Chars> <UDblQuote>
<IncludeLines>    ::= <CtrlChar> "include" <UniFile> <EOL>
<UniFile>         ::= <UDblQuote> <UniFilename> <UDblQuote>
<UniFilename>     ::= <FilenameChars> <MoreFNameChars>* {".uni"} {".UNI"}
<FilenameChars>   ::= {<Letter>} {<Digit>}
<MoreFNameChars>  ::= {<Letter>} {<Digit>} {"_"}
<CtrlChar>        ::= "/"
<ControlRefactor> ::= <CtrlChar> "=" <NewCtrlChar> <EOL>
<NewCtrlChar>     ::= (0x0021 - 0xF6FF)
```

NOTE: Unicode files that are used for generating HII String Packs are the only type of Unicode file that allows for refactoring the control character (providing backward compatibility), `<CtrlChar>`.

3.1 Example file

```
//
// Cpu I/O Strings
//
// Copyright (c) 2006, Intel Corporation. All rights reserved.<BR>
//
// This program and the accompanying materials are licensed and made
// available under the terms and conditions of the BSD License which
// accompanies this distribution. The full text of the license may
// be found at:
//    http://opensource.org/licenses/bsd-license.php
//
// THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
// WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS
// OR IMPLIED.
//

/= #
#langdef  en-US  "English, US"
#langdef  fr-FR  "Français"

#string STR_PROCESSOR_VERSION
#language en-US "NT32 Emulated Processor"
```

```
#language fr-FR "Processeur Émulé par NT32"
```

4 REDACTED

Formerly "Meta-Data UNI Files". This chapter intentionally removed.

5 FONT SUPPORT

This chapter defines the optional attributes and entries in EDK II Unicode files to support font selection.

Syntax

The following sections describe extensions to the .uni format.

The extensions add support for fonts into the .uni format by introducing the `#fontdef` and `#font` commands, extending the `#string` command and adding new escape characters into the strings.

Fonts

Every string is associated with a font. Each font has a font identifier, a font name, a size (in pixel height) and a style (normal, bold, etc.). By default, strings will be associated with the font identifier `sysdefault`. Usually this is associated with the font sysdefault, 19, normal (the standard UEFI font).

The default font associated with strings can be changed from sysdefault to another font identifier using the `#font` command. All strings after the `#font` command will use the specified font identifier.

Strings can use a different `#font` identifier by using the font attribute of the `#string` command (before the first `#language` attribute). A string in a specific language can use a different font identifier by using the `#font` attribute after the language attribute.

Characters within a string can use a different font identifier, a different font size or a different font style by using the `\f` escape sequences described below. These escape characters extend those described in the EDK2 Build Specification.

Table 1 .uni File Font Escape Characters

Font Control Character	Description
\"	Insert a double-quote.
\\	Insert a single backslash.
\br	Breaking code.
\f!identifier!	Select the font identifier for the characters which follow.
\fb	Toggle the current bold style for characters that follow in the current string.
\fd	Toggle the current double-underline style. If the current style is underline, the style becomes double-underline.
\fe	Toggle the current emboss style for the characters that follow.
\fh!integer!	Select the font size (in pixels) for the characters that follow.
\fi	Toggle the current italic style for the characters that follow in the current string.
\fs	Toggle the current shadow style for the characters that follow in the current string.
\fu	Toggle the current underline style for the characters that follow in the current string.
\n	Insert a carriage-return and line-feed.
\narrow	Display the following characters as "narrow" characters.

\nbr	Non-breaking code.
\r	Insert a carriage-return.
\wide	Display the following characters as "wide" characters

Font identifiers are created by using the `#fontdef`.

5.1 #font

Set the default font to use with all subsequent `#strings`.

Syntax

```
"#font" <MS> font-identifier
```

Attributes

font-identifier

C style identifier associated with the font.

5.2 #fontdef

Associated a font identifier with a specific font family, size and style.

Syntax

```
"#fontdef" <MS> _font-identifier_ <MS> <FontOptions> <EOL>

<FontOptions>   ::= font-name <MS> font-size [<MS> font-style-list]
font-style-list ::= <UDblQuote> [fs-entries] <UDblQuote>
fs-entries      ::= font-style ["|" font-style]*
font-style      ::= {"bold"} {"italic"} {"underline"} {"dblunder"}
                    {"shadow"} {"emboss"} {"normal"}
font-size       ::= (1-9) (0-9)*
```

Attributes

font-identifier

C-style identifier.

font-name

Quoted string that specifies a font family name. For example, "Arial" or "Times New Roman"

font-size

Unsigned integer that specifies the height of the font character cell, in pixels. For example, the UEFI standard font is size 19 because the cell is 19 pixels high.

font-style

Quoted string that contains zero or more keywords that specify the font style, separated by a "|". If "normal" is used, then it may not be combined with any other font style. If there is no font style specified, then "normal" is assumed.

5.3 #string Extensions

The EDK II build command is responsible for parsing the .uni files specified in INF files' `[Sources]` sections. The tool uses python objects to convert the syntax in the HII string files to byte arrays in the AutoGen.c file for each module.

Refer to the EDK II Build Specification for details on the process of creating the byte arrays.

Syntax

```
<UnicodeLines> ::= "#string" <MS> <Identifier> <ME>
                   [<FontId>]
                   [<LangLine>]+
<LangLine>     ::= "#language" <MS> lang-code <ME> <FontString>
<FontString>   ::= [<FontId>] [<strings>]+
<FontId>       ::= ["#font" <MS> font-identifier> <ME>]
<strings>      ::= <String> <ME>
```

[Extension to #string command in the EDK2 Build Specification]

The font attribute specifies the default font that will be used for the characters in string. If `#font` is not specified, then the default font identifier will be used.

If the `#font` attribute appears before the first `#language` identifier, then it applies to all characters for all languages. If the `#font` attribute appears after a `#language` identifier, it applies only to the string characters in that language. It is permissible for `#font` to appear in more than one place, in which case the language-specific font identifier will have priority.

Description

The `#fontdef` command introduces a font identifier and associates it with a font of a particular family, size and style. If the font identifier has been previously defined, then the new definition is ignored.