intel®

Intel® Platform Innovation Framework for EFI Human Interface Infrastructure Specification

Version 0.9

September 16, 2003

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Except for a limited copyright license to copy this specification for internal use only, no license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this specification. Intel does not warrant or represent that such implementation(s) will not infringe such rights.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

Intel, the Intel logo, and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2000–2003, Intel Corporation.

Revision History

Revision	Revision History	Date
0.9	First public release.	9/16/03

Contents

1 Introduction 9
Purpose 9
Overview 10
Glossary 10
References 12
Conventions Used in This Document 13
Data Structure Descriptions 13
Protocol Descriptions 14
Procedure Descriptions 14
Pseudo-Code Conventions 15
Typographic Conventions 15
2 Design Discussion 17
Design Rationale 17
Introduction 17
String Management 17
Localization Issues 19
User Input 20
Forms 21
Human Interface Overview 22
Introduction 22
Package Header 23
Package Manipulation 23
Packages Definition 23
Human Interface Infrastructure (HII) Protocol 23
Font Package 23
Introduction 23
Glyph Sizes 23
Glyph Representation 24
Strings 24
Introduction 24
Internal String Representation 24
Form Packages 25
Goals 25
Forms and Form Sets 27
Semantics and Tag Structures 27
One-Of 28
Checkbox: <checkbox> 29
Numeric: <numeric> 29
Password: <password> 29
Hidden: <hidden> 29
Ordering: <list> 29
Hypertext: <goto> 31

Dynamic Data ... 34
Advanced Operations (Optional) ... 34
Dynamic Processing of NV/IFR Data ... 35
Form Callback Protocol ... 35
Browser Interface ... 35
Form Browser Protocol ... 35
Runtime Representations ... 35
Using IFR at Runtime ... 35
Limitations of Presentation Mechanisms ... 36

3 Code Definitions ... 37

Packages ... 37
Package Header ... 37
EFI_HII_PACK_HEADER ... 37
Packages Definition ... 38
EFI_HII_PACKAGES ... 38
Human Interface Infrastructure (HII) Protocol ... 40
EFI_HII_PROTOCOL ... 40
EFI_HII_PROTOCOL.NewPack() ... 43
EFI_HII_PROTOCOL.RemovePack() ... 44
EFI_HII_PROTOCOL.FindHandles() ... 45
EFI_HII_PROTOCOL.ExportDatabase() ... 46
Font Package ... 50
Glyph Representation ... 50
EFI_NARROW_GLYPH ... 50
EFI_WIDE_GLYPH ... 51
EFI_HII_FONT_PACK ... 52
HII Protocol Font-Related Entries ... 53
EFI_HII_PROTOCOL (Font-Related Entries) ... 53
EFI_HII_PROTOCOL.NewPack() ... 54
EFI_HII_PROTOCOL.TestString() ... 55
EFI_HII_PROTOCOL.GetGlyph() ... 56
EFI_HII_PROTOCOL.GlyphToBlt() ... 58
Strings ... 60
String ... 60
EFI_STRING ... 60
String Package Structure ... 61
EFI_HII_STRING_PACK ... 61
HII Protocol String Functions ... 64
EFI_HII_PROTOCOL (String Functions) ... 64
EFI_HII_PROTOCOL.NewPack() ... 66
EFI_HII_PROTOCOL.NewString() ... 67
EFI_HII_PROTOCOL.GetPrimaryLanguages() ... 68
EFI_HII_PROTOCOL.GetSecondaryLanguages() ... 69
EFI_HII_PROTOCOL.GetString() ... 70
EFI_HII_PROTOCOL.GetLine() ... 72

Form Packages....74
Form Language Syntax....74
Meta-Syntax....74
Internal Form Representation (IFR) Language Syntax Definition....75
EFI_IFR_OP_HEADER....75
Form Package Syntax....77
Form Tag....78
EFI_IFR_SUBTITLE....79
EFI_IFR_TEXT....80
EFI_IFR_ONE_OF....81
EFI_IFR_CHECKBOX....83
EFI_IFR_NUMERIC....85
EFI_IFR_PASSWORD....87
EFI_IFR_ORDERED_LIST....89
EFI_IFR_REF....91
EFI_IFR_HIDDEN....92
EFI_IFR_GRAY_OUT....93
EFI_IFR_SUPPRESS....94
EFI_IFR_INCONSISTENT....95
EFI_IFR_LABEL....96
EFI_IFR_VARSTORE....97
EFI_IFR_VARSTORE_SELECT....98
EFI_IFR_VARSTORE_SELECT_PAIR....99
Boolean Expressions....100
EFI HII Protocol Forms Entries....102
EFI_HII_PROTOCOL (Forms Entries)....102
EFI_HII_PROTOCOL.NewPack()....104
EFI_HII_PROTOCOL.GetForms()....105
EFI_HII_PROTOCOL.UpdateForm()....107
Dynamic Processing of NV/IFR Data....108
Form Callback Protocol....108
EFI_FORM_CALLBACK_PROTOCOL....108
EFI_FORM_CALLBACK_PROTOCOL.NvRead()....111
EFI_FORM_CALLBACK_PROTOCOL.NvWrite()....113
EFI_FORM_CALLBACK_PROTOCOL.CallBack()....115
Browser Interface....117
Form Browser Protocol....117
EFI_FORM_BROWSER_PROTOCOL....117
EFI_FORM_BROWSER_PROTOCOL.SendForm()....119
EFI_FORM_BROWSER_PROTOCOL.CreatePopUp()....121
Appendix A Conventions for IFR to HTML Translation....123

Figures

Figure 2-1. Managing Human Interface Components 22

Tables

Table 2-1. Localization Issues 19
Table 2-2. Differences between HTML and IFR...21
Table 3-1. Value Passed in the `Data` Pointer 116
Table A-2. Suggested Translations between IFR and HTML 123

1 Introduction

Purpose

This document describes the mechanisms by which the Intel® Platform Innovation Framework for EFI (the "Framework") manages user input. The major areas described include the following:

- String and font management.
- User input abstractions (for keyboards and mice), mainly those used during the Driver Execution Environment (DXE) and Boot Device Selection (BDS) phases.
- Internal representations of the *forms* (in the HTML sense) that are used for running a preboot setup
- External representations, and the derivations of those representations, of the forms that are used to pass configuration information to runtime applications and the mechanisms to allow the results of those applications to be driven back into the firmware.

General goals include:

- Simplified *localization,* the process by which the interface is adapted to a particular language.
- A "forms" representation mechanism that is rich enough to support the complex configuration issues encountered by platform developers, including stock keeping unit (SKU) management and interrelationships between questions in the forms.
- Definition of a mechanism to allow most or all the configuration of the system to be performed during boot (DXE/BDS), at runtime, and remotely. Where possible, the forms describing the configuration should be expressed using existing standards such as XML.
- Ability for the different drivers (including those from add-in cards) and applications to contribute forms, strings, and fonts in a uniform manner while still allowing innovation in the look and feel for Setup.
- Encourage a "walk up and use" (WUU) user interface. Most applications are designed to be used repeatedly. User interface designers must trade off learnability for usability. The goal of WUU applications is to be instantly usable without a learning curve or other documentation. Design characteristics include the following:
    - — A simplified interface.
    - — Continual display of both keys and context-sensitive help, rather than having the user ask for it.
    - — Minimal shortcuts (most people become confused by more than one method for doing things).
    - — An interface that is analogous to a common interface. At this time, a generic web browser is probably the most universal nonproprietary interface.

Overview

This document describes the following:

- General design rationale and concepts
- Data structures. They are described more or less bottom up, in the following order:
  - Fonts
  - Strings
  - Internal Form Representations (IFRs)
  - Mechanisms to map internal representations
  - Mechanisms to map to external representations (such as XHTML).
- Code interfaces

It is important to note which concepts are required by the architecture and which are considered possible implementations. In general, all of the definitions expressed in the Extensible Firmware Interface (EFI) standard protocol/member function format are architectural. Except where noted, database information and representations are architectural. The tools are not architectural, nor is, of course, the rationale. Variances from these general rules are noted.

Glossary

The following definitions, except where noted, are not EFI specific. See the master glossary in the Framework Interoperability and Component Specifications help system for additional terms; see "Typographic Conventions" later in this chapter for the URL.

DBCS

Double Byte Character Set.

font

A graphical representation corresponding to a character set, in this case Unicode. The following are the same Latin letter in three fonts using the same size (14):

A

A

A

font glyph

The individual elements of a font corresponding to single characters are called *font glyphs* or simply *glyphs*. The first character in each of the above three lines is a *glyph* for the letter "A" in three different fonts.

form

A description of a page or pages which describe fields for user input. See e.g. [HTML] Chapter 10.

glyph

The individual elements of a font corresponding to single characters. May also be called *font glyphs.* Also see *font glyph* above.

HII

Human Interface Infrastructure.

HTML

Hypertext Markup Language. A particular implementation of SGML focused on hypertext applications. HTML is a fairly simple language that enables the description of pages (generally Internet pages) that include links to other pages and other data types (such as graphics). When applied to a larger world, HTML has many shortcomings, including localization (q.v.) and formatting issues. The HTML *form* concept is of particular interest to this application.

IFR

Internal Form Representation. Used to represent forms in EFI so that it can be interpreted as is or expanded easily into XHTML.

IME

Input Method Editor. A program or subprogram that is used to map keystrokes to logographic characters. For example, IMEs are used (possibly with user intervention) to map the Kana (Hirigana or Katakana) characters on Japanese keyboards to Kanji.

internationalization

In this context, is the process of making a system usable across languages and cultures by using universally understood symbols. Internationalization is difficult due to the differences in cultures and the difficulty of creating obvious symbols; for example, why does a red octagon mean “Stop”?

localization

The process of focusing a system in so that it works using the symbols of a language/culture. The following design is influenced in major part by the requirements of localization.

logographic

A character set that uses characters to represent words or parts of words rather than syllables or sounds. Kanji is logographic but Kana characters are not.

NV

Nonvolatile.

scan-code

A value representing the *location* of a key on a keyboard. Scan-codes may also encode make (key press) and break (key release) and auto-repeat information.

SGML

Standard Generalized Markup Language. A Markup Language for defining Markup Languages

SKU

Stock keeping unit.

string

A null-terminated ordered list of 16-bit Unicode characters.

UGA

Universal Graphics Adapter.

VFR

Visual Forms Representation.

WUU

Walk up and use. A user interface in which the goal is to be instantly usable without a learning curve or other documentation.

XHTML

Extensible HTML. XHTML "will obey all of the grammar rules of XML (properly nested elements, quoted attributes, and so on), while conforming to the vocabulary of HTML (the elements and attributes that are available for use ant their relationships to one another)." [PXML, pg., 153]. Although not completely defined, XHTML is basically the intersection of XML and HTML and *does* support forms.

XML

Extensible Markup Language. A subset of SGML. Addresses many of the problems with HTML but does not currently (1.0) support forms in any specified way.

References

This section lists user-interface-related information that may be useful to you or that is referenced in this specification. See the master references in the Framework Interoperability and Component Specifications help system for additional references; see "Typographic Conventions" later in this chapter for the URL.

- **User Interface:**
  - [PUI] *Programming the User Interface: Principles and Examples*, Judith R. Brown, Steve Cunningham, John Wiley & Sons, 1989, ISBN: 0-471-63843-9.
  - [Tufte83] *The Visual Display of Quantitative Information*, Edward R. Tufte, Graphics Press, 1983.
  - [Tufte90] *Envisioning Information*, Edward R. Tufte, Graphics Press, 1990.
  - [Tufte97] *Visual Explanations*, Edward R. Tufte, Graphics Press, 1997.
- **Localization:**
  - [DBCS] Japanese Language DBCS (Double Byte Character Set): MS-DOS Version, Sizuoka Information Industry, AX Conference, 1991.
  - [DIS] Developing International Software For Windows 95* and Windows NT*, Nadine Kano, Microsoft Press, 1995, ISBN: 1-55615-840-8.

- **Markup Languages:**
  - [HTML] *HTML: The Definitive Guide, 2nd Edition*, Chuck Musciano and Bill Kennedy, O'Reilly and Associates, Inc., 1997, ISBN: 1-56592-235-2.
  - [PXML] *Professional XML*, Didier Martin, Mark Birbeck, et. al., Wrox Press, April, 2000, ISBN: 1-861003-11-0.
  - [XMLP] *XML: A Primer*, Simon St. Laurent, MIS:Press, 1998, ISBN:1-5582-8592-X.
  - [JavaScript] *JavaScript: The Definitive Guide, 3rd Edition*, David Flanagan, O'Reilly and Associates, Inc., 1998, ISBN: 1-56592-392-8.
- **Other References:**
  - [SVGA] *Super VGA Graphics Programming Secrets*, Steve Rimmer, Windcrest / McGraw-Hill, 1993, ISBN: 0-8306-4428-8.
  - *The Annotated Alice: Alice's Adventures in Wonderland* and *Through the Looking Glass*, Lewis Carroll, Martin Gardner, Meridian, 1960.

Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

Data Structure Descriptions

Intel® processors based on 32-bit Intel® architecture (IA-32) are "little endian" machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Intel® Itanium® processor family may be configured for both "little endian" and "big endian" operation. All implementations designed to conform to this specification will use "little endian" operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

The data structures described in this document generally have the following format:

STRUCTURE NAME: The formal name of the data structure.

Summary: A brief description of the data structure.

Prototype: A "C-style" type declaration for the data structure.

Parameters: A brief description of each field in the data structure prototype.

Description: A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this data structure.

In C structure definitions, the construct *[…]* indicates a variable length array, rather than a pointer to a variable length array. The number of elements can be discerned from other elements in the array. For example:

```
UINT16        NumberOfNarrowGlyphs;
UINT16        NumberOfWideGlyphs;
NARROW_FONT   NarrowGlyphs[…];
WIDE_FONT     WideGlyphs[…]
```

The number of elements in *NarrowGlyphs* is defined by *NumberOfNarrowGlyphs*.

Protocol Descriptions

The protocols described in this document generally have the following format:

Protocol Name: The formal name of the protocol interface.

Summary: A brief description of the protocol interface.

GUID: The 128-bit Globally Unique Identifier (GUID) for the protocol interface.

Protocol Interface Structure:
A "C-style" data structure definition containing the procedures and data fields produced by this protocol interface.

Parameters: A brief description of each field in the protocol interface structure.

Related Definitions: The type declarations and constants that are used in the protocol interface structure or any of its procedures.

Description: A description of the functionality provided by the interface including any limitations and caveats of which the caller should be aware.

Procedure Descriptions

The procedures described in this document generally have the following format:

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

Prototype: A "C-style" procedure header defining the calling sequence.

Parameters: A brief description of each field in the procedure prototype.

Description: A description of the functionality provided by the interface including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this procedure.

Status Codes Returned: A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

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*.

Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the online help version of this specification, 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. Note that these links are *not* active in the PDF of the specification.
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`	In the online help version of this specification, words in a **`Bold 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. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **`Bold Monospace`** appearance that is

	underlined but in dark red. Again, these links are not active in the PDF of the specification.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
`Plain Monospace`	In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

See the master Framework glossary in the Framework Interoperability and Component Specifications help system for definitions of terms and abbreviations that are used in this document or that might be useful in understanding the descriptions presented in this document.

See the master Framework references in the Interoperability and Component Specifications help system for a complete list of the additional documents and specifications that are required or suggested for interpreting the information presented in this document.

The Framework Interoperability and Component Specifications help system is available at the following URL:

http://www.intel.com/technology/framework/spec.htm

2
Design Discussion

Design Rationale

Introduction

This section explains the design decisions that are incorporated into the interfaces defined in chapter 3, "Code Definitions."

String Management

The standard representation for string characters in the Framework environment is Unicode 16 (UTF-16). At first glance that statement would seem to be enough discussion on string and font representation. Unicode is a well-defined standard, so it would seem to be a simple job to display the characters. It is not, however, for a number of reasons:

- First, if the Framework were to require that all of Unicode's 65,535 characters (zero is used as a terminator) to be carried, it would occupy around 2.5 MB (at 16x19 font noncompressed).
- Second, Unicode characters are usually presented in variable pitch fonts. If we simply decided that all characters were the same width, a "1" character and a complex logographic glyph would take the same width. This size would make it very hard to read the narrow characters and limit the number of narrow characters (Latin characters, for example) to about half of what normally fits on a row of text.
- Third, we need to avoid duplicating forms (internally) simply because we need to carry more than one language. Forms can require a fair amount of storage themselves. Further, consistency among forms for different languages should reduce errors.

Limiting Glyphs in Firmware Volumes

Strings in the Framework environment can be presented in differing environments with very different limitations. The most constrained environment is in the DXE and BDS spaces prior to discovery of a boot device with a system partition. The main limitation in this environment is storage space. If unexpected strings could be displayed before a system partition was available, the Framework would have to store glyphs for all characters in a Unicode font. Presumably, the system partition will have all glyphs available.

The benefit that a relatively closed environment such as DXE or BDS provides is that, with some careful user interface design, the number of unexpected characters that the system could be called on to display can be limited to a manageable number. By knowing what strings we are going to display, we can limit the number of glyphs we are required to carry.

It is also clear that, with careful design, we can support a system where a limited number of strings are displayed before a system partition is available, while still enabling the input and display of large numbers of characters/glyphs using a full font file stored on the system partition. In such a situation, the designer must be careful to ensure that enough information can be displayed and that the configuration can be changed using only the information found in firmware volumes (FVs) to obtain access to a satisfactory system partition.

Unicode

Unicode (as defined by UTF-16) has some interesting issues.

Unicode does not distinguish between characters of various widths, which is a reasonable concept if one has enough storage space to do font scaling but is a mess for the preboot environment. The solution here is to limit fonts to two widths and one height.

Unicode defines a *private use* area of 6500 characters that may be defined for local uses. Suggested uses include Egyptian Hieroglyphics; see *Developing International Software For Windows 95** *and Windows NT** for more information. Use of this area is prohibited for the Framework because a centralized font database that is accumulated from the various drivers (a valid implementation) would end up with collisions in the private use area and these characters generally could not be displayed in an XML browser.

UTF-16 defines *surrogate areas* (see page 56 in *Professional XML*) that allow for expanded character representations of the 16-bit Unicode. These character representations are very similar to Double Byte Character Set (DBCS)—2048 Unicode values split into two groups (D000–DBFF and DC00–DFFF). They are defined to have 16 additional bits of value to make up the character, for a total of about one million extra characters. Surrogate characters are *not* legal XML and are not supported in the Framework.

Unicode uses the concept of a *nonspacing* character. These glyphs are used to add accents, and so on, to other characters by what amounts to logically OR'ing the glyph over the previous glyph. There does not appear to be any predictable range in the Unicode encoding to determine nonspacing characters, yet these characters appear in many languages. Further, these characters enable spelling of several languages including many African languages and Vietnamese.

Localization Issues

Localization is the process by which the interface is adapted to a particular language. Table 2-1 discusses issues with localization and provides possible solutions.

Table 2-1. Localization Issues

Issue	Example	Solution	Comment
Directional display	Right to left printing for Hebrew.	Printing direction is a function of the language.	The display engine may or may not support all display techniques. If a language supports a display mechanism that the display engine does not, the language that uses the font must be selected.
Punctuation	Punctuation is directional. A comma in a right-to-left language is different from a comma in a left-to-right language.	Character choice is the choice of the author or translator.	
Line breakage	Rules vary from language to language.	Little or no formatting is performed by the Framework preboot GUI.	The runtime display is up to the runtime browser and is not defined here.
Date and time	Most Europeans would write July 4, 1776, as 4/7/1776 while the United States would write it 7/4/1776 and others would write 1776/7/4. The separator characters between the parts of both date and time vary as well.	Generally left to the creator of the user interface.	
Numbers	12,345.67 in one language is presented as 12.345,67 in another.	Print only integers and do not insert separator characters.	This solution is becoming accepted around the world as more people use computers.

User Input

To limit the number of required glyphs, we must also limit the amount and type of user input.

We can generally expect user input to come from the following two main types of devices:

- Keyboards
- Mouse-like pointing devices

Input from other devices, such as limited keys on a front panel, can be handled in two manners:

- Treat the limited keys as special-purpose devices with completely unique interfaces.
- Programmatically make the limited keys mimic a keyboard or mouse-like pointing device.

Pointing devices require no localization. They are universally understood by the subset of the world population we are addressing. For example, if someone does not know how to use a mouse or other pointing device, it is probably not a good idea to allow that person to change a system's configuration.

Keyboards, on the other hand, are localized at the keycaps but not at the electronics. In other words, a French keyboard and a German keyboard might have very different keys but there is no way for the software inside the keyboard, let alone the software in the system at the other end of the wire, to know which set of keycaps are installed.

The general solution proposed here is to use the keys that are common between keyboards and to ignore the language-specific keys. Keys that are available on USB keyboards in preboot mode include the following:

- Function keys (F1 – F12)
- Number keys (0-9)
- "Upside down T" cursor keys (the arrows, home, end, page up, page down)
- Numeric keypad keys
- The Enter, Space, Tab, and Esc keys
- Modifier keys (shifts, alts, controls, Windows*)
- Number lock

The scan-codes for these keys do not vary from language to language. These keys are the standard keys used for browser navigation although most end-users are unaware of this fact. Help for form-entry-specific keys must be provided to enable a useful keys-only interface. The one case where other, language-specific keys may be used is to enter passwords. Because passwords are never displayed, there is no requirement to translate scan-code to Unicode (keyboard localization) or scan-code to font.

Additional data can be provided to enable a richer set of input characters. This input is necessary to support features such as arbitrary text input and passwords.

Forms

HTML and IFR

The Framework forms, or Internal Form Representation (IFR), are data structures that are used to describe models of menus of input. The data structures define a language that is used to describe the allowed user input.

IFR is loosely based on HTML and its more recent equivalent, XHTML. IFR differs from HTML in several important ways, as listed in Table 2-2.

Table 2-2. Differences between HTML and IFR

HTML	IFR
Text is interspersed with meta-commands.	Supports text as a separate command. This support makes IFR easier to localize because IFR refers to strings by token to use the rest of the localization support.
Meta-commands are textual (“<input type=radio…”).	Commands are binary.
Supports a rich set of commands.	Set of commands is mainly a subset.
For most semantic checking and visibility control, requires the designer to resort to a scripting language such as Java* or JavaScript*.	Uses internal commands for the specialized semantic checking that it supports.

One of the design goals of IFR is that it be fairly easily translated into HTML.

Results Routing

IFR and HTML share many features. A major feature that both share is the transportation of routing information that describes how to process the results. Because the target environments for HTML and IFR are very different, the pieces of routing information that are carried are also very different.

IFR carries its routing information in the form of GUIDs. These GUIDs are not normally displayed to the user but are included in the (very HTML-like) output, which is then processed using a protocol described in this document.

Human Interface Overview

Introduction

Figure 2-1 depicts the model that is used inside the Framework to manage human interface components.

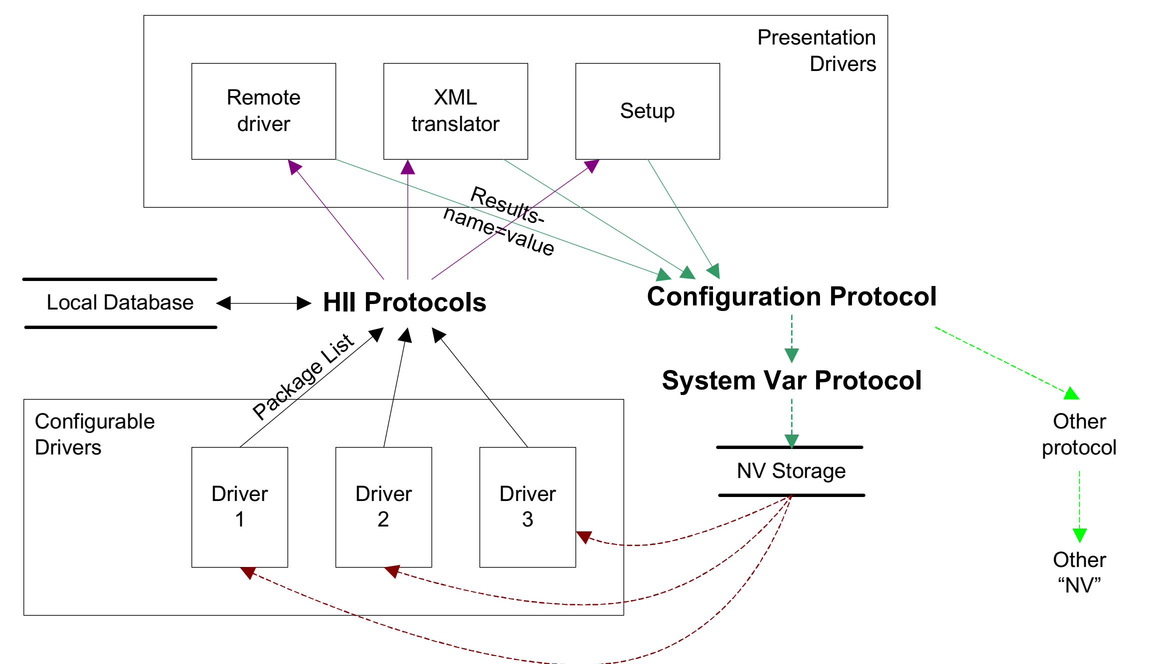

Figure 2-1. Managing Human Interface Components

Human interface data is divided into the following

- Input
- Fonts
- Strings
- Forms

Each of these is represented by a variable length data structure known as a *package* or simply a *pack*. Each package starts with a header, which is described in Section 0.

The definition of package-specific protocols is left for later in this section, after the packs that make up a package are introduced. Each of the various packs supports the separate registration of the pack type. The pack also has a package registration mechanism that allows for bulk registration.

See chapter 3, "Code Definitions," for the definitions of all human interface–related code that is referenced in this chapter.

Package Header

The package header starts the variable-length data structure that contains each of the human interface data components. The package header is defined in "Package Header" in chapter 3, "Code Definitions."

Package Manipulation

Package lists are expected to be separate sections that are stored in the same files as driver executables, although this implementation is not required.

Package lists are submitted to the EFI Human Interface Infrastructure (HII) Protocol to be stored in a database. Different packages inside the list are handled differently. Font packages are integrated into existing font data, expanding the available font characters. String and form information is handled by assigning a handle to the "subdatabase." These handles are then used to refer to the strings by the drivers themselves, as well as other drivers that make use of the database information.

Packages Definition

The packages that are passed to the HII database are self describing and their definition is intended to be extensible so that future types of packages can be added seamlessly. Type **EFI_HII_PACKAGES** is defined in "Packages Definition" in chapter 3, "Code Definitions."

Human Interface Infrastructure (HII) Protocol

The Human Interface Infrastructure Protocol (**EFI_HII_PROTOCOL**) manages the structures in the HII database. A number of functions are defined under **EFI_HII_PROTOCOL** to manipulate the data in the HII database. Type **EFI_HII_PROTOCOL** is defined in "Human Interface Infrastructure (HII) Protocol" in chapter 3, "Code Definitions."

Font Package

Introduction

This section describes the general format for the storage of fonts. A font package consists of a header and two types of glyph structures—standard-width (narrow) glyphs and wide glyphs.

Glyph Sizes

There are a number of factors to consider when choosing a standard glyph size:

- The glyphs must be readable by a large percent of the population in a standard screen format. Currently this format is expected to be 800x600 pixels.
- The glyphs should not be too squat or elongated.
- The maximum glyph width must be large enough to accommodate logographic characters. This width is around 15 or 16 pixels in either dimension.
- The glyphs must not be so large that they use a large amount of space in the firmware device.
- It would be nice if one of the dimensions were a multiple of 8 so that the characters would fit in the byte-wide storage of the target architecture.

Given these factors, the preferred dimensions are 8x19 for narrow glyphs and 16x19 for wide glyphs. These dimensions yield about 31 lines of 100 narrow characters on an 800x600 screen.

The representation is designed to be extensible to other formats as needed in the future.

Glyph Representation

There are two sizes of glyphs. There is one structure (**EFI_NARROW_GLYPH**, **EFI_WIDE_GLYPH**) for each glyph size. See “Glyph Representation” in chapter 3, “Code Definitions,” for the definitions of these two structures.

Strings

Introduction

A string package defines a list of strings in a particular language or related set of languages. Numerous string packages may exist in a single package to implement support for multiple language sets.

A string is generally a C-style Unicode string, although it may contain special EFI-specific formatting characters as well.

A string is referred to by a **STRING_TOKEN**, which is a constant usually assigned during the build process. A **STRING_TOKEN** is contained in a variable of type **STRING_REF**. The difference in the two makes it simpler to determine if an element is referring to a string or a container for a reference to a string, which makes implementing the build tools easier.

Internal String Representation

This section examines the internal storage format of strings and indicates how this format is used for the functions that enable programs to extract strings and parts of strings once a string package has been handed off to be managed. It uses the following text (from *Alice in Wonderland*) in its examples:

```
Twinkle, twinkle, little bat!
How I wonder what you're at!
Up above the world you fly,
Like a tea-tray in the sky.
```

Internal storage would look like:

```
Twinkle,<cr>twinkle,<cr>little<cr>bat!<cr><lf>How<cr>I<cr>…
```

where **<cr>** indicates carriage return and **<lf>** indicates line feed. English text can be broken at any space. Text in other languages may or may not be broken at spaces. Assume that English had a rule that spaces before words starting with *w* are nonbreaking. The representation would then be:

```
… <cr>bat!<cr>How<cr>I<sp>wonder<sp>what<cr>you're<cr>…
```

The partial string interface treats nonspacing, separated words as single words.

As noted above, some languages support narrow or wide characters and have commonly used stylistic guidelines for how narrow and wide glyphs are intermixed. In particular, most languages have adopted the rule that characters should be the same width. For example, an *1* would typically

be a narrow character but would be printed as a wide character if the characters surrounding it were wide. Unicode does not have the concept of narrow or wide characters, so it is generally left up to sophisticated operating system (OS)–present drivers to determine the applicability of the width of characters. Due to the limited size available to many of the target environments, the EFI environment cannot rely on such a rich heuristic mechanism. Instead, it supports the use of special <narrow> and <wide> characters (defined later in chapter 3) that indicate the preference for character widths. In essence they define the search pattern—if in the default <narrow> mode, the narrow characters are searched first; if in <wide> mode, the wide characters are searched first.

Consider the case of a firmware-based, 80x25-line, character-oriented presentation driver that has split the screen into three roughly equal columns of 26 characters each. The first column is for prompts, the second is for the currently selected option, and the third column is for help text. Assume the top and bottom two lines are used for other purposes. This setup means that the help text can occupy 26x21 lines. The parameters to the extract functions would then indicate a *StartWordIndex* of 0 (first word), a *NumberOfLines* of 21, and a *LineWidth* of 26. The **GetLine()** function fills each line with as many space-separated, nonsplitting "words" as can be fit on each line before moving to the next line, adding spaces between each. "Words" that cannot fit on a line alone are split so that the line width will align most closely to the maximum line width but not expand over.

GetString() has options to extract the raw string (as described above) or with spaces in the normal **<cr>** locations and with having special overrides removed.

In the case of translating the text to HTML, it is assumed that the browser can handle its own line breaks. In this case, the *StartWordIndex* would be 0, the *NumberOfLines* would be 0 (all lines), and the *LineWidth* would be 0 (infinite), thus generating lines as long as the text allows.

Form Packages

Goals

During the boot of a Framework-based system, the following types of data might be displayed and, hence, must be supported by the user interface:

- Graphical displays—in particular, logos that are displayed during boot to provide a pleasant end-user experience and advertising.
- Text, such as a copyright, on a power-on screen.
- A query and response dialog during boot. These queries usually take the form, "This error was found. Press a key to continue." It is typical to switch to a text screen from the logo screen to display such information.
- Setup, which provides several interface types itself:
  - Columnar data, such as
    "Processor Speed   2.4 GHz"
    and
    "Memory Size   512 MB"
  - Subtitles, such as "Ports," "Power Management," and so on

- Questions, including the following:
    - A prompt, such as “Parallel port address”
    - Question-specific help text
    - Some mechanism for actual input, including the following:
        - “One-of” selection (like a radio button): The most common input mechanism, where the user must select one item from a menu of options.
        - Check box: The user can select or clear an option individually. It is commonly used to enable or disable a mode. When grouped, check boxes support multiple option sets where more than one option can be selected simultaneously.
        - Decimal number within a range.
        - Password.
        - Generalized character strings (“text boxes”). Passwords are, in fact, generally treated as a subset of strings in HTML.

This list does not actually define user-interface issues. For example, help text is generally necessary whether it is displayed along with the question or only in response to a keystroke. Keys help (the functions associated with individual keys) are not defined because they are user-interface specific.

It is important to define the boundary between what is provided internally and what is a part of a user interface. For example, are radio buttons required with “one-of” choices, or are drop-down combo boxes also legal? Are the number of choices limited for a “one-of” question? A developer might want a “one-of” button to input the day of the month. Thirty-one radio buttons is excessive but a drop-down combo box with a slider (as used in Font Selection in Microsoft Word*) is not.

The effort becomes more complex if one attempts to handle interrelated questions. It is common for one question to be meaningful only if a particular option is selected on a different question. Forms languages such as HTML are not rich enough to express this relation and, as such, do not provide sufficient hints for the browser to “gray-out” the secondary question if a different option is chosen in the primary question. Typical HTML Web forms are primitive enough that this issue rarely arises. Unfortunately, the questions in Setup tend to reflect the underlying interrelationship of the hardware and, as such, tend to create interrelated questions.

IFR supports mechanisms to describe the default values for questions. As in HTML, it is up to the presentation engine (“browser”) to provide an interface to allow these values to be set.

Different browser environments have different facilities and mechanisms for causing the form to be submitted. A mechanism to perform this task is required by each IFR browser but left up to the browser for implementation.

The syntax of the output in XHTML is a sequence of *UNICODE name=value* pairs separated by the “&” character. IFR supports a subset of this easily parsed standard mechanism to encode its results as well. The mechanism encodes identifier, offset, and width information in the name part. The value part is typically decimal integers, except for fonts and strings.

Forms and Form Sets

An IFR is used to represent forms in the Framework. This representation is designed so that it can be interpreted as is or expanded easily into XHTML.

In most markup languages, a form is submitted to a server for processing when the user completes it. In many of the "use" cases that IFR targets, the equivalent of the server is not available. For this reason, the forms package can contain one or more forms.

Semantics and Tag Structures

Form Packages and Scoping

The form is the basic encapsulation of configuration data. A form package consists of one or more forms. The form package provides scoping for identifiers in the forms, including **<name-id>** and string tokens in particular. The intent is for the driver or drivers creating a form set to be cooperative and to avoid the definition of these identifiers from being duplicated unexpectedly. Different form packages are in essence invisible to each other. For example, one form set cannot go to another form set.

The first form in the form set is known as the *parent form.* All other forms are *child forms.* When interpreting forms, it is up to the interpreter to create a "main page" through which all parent forms from all form sets are accessible. Child forms are accessed using hypertext references (using the "go-to" operation defined in chapter 3) from the parent page or other child pages. The interpreter is responsible for creating references from the parent page back to the main page and for retaining a "back" list of previously visited pages. Other exits from child pages must be through explicit IFR hypertext references.

Note that it is legal for a form package to contain forms that cannot be reached from the parent form. These forms may be used in more dynamic cases by drivers to take advantage of the user interface capabilities that are already useful for configuration in the system.

Forms

Forms must be position independent because they can be copied from place to place. Further, position independence of the parts of the forms (operations) enables insertion of new data between precompiled form text.

Device Descriptions

A device description operation allows a form or forms to be associated with its corresponding firmware. The format of the contents of the **<dev-desc-data>** are defined in the *Intel® Platform Innovation Framework for EFI Device Description Specification.*

Titles, Subtitles, and Text: <subtitle>, <text>

Each form must have a title. Subtitles can be placed throughout the forms to provide visual separation of the elements. Text may be inserted as well.

The exact use of the title, subtitle, and text elements is defined by individual presentation drivers (the "browsers" for the language) as is the presentation to the user. It is suggested that subtitle be translated into HTML <h3>.

✏ **NOTE**

Note that, unlike HTML, text has its own opcode (tag). The Text Tag exists in IFR (but not in HTML) to facilitate localization of text for different languages.

Questions

The intent of a question, from a driver's perspective, is to associate an ID with a value.

Experience has shown that very few types of questions are required to obtain the information that is necessary to configure a system. The parameters for question operations follow a standard form. The first byte is the opcode. This byte is followed by an ID that serves as an internal mechanism to refer to the question and as a part of the results generation. String tokens to provide a prompt (a short description of the question) and context-sensitive help text are then provided. Note that there is no way to provide "keys" help as that is the responsibility of the presentation driver.

The following subsections describe the different types of question tags. The different types of question tags are as follows:

- One-of
- Checkbox: <checkbox>
- Numeric: <numeric>
- Password: <password>
- Hidden: <hidden>
- Ordering: <list>
- Hypertext: <goto>

See "Internal Form Representation (IFR) Language Syntax Definition" in chapter 3, "Code Definitions," for definitions if these tags.

One-Of

HTML has several one-of types of tags, including *<input type=radio…>* and *<select…>*.

The most commonly used type is equivalent to an HTML radio button where the user is asked to pick one item from a series of items. In IFR, this model is known as a *one-of* selection. Flags that are associated with each option are split between standard definitions and user definitions. The two standard definitions are "default" and "current selection."

Checkbox: <checkbox>

The HTML tag for the checkbox type is *<input type=checkbox…>*.

The checkbox type is used in two ways. The first is as an equivalent to an "on/off" radio button. The second is as a series of checkboxes to present the equivalent of a radio button except that more than one item may be checked at a time.

Numeric: <numeric>

The numeric type has no exact analogy in HTML. The closest type is *<input type=text…>*.

Numeric questions allow for the input of bounded positive (or 0) decimal numbers. The minimum and maximum values are specified, as well as a step value. The step value is used to allow the browser to do more complete validation in cases where legal input values are not monotonically increasing. For example, consider a case where only odd values were required (between 1 and 15, for example). The minimum value would be 1, maximum of 15, and the step would be 2. A number *n* is valid if:

```
(minimum <= n && n<= maximum)
```

and

```
int ((n-minimum) / step) == (n-minimum)/step
```

Password: <password>

Password questions allow for the input of passwords. Many browsers (mainly remote and OS-present) will not be secure enough for passwords. It is up to the presentation driver to edit out password operations in these cases. The encoding mechanisms are TBD.

Hidden: <hidden>

Hidden questions are questions that have no options and are the equivalent of constants. The browser must hand the ID and value back as with a normal question. The hidden construct is from HTML and allows the generating driver to send a message to the driver responsible for processing the output of the browser.

Ordering: <list>

HTML has no analogous *<list>* tag.

This input type enables ordered input from a list of choices. The construct is intended to support unique lists where a choice may appear in the list only once (e.g. a list of boot devices), or lists where a choice may appear several times. The syntax is designed to enable a number of different visual representations.

The question format consists of the following:

- A header
- A list of choices
- A list of containers

Each container has a reference to a choice.

Header

The header contains the usual header information—ID, prompt, and help text. The ID does not end up being output. The flags that are defined include the following:

- Unique: Each choice may be used at most once.
- NoNull: All containers must be filled with a selection.
- A "null choice" value rounds out the header. This value is legal input for a container if the NoNull flag is off.

List of Choices

Each choice consists of a string reference and a value. The string reference is used to describe the choice and the value is the value to put in the container if the choice is selected. A null string ends the choice list.

List of Containers

Each container consists of the following:

- **String reference:** Describes the container (usually like "third boot option")
- **Id-offset-width:** Defines a resulting name that corresponds to the order
- **Default value:** The initial value for the choice

The presentation driver should not evaluate uniqueness while the user is still changing the configuration of a particular question.

Examples

Following is an example of a text display (character oriented):

Names for Kings (0 = None)

1. Harold
2. Andrew
3. Mark
4. Alfred
5. George
6. Ethelred
7. Wilhelm

First Name: [6]
Second Name: [2]
Third Name: [3]
Fourth Name: [ ]

This text display might be represented with the following syntax (with syntactic sugar and with actual strings substituted for string references to improve readability):

```
List id, "Names for Kings", "Help",  0, Unique
choices
      "Harold", 1
      "Andrew", 2
      "Mark", 3
      "Alfred", 4
      "George", 5
      "Ethelred", 6
      "Wilhelm", 7
containers
      id1, "First Name:", 1
      id2, "Second Name:", 0
      id3, "Third Name:", 0
      id4, "Fourth Name:", 0
EndList
```

Given the above example, the results would be `…&id1=6&id2=2&id3=3&id4=0&…`

Hypertext: <goto>

The HTML tag for the go-to type is *<a href…>*.

The go-to command implements the ability to refer to a form from another form. The parameter is a *form* identifier, meaning that the go-to may only reference another form and not a place inside the form. In particular, the go-to reference may not be a label. If nothing else, this design eliminates confusion with jumping into the middle of nesting constructs inside IFR forms.

Image

The HTML tag for the image type is *<image align=left src=…>*.

This type inserts an image into the form. If the form cannot display graphics, it may substitute the `<text-only-string-ref>` tag instead. Text is not wrapped around the image.

Background

The HTML tag for the image type is : *<body background=…>*.

As in HTML, the background is tiled across the full screen. Text scrolls over the background.

Visibility Control: <grayout>, <suppress>

There is no HTML analogy for visibility control.

HTML does not support the ability to control whether a particular part of a form should be made visible to the user or "grayed out" (printed in a muted tone or made invisible).

Visibility control is implemented via the *grayout* construct. This construct is block structured and analogous to an "if" statement in C. The hide construct has an opcode and a Boolean expression. These are followed by a series of other operators and finally a termination opcode. If the Boolean expression is true, the encompassed operations should be grayed out. If it is false, they should be made visible.

```
grayoutif serport == 0
      oneof id=serport2 prompt=sp2str help=sp2helpstr
      ...
```

The *suppress* operation is similar to the hide construct except that the enclosed items must not be displayed.

Neither *suppress* nor *grayout* affect the output of the results.

Boolean Expressions

The Boolean expressions (involving only true and false) are presented internally in Reverse Polish Notation (RPN [postfix]) form. The Boolean operators are limited to "and," "or," and "not." The following three primitives are used to query the current state of the configuration:

- **ID/Value compare:** The current configuration ("value" in HTML) of the question corresponding to the ID is compared to the value operand. The primitive results in `TRUE` if they are the same and `FALSE` otherwise. In the case of a "many-of" instance, if the value is selected (even if other values are also selected), the primitive returns `TRUE`. (By "returns," we mean "evaluates to" or, from the common implementation method, "pushes on the stack.")
- **ID/List compare:** The current value of the question corresponding to the ID is compared to a list of values. If the value is in the list, `TRUE` is returned. If not, `FALSE` is returned. This operation is valid only on "one-of" and numeric questions. The list itself consists of a `UINT16` count followed by that many `UINT16` values.
- **ID/ID compare:** The current values of the questions corresponding to the two IDs are compared. If the questions are of different type, `FALSE` is returned. This value actually is not really valid, but it is a clean way to recover. If the values are identical, `TRUE` is returned. Otherwise `FALSE` is returned. In the case of a "many-of" instance, all values must correspond. Those values selected in one must be selected in the other and those not selected in one must also be not selected in the other.

Using Grayed-Out Parts of a Form

There are two main reasons that an area might be grayed out:

- The driver might support a subset of the options available on a particular system.
- The value of one question has a role in determining if another question should be grayed out.

In the first reason, the driver might sense which subset a particular system has and need to display only those options. This action can be accomplished by editing the form dynamically or by simply modifying a hidden question value and using a hide operation to do an ID/value comparison on the hidden question.

The second reason, when the value of one question has a role in determining if another question should be grayed out, is more familiar to the user. This issues is common in OS-present applications as well. Unfortunately, HTML punts on grayed-out control, relying on JavaScript or a similar tool for assistance. Consider two "one-of" questions. The first asks if the onboard USB should be enabled or disabled. The second asks if the onboard USB should be searched for boot devices at power up. If the onboard USB is disabled, the second question does not make sense. This case could be resolved using the provided primitives (assuming some syntactic sugar), as shown in the following example:

```
OneOf USB_EN_DIS  EnDisString, EnDisHelpString
     EnabledString, 1, Default+Selected
     DisabledString, 0, 0
EndOneOf
GrayOutIf   USB_EN_DIS == 1
     OneOf id=USB_FIND_BOOT prompt=
          FindBootString help=FindBootHelpString
               EnabledString, 1, Default+Selected
               DisabledString, 0, 0
     EndOneOf
EndGrayout
```

Consistency Checking

As well as controlling visibility, questions have other effects on each other. Consider three numeric questions: year, month, day. The range for month is 1 to 12 and the range for day is 1 to 31. The problem is that June 31 is not valid, nor is February 29, 2003, although February 29, 2004, is acceptable.

IFR addresses such issues with consistency expressions. Consistency expressions are Boolean expressions with associated strings. If the expression becomes `TRUE`, it indicates that an inconsistency has occurred. The associated string is a useful example of a pop-up indicating the issue.

Using the date as an example (and again with syntactic sugar):

```
Numeric id=YEAR prompt=YearString help=YearHelpString start=2000 \
 end=2039 step=1 default=2001
Numeric id=MONTH prompt=MonthString help=MonthHelpString start=1 end=12 \
  Step=1 default=1
Numeric id=DAY prompt=DayString help=DayHelpString start=1 end=31 \
  Step=1 default=1
Inconsistent If=(DAY == 31 && MONTH == [2, 4, 6, 9, 11]) text=BadDayString
Inconsistent If=(DAY == 30 && MONTH == 2) text=Feb30String
Inconsistent If=(DAY == 29 && MONTH == 2 &&     \
      !YEAR == [2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036]),
      text=LeapYearString
```

The year ranges from 2000 to 2039, the month from 1 to 12, and the day from 1 to 31. Some months have only 30 days and February (`MONTH == 2`) has only 28 or 29. The Var/List operation (syntactically cleansed here using the example "`MONTH == [2, 4,…`") is particularly useful here.

Dynamic Data

Labels

Most of the contents of forms can be created at build time. Some, however, cannot be defined statically. For example, the list of boot devices cannot be known ahead of time.

The mechanism that is defined for inserting new form operations into an existing form is to use the *label* operation. The driver must create IFR operations on the fly. A function allows this dynamic data to be inserted into the driver's IFR *before* a given label.

Advanced Operations (Optional)

The operations described thus far define the minimum level of IFR to be supported on all compliant systems. The following operations are optional. Implementations of IFR browsers that do not support these operations should ignore them (skip over them using the length field).

String Input

```
<string-input> ::= <string-op> <question-header> <min-length> <max-length>
```

String input is optional as it is difficult to support localized general-purpose keyboard input. Strings up to `<byte-width>` (255 characters) are supported so `<min-length>` and `<max-length>` are bytes.

No inconsistency checking operations are supported on strings.

Dynamic Processing of NV/IFR Data

Form Callback Protocol

The Form Callback Protocol provides an interface to hardware-specific drivers that control access to nonsystem nonvolatile storage (NVS) and support callbacks from the browser or Human Interface Infrastructure (HII). Type **EFI_FORM_CALLBACK_PROTOCOL** is defined in "Dynamic Processing of NV/IFR Data" in chapter 3, "Code Definitions."

Browser Interface

Form Browser Protocol

The Form Browser Protocol is the interface to call for drivers to leverage the EFI Configuration Driver interface. Type **EFI_FORM_BROWSER_PROTOCOL** is defined in "Browser Interface" in chapter 3, "Code Definitions."

Runtime Representations

Using IFR at Runtime

How should a presentation driver use the semantics provided by the IFR and its subordinates? The only real answer is, "as well as it can." The intent of the design of IFR in particular was to provide a rich enough language to address the requirements of the pre-OS space while enabling a large number of types of presentation drivers to address different configuration mechanisms.

The following are examples of configuration mechanisms:

- Standard system setup
- Remote setup over a serial connection to e.g. VT100 terminal emulation
- Remote configuration over a modem to a technical support center (via shared voice data).
- Remote setup over a network card
- OS-present setup
- Automatic system configuration during board-level manufacturing
- Automatic system configuration during system integration

These alternative mechanisms vary widely in such areas as the following:

- The bandwidth of the communication media between the user and the system. A remote system cannot necessarily handle the bandwidth of data that a game machine with a graphics accelerator can.
- The time delay between when the IFR was created and when the user sees it. If the IFR is turned into HTML, it may be hours or days between the time the forms package was created and the time it is used.
- The capabilities for display and input of the communication media (VT-100 has limited graphics capabilities).
- The limited semantics of the representation into which IFR is translated. IFR was designed to be easily translated into (X)HTML but, as noted above, a simple translation (one that does not include JavaScript generation, for example) would not be able to perform consistency checks and gray-out options.

Limitations of Presentation Mechanisms

Both developers of forms and developers of presentation drivers must understand the limitations that the existing presentation mechanisms impose in order to create forms that are useful in a wide variety of settings.

The driver writer, for example, can use help text to insulate the customer against confusion when inconsistency checking is dropped by an HTML presentation driver.

3 Code Definitions

Packages

Package Header

EFI_HII_PACK_HEADER

Summary

The header found at the start of each package.

Prototype

```
typedef struct {
  UINT32        Length;
  UINT16        Type;
} EFI_HII_PACK_HEADER;
```

Parameters

Length

The size of the package in bytes.

Type

See "Related Definitions" below.

Description

Each package starts with a header, as defined above, that indicates the size and type of the package. When added to a pointer pointing to the start of the header, *Length* points at the next package.

When concatenated together and terminated with an **EFI_HII_PACK_HEADER** with a *Length* of zero, the package lists form a localization package list.

Related Definitions

```
//*******************************************************
// Defined Type values
//*******************************************************
#define EFI_HII_FONT         1
#define EFI_HII_STRING       2
#define EFI_HII_IFR          3
#define EFI_HII_KEYBOARD     4
#define EFI_HII_HANDLE_PACK  5
```

Packages Definition

EFI_HII_PACKAGES

Summary

Definition of the packages structure that will be used to pass contents into the HII database. There are a variable number of packages that can be defined in the **EFI_HII_PACKAGES** structure. Each package will have a header that will identify the type of package that is being sent to the database.

Prototype

```
typedef struct {
  UINT32               NumberOfPackages;
  EFI_GUID             *GuidId;
  EFI_HII_HANDLE_PACK  *HandlePack;
} EFI_HII_PACKAGES;
```

Parameters

NumberOfPackages

The number of packages being defined in **EFI_HII_PACKAGES**.

GuidId

The GUID to be used to identify this set of packages that are being exported to the HII database.

HandlePack

The package that is intended to enable the passing in of pertinent driver model data so that the package contents can be associated with other system data and also provides a simple means by which a Callback handle can be passed into the database.

Description

Because the packages that are defined in the above definition are the only required definitions, each optional entry is defined in its own section. See “Related Definitions” below.

Related Definitions

```
//*******************************************************
// EFI_HII_HANDLE_PACK
//*******************************************************
typedef struct {
  EFI_HII_PACK_HEADER   Header;             // Must be filled in
  EFI_HANDLE            ImageHandle;        // Must be filled in
  EFI_HANDLE            DeviceHandle;       // Optional
  EFI_HANDLE            ControllerHandle;   // Optional
  EFI_HANDLE            CallbackHandle;     // Optional
} EFI_HII_HANDLE_PACK;
```

Header

The structure that defines the type of package being described, as well as the length of the overall package.

ImageHandle

The image handle of the driver to which the package is referring.

DeviceHandle

The handle of the device that is being described by this package.

ControllerHandle

The handle of the parent of the device that is being described by this package.

CallbackHandle

The handle that was registered to receive **EFI_FORM_CALLBACK_PROTOCOL** calls from other drivers. A callback would commonly occur from a browser to provide user-input data back to the driver that registered the callback handle.

Human Interface Infrastructure (HII) Protocol

EFI_HII_PROTOCOL

Summary

The HII Protocol manages the HII database, which is a repository for data having to do with fonts, strings, forms, keyboards, and other future human interface items.

GUID

```
// {B5F16136-1144-4d6a-BBBB-41F2FF1E1D04}
#define EFI_HII_PROTOCOL_GUID  \
   { 0xb5f16136, 0x1144, 0x4d6a, 0xbb, 0xbb, 0x41, 0xf2, \
   0xff, 0x1e, 0x1d, 0x4 }
```

Protocol Interface Structure

```
typedef struct _EFI_HII_PROTOCOL {
  EFI_HII_NEW_PACK                 NewPack;
  EFI_HII_REMOVE_PACK              RemovePack;
  EFI_HII_FIND_HANDLES             FindHandles;
  EFI_HII_EXPORT                   ExportDatabase;

  EFI_HII_TEST_STRING              TestString;
  EFI_HII_GET_GLYPH                GetGlyph;
  EFI_HII_GLYPH_TO_BLT             GlyphToBlt;

  EFI_HII_NEW_STRING               NewString;
  EFI_HII_GET_PRI_LANGUAGES        GetPrimaryLanguages;
  EFI_HII_GET_SEC_LANGUAGES        GetSecondaryLanguages;
  EFI_HII_GET_STRING               GetString;
  EFI_HII_GET_LINE                 GetLine;
  EFI_HII_GET_FORMS                GetForms;
  EFI_HII_GET_DEFAULT_IMAGE        GetDefaultImage;
  EFI_HII_UPDATE_FORM              UpdateForm;

  EFI_HII_GET_KEYBOARD_LAYOUT      GetKeyboardLayout;
} EFI_HII_PROTOCOL;
```

Parameters

NewPack

Extracts the various packs from a package list. See the **NewPack()** function description.

RemovePack

Removes a package from the HII database. See the **RemovePack()** function description.

FindHandles

Determines the handles that are currently active in the database. See the **FindHandles()** function description.

ExportDatabase

Export the entire contents of the database to a buffer. See the **ExportDatabase()** function description.

TestString

Tests if all of the characters in a string have corresponding font characters. See the **TestString()** function description.

GetGlyph

Translates a Unicode character into the corresponding font glyph. See the **GetGlyph()** function description.

GlyphToBlt

Converts a glyph value into a format that is ready for a UGA BLT command. See the **GlyphToBlt()** function description.

NewString

Allows a new string to be added to an already existing string package. See the **NewString()** function description.

GetPrimaryLanguages

Allows a program to determine the primary languages that are supported on a given handle. See the **GetPrimaryLanguages()** function description.

GetSecondaryLanguages

Allows a program to determine which secondary languages are supported on a given handle for a given primary language. See the **GetSecondaryLanguages()** function description.

GetString

Extracts a string from a package that is already registered with the EFI HII database. See the **GetString()** function description.

GetLine

Allows a program to extract a part of a string of not more than a given width. See the **GetLine()** function description.

GetForms

Allows a program to extract a form or form package that has been previously registered. See the **GetForms()** function description.

GetDefaultImage

Allows a program to extract the nonvolatile image that represents the default storage image. See the **GetDefaultImage()** function description.

UpdateForm

Allows a program to update a previously registered form. See the **UpdateForm()** function description.

GetKeyboardLayout

Allows a program to extract the current keyboard layout. See the **GetKeyboardLayout()** function description.

Description

The HII Protocol is used as a repository of content that is both provided by built-in firmware content as well as option ROMs.

Related Definitions

```
//******************************************
// EFI_HII_HANDLE
//******************************************
typedef UINT16          EFI_HII_HANDLE;


//******************************************
// EFI_FORM_LABEL
//******************************************
typedef UINT16         EFI_FORM_LABEL;
```

EFI_HII_PROTOCOL.NewPack()

Summary

Registers the various packs which are passed in via the Package parameter.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_NEW_PACK) (
  IN  EFI_HII_PROTOCOL           *This,
  IN  EFI_HII_PACKAGES           *Packages,
  OUT EFI_HII_HANDLE             *Handle
  );
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Packages

A pointer to an **EFI_HII_PACKAGES** package instance.

Handle

A pointer to the **EFI_HII_HANDLE** instance.

Description

With the exception of font and keyboard data, this function adds the contents of the package list to the database and returns a handle back to the data. Font and keyboard data is kept in a common pool and will have a **NULL** handle associated with them. In the case where *Packages* contains both pooled data and database data, a valid handle will be returned upon the addition of the appropriate data into the database.

Status Codes Returned

EFI_SUCCESS	Data was extracted from *Packages*, the database was updated with the data, and *Handle* returned successfully.
EFI_INVALID_PARAMETER	The content of *Packages* was invalid.

EFI_HII_PROTOCOL.RemovePack()

Summary

Removes a package from the HII database.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_REMOVE_PACK) (
  IN  EFI_HII_PROTOCOL          *This,
  IN  EFI_HII_HANDLE            Handle
  );
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

The handle that was registered to the data that is requested for removal.

Description

This function removes the string and/or form data that is associated with a handle from the HII database. This function has no effect on keyboard or font data that may have been registered with the **NewPack()** function.

Status Codes Returned

EFI_SUCCESS	The data associated with the *Handle* was removed from the HII database.
EFI_INVALID_PARAMETER	The *Handle* was not valid.

EFI_HII_PROTOCOL.FindHandles()

Summary

Determines the handles that are currently active in the database.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_FIND_HANDLES) (
  IN      EFI_HII_PROTOCOL          *This,
  IN OUT UINT16                     *HandleBufferLength,
  OUT     EFI_HII_HANDLE            *Handle
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

HandleBufferLength

On input, a pointer to the length of the handle buffer. On output, the length of the handle buffer that is required for the handles found.

Handle

An array of **EFI_HII_HANDLE** instances returned.

Description

This function determines the handles that are currently active in the database. For example, a program wishing to create a Setup-like configuration utility would use this call to determine the handles that are available. It would then use calls defined in the forms section below to extract forms and then interpret them.

Status Codes Returned

EFI_SUCCESS	*Handle* was updated successfully.
EFI_BUFFER_TOO_SMALL	The *HandleBufferLength* parameter indicates that *Handle* is too small to support the number of handles. *HandleBufferLength* is updated with a value that will enable the data to fit.

EFI_HII_PROTOCOL.ExportDatabase()

Summary

Exports the contents of the database into a buffer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_EXPORT) (
  IN      EFI_HII_PROTOCOL          *This,
  IN      EFI_HII_HANDLE            *Handle,
  IN OUT  UINTN                     *BufferSize,
  OUT     VOID                      *Buffer
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

An **EFI_HII_HANDLE** that corresponds to the desired handle to export. If the value is 0, the entire database will be exported. In either case, the data will be exported in a format described by the structure definition of **EFI_HII_DATA_TABLE**.

BufferSize

On input, a pointer to the length of the buffer. On output, the length of the buffer that is required for the export data.

Buffer

A pointer to a buffer that will contain the results of the export function.

Description

This function will retrieve the contents of the HII database and export it in a well-defined format. This format encompasses a means by which the data is well described and provides for seamless integration of additional export data as content evolves.

Related Definitions

```
//*******************************************************
// EXPORT_TABLE
//*******************************************************
typedef struct {
  UINTN                NumberOfHiiDataTables;
//EFI_HII_DATA_TABLE HiiDataTable[];
} EXPORT_TABLE
```

NumberOfHiiDataTables

Number of **EFI_HII_DATA_TABLE** entries defined in the **EXPORT_TABLE** structure.

HiiDataTable

Variable count of **EFI_HII_DATA_TABLE** entries. The amount in the table corresponds to the value in *NumberOfHiiDataTables*.

```
//*******************************************************
// EFI_HII_DATA_TABLE
//*******************************************************
typedef struct {
  EFI_HII_HANDLE          HiiHandle;
  UINTN                   IfrDataOffset;
  UINTN                   StringDataOffset;
  UINTN                   NumberOfVariableData;
  UINTN                   NumberOfLanguages;
  EFI_DEVICE_PATH         DevicePath;
//EFI_VARIABLE_CONTENTS VariableData[];
//EFI_IFR_CONTENTS      IfrData;
//EFI_STRING_CONTENTS   StringData[];
} EFI_HII_DATA_TABLE;
```

HiiHandle

Unique value that correlates to the original HII handle.

IfrDataOffset

Byte offset from the start of this structure to the IFR data.

StringDataOffset

Byte offset from the start of this structure to the string data.

NumberOfVariableData

Number of *VariableData[]* elements in the array.

NumberOfLanguages

The number of language string packages.

DevicePath

Describes a logical path to a device from a known starting point

VariableData

Contents of the variable information for this entry—GUID/name/data.

IfrData

Contents of the IFR data for this entry.

StringData

Contents of the string data. There may be multiple instances of the **EFI_STRING_CONTENTS** structure, defining multiple languages

```
//*******************************************************
// EFI_VARIABLE_CONTENTS
//*******************************************************
typedef struct {
  EFI_GUID       VariableGuid;
  UINT32         VariableNameLength;
  UINT32         VariableDataLength;
  UINT16         VariableId;
  CHAR16         VariableName[40];
} EFI_VARIABLE_CONTENTS;
```

VariableGuid

GUID of the EFI variable.

VariableNameLength

Length in bytes of the EFI variable.

VariableDataLength

Length of data that follows the **EFI_VARIABLE_CONTENTS** structure. Because this data is variable length, it is not included in the structure definition, but the *VariableDataLength* amount of bytes needs to be assumed to immediately follow this header.

VariableId

The unique value for this variable, which will be later referenced by other IFR content to determine which variable is actively being referenced.

VariableName

The name of the variable, which will have a maximum size of 40 Unicode characters. Data starts after the *VariableName* parameter.

```
//*******************************************************
// EFI_IFR_CONTENTS
//*******************************************************
typedef struct {
  UINT32          IfrDataLength;
} EFI_IFR_CONTENTS;
```

IfrDataLength

Length of the data that follows. Each opcode is self describing and the first opcode definition that should be encountered is an **EFI_IFR_FORM_SET**. The last opcode definition should be **EFI_IFR_END_FORM_SET**.

```
//*******************************************************
// EFI_STRING_CONTENTS
//*******************************************************
typedef struct {
  CHAR16          Language[3];
  CHAR16          Pad;
  UINT32          NumberOfStrings;
} EFI_STRING_CONTENTS;
```

✏ NOTE

This structure will have **HII_DATA_TABLE->***NumberOfStringData* *occurrences.*

Language

Language definition for the strings immediately after **EFI_STRING_CONTENTS**.

Pad

A pad value to ensure that access to member contents is aligned.

NumberOfStrings

Number of **NULL**-terminated strings that are contained immediately following after the **EFI_STRING_CONTENTS** definition.

Font Package

Glyph Representation

EFI_NARROW_GLYPH

Summary

The **EFI_NARROW_GLYPH** has a preferred dimension (w x h) of 8 x 19 pixels.

Prototype

```
typedef struct {
  CHAR16         UnicodeWeight;
  UINT8          Attributes;
  UINT8          GlyphCol1[19];
} EFI_NARROW_GLYPH;
```

Parameters

UnicodeWeight

The Unicode representation of the glyph. The term *weight* is the technical term for a character value.

Attributes

The data element containing the glyph definitions; see "Related Definitions" below.

GlyphCol1

The column major glyph representation of the character. Bits with values of one indicate that the corresponding pixel is to be on when normally displayed; those with zero are off.

Description

Glyphs are represented by two structures, one each for the two sizes of glyphs. The narrow glyph (**EFI_NARROW_GLYPH**) is the normal glyph used for text display.

Related Definitions

```
// Contents of EFI_NARROW_GLYPH.Attributes
#define GLYPH_NON_SPACING   0x01
#define GLYPH_WIDE          0x02
```

Following is a description of the fields in the above definition:

GLYPH_NON_SPACING	This symbol is to be printed "on top of" (**OR**'d with) the previous glyph before display.
GLYPH_WIDE	This symbol uses 16x19 formats rather than 8x19.

EFI_WIDE_GLYPH

Summary

The **EFI_WIDE_GLYPH** has a preferred dimension (w x h) of 16 x 19 pixels which is large enough to accommodate logographic characters.

Prototype

```
typedef struct  {
  CHAR16        UnicodeWeight;
  UINT8         Attributes;
  UINT8         GlyphCol1[GLYPH_HEIGHT];
  UINT8         GlyphCol2[GLYPH_HEIGHT];
  UINT8         Pad[3];
} EFI_WIDE_GLYPH;
```

Parameters

UnicodeWeight

The Unicode representation of the glyph. The term *weight* is the technical term for a character value.

Attributes

The data element containing the glyph definitions; see "Related Definitions" in **EFI_NARROW_GLYPH** for attribute values.

GlyphCol2 and *GlyphCol2*

The column major glyph representation of the character. Bits with values of one indicate that the corresponding pixel is to be on when normally displayed; those with zero are off.

Pad

Ensures that **sizeof(EFI_WIDE_GLYPH)** is twice the **sizeof(EFI_NARROW_GLYPH)**. The contents of *Pad* must be zero.

Description

Glyphs are represented via the two structures, one each for the two sizes of glyphs. The wide glyph (**EFI_WIDE_GLYPH**) is large enough to display logographic characters.

EFI_HII_FONT_PACK

Summary

A font list consists of a font header followed by a series of glyph structures. Note that fonts are not language specific.

Prototype

```
typedef struct {
  EFI_HII_PACK_HEADER        Header;
  UINT16                     NumberOfNarrowGlyphs;
  UINT16                     NumberOfWideGlyphs;
  //NARROW_GLYPH             NarrowGlyphs[];
  //WIDE_GLYPH               WideGlyphs[];
} EFI_HII_FONT_PACK;

Header.Type = EFI_HII_FONT;
```

Parameters

Header

The header contains a *Length* and *Type* field. In the case of a font package, the type will be **EFI_HII_FONT** and the length will be the total size of the font package including the size of the narrow and wide glyphs.

NumberOfNarrowGlyphs

The number of *NarrowGlyphs* that are included in the font package.

NumberOfWideGlyphs

The number of *WideGlyphs* that are included in the font package.

NarrowGlyphs

An array of **EFI_NARROW_GLYPH** entries. The number of entries is specified by *NumberOfNarrowGlyphs*.

WideGlyphs

An array of **EFI_WIDE_GLYPH** entries. The number of entries is specified by *NumberOfWideGlyphs*. To calculate the offset of *WideGlyphs*, use the offset of *NarrowGlyphs* and add the size of **EFI_NARROW_GLYPH** multiplied by the *NumberOfNarrowGlyphs*.

Description

The fonts must be presented in Unicode sort order. That is, the primary sort key is the UnicodeWeight and the secondary sort key is the SurrogateWeight.

It is up to developers who manage fonts to choose efficient mechanisms for accessing fonts. The contiguous presentation can easily be used because narrow and wide glyphs are not intermixed, so a binary search is possible (hence the requirement that the glyphs be sorted by weight);

HII Protocol Font-Related Entries

The functions described in this section are a part of the larger **EFI_HII_PROTOCOL**. This section describes the font-related entries.

EFI_HII_PROTOCOL (Font-Related Entries)

Summary

A common font database is maintained via the EFI HII protocol. The font-related entries in the protocol allow new font glyphs to be added to the database and the database to be queried.

Protocol Interface Structure

```
typedef struct _EFI_HII_PROTOCOL {
  EFI_HII_NEW_PACK                NewPack;
  EFI_HII_TEST_STRING             TestString;
  EFI_HII_GET_GLYPH               GetGlyph;
  EFI_HII_GLYPH_TO_BLT             GlyphToBlt;
  …
} EFI_HII_PROTOCOL;
```

Parameters

[MAR2]NewPack

Adds new glyphs to the database.

TestString

Checks to see if all of the Unicode characters to actualize a string are available.

GetGlyph

Translates a Unicode character into the corresponding font glyph.

GetGlyph

Translates a glyph into the format required for input to the Universal Graphics Adapter (UGA) Block Transfer (BLT) routines.

Description

The **EFI_HII_PROTOCOL** is also used as a central repository for all fonts within the environment. Glyphs may be added to the database. Two extraction mechanisms are provided, with the following differences:

- In one, a buffer is simply filled and formatting is performed externally to the mechanism.
- In the second, a buffer is filled and expanded with data.

The buffer is filled differently depending on language directionality.

EFI_HII_PROTOCOL.NewPack()

Summary

Extracts the various packs from a package list.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_NEW_PACK) (
  IN   EFI_HII_PROTOCOL            *This,
  IN   EFI_HII_PACK_LIST           *Package,
  OUT  EFI_HII_HANDLE              *Handle
  );
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Package

A pointer to an **EFI_HII_PACK_LIST** package instance.

Handle

A pointer to the **EFI_HII_HANDLE** instance.

Description

With the exception of font and keyboard data, this function adds the contents of the package list to the database and returns a handle back to the data. Font and keyboard data is kept in a common pool and will have a **NULL** handle associated with them. In the case where a *Package* contains both pooled data and database data, a valid handle will be returned upon the addition of the appropriate data into the database.

Status Codes Returned

EFI_SUCCESS	Data was extracted from the *Package*, the database was updated with the data, and *Handle* returned successfully.
EFI_INVALID_PARAMETER	The content of the *Package* was invalid.

EFI_HII_PROTOCOL.TestString()

Summary

Tests if all of the characters in a string have corresponding font characters.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_TEST_STRING) (
  IN      EFI_HII_PROTOCOL         *This,
  IN      CHAR16                   *StringToTest,
  IN OUT  UINT32                   *FirstMissing,
  OUT     UINT32                   *GlyphBufferSize
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

StringToTest

A pointer to a Unicode string.

FirstMissing

A pointer to an index into the string. On input, the index of the first character in the *StringToTest* to examine. On exit, the index of the first character encountered for which a glyph is unavailable. If all glyphs in the string are available, the index is the index of the terminator of the string.

GlyphBufferSize

A pointer to a value. On output, if the function returns **EFI_SUCCESS**, it contains the amount of memory that is required to store the string's glyph equivalent.

Description

This function may be called repeatedly to determine subsequent missing characters. Note that the index pointed to by *FirstMissing* must be incremented between calls. Line separator characters are ignored.

Status Codes Returned

EFI_SUCCESS	All glyphs are available. Note that an empty string always returns this value.
EFI_NOT_FOUND	A glyph was not found for a character.

EFI_HII_PROTOCOL.GetGlyph()

Summary

Translates a Unicode character into the corresponding font glyph.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_GET_GLYPH) (
  IN      EFI_HII_PROTOCOL        *This,
  IN      CHAR16                  *Source,
  IN OUT  UINT16                  *Index,
  OUT     UINT8                   **GlyphBuffer,
  OUT     UINT16                  *BitWidth,
  IN OUT  UINT32                  *InternalStatus
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Source

A pointer to a Unicode string.

Index

On input, the offset into the string from which to fetch the character. On successful completion, the index is updated to the first character past the character(s) making up the just extracted glyph.

GlyphBuffer

Pointer to an array where the glyphs corresponding to the characters in the source may be stored. *GlyphBuffer* is assumed to be wide enough to accept a wide glyph character.

BitWidth

If **EFI_SUCCESS** was returned, the **UINT16** pointed to by this value is filled with the length of the glyph in pixels. It is unchanged if the call was unsuccessful.

InternalStatus

To save the time required to read the string from the beginning on each glyph extraction (for example, to ensure that the narrow versus wide glyph mode is correct), this value is updated each time the function is called with the status that is local to the call. The cell pointed to by this parameter must be initialized to zero prior to invoking the call the first time for any string.

Description

This function translates a Unicode character into the corresponding font glyph. The data returned is the format required for input to the Universal Graphics Adapter (UGA) Block Transfer (BLT) routines.

Status Codes Returned

EFI_SUCCESS	It worked.
EFI_NOT_FOUND	A glyph for a character was not found.

EFI_HII_PROTOCOL.GlyphToBlt()

Summary

Translates a glyph into the format required for input to the Universal Graphics Adapter (UGA) Block Transfer (BLT) routines.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_GLYPH_TO_BLT) (
  IN      EFI_HII_PROTOCOL         *This,
  IN      UINT8                    *GlyphBuffer,
  IN      EFI_UGA_PIXEL            Foreground,
  IN      EFI_UGA_PIXEL            Background,
  IN      UINTN                    Count,
  IN      UINTN                    Width,
  IN      UINTN                    Height,
  IN OUT  EFI_UGA_PIXEL            *BltBuffer
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

GlyphBuffer

A pointer to the buffer that contains glyph data.

Foreground

The foreground setting requested to be used for the generated **BltBuffer** data.

Background

The background setting requested to be used for the generated **BltBuffer** data.

Count

The entry in the *BltBuffer* upon which to act.

Width

The width in bits of the glyph being converted.

Height

The height in bits of the glyph being converted

BltBuffer

A pointer to the buffer that contains the data that is ready to be used by the UGA Block Transfer (BLT) routines.

Description

This function translates a glyph into the format required for input to the Universal Graphics Adapter (UGA) Block Transfer (BLT) routines.

Related Definitions

```
//**********************************************
// EFI_UGA_PIXEL
//**********************************************
typedef struct {
  UINT8    Blue;
  UINT8    Green;
  UINT8    Red;
  UINT8    Reserved;
} EFI_UGA_PIXEL
```

Status Codes Returned

EFI_SUCCESS	It worked.
EFI_NOT_FOUND	A glyph for a character was not found.

Strings

String

EFI_STRING

Summary

A string is a zero-terminated array of Unicode characters.

Prototype

```
typedef CHAR16 *             EFI_STRING;
```

Description

STRING is the basis of localization.

String Package Structure

EFI_HII_STRING_PACK

Summary

A string package is used to localize strings to a particular language. The package is associated with a particular driver or set of drivers. Tools are used to associate tokens with string references in forms and in programs. These tokens are language agnostic. When paired directly or indirectly with a language pack, the string token resolves into an actual Unicode string. When passing this package as a component of the **EFI_HII_PACKAGES** structure, multiple **EFI_HII_STRING_PACK** entries are allowed.

Prototype

```
typedef struct {
  EFI_HII_PACK_HEADER    Header;
  RELOFST                LanguageNameString;
  RELOFST                PrintableLanguageName;
  UINT32                 NumStringPointers;
  UINT32                 Attributes;
  //RELOFST              StringPointers[];
  //EFI_STRING           Strings[];
} EFI_HII_STRING_PACK;
```

Parameters

Header

The header contains a *Length* and *Type* field. In the case of a font package, the type will be **EFI_HII_STRING** and the length will be the total size of the string package, including the size of the strings.

LanguageNameString

The string containing one or more ISO 639-2 three-character designator(s) of the language or languages whose translations are contained in this language pack. The first designator indicates the primary language while the others are secondary languages.

PrintableLanguageName

Contains the offset into this structure of a printable name of the language for use when prompting the user. The language printed is to be the primary language.

NumStringPointers

The number of *Strings* and *StringPointers* contained within the string package.

Attributes

Indicates the direction the language is to be printed. See “Related Definitions” below.

StringPointers

An array of strings that is indexed using string indexes that are **UINT16** tokens resolved to the various strings in the package. Each array entry is an offset from the beginning of the string package and points to the start of a Unicode string. The number of *StringPointers* in the array is defined by *NumStringPointers*.

Strings

The **NULL**-terminated Unicode strings themselves.

Description

The key element of this structure is the *StringPointer* array. This array provides the level of abstraction between the language-independent string token and the translation of that string in a particular language. The string tokens are used as indexes (0, 1, …) and not as offsets.

The actual organization of the **EFI_HII_STRING_PACK** structure may not be apparent from the structure definition. In fact, it consists of a fairly small header, an *n* entry array of string offsets, and *n* strings. Note that the only meaning associated to the strings is through the string offsets using the **STRING_TOKEN** values.

A string reference (**STRING_REF**) is a **UINT16** value defining a string to be manipulated. The string handle does not define a particular representation. Only the union of a string handle and a language name targets a particular representation (either Unicode or pixels).

Related Definitions

```
//**************************************************
// RELOFST
//**************************************************
#define    RELOFST          UINT32


//**************************************************
// STRING_REF
//**************************************************
#define    STRING_REF       UINT16


//**************************************************
// contents of EFI_HII_STRING_PACK.Attributes
//**************************************************
#define    LANG_RIGHT_TO_LEFT    0x00000001
```

Following are descriptions of the fields in the above definitions.

RELOFST	A 32-bit offset relative to the start of the encompassing string pack structure, thus providing position independence for the entire structure.
STRING_REF	A variable that can contain a **STRING_TOKEN**. When used in programs, string tokens are fundamentally constants.
LANG_RIGHT_TO_LEFT	If on, the language is intended to be printed right to left. The default (off) is to print left to right.

HII Protocol String Functions

EFI_HII_PROTOCOL (String Functions)

Summary

The HII Protocol maintains a database of strings. Strings are referred to by a triple consisting of a handle that is unique to the string pack, a **STRING_REF**, and a language.

Protocol Interface Structure

```
typedef struct _EFI_HII_PROTOCOL {
  …
  EFI_HII_NEW_PACK              NewPack;
  EFI_HII_NEW_STRING            NewString;
  EFI_HII_GET_PRI_LANGUAGES     GetPrimaryLanguages;
  EFI_HII_GET_SEC_LANGUAGES     GetSecondaryLanguages;
  EFI_HII_GET_STRING            GetString;
  EFI_HII_GET_LINE              GetLine;
  …
} EFI_HII_PROTOCOL;
```

Parameters

NewPack

Adds a new language pack to the database. See the **NewPack()** function description.

NewString

Adds a new string to an existing string pack in the database. See the **NewString()** function description.

GetPrimaryLanguages

Determines the primary languages supported by this package. See the **GetPrimaryLanguages()** function description.

GetSecondaryLanguages

Determines the secondary languages supported by a primary language in this package. See the **GetSecondaryLanguages()** function description.

GetString

Extracts a string from the string database. See the **GetString()** function description.

GetLine

Extracts enough of a string to fill a defined width. See the **GetLine()** function description.

Description

A common database is provided for the management of strings. Unlike fonts, strings are specific to specific applications or drivers. The string database performs two basic functions:

- Provides generalized extraction routines for managing and using string packs.
- Provides mechanisms for strings to be registered by one driver (via **`NewPack()`**, for example) and accessed by other drivers (particularly when used in conjunction with forms).

EFI_HII_PROTOCOL.NewPack()

Summary

Extracts the various packs from a package list.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_NEW_PACK) (
  IN   EFI_HII_PROTOCOL             *This,
  IN   EFI_HII_PACK_LIST            *Package,
  OUT  EFI_HII_HANDLE               *Handle
  );
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Package

A pointer to an **EFI_HII_PACK_LIST** package instance.

Handle

A pointer to the **EFI_HII_HANDLE** instance.

Description

With the exception of font and keyboard data, this function adds the contents of the package list to the database and returns a handle back to the data. Font and keyboard data is kept in a common pool and will have a **NULL** handle associated with them. In the case where a *Package* contains both pooled data and database data, a valid handle will be returned upon the addition of the appropriate data into the database.

Status Codes Returned

EFI_SUCCESS	Data was extracted from the *Package*, the database was updated with the data, and *Handle* returned successfully.
EFI_INVALID_PARAMETER	The content of the *Package* was invalid.

EFI_HII_PROTOCOL.NewString()

Summary

Allows a new string to be added to an already existing string package.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_NEW_STRING) (
  IN EFI_HII_PROTOCOL        *This,
  IN CHAR16                  *Language,
  IN EFI_HII_HANDLE          Handle,
  IN STRING_REF              *Reference,
  IN CHAR16                  *NewString
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Language

Pointer to a **NULL**-terminated string containing a single ISO-639-2 language identifier, indicating the language in which the string is translated in. A string consisting of all spaces indicates that the string is applicable to all languages.

Handle

The handle of the language pack to which the string is to be added.

Reference

The identifier of the string to be added. If the reference value is zero, then the string will be assigned a new identifier on that handle for the language specified. Otherwise, the string will be updated with the *NewString* Value.

NewString

The string to be added.

Description

This routine adds a new string to a string package already submitted using **NewPack()**. This string effectively overwrites existing strings.

Status Codes Returned

EFI_SUCCESS	The string effectively registered.
EFI_INVALID_PARAMETER	The *Handle* was unknown.

EFI_HII_PROTOCOL.GetPrimaryLanguages()

Summary

Allows a program to determine the primary languages that are supported on a given handle.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_GET_PRI_LANGUAGES) (
  IN  EFI_HII_PROTOCOL            *This,
  IN  EFI_HII_HANDLE              Handle,
  OUT EFI_STRING                  *LanguageString
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

The handle on which the strings reside.

LanguageString

A string allocated by **GetPrimaryLanguages()** that contains a list of all primary languages registered on the handle. The routine will not return the three-spaces language identifier used in other functions to indicate non-language-specific strings.

Description

This routine is intended to be used by drivers to query the interface database for supported languages. This routine returns a string of concatenated 3-byte language identifiers, one per string package associated with the handle.

Status Codes Returned

EFI_SUCCESS	*LanguageString* was correctly returned.
EFI_INVALID_PARAMETER	The *Handle* was unknown.

EFI_HII_PROTOCOL.GetSecondaryLanguages()

Summary

Allows a program to determine which secondary languages are supported on a given handle for a given primary language.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_GET_SEC_LANGUAGES) (
  IN   EFI_HII_PROTOCOL           *This,
  IN   EFI_HII_HANDLE             Handle,
  IN   CHAR16                     *PrimaryLanguage,
  OUT  EFI_STRING                 *LanguageString
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

The handle on which the strings reside.

PrimaryLanguage

Pointer to a **NULL**-terminated string containing a single ISO-639-2 language identifier, indicating the primary language.

LanguageString

A string allocated by **GetSecondaryLanguages()** containing a list of all secondary languages registered on the handle. The routine will not return the three-spaces language identifier used in other functions to indicate non-language-specific strings, nor will it return the primary language. This function succeeds but returns a **NULL** *LanguageString* if there are no secondary languages associated with the input *Handle* and *PrimaryLanguage* pair.

Description

Each string package has associated with it a single primary language and zero or more secondary languages. This routine returns the secondary languages associated with a string package. The string package is identified by the package list handle and the (currently three-character ISO-639-2 primary language identifier.

Status Codes Returned

EFI_SUCCESS	*LanguageString* was correctly returned.
EFI_INVALID_PARAMETER	The *Handle* was unknown.

EFI_HII_PROTOCOL.GetString()

Summary

Extracts a string from a package already registered with the EFI HII database.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_GET_STRING) (
  IN      EFI_HII_PROTOCOL        *This,
  IN      EFI_HII_HANDLE          Handle,
  IN      STRING_REF              Token,
  IN      BOOLEAN                 Raw,
  IN      CHAR16                  *LanguageString,
  IN OUT  UINT16                  *BufferLength,
  OUT     EFI_STRING              *StringBuffer
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

The handle on which the string resides.

Token

The string token assigned to the string.

Raw

If **TRUE**, the string is returned unedited in the internal storage format described above. If false, the string returned is edited by replacing <cr> with <space> and by removing special characters such as the <wide> prefix.

LanguageString

Pointer to a **NULL**-terminated string containing a single ISO-639-2 language identifier, indicating the language to print. If the *LanguageString* is empty (starts with a **NULL**), the default system language will be used to determine the language.

BufferLength

Length of the *StringBuffer*. If the status reports that the buffer width is too small, this parameter is filled with the length of the buffer needed.

StringBuffer

The buffer designed to receive the characters in the string.

Description

This routine extracts a string from the package database. The string may be extracted in internal or external formats.

Status Codes Returned

EFI_SUCCESS	*StringBuffer* is filled with a **NULL**-terminated string.
EFI_INVALID_PARAMETER	The handle or string token is unknown.
EFI_BUFFER_TOO_SMALL	The buffer provided was not large enough to allow the entire string to be stored.

EFI_HII_PROTOCOL.GetLine()

Summary

Allows a program to extract a part of a string of not more than a given width.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_GET_LINE) (
  IN      EFI_HII_PROTOCOL        *This,
  IN      EFI_HII_HANDLE          Handle,
  IN      STRING_REF              Token,
  IN OUT  UINT16                  *Index,
  IN      UINT16                  LineWidth,
  IN      CHAR16                  *LanguageString,
  IN OUT  UINT16                  *BufferLength,
  OUT     EFI_STRING              *StringBuffer
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

The handle on which the string resides.

Token

The string token assigned to the string.

Index

On input, the offset into the string where the line is to start. On output, the index is updated to point to beyond the last character returned in the call. The interface is designed so that repeated calls will fill the buffer with subsequent parameters.

LineWidth

The maximum width of the line in units of narrow glyphs. Specific line breaks (as in the case of two carriage returns) are still honored resulting in separate lines. The buffer is padded to the length in narrow spaces.

LanguageString

Pointer to a **NULL**-terminated string containing a single ISO-639-2 language identifier, indicating the language to print. If the *LanguageString* is empty (starts with a **NULL**) the default system language will be used to determine the language.

BufferLength

Pointer to the length of the *StringBuffer*. If the status reports that the buffer width is too small, this parameter is filled with the length of the buffer needed.

StringBuffer

The buffer designed to receive the characters in the string.

Description

This function is used to extract parts of a string so that those parts of strings fit inside a column of a defined width. With repeated calls, this design allows a calling program to extract "lines" of text that fit inside columns. The effort of measuring the fit of strings inside columns is localized to this call. This functionality is commonly used in menuing applications.

Status Codes Returned

EFI_SUCCESS	*StringBuffer* filled with characters that will fit on the line.
EFI_NOT_FOUND	The font glyph for at least one of the characters in the string is not in the font database.
EFI_BUFFER_TOO_SMALL	The buffer provided was not large enough to allow the entire string to be stored. Note that the *BufferWidth* may need to be larger than the *LineWidth* due to, for example, nonspacing characters.

Form Packages

Form Language Syntax

The language described here is the “machine language” of the form set. Syntactic sugar to hide the complexities of the language (for example, use of infix rather than postfix notation for the expressions) is beyond the scope of this document.

Meta-Syntax

The description of the syntax uses Backus-Naur Form (BNF) with the following extensions:

- **[** to **]*** indicates zero or more repetitions of the contents.
- **[** to **]+** indicates one or more repetitions of the contents.
- **[** to **]n…m** (n and m are integers) indicates **n** through **m** repetitions.
- **[** to **]n…** (n is an integer) indicates at least **n** repetitions (so **[x]+** is equivalent to **[x]1…**).
- **[** to **]** indicates that the contents are optional.

Internal Form Representation (IFR) Language Syntax Definition

EFI_IFR_OP_HEADER

Summary

Defines the form tag header.

Prototype

```
typedef struct _EFI_IFR_OP_HEADER {
  UINT8                     OpCode;
  UINT8                     Length;
} EFI_IFR_OP_HEADER;
```

Parameters

OpCode

Defines which type of operation is being described by this header. See "Related Definitions" below for the defined IFR opcodes, which are then defined in the following sections.

Length

Defines the number of bytes in the tag, including the opcode.

Description

Forms are represented in a binary format roughly similar to processor instructions. Each IFR instruction is interchangeably called a *tag* or an *operation*. *Tag* is preferred because the functionality is analogous to tags in higher-level markup languages.

Each tag starts with an opcode followed by a **UINT8** constant and then a **UINT8** length. The length defines the number of bytes in the tag, including the opcode. The length is used so that new opcodes can be added. An IFR browser is responsible for skipping over tags that it does not understand.

Question tags are those that allow user input that is visible in the results when a browser processes a form. Question tags use a triple containing the following information to describe the tag:

- An ID assigned to the question that is unique inside the form package.
- An offset into some sort of NVRAM storage (in bytes).
- A storage width (in bytes).

Although not required, it is expected that a tool will assign these values. Note that the utility of the offset and width values, in particular, varies with how the results are to be processed.

Related Definitions

```
//
// IFR Opcodes
//
#define EFI_IFR_FORM_OP                0x01
#define EFI_IFR_SUBTITLE_OP            0x02
#define EFI_IFR_TEXT_OP                0x03
#define EFI_IFR_GRAPHIC_OP             0x04
#define EFI_IFR_ONE_OF_OP              0x05
#define EFI_IFR_CHECKBOX_OP            0x06
#define EFI_IFR_NUMERIC_OP             0x07
#define EFI_IFR_PASSWORD_OP            0x08
#define EFI_IFR_ONE_OF_OPTION_OP       0x09    // ONEOF OPTION field
#define EFI_IFR_SUPPRESS_IF_OP         0x0A
#define EFI_IFR_END_FORM_OP            0x0B
#define EFI_IFR_HIDDEN_OP              0x0C
#define EFI_IFR_END_FORM_SET_OP        0x0D
#define EFI_IFR_FORM_SET_OP            0x0E
#define EFI_IFR_REF_OP                 0x0F
#define EFI_IFR_END_ONE_OF_OP          0x10
#define EFI_IFR_INCONSISTENT_IF_OP     0x11
#define EFI_IFR_EQ_ID_VAL_OP           0x12
#define EFI_IFR_EQ_ID_ID_OP            0x13
#define EFI_IFR_EQ_ID_LIST_OP          0x14
#define EFI_IFR_AND_OP                 0x15
#define EFI_IFR_OR_OP                  0x16
#define EFI_IFR_NOT_OP                 0x17
#define EFI_IFR_END_IF_OP              0x18
#define EFI_IFR_GRAYOUT_IF_OP          0x19
#define EFI_IFR_DATE_OP                0x1A
#define EFI_IFR_TIME_OP                0x1B
#define EFI_IFR_STRING_OP              0x1C
#define EFI_IFR_LABEL_OP               0x1D
#define EFI_IFR_SAVE_DEFAULTS_OP       0x1E
#define EFI_IFR_RESTORE_DEFAULTS_OP    0x1F
#define EFI_IFR_BANNER_OP              0x20
#define EFI_IFR_INVENTORY_OP           0x21
```

Form Package Syntax

Prototype

```
<form-package> ::= <form-package-header> [<form>]* <end-form-package-op>
       <form-package-header> ::= <form-package-op> <length> <form-id>
              <form-package-op> ::= UINT8-constant
<end-form-package-op> ::= UINT8-constant <length>
```

Description

The form package consists of a header, a set of forms, and an end-of-form operation. The **form-id** parameter defines the form package's initial form when used by a browser.

Form Tag

Prototype

```
<form> ::= <form header> <form-body> <end-form>
<form-body> ::= [<form-stmt>]*

typedef struct {
  EFI_IFR_OP_HEADER           Header;
  UINT16                      FormId;
  STRING_REF                  FormTitle;
} EFI_IFR_FORM;

typedef struct {
  EFI_IFR_OP_HEADER           Header;
} EFI_IFR_END_FORM;

<form-stmt> ::= <subtitle> | <text> | <one-of> | <many-of> | <numeric> |
<password> | <consistency> | <list> | <grayout> | <hidden> | <label> | <ref> |
<suppress> | <img>
```

Parameters

Header

The sequence that defines the type of opcode as well as the length of the opcode being defined.

FormId

The unique identification for this particular form.

FormTitle

The string token reference to the title of this particular form.

Description

A form is the encapsulation of what amounts to a browser page. The header defines a *FormId*, which is referenced by the form package, among others. It also defines a *FormTitle*, which is a string to be used as the title for the form.

EFI_IFR_SUBTITLE

Summary

Defines the subtitle tag.

Prototype

```
typedef struct {
  EFI_IFR_OP_HEADER          Header;
  STRING_REF                 SubTitle;
} EFI_IFR_SUBTITLE;
```

Parameters

Header

The sequence that defines the type of opcode as well as the length of the opcode being defined.

SubTitle

The string token reference to a subtitle opcode.

Description

Subtitle strings are intended to be used by authors to separate sections of questions into semantic groups.

EFI_IFR_TEXT

Summary

Defines the text tag.

Prototype

```
typedef struct {
  EFI_IFR_OP_HEADER     Header;
  STRING_REF            Help;
  STRING_REF            Text;
  STRING_REF            TextTwo;
  UINT8                 Flags;
  UINT16                Key;
} EFI_IFR_TEXT;
```

Parameters

Header

The sequence that defines the type of opcode as well as the length of the opcode being defined.

FormTitle

The string token reference to the title of this particular form.

Help

The string token reference to the help string for this opcode

Text

The string token reference to the primary string for this opcode.

TextTwo

The string token reference to the secondary string for this opcode

Flags

This parameter is included solely for dynamic support.

Key

The value to be passed to the caller to identify this particular opcode.

Description

Unlike HTML, text is simply another tag. This tag type enables IFR to be more easily localized.

EFI_IFR_ONE_OF

Summary

Defines the one-of tag.

Prototype

```
<one-of> ::=  <one-of-tag> [<one-of-body-tags>]2… <one-of-end-tag>

typedef struct {
  EFI_IFR_OP_HEADER          Header;
  UINT16                     QuestionId;
  UINT8                      Width;
  STRING_REF                 Prompt;
  STRING_REF                 Help;
} EFI_IFR_ONE_OF;

typedef struct {
  EFI_IFR_OP_HEADER          Header;
  STRING_REF                 Option;
  UINT16                     Value;
  UINT8                      Flags;
  UINT16                     Key;
} EFI_IFR_ONE_OF_OPTION;

typedef struct {
  EFI_IFR_OP_HEADER          Header;
} EFI_IFR_END_ONE_OF;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

QuestionId

The unique value that identifies the particular question being defined by the opcode. This value will correspond to the starting offset in nonvolatile RAM (NVRAM) from which the settings for this question are being read and written to.

Width

Identifies the size of NVRAM.

Prompt

The string token reference to the prompt string for this particular opcode.

Help

The string token reference to the help string for this particular opcode.

Option

The string token reference to the option description string for this particular opcode.

Value

The value associated with the **EFI_IFR_ONE_OF_OPTION** that was chosen. This value is what is used to determine which option is currently active.

Flags

A bit-mask that determines which unique settings are active for this opcode. See "Related Definitions" below.

Key

A unique value that the browser passes back to a consumer by the browser if the **EFI_IFR_FLAG_INTERACTIVE** flag is set and a user selects this opcode. Type **EFI_IFR_FLAG_INTERACTIVE** is defined in "Related Definitions" below.

Description

The one-of tag is a nested question type. It consists of a one-of header operation, several one-of-body operations, and an end tag.

Related Definitions

```
#define EFI_IFR_FLAG_DEFAULT            0x01
#define EFI_IFR_FLAG_MANUFACTURING      0x02
#define EFI_IFR_FLAG_INTERACTIVE        0x04
#define EFI_IFR_FLAG_NV_ACCESS          0x08
#define EFI_IFR_FLAG_RESET_REQUIRED     0x10
#define EFI_IFR_FLAG_LATE_CHECK         0x20
```

EFI_IFR_CHECKBOX

Summary

Defines the checkbox tag.

Prototype

```
typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    QuestionId;
  UINT8                     Width;
  STRING_REF                Prompt;
  STRING_REF                Help;
  UINT8                     Flags;
  UINT16                    Key;
} EFI_IFR_CHECKBOX;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

QuestionId

The unique value that identifies the particular question being defined by the opcode. This value will correspond to the starting offset in NVRAM from which the settings for this question are being read and written to.

Width

Identifies the size of nonvolatile RAM.

Prompt

The string token reference to the prompt string for this particular opcode.

Help

The string token reference to the help string for this particular opcode.

Flags

A bit-mask that determines which unique settings are active for this opcode. See "Related Definitions" below for defined flags for this opcode.

Key

A unique value that the browser passes back to a consumer if the **EFI_IFR_FLAG_INTERACTIVE** flag is set and a user selects this opcode. Type **EFI_IFR_FLAG_INTERACTIVE** is defined in **EFI_IFR_ONE_OF**.

Description

The checkbox tag returns zero if the box is not checked and one if it is. The default is stored in bit position zero of the flag.

Related Definitions

```
#define EFI_IFR_CHECKBOX_DEFAULT  1
```

EFI_IFR_NUMERIC

Summary

Defines the numeric tag.

Prototype

```
typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    QuestionId;
  UINT8                     Width;
  STRING_REF                Prompt;
  STRING_REF                Help;
  UINT8                     Flags;
  UINT16                    Key;
  UINT16                    MinValue;
  UINT16                    MaxValue;
  UINT16                    Step;
  UINT16                    Default;
} EFI_IFR_NUMERIC;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

QuestionId

The unique value that identifies the particular question being defined by the opcode. This value will correspond to the starting offset in NVRAM from which the settings for this question are being read and written to.

Width

Identifies the size of nonvolatile RAM.

Prompt

The string token reference to the prompt string for this particular opcode.

Help

The string token reference to the help string for this particular opcode.

Flags

A bit-mask that determines which unique settings are active for this opcode.

Key

A unique value which is passed back to a consumer by the browser if the **EFI_IFR_FLAG_INTERACTIVE** flag is set and a user selects this opcode. Type **EFI_IFR_FLAG_INTERACTIVE** is defined in **EFI_IFR_ONE_OF**.

MinValue

The minimum value to be accepted by the browser for this opcode.

MaxValue

The maximum value to be accepted by the browser for this opcode.

Step

Defines the amount to increment or decrement the value each time a user requests a value change. If the step value is 0, then the input mechanism for the numeric value is to be free-form and require the user to type in the actual value.

Default

The default value for this opcode.

Description

The parameters allow for expression of a rich variety of numeric inputs that may be validated by the browser prior to submission. Valid input (*n*) is:

```
MinValue <= n <= MaxValue
int((n-MinValue)/Step) = (n-MinValue)/Step
```

The range data may be used to provide better keys help for the user as well as for internal validation. HTML has no equivalent of a numeric tag, so a string tag is used along with scripting to provide limit checking.

EFI_IFR_PASSWORD

Summary

Defines the password tag.

Prototype

```
typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    QuestionId;
  UINT8                     Width;
  STRING_REF                Prompt;
  STRING_REF                Help;
  UINT8                     Flags;
  UINT16                    Key;
  UINT8                     MinSize;
  UINT8                     MaxSize;
  UINT16                    Encoding;
} EFI_IFR_PASSWORD;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

QuestionId

The unique value that identifies the particular question being defined by the opcode. This value will correspond to the starting offset in NVRAM from which the settings for this question are being read and written to.

Width

Identifies the size of nonvolatile RAM.

Prompt

The string token reference to the prompt string for this particular opcode.

Help

The string token reference to the help string for this particular opcode.

Flags

A bit-mask that determines which unique settings are active for this opcode.

Key

A unique value that is passed back to a consumer by the browser if the **EFI_IFR_FLAG_INTERACTIVE** flag is set and a user selects this opcode. Type **EFI_IFR_FLAG_INTERACTIVE** is defined in **EFI_IFR_ONE_OF**.

MinSize

The minimum number of characters that can be accepted for this opcode.

MaxSize

The maximum number of characters that can be accepted for this opcode.

Encoding[ceu5]

A value to determine if password encoding is required. If **TRUE**, then the processing of the password by the browser will be run through a built-in encoding mechanism. Otherwise, the data will be processed in its raw form.

Description

This opcode provides the ability to define password capability and its associated storage offsets. In addition, this opcode provides the ability to have the contents that are being read and written to either be encoded or not.

EFI_IFR_ORDERED_LIST

Summary

Defines the ordered list tag.

Prototype

```
<ordered-list> ::= <one-of-tag> [<one-of-body-tags>]2… <one-of-end-tag>

typedef struct {
  EFI_IFR_OP_HEADER          Header;
  UINT16                     QuestionId;
  UINT8                      MaxEntries;
  STRING_REF                 Prompt;
  STRING_REF                 Help;
} EFI_IFR_ORDERED_LIST;

typedef struct {
  EFI_IFR_OP_HEADER           Header;
  STRING_REF                  Option;
  UINT16                      Value;
  UINT8                       Flags;
  UINT16                      Key;
} EFI_IFR_ONE_OF_OPTION;

typedef struct {
  EFI_IFR_OP_HEADER           Header;
} EFI_IFR_END_ONE_OF;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

QuestionId

The unique value which identifies the particular question being defined by the opcode. This value will correspond to the starting offset in non-volatile RAM that the settings for this question are being read from and written to.

MaxEntries

The maximum number of entries for which this tag will maintain an order. This value also identifies the size of the storage associated with this tag's ordering array.

Prompt

The string token reference to the prompt string for this particular opcode.

Help

The string token reference to the help string for this particular opcode.

Option

The string token reference to the option description string for this particular opcode.

Value

The value associated with the **EFI_IFR_ONE_OF_OPTION** that was chosen. This value is what is used to determine which option is currently active. For ordered lists, the value of 0 is reserved and should not be used.

Flags

A bit-mask that determines which unique settings are active for this opcode. See "Related Definitions" below.

Key

A unique value that is passed back to a consumer by the browser if the **EFI_IFR_FLAG_INTERACTIVE** flag is set and a user selects this opcode. Type **EFI_IFR_FLAG_INTERACTIVE** is defined in **EFI_IFR_ONE_OF**.

Description

The ordered list does not have a direct analogy in HTML. It is intended to be used for cases such as defining the boot order. This opcode's use is very similar to the **EFI_IFR_ONE_OF** opcode where there are corresponding options contained within this particular opcode. The values of each option are what is recorded in the nonvolatile variable that is associated with this opcode. For example, if this opcode has 3 options associated with it, and the values were 3, 4, and 5, one might expect the storage destination to look like "345." If the order of these opcodes is changed, the settings would potentially be something such as "534." One thing to note is that valid values for the options in ordered lists should never be a 0. The value of 0 is used to determine if a particular "slot" in the array is empty. Therefore, if in the previous example 3 was followed by a 4 and then followed by a 0, the valid options to be displayed would be 3 and 4 only.

EFI_IFR_REF

Summary

Defines the ref tag.

Prototype

```
<ref> ::= <ref-op> <length> <form-id> <string>
<ref-op> ::= UINT8-constant

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    FormId;
  STRING_REF                Prompt;
  STRING_REF                Help;
  UINT8                     Flags;
  UINT16                    Key;
} EFI_IFR_REF;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

FormId

The unique value which identifies the form this opcode is referring to.

Prompt

The string token reference to the prompt string for this particular opcode.

Help

The string token reference to the help string for this particular opcode.

Flags

A bit-mask that determines which unique settings are active for this opcode.

Key

A unique value which is passed back to a consumer by the browser if the **EFI_IFR_FLAG_INTERACTIVE** flag is set and a user selects this opcode. Type **EFI_IFR_FLAG_INTERACTIVE** is defined in **EFI_IFR_ONE_OF**.

Description

The ref tag is the equivalent of an HTML hypertext link. IFR limits links to the start of other forms whereas HTML supports arbitrary hypertext links.

EFI_IFR_HIDDEN

Summary

Defines the hidden tag.

Prototype

```
typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    Value;
  UINT16                    Key;
} EFI_IFR_HIDDEN;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

Value

A value to associate with this particular opcode. This value is typically used for revision information and will not affect the user interface.

Key

A unique value that can be used to identify a particular hidden opcode and determine its *Value*. This uniqueness is essential when multiple hidden opcodes exist each with a different intention.

Description

Hidden input allows for communication of revision data between the creator of the tags and the consumer, for example. The user generally should not see hidden tags. Hidden tags can be used inside IFR along with the `<grayout>` and `<suppress>` tags to control display of optional data.

EFI_IFR_GRAY_OUT

Summary

Defines the grayout tag.

Prototype

```
<grayout> ::= <grayout-op> <length> <RPN expression>

typedef struct {
  struct _EFI_IFR_OP_HEADER      Header;
} EFI_IFR_GRAY_OUT;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

Description

The **<grayout>** tag causes the following tag to be displayed in a special display form that is used for inaccessible options if the Boolean expression evaluates to **TRUE**. Developers writing IFR should realize that different browsers will support this option to varying degrees. In particular, HTML has no similar construct so it may not support this facility.

EFI_IFR_SUPPRESS

Summary

Defines the suppress numeric tag.

Prototype

```
<suppress> ::= supress-op <length> <rpn-bool-expr> \
               <form-stmts> end-suppress-op

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    QuestionId;
  UINT16                    Value;
} EFI_IFR_SUPRESS;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined

QuestionId

The unique value that identifies the particular question being defined by the opcode. This value will correspond to the starting offset in NVRAM for which the settings for this question are being read and written to.

Value

The value against which the contents of the *QuestionId* will be compared.

Description

The suppress tag causes the following tag to be hidden from the user if the Boolean expression evaluates to **TRUE**. As with **<grayout>**, the quality of support may vary from browser to browser. HTML itself does not have a mechanism to provide this functionality

EFI_IFR_INCONSISTENT

Summary

Defines the inconsistency tag.

Prototype

```
<inconsistency> ::= inconsistency-op <length> <Popup> <rpn-bool-expr>
```

```
typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  STRING_REF                Popup;
} EFI_IFR_INCONSISTENT;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

Popup

The string token reference to the string that will be used for the consistency check message.

Description

This tag uses a Boolean expression to allow the IFR creator to check options in a richer manner than provided by the question tags themselves. For example, this tag might be used to validate that two options are not using the same address or that the numbers that were entered align to some pattern (such as leap years and February in a date input field). The tag provides a string to be used in a “pop-up” display to alert the user to the issue. Inconsistency tags might be evaluated when the user traverses from tag to tag or only upon submission. The user should not be allowed to submit the results of a form inconsistency.

EFI_IFR_LABEL

Summary

Defines the label tag.

Prototype

```
typedef struct {
  struct _EFI_IFR_OP_HEADER      Header;
  EFI_FORM_LABEL                 LabelId;
} EFI_IFR_LABEL;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

Label

A unique value that does not affect the user interface but provides a location to which IFR can be added or removed via the **EFI_HII_PROTOCOL.UpdateForm()** function.

Description

This tag is used to provide a base for possible runtime additions to the form. The label must be unique to the form package in which it resides.

EFI_IFR_VARSTORE

Summary

Defines the variable store tag.

Prototype

```
typedef struct {
  EFI_IFR_OP_HEADER            Header;
  EFI_GUID                     Guid;
  UINT16                       VarId;
  UINT16                       Size;
  //UINT8                      Name[];
} EFI_IFR_VARSTORE;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

Guid

The variable's GUID definition. This field comprises one half of the variable name, with the other half being the human-readable aspect of the name, which is represented by the string immediately following the *Size* field.

VarId

The variable storage ID. This field is the value that is used to uniquely identify this **EFI_IFR_VARSTORE** definition instance from others. Opcodes such as **EFI_IFR_VARSTORE_SELECT**, which is the variable store selection opcode, will refer to this field to designate which is the active variable that is being used.

Size

The size of the variable storage repository.

Name

This field is actually not defined in the structure but is included here to illustrate the content of the encoding for this opcode. Because this field is variable in length, the string is a **NULL**-terminated string and the overall size will be reflected in the opcode's *Header* field. Additionally, there is an expectation that this field will not exceed 40 characters in length.

Description

This tag is used to provide a definition of a variable that can be used for purposes of establishing custom nonvolatile storage destinations. These opcodes will generally be used once in a given form set and will apply globally across the form set.

EFI_IFR_VARSTORE_SELECT

Summary

Defines the variable store select tag.

Prototype

```
typedef struct {
  EFI_IFR_OP_HEADER              Header;
  UINT16                         VarId;
} EFI_IFR_VARSTORE_SELECT;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

VarId

The variable storage ID. This field is the value that is used to uniquely identify the **EFI_IFR_VARSTORE** definition instance that opcodes are to use until a time such as another variable storage select opcode appearing.

Description

This tag is used to define what the active variable storage definition is to use for the opcodes that follow this tag. All opcodes that refer to configuration settings that are stored in variables will be affected by this tag. To avoid having each and every opcode be burdened with a field that specifies which variable storage the opcode uses, this tag is intended as a means by which the IFR compiler can set the "active" variable storage to use for a given opcode. When the context of an opcode's storage must change, this tag will again be embedded with the appropriate **VarId** information for the opcodes that follow.

EFI_IFR_VARSTORE_SELECT_PAIR

Summary

Defines the variable store select pair tag.

Prototype

```
typedef struct {
  EFI_IFR_OP_HEADER                Header;
  UINT16                           VarId;
  UINT16                           SecondaryVarId;
} EFI_IFR_VARSTORE_SELECT_PAIR;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

VarId

The variable storage ID. This field is the value that is used to uniquely identify the **EFI_IFR_VARSTORE** definition instance that is to be used by opcodes until such a time as another variable storage select opcode appears.

SecondaryVarId

The variable storage ID. This field is the value that is used to uniquely identify the **EFI_IFR_VARSTORE** definition instance that is to be used by opcodes until such a time as another variable storage select opcode appears.

Description

This tag is used primarily in the case where a Boolean expression needs to be interpreted where the value of two opcode settings need to be compared and each of the opcodes reside in a different variable storage. This opcode does not affect the “active” variable setting and will only apply to the following opcode, which is a Boolean expression that compares the settings of two different variable IDs.

Boolean Expressions

Summary

Defines Boolean expressions.

Prototype

```
<rpn-bool-expr> ::=  <bool-expr> <rpn-expr-end-op>
           <rpn-expr-end-op> ::= UINT8-constant
           <bool-expr> ::= <bool-primitive> |
           <bool-expr> <not-op> |
           <bool-expr> <bool-expr> <and-op> |
           <bool-expr> <bool-expr> <or-op>
           <not-op>, <and-op>, <or-op> ::= UINT8-constant
<bool-primitive> ::=  <id-val> | <id-val-list> | <id-id>
<id-val> ::= <id-val-op> <name-id>  <value>
<id-val-list> ::= <id-val-list-op> <name-id> <value-list-length>
<value-list> ::= <length> [<value>]n..n
<id-id> ::= <id-id-op> <name-id> <name-id> <id-val-op>, \
            <id-val-list-op>, \
            <id-id-op> ::= UINT8-constant
<length> ::= UINT-8-value

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    QuestionId;
  UINT16                    Value;
} EFI_IFR_EQ_ID_VAL;

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    QuestionId;
  UINT16                    ListLength;
  UINT16                    ValueList[1];
} EFI_IFR_EQ_ID_LIST;

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
  UINT16                    QuestionId1;
  UINT16                    QuestionId2;
} EFI_IFR_EQ_ID_ID;

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
} EFI_IFR_AND;
```

```
typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
} EFI_IFR_OR;

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
} EFI_IFR_NOT;

typedef struct {
  struct _EFI_IFR_OP_HEADER Header;
} EFI_IFR_END_EXPR;
```

Parameters

Header

The byte sequence that defines the type of opcode as well as the length of the opcode being defined.

QuestionId

The unique value that identifies the particular question being referenced by the opcode.

QuestionId2

The unique value that identifies the particular question being referenced by the opcode.

Value

The value to which the question being referenced will be compared.

ListLength

The length of the list of values against which to be compared.

ValueList

The list of values against which a particular question will be compared.

Description

A Boolean expression is a postfix (Reverse Polish Notation) equation that evaluates to true or false. The terminal entries allow for assertions that two questions contain the same data values, that a question's value equals a constant, and that a question's value is in a list of constant values. Higher-level operations are **AND**, **OR**, and **NOT**.

The value of **<length>** is the number of bytes from and including the opcode to the end of the operation. The "end of the operation" is defined to be the byte preceding an operation with its opcode and length field.

EFI HII Protocol Forms Entries

EFI_HII_PROTOCOL (Forms Entries)

Summary

The EFI HII protocol maintains a database of forms. Forms packages are referred to by a handle while forms are referred to by a handle and the form id.

Protocol Interface Structure

```
typedef struct _EFI_HII_PROTOCOL {
  …
  EFI_HII_NEW_PACK                NewPack;
  EFI_HII_GET_FORMS               GetForms;
  EFI_HII_UPDATE_FORM             UpdateForm;
  …
} EFI_HII_PROTOCOL;
```

Parameters

NewPack

Adds a new pack to the database. See the **NewPack()** function description.

GetForms

Extracts one or more forms from the database. See the **GetForms()** function description.

UpdateForm

Adds new elements to the form. See the **UpdateForm()** function description.

Description

The forms functions allow for the addition of forms to the HII database, for extraction of those forms, and for update.

Related Definitions

```
//*******************************************************
// EFI_FORM_ID
//*******************************************************
typedef UINT16        EFI_FORM_ID;
```

```
//*******************************************************
// EFI_HII_UPDATE_DATA
//*******************************************************
typedef struct {
  BOOLEAN           FormSetUpdate;
  EFI_PHYSICAL_ADDRESS  FormCallbackHandle;
  BOOLEAN           FormUpdate;
  UINT16            FormValue;
  STRING_REF        FormTitle;
  UINT16            DataCount;
  UINT8             *Data;
} EFI_HII_UPDATE_DATA;
```

FormSetUpdate

If **TRUE**, indicates that the *FormCallbackHandle* value will be used to update the contents of the *CallBackHandle* entry in the form set.

FormCallbackHandle

This parameter is valid only when *FormSetUpdate* is **TRUE**. The value in this parameter will be used to update the contents of the *CallbackHandle* entry in the form set.

FormUpdate

If **TRUE**, indicates that the *FormTitle* contents will be used to update the *FormValue*'s title.

FormValue

Specifies which form is to be updated if the *FormUpdate* value is **TRUE**.

FormTitle

This parameter is valid only when the *FormUpdate* parameter is **TRUE**. The value in this parameter will be used to update the contents of the form title.

DataCount

The number of *Data* entries in this structure.

Data

An array of 1+ opcodes, specified by *DataCount*.

EFI_HII_PROTOCOL.NewPack()

Summary

`NewPack()` is the common method for submitting new packages to the HII Protocol for addition to the database. For forms, the form package is registered with the protocol.

No additional status returns are defined due to forms.

EFI_HII_PROTOCOL.GetForms()

Summary

This function allows a program to extract a form or form package that has previously been registered with the HII database.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_GET_FORMS) (
  IN      EFI_HII_PROTOCOL        *This,
  IN      EFI_HII_HANDLE          Handle,
  IN      EFI_FORM_ID             FormId,
  IN OUT  UINT16                  *BufferLength,
  OUT     UINT8                   *Buffer
);
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

Handle on which the form resides.

FormId

The ID of the form to return. If the ID is zero, the entire form package is returned.

BufferLength

On input, the length of the *Buffer*. On output, the length of the returned buffer, if the length was sufficient and, if it was not, the length that is required to fit the requested form(s).

Buffer

The buffer designed to receive the form(s).

Description

This function is used to extract a form or forms.

Status Codes Returned

EFI_SUCCESS	*Buffer* filled with the requested forms. *BufferLength* updated.
EFI_INVALID_PARAMETER	Unknown handle.
EFI_NOT_FOUND	A form on the requested handle cannot be found with the requested *FormId*.
EFI_BUFFER_TOO_SMALL	The buffer provided was not large enough to allow the form to be stored.

EFI_HII_PROTOCOL.UpdateForm()

Summary

This function allows the caller to update a form or form package that has previously been registered with the EFI HII database.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HII_UPDATE_FORM) {
  IN EFI_HII_PROTOCOL        *This,
  IN EFI_HII_HANDLE          Handle,
  IN EFI_FORM_LABEL          Label,
  IN BOOLEAN                 AddData,
  IN EFI_HII_UPDATE_DATA     *Data
};
```

Parameters

This

A pointer to the **EFI_HII_PROTOCOL** instance.

Handle

Handle of the package where the form to be updated resides.

Label

The label inside the form package where the update is to take place.

AddData

If **TRUE**, adding data at a given *Label*; otherwise, if **FALSE**, removing data at a given *Label*. If **FALSE**, it will not allow the removal of the end of a form.

Data

The buffer containing the new tags to insert after the *Label*.

Description

This function allows a program to update a form at runtime. The form must have been built expecting the update, because a label tag is required. The tags in *Buffer* are inserted into the form just after the label tag.

Status Codes Returned

EFI_SUCCESS	The form was updated with the new tags.
EFI_INVALID_PARAMETER	The buffer for the buffer length does not contain an integral number of tags.
EFI_NOT_FOUND	The *Handle*, *Label*, or *FormId* was not found.

Dynamic Processing of NV/IFR Data

Form Callback Protocol

EFI_FORM_CALLBACK_PROTOCOL

Summary

The **EFI_FORM_CALLBACK_PROTOCOL** is the defined interface for access to custom NVS devices as well as communication of user selections in a more interactive environment. This protocol should be published by hardware-specific drivers that want to export access to custom hardware storage or publish IFR that has a requirement to call back the original driver.

GUID

```
#define EFI_FORM_CALLBACK_PROTOCOL_GUID  \
   { 0xf3e4543d, 0xcf35, 0x6cef, 0x35, 0xc4, 0x4f, 0xe6, \
   0x34, 0x4d, 0xfc, 0x54 }
```

Protocol Interface Structure

```
typedef struct _EFI_FORM_CALLBACK_PROTOCOL {
  EFI_NV_READ                   NvRead;
  EFI_NV_WRITE                  NvWrite;
  EFI_FORM_CALLBACK             Callback;
} EFI_FORM_CALLBACK_PROTOCOL;
```

Parameters

NvRead

The read operation to access the NV data serviced by a hardware-specific driver. See the **NvRead()** function description.

NvWrite

The write operation to access the NV data serviced by a hardware-specific driver. See the **NvWrite()** function description.

Callback

The function that is called from the configuration browser to communicate key value pairs. See the **Callback()** function description.

Description

This interface is provided by hardware-specific drivers that control access to nonsystem NVS and support callbacks from the browser or HII.

Related Definitions

```
//*******************************************************
// EFI_HII_CALLBACK_PACKET
//*******************************************************
typedef union {
  EFI_IFR_DATA_ARRAY    DataArray;
  EFI_IFR_PACKET        DataPacket;
  CHAR16                *String;
} EFI_HII_CALLBACK_PACKET;
```

DataArray

Refers to an array of entries that describes the current configuration settings as well as directives that are communicated back to the browser

DataPacket

Describes string and IFR content that is being passed back to the browser to display. This content is used mainly by drivers that need to interact directly with the browser without using the HII repository as an intermediary.

String

If a callback will return with an error and the driver wants the browser to display if returning an error, it fills the string with null-terminated contents.

```
//*******************************************************
// EFI_IFR_DATA_ARRAY
//*******************************************************
typedef struct {
  VOID                *NvRamMap;
  UINT32              EntryCount;
} EFI_IFR_DATA_ARRAY;
```

NvRamMap

If the flag of the opcode specified to retrieve a copy of the NVRAM map, this parameter is a pointer to a buffer copy.

EntryCount

The number of **EFI_IFR_DATA_ENTRY** entries.
Note that immediately following the *EntryCount* is an array of **EFI_IFR_DATA_ENTRY** structures. The number of iterations is defined by the *EntryCount* value.

```
//*******************************************************
// EFI_IFR_DATA_ENTRY
//*******************************************************
typedef struct {
  UINT8     OpCode;
  UINT8     Length;
  UINT16    Flags;
  VOID      *Data;
} EFI_IFR_DATA_ENTRY;
```

OpCode

The type of opcode. The opcode type is likely string, numeric, or one-of.

Length

Length of the **EFI_IFR_DATA_ENTRY** packet.

Flags

Flags settings to determine what behavior is desired from the browser after the callback.

Data

The data in the form based on the opcode type. This parameter is not a pointer to the data; the data follows immediately.

If the *OpCode* is a one-of or numeric type, *Data* is a **UINT16** value.

If the *OpCode* is a string type, *Data* is a **CHAR16[x]** type.

If the *OpCode* is a checkbox type, *Data* is a **UINT8** value.

If the *OpCode* is an NV access type, *Data* is a **EFI_IFR_NV_DATA** structure.

EFI_FORM_CALLBACK_PROTOCOL.NvRead()

Summary

Returns the value of a variable.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_NV_READ) (
  IN        EFI_FORM_CALLBACK_PROTOCOL  *This,
  IN        CHAR16                      *VariableName,
  IN        EFI_GUID                    *VendorGuid,
  OUT       UINT32                      *Attributes OPTIONAL,
  IN OUT    UINTN                       *DataSize,
  OUT       VOID                        *Buffer
  );
```

Parameters

This

A pointer to the **EFI_FORM_CALLBACK_PROTOCOL** instance.

VariableName

A **NULL**-terminated Unicode string that is the name of the vendor's variable.

VendorGuid

A unique identifier for the vendor.

Attributes

If not **NULL**, a pointer to the memory location to return the attributes bit-mask for the variable. See "Related Definitions."

DataSize

The size in bytes of the *Buffer*. A size of zero causes the variable to be deleted.

Buffer

The buffer to return the contents of the variable.

Description

Each vendor may create and manage its own variables without the risk of name conflicts by using a unique *VendorGuid*.. When a variable is set, its *Attributes* are supplied to indicate how the data variable should be stored and maintained by the system. Any attempts to access a variable that does not have the attribute set for runtime access will yield the **EFI_NOT_FOUND** error.

Related Definitions

```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE          0x0000000000000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS    0x0000000000000002
#define EFI_VARIABLE_RUNTIME_ACCESS        0x0000000000000004
```

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The *DataSize* is too small for the result. *DataSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.

EFI_FORM_CALLBACK_PROTOCOL.NvWrite()

Summary

Sets the value of a variable.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_NV_WRITE) (
  IN EFI_FORM_CALLBACK_PROTOCOL  *This,
  IN CHAR16                      *VariableName,
  IN EFI_GUID                    *VendorGuid,
  IN UINT32                      *Attributes,
  IN     UINTN                   DataSize,
  IN     VOID                    *Buffer,
  OUT    BOOLEAN                 *ResetRequired
  );
```

Parameters

This

A pointer to the **EFI_FORM_CALLBACK_PROTOCOL** instance.

VariableName

A **NULL**-terminated Unicode string that is the name of the vendor's variable. Each *VariableName* is unique for each *VendorGuid*.

VendorGuid

A unique identifier for the vendor.

Attributes

Attributes bit-mask to set for the variable. See **EFI_FORM_CALLBACK_PROTOCOL.NvRead()**.

DataSize

The size in bytes of the *Buffer*. A size of zero causes the variable to be deleted.

Buffer

The buffer containing the contents of the variable.

ResetRequired

Returns a value from the driver that abstracts this information and will enable a system to know if a system reset is required to achieve the configuration changes being enabled through this function.

Description

Variables are stored by the firmware and may maintain their values across power cycles. Each vendor may create and manage its own variables without the risk of name conflicts by using a unique *VendorGuid*.

Status Codes Returned

EFI_SUCCESS	The firmware has successfully stored the variable and its data as defined by the *Attributes*.
EFI_INVALID_PARAMETER	An invalid combination of Attribute bits was supplied, or the *DataSize* exceeds the maximum allowed.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.

EFI_FORM_CALLBACK_PROTOCOL.CallBack()

Summary

The function that is called to provide results data to the driver. This data consists of a unique key which is used to identify what data is either being passed back or being asked for.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FORM_CALLBACK) (
  IN EFI_FORM_CALLBACK_PROTOCOL  *This,
  IN UINT16                      KeyValue,
  IN EFI_IFR_DATA_ARRAY          *Data,
  OUT EFI_HII_CALLBACK_PACKET    **Packet
);
```

Parameters

This

A pointer to the **EFI_NV_ACCESS_PROTOCOL** instance.

KeyValue

A unique value which is sent to the original exporting driver so that it can identify the type of data to expect. The format of the data tends to vary based on the opcode that generated the callback.

Data

A pointer to the data being sent to the original exporting driver. Type **EFI_IFR_DATA_ARRAY** is defined in **EFI_FORM_CALLBACK_PROTOCOL**.

Packet

A pointer to a packet of information which a driver passes back to the browser

Related Definitions

The *Data* format will be based on the opcode type that the *KeyValue* references. Table 3-1 lists the value passed in the *Data* pointer for each opcode type.

Table 3-1. Value Passed in the *Data* Pointer

If the opcode is…	The following is being passed in the *Data* pointer…		Comment
OneOf	**UINT16**	**Value**	
Text	**NULL**		There is no user initiated data to be sent other than the *KeyValue*. There should be a reasonable expectation that a response to this callback will be that a message gets posted with a particular key value and string. This posting would be done in the **EFI_FORM_BROWSER_PROTOCOL**.
String	**CHAR16**	***String**	
Numeric	**UINT16**	**Value**	

Status Codes Returned

EFI_SUCCESS	The firmware has successfully stored the variable and its data as defined by the *Attributes*.
EFI_INVALID_PARAMETER	An invalid combination of attribute bits was supplied, or the *DataSize* exceeds the maximum allowed.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.

Browser Interface

Form Browser Protocol

The **EFI_FORM_BROWSER_PROTOCOL** is the interface to call for drivers to leverage the EFI Configuration Driver interface.

EFI_FORM_BROWSER_PROTOCOL

Summary

The **EFI_FORM_BROWSER_PROTOCOL** is the interface to the EFI Configuration Driver. This will allow the caller to direct the configuration driver to use either the HII database or use the passed-in packet of data.

GUID

```
#define EFI_FORM_BROWSER_PROTOCOL_GUID  \
   { 0xe5a1333e, 0xe1b4, 0x4d55, 0xce, 0xeb, 0x35, 0xc3, \
   0xef, 0x13, 0x34, 0x43 }
```

Protocol Interface Structure

```
typedef struct _EFI_FORM_BROWSER_PROTOCOL {
  EFI_SEND_FORM                    SendForm;
  EFI_CREATE_POP_UP                CreatePopUp;
} EFI_FORM_BROWSER_PROTOCOL;
```

Parameters

SendForm

Provides direction to the configuration driver whether to use the HII database or to use a passed-in set of data. This functions also establishes a pointer to the calling driver's callback interface. See the **SendForm()** function description.

CreatePopUp

Routine used to abstract a generic dialog interface and return the selected key or string. See the **CreatePopUp()** function description.

Description

This protocol is the interface to call for drivers to leverage the EFI Configuration Driver interface.

Related Definitions

```
//***********************************************
// SCREEN_DESCRIPTOR
//***********************************************
typedef struct {
  UINTN          LeftColumn;
  UINTN          RightColumn;
  UINTN          TopRow;
  UINTN          BottomRow;
} SCREEN_DESCRIPTOR;
```

EFI_FORM_BROWSER_PROTOCOL.SendForm()

Summary

Provides direction to the configuration driver whether to use the HII database or a passed-in set of data. This function also establishes a pointer to the calling driver's callback interface.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SEND_FORM) (
  IN EFI_FORM_BROWSER_PROTOCOL        *This,
  IN BOOLEAN                          UseDatabase,
  IN EFI_HII_HANDLE                   *Handle,
  IN UINTN                            HandleCount,
  IN EFI_IFR_PACKET                   *Packet, OPTIONAL
  IN EFI_HANDLE                       CallbackHandle, OPTIONAL
  IN UINT8                            *NvMapOverride, OPTIONAL
  IN SCREEN_DESCRIPTOR                *ScreenDimensions, OPTIONAL
  OUT BOOLEAN                         *ResetRequired OPTIONAL
  );
```

Parameters

This

A pointer to the **EFI_FORM_BROWSER_PROTOCOL** instance.

UseDatabase

Determines whether the HII database is to be used to gather information. If the value is **FALSE**, the configuration driver will get the information provided in the passed-in *Packet* parameters.

Handle

A pointer to an array of HII handles to display. This value should correspond to the value of the HII form package that is required to be displayed.

HandleCount

The number of handles in the array specified by *Handle*.

Packet

A pointer to a set of data containing pointers to IFR and/or string data. This parameter is used only when the *UseDatabase* parameter is **FALSE** and an application is trying to pass information directly back and forth to the browser.

CallbackHandle

The handle to the driver's callback interface. This parameter is used only when the *UseDatabase* parameter is **FALSE** and an application wants to register a callback with the browser.

NvMapOverride

This buffer is used only when there is no NV variable to define the current settings and the caller needs to provide to the browser the current settings for the “fake” NV variable. If used, no saving of an NV variable will be possible. This parameter is also ignored if *Handle* is zero.

ScreenDimensions

Allows the browser to be called so that it occupies a portion of the physical screen instead of dynamically determining the screen dimensions.

ResetRequired

This **BOOLEAN** value will tell the caller if a reset is required based on the data that might have been changed. The *ResetRequired* parameter is primarily applicable for configuration applications, and is an optional parameter.

Related Definitions

```
//****************************************************
// EFI_IFR_PACKET
//****************************************************
typedef struct {
  EFI_HII_IFR_PACK          *IFRData;
  EFI_HII_STRING_PACK       *StringData;
} EFI_IFR_PACKET;
```

Status Codes Returned

EFI_SUCCESS	The function completed successfully
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The *DataSize* is too small for the result. *DataSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.

EFI_FORM_BROWSER_PROTOCOL.CreatePopUp()

Summary

Routine used to abstract a generic dialog interface and return the selected key or string.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CREATE_POP_UP) (
  IN   UINTN                          NumberOfLines,
  IN   BOOLEAN                        HotKey,
  IN   UINTN                          MaximumStringSize,
  OUT  CHAR16                         *StringBuffer,
  OUT  EFI_INPUT_KEY                  KeyValue,
  IN   CHAR16                         *String,
  ...
  );
```

Parameters

NumberOfLines

The number of lines for the dialog box.

HotKey

Defines whether a single character is parsed (**TRUE**) and returned in *KeyValue* or if a string is returned in *StringBuffer*. Two special characters are considered when entering a string—a **SCAN_ESC** and a **CHAR_CARRIAGE_RETURN**. **SCAN_ESC** terminates string input and returns while **CHAR_CARRIAGE_RETURN** commits the entered string.

MaximumStringSize

The maximum size in bytes of a typed-in string. Because each character is a **CHAR16**, the minimum string returned is two bytes.

StringBuffer

The passed-in pointer to the buffer that will hold the typed in string if *HotKey* is **FALSE**.

KeyValue

The **EFI_KEY** value returned if *HotKey* is **TRUE**.

String

The pointer to the first string in the list of strings that comprise the dialog box.

...

A series of *NumberOfLines* text strings that will be used to construct the dialog box.

Description

This function is intended for use by applications that might have a need for the creation of a simple dialog box but may not need to complete services of a form-based browser and all the inputs that are required for the form-based browser such as IFR and localization.

Status Codes Returned

EFI_SUCCESS	Displayed dialog and received user interaction
EFI_DEVICE_ERROR	User typed in an ESC character to exit the routine.
EFI_INVALID_PARAMETER	One of the parameters was invalid (e.g., **(*StringBuffer* == NULL) && (*HotKey* == FALSE))**.

Appendix A
Conventions for IFR to HTML Translation

Table A-2 defines suggested translations between IFR and HTML.

Table A-2. Suggested Translations between IFR and HTML

IFR	HTML
String in *form* operand	Both <title> and <h1>
Subtitle	<h3>
Text	Standard text
One-of	Either radio button or drop down
Checkbox	Single selection check box
Numeric	Text input sized to fit the maximum number of digits in the number along with JavaScript or equivalent validation
Password	No recommendation
Go-to	<a href…>