Boot Setting File (BSF) Specification

March 2016

Revision 1.0

Revision History

Revision	Revision History	Date
1.0	Public release. Align w/ FSP	March 2016

Contents

Figures

Tables

§

1 Introduction

1.1 Overview

This document describes the format of the Boot Setting File (BSF) used to specify features, settings and tool display information for the Intel® BCT Firmware Support Package (FSP) and tools like the Binary Configuration Tool (BCT) BCT.

While the BCT tool may read BSF files created for previous versions of tools, this BSF file format is not compatible with previous versions of tools.

This document also describes the 'As Built' BSF file, which contains the feature selections and binary setting values that are selected by the developer. This 'As Built' file is an input to patch utilities that can modify a binary firmware image file.

This document describes the structure and content of BSF format files, which has the following goal:

1.1.1 Simplified Firmware Configuration

The primary goal of this file format is to simplify the configuration of features and settings for a given firmware image.

1.2 Target Audience

This document is intended for persons creating firmware images. It is most likely only of interest in the event that a developer needs to create customized features and expose binary settings for delivery to customers, or that a new SOC device is being enabled.

1.3 Related Information

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

Intel Firmware Support Package (FSP) www.intel.com/fsp

Intel® Binary Configuration Tool (BCT)
http://www.intel.com/content/www/us/en/embedded/software/fsp/binary-configuration-tool-release-notes.html

1.4 Terms

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

AS_BUILT

The 'AS_BUILT' is a 'BSF format' file that contains configuration information about an image. This configuration information is used to modify a binary image to create a final firmware image. The term is also used as a label within the file itself to retain the configuration knobs selected by the user.

Boolean

A 1 byte value containing a 0 to represent FALSE/DISABLED or a 1 to represent TRUE/ENABLED. Other values are undefined.

BIN

A value that must be expressed as binary data ([0-1],) base 2 with a '0b' prefix character, as in 0b10010001.

BSF

The Boot Setting File is a text file that is consumed by the BCT tool that permits the user to select features and/or modify configuration settings in a binary image.

BNF

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

DEC

A value that must be expressed as a non-negative integer ([0-9],) base 10.

EBIN

A value (end-bin) that must be expressed as binary data ([0-1],) base 2. with a 'b' suffix character, as in 10010001b.

EBNF

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

EHEX

A value (end-hex) that must be expressed as hexadecimal data ([a-fA-F0- 9],) base 16, suffixed by the 'h' character.

Feature Selection Knobs

Configuration elements that are used by tools to include code during the build process. Features are either enabled or disabled. These feature elements are used to control what gets built. Some features which are included in the binary image created by the build may also be disabled in the binary, however disabling of these features does not shrink the final binary image size. Typical feature names might include USB, Memory, etc.

HEX

A value that must be expressed as hexadecimal data ([a-fA-F0-9],) base 16, preceded by the '0x' character string.

Localization

Customization of graphic applications for a particular language or region.

PcdName

A PcdName is a formatted entry to specify EDK II Platform Configuration Data variables. The PcdName consists of the C variable name of the Token Space GUID (a name space for a set of PCDs) followed by a dot '.' character, then the C variable name of the PCD that represents a token number that is unique to the token space.

Binary Configuration Setting Knobs

Binary configuration elements are associated with values (although enabling and disabling of a setting is also permitted.) Binary configuration data is contained within a binary file created by a build. The user is permitted to modify any of the exposed values which will be modified by a patch tool what will replace the value within the binary file.

1.5 Conventions Used in this Document

This document uses typographic and illustrative conventions described below.

1.5.1 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 multi-byte 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:

An EBNF-type declaration for the data structure.

Example:

Sample data structure using the 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.

1.5.2 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 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 Specification*.

1.5.3 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)

Any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyper link.

Bold

In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.

Italic

In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.

BOLD Monospace

Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.

`Bold Monospace`

Words in a **`Bold Monospace`** typeface that is underlined and in blue indicate an active hyper link to the code definition for that function or type definition. Click on the word to follow the hyper link.

$(VAR)

This symbol VAR defined by the utility or input files.

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

`Italic Monospace`

In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).

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

[item]	Square brackets denote the enclosed item is optional.
{item}	Curly braces denote a choice or selection item, only one of which may occur on a given line.
<item>	Angle brackets denote a name for an item.
(range-range)	Parenthesis with characters and dash characters denote ranges of values, for example, (a-zA-Z0-9) indicates a single alphanumeric character, while (0-9) indicates a single digit.
“item”	Characters within quotation marks are the exact content of an item, as they must appear in the output text file.
?	The question mark denotes zero or one occurrences of an item.
*	The star character denotes zero or more occurrences of an item.
+	The plus character denotes one or more occurrences of an item.
item$^{\{n\}}$	A superscript number, n, is the number occurrences of the item that must be used. Example: (0-9)^{8} indicates that there must be exactly eight digits, so 01234567 is valid, while 1234567 is not valid.
item$_{\{n,\}}$	A superscript number, n, within curly braces followed by a comma “,” indicates the minimum number of occurrences of the item, with no maximum number of occurrences.
item$_{\{,n\}}$	A superscript number, n, within curly braces, preceded by a comma “,”indicates a maximum number of occurrences of the item.
item$_{\{n,m\}}$	A super script number, n, followed by a comma “,“and a number, m, indicates that the number of occurrences can be from n to m occurrences of the item, inclusive.

2 Design Discussion

This section of the document provides an overview of the Boot Setting File (BSF) usage and format. The BSF file is used by tools to display user modifiable feature selections and binary configuration settings. Tools provide user interface controls that allow the user to select features for build and edit the binary data in a user-friendly manner. Therefore, writing the BSF file correctly is essential to proper use of the utility.

The BSF file is a text file that has a .BSF extension. Each BSF file contains an optional global data definition section, an optional feature definition section, an optional BSF information block, an optional structure definition section, an optional list definition section and a required page definition section in that order. If any errors in the syntax are found when the file is loaded, the file will stop loading and the error message with the line number of the error will be printed to the help view.

Note: *If a BSF file does not contain a structure definition section, the BSF file can only be used to add or remove features and configuration of binary settings will not be possible.*

Directives can be used throughout the file to modify the execution path based on specified conditions. These statements can even be used to vary entire sections, such as a List section. However, they cannot be used to vary the global data definition section. Comments can also be used throughout the file to place useful text statements that will be ignored by the processing tools.

Earlier versions of tools, such as the BMP tool, may not be able to process the new format if new features, such as the global data and feature sections, or using profiles to predefine values based on profile (global data) selections.

In addition to providing selection information, an 'As Built' version of the file will also be used to store selections of features and setting values that can be processed by patch utilities to update a binary firmware image file.

Note: *The original BMP tool required that the BSF file be a DOS format file, new tools must remove this restriction, and be able to process files created under a Unix style environment (Linux [LF] or Mac OS/X [CR]) as well as the DOS format [CRLF] file.*

2.1 Usage

The BSF file is used by the Intel® BCT tool to help users select features that will be built into a binary file and modify binary configuration settings within an existing firmware image.

The main components of the BSF file are:

A global data definition section that is used to define filters used to limit displayed binary configuration settings, providing relationship data for handling of multiple one-of selections, profiles that can be used for pre- defining a set of features and values for binary configuration settings, as well as different hardware configuration (SKUs) settings. Additionally, the profile identifiers may be used in directive statements that control what information is available for feature selection and/or binary configuration settings, while filters are used to mask a user's view of modifiable binary configuration settings.

A features definition section that is used to specify available features. These feature values are used as part of the build. They may also be used as an argument to a directive or as a variable name element within a data structure.

An inconsistency definition section that is used to specify the conditions which would create error conditions. This would be used in the case where separate configuration settings have relationships which might affect what is considered a valid set of values. The most common example for such a thing would be where one might establish a rule for determining leap years, or to ensure that two serial port addresses cannot be the same.

A data structure within the binary image that contains both configurable and non-configurable data. Non-configurable data is located in the data structure using a keyword, 'skip' rather than reserved (which usually denotes values of zero.) The data structures always start with an ascii signature such as "Begin", to identify the location within a firmware binary of the data structure. Multiple data structures are permitted within a firmware image. Special labels, '$_DEFAULT_' and '$_AS_BUILT_' are used with data structure elements to retain information across multiple sessions or between process steps.

A list definition section that will be used to list valid value selections for a binary configuration setting.

An information block section that identifies the image.

A page definition display that provides information about each of the pages that can be displayed. For each item that will be displayed for a given page, there is an association of data structure element and prompt, and optionally, the name of the list and a help text string for the setting.

Conditional directive statements are used to limit the scope of the BSF file.

The file is created by the firmware developer of the binary, and is used to expose features and settings that are configurable by a user.

Two reserved Labels, $_DEFAULT_ and $_AS_BUILT_, can be used throughout the BSF document's FeatureDef and StructDef sections to store the initial ($_DEFAULT_) values, as well as the values selected ($_AS_BUILT_) by a user. This also permits the user to modify the 'As Built' version of the file using a text editor, rather than tools.

2.2 Development Environments

The BCT development environment is primarily focused on Linux development workstations. However, in the future, the build environment must support development workstations running Microsoft* Windows operating systems and/ or the Linux operating systems. In addition, multiple compiler tools chains from Microsoft, Intel and GCC must be supported. All provided source code should be Posix compliant. If assembly source code is used, both GCC (AT&T) and Microsoft (Intel) style assembly files should be provided.

2.3 Firmware Images

The primary function of the firmware image is to initialize the hardware and boot the operating system.

Firmware images are created from C and Assembly code files that are built using standard makefile formats. While most of the images delivered to customers are binary, some C code is provided to permit minor customization of the final binary image. Typical binary images for SOC devices are less than 256Kb.

In order to provide configuration of the SOC device, data structures with known signatures within the binary image are created that contains the SOC device's binary configuration settings.

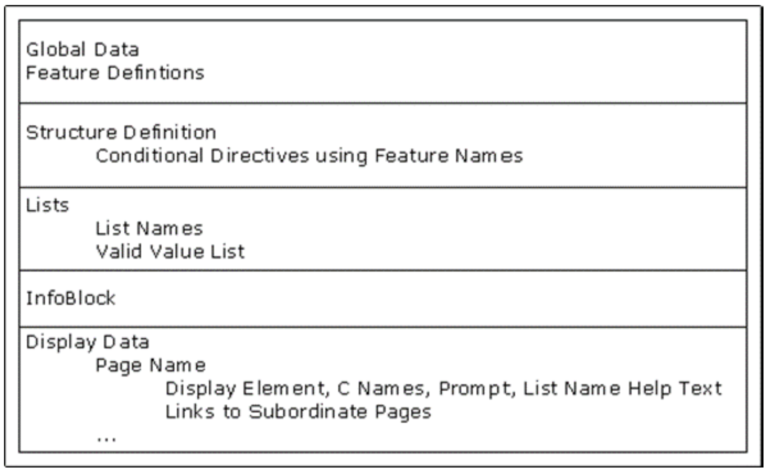
This specification covers the format of the text file that is used by tools to document the data structure, possible values, selection lists as well as the format of user visible information (prompt and help text) used by tools during the feature selection and the binary configuration processes.

2.4 Typical Boot Setting File Layout

The BSF layout consists of comment lines, the data structure, which always starts with a signature that can be recognized by tools, followed by information used by tools to provide configuration information to the user, including selection values and information display-able to the tool user.

The major parts of the file are shown in Figure 1.

Figure 1. High Level BSF Format Block Diagram


Global Data
Feature Defintions
Structure Definition
Conditional Directives using Feature Names
Lists
List Names
Valid Value List
InfoBlock
Display Data
Page Name
Display Element, C Names, Prompt, List Name Help Text
Links to Subordinate Pages
...

Additional information may be provided to assist the tools in processing the firmware image. The following paragraphs describe, at a high level, the content of the BSF file. The next chapter describes the valid syntax for a BSF file.

Note: *In the examples that follow, not all required elements are present. For example, with a structure definition, a Find or Find_Ptr_Ref element is required before any other variables can be listed, in the examples, this element may not be present.*

2.4.1 Conditional Directives

Conditional directive statements may be used within the BSF file to provide conditional control of content within the file. The BSF file supports only the following conditional directive statements: #if,

#elif, #elseif, #else and #endif. Directive statement expressions are either arithmetic or logical comparisons that must evaluate to either True (1) or False (0). Directive argument names MUST be defined prior to use in a directive expression. String comparison expressions are not permitted within the BSF file. The #define, #include, #ifdef and #ifndef statements are not supported directives.

Similar to the ANSI C conditional directive usage, any directives within the BSF file must not break the validity of the BSF format. Also, in keeping with the ANSI C specification, a directive's expression may span multiple lines, by using the back slash character to extend processing of the line.

Examples of bad usage include:

Using a #if statement without a #endif statement.

Using a variable name in a directive statement comparison that encapsulates the variable.

Using a StructDef variable name in a comparison before it has been defined in previous statements.

The comparisons (both arithmetic and logical) may use variable names and constants, may be combined (use of parenthesis is encouraged to remove ambiguity,) and are in-fix processed.

Directive statements are permitted within a FeatureDef section, but are not permitted to encapsulate a FeatureDef section (only one FeatureDef section is permitted.) Directive comparison variables within the FeatureDef section are limited to variables defined in the GlobalDataDef section or FeatureDef statement variables previously defined.

Directive statements may be used within a StructDef section and are permitted to encapsulate the StructDef section provided another StructDef section can be used (only one valid StructDef section is permitted.) Directive comparison variables within the StructDef section are limited to variables defined in the GlobalDataDef, FeatureDef and variables within the StructDef section.

Directive statements may encapsulate List sections and within a List definition section only if used to determine a value of a selection statement (the directive section must contain an #else directive.) Directive comparison variables for List sections are limited to variables defined in the GlobalDataDef, FeatureDef and variables within the StructDef section.

Directive statement may be used around PageDef sections, as well as within a PageDef section. Directive comparison variables within and around PageDef sections are limited to variables defined in the GlobalDataDef, FeatureDef and variables within the StructDef section.

Directive statements are not permitted to encapsulate either a GlobalDataDef section or an InfoBlock section.

2.4.2 Default Values

Default values are defined using a special label, $_DEFAULT_, and, if present, are the recommended default values suggested by the platform provider. The default label is valid in FeatureDef and StructDef sections only. This label is not required.

2.4.3 Global Data

This area is used to define filters, one-of selection relationships, profiles and SKUs.

Filters are used to mask some binary configuration settings. This permits specifying different classes of settings, such as Expert, Safe, etc. The tools used to query the user for setting values can use these filters to limit what content can be viewed into different categories. The value assigned to the filter is used as a bitwise mask. Tools should only permit zero or one filters to be selected.

One-of selection relationships are used to associated multiple items as a one-of set of choices. Common scenarios where this would show up is where a user would be presented with several possible choices and only one of those choices is allowed to be active at a time. This also provides a hint to a BSF parser that such a relationship could also be displayed as a radio selection.

Profiles are used to define pre-set values, and may be needed by different groups, such as SV, CV and Manufacturing. Each of these groups may pre- defined binary configuration setting values. This area defines profiles that can be used in other sections of the BSF file as labels to define values for each of these profiles. Profile values can be applied regardless of any filter selected. Tools should permit zero or one profiles to be selected.

Different hardware based on a single platform are referenced by SKUs. These SKUs are treated as labels. One and only one SKU may be selected by the user; creating a single firmware image that supports more than one SKU is not permitted.

Global labels must be defined before they can be used.

The following is an example of the syntax for the BSF Global Data section:

```
CategoryID = variable , HEX , UserInterface
ViewID = variable , HEX , UserInterface UserView = %ADVANCED
DefaultID = variable , UserInterfaceName SKUID = value , UserInterfaceName
```

Example:

```
GlobalDataDef
    ViewID            = %ADVANCED , 0xFFFFFFFF , ''Advanced View''
    ViewID            = %INTERMEDIATE , 0x00000001 , ''Inter. View''
    ViewID            = %SAFE , 0x00000002 , ''Safe View''
    CategoryID        = %SATA , 0x00000001 , ''SATA Related''
    CategoryID        = %USB , 0x00000002 , ''USB Related''
    CategoryID        = %NAND , 0x00000004 , ''NAND Related''
    CategoryID        = %STORAGE , 0x00000007 , ''Store Related''
    DefaultID = $MANUF , ''Manufacturing Defaults''
    DefaultID = $USER1 , ''Mike's Preferred Defaults''
    SKUID = 0x00, ''Menlow''
    SKUID = 0x01, ''Crown Beach''
EndGlobalData
```

In the above example, several View IDs are defined. A view is the ability to define a means of associating a configuration setting to a user access level. With this concept, there is a presumption that a tool would comprehend what user access level a given user might be. For a particular configuration setting, a view setting can be associated with it.   It is also legal to have an

_AS_BUILT_ label associated with a ViewID so that this state information can be carried by the BSF file itself. That being the case, an algorithm would be used by the BSF parsing tool to determine if a configuration setting is visible to a user. It should also be noted it is possible to "lock" the user's view by using the UserView keyword such that the user's view level is defined by the BSF file and not by the tool querying the user as a tool setting attribute.

To properly interpret the View data, the algorithm would be:

If the User's level is bit-wise AND'ed with the configuration setting's ViewID and the results equals the configuration setting's VIEWID, then the setting is visible.

```
if ((USER_LVL & CFG_SETTING) == CFG_SETTING) {Display Question}
```

In other words, if a user level was determined to be equivalent to an %ADVANCED user, and a configuration setting was labeled with %SAFE you would evaluate the algorithm as such:

```
if ((%ADVANCED & %SAFE) == %SAFE) { Display Question }
```

The same can be said of the Category IDs that are defined above. Configuration settings can have categories associated with them. With that in mind, making such an association facilitates the ability to view all of a certain "type" of question. For instance, a user might want to view all of the $USB related questions. Since these IDs are a mask, the concept of having a category which encompasses several types of devices makes sense. In the above example, $SATA, $USB, and $NAND are three independent types of categories, and a fourth category is shown which would encompass both of the former types. This is practical in the case where the user might want to view all questions associated with storage. So in this case, the

$STORAGE category might be an excellent choice of a mask for such a purpose. That being the case, an algorithm would be used by the BSF parsing tool to determine if a configuration setting is visible to a user. The algorithm would be:

If the current Category ID setting selected by the tool is bit-wise AND'ed with the configuration setting's Category ID and the results equals the configuration setting's Category ID, then the setting is visible.

```
if ((TOOL_VIEW & CFG_VIEW) == CFG_VIEW) { Display Question }
```

In other words, if a category view for the tool was determined to be equivalent to the %USB view, and a configuration setting was labeled with %NAND you would evaluate the algorithm as such:

```
if ((%USB & %NAND) == %NAND) { Display Question }
```

```
StructDef
    Find ''Begin''
    #if SKUID == 0x00
      $Var1 1 byte $_DEFAULT_ = 0x08 $MANUF= 0x08 $USER1= 0x05
    #elif SKUID == 0x01
      $Var1 1 byte $_DEFAULT_ = 0x08 $MANUF= 0x08 $USER1= 0x03
      $Var2 1 byte $_DEFAULT_ = 0x0F $MANUF= 0x0F
    #else
      $Var1 1 byte $_DEFAULT_ = 0x00
    #endif
      $Var3 1 byte %ADVANCED $_DEFAULT_ = 0x02 $MANUF = 0x03
      $Var4 1 byte %INTERMEDIATE $SAFE $_DEFAULT_ = 0xFF
      $Var5 1 byte %INTERMEDIATE $_DEFAULT_ = 0x11
      $Var6 1 byte $_DEFAULT_ = 0xFF
      $Var7 1 byte $_DEFAULT_ = 0xFE
      $Var8 1 byte $_DEFAULT_ = 0xFD
EndStruct
```

In this example, two different hardware platforms, have been defined. If the user selects the "Menlow" SKU, and selects the profile, "Manufacturing Defaults", the 'used' structure is:

```
StructDef
    Find ''Begin''
    $Var1 1 byte $MANUF= 0x08 $USER1= 0x05
EndStruct
```

And the value for $Var1 within the binary will be set to 0x08 in the 'As Built' BSF file.

To describe the functionality of the View ID masking, page elements for $Var1,

$Var2 and $Var8 are eligible for modification at all times, while $Var3 is only eligible for modification if the View ID is set to %ADVANCED. $Var4 is only eligible for modification if the View ID is set to %ADVANCED, %INTERMEDIATE or %SAFE, while $Var5 is only eligible for modification if the View ID is set to %ADVANCED or %INTERMEDIATE.

The following example shows an 'As Built' version of BSF file example above.

```
GlobalDataDef
    ViewID    = %ADVANCED , 0xFFFFFFFF , ''Advanced View''
    ViewID    = %INTERMEDIATE , 0x00000001 , ''Inter.  View''
    ViewID    = %SAFE , 0x00000002 , ''Safe View''
    DefaultID = $MANUF , ''Manufacturing Defaults''
    DefaultID = $USER1 , ''Mike's Preferred Defaults''
    SKUID = 0x00, ''Menlow''
    SKUID = 0x01 $_AS_BUILT_ = 1, ''Crown Beach''
EndGlobalData

StructDef
  Find ''Begin''
    $Var1 1 byte $_DEFAULT_ = 0x08 $_AS_BUILT_ = 0x08 $MANUF= 0x08
    $Var2 1 byte $_DEFAULT_ = 0x0F $_AS_BUILT_ = 0x0F $MANUF= 0x0F
    $Var3 1 byte %ADVANCED $_DEFAULT_ = 0x02 $_AS_BUILT_ = 0x03 \
                 $MANUF=0x03
    $Var4 1 byte %INTERMEDIATE $SAFE $_DEFAULT_ = 0xFF
    $Var5 1 byte %INTERMEDIATE $_DEFAULT_ = 0x11
    $Var6 1 byte $_DEFAULT_ = 0xFF
    $Var7 1 byte $_DEFAULT_ = 0xFE
    $Var8 1 byte $_DEFAULT_ = 0xFD
EndStruct
```

In this example, only one of the hardware platforms has been defined. The user selected the "Crown Beach" SKU, and selected the profile, "Manufacturing Defaults". (The back slash at the end of the line is shown here to indicate the line was continued. In the real file, the content will most likely appear on a single line.)

2.4.4 Feature List

Individual features may or may not be included in a build. In some cases, features may be included in the build, and may also be disabled in the binary. Features must be defined before they can be used. This section defines Feature tags and prompt information to permit a user to select or deselect any available features. For every Feature tag being defined, they are to be evaluated as a BOOLEAN type, meaning that this feature is either evaluated as Enabled (1) or Disabled (2). The section must also include a default value, as a code-base may be delivered in which it is desirable to have a feature not be included by default.

View and Category-based Filters defined in the GlobalDataDef section may also be applied to individual feature elements.

The following is the syntax for the BSF Feature section:

```
  FeatureName [, [Filter] Default] , prompt [ , Help ]
```

Example:

```
FeatureDef
  $USB_FEATURE, $_DEFAULT_ = 1, ‘‘Enable USB?’’,
     ‘‘Enables input and output USB devices.’’
     ‘‘Feature is always in Flash’’
  $TOUCH_SCREEN_FEATURE, $_DEFAULT_ = 0, ‘‘Enable Touch Screen Input?’’,
      ‘‘Feature is not included in the flash part if it is disabled’’
EndFeature

StructDef
  Find ‘‘Begin’’
  $Var1               1 byte
  $Var2               4 bits
  $USB_FEATURE        1 bit
  ALIGN
  #if $USB_FEATURE
    $Var3             20 bytes
  #else
    Skip              20 bytes
  #endif
  #if $TOUCH_SCREEN_FEATURE
    Find ‘‘TSFBegin’’
    $Var4             1 bit
    $Var5             1 bit
    ALIGN
    $Var6             1 byte
  #endif
  Find ‘‘Next’’
  $Var7               1 byte
  $Var8 32 bits
EndStruct
```

In this example, two different features may be selected. By default, the $TOUCH_SCREEN_FEATURE would be disabled, and the end user would be required to enable the feature prior to building. If the features are selected prior to linking in different object files (one or more files per feature) then if the USB feature is enabled and the Touch Screen feature is disabled, the structure after the build will look like:

```
FeatureDef
  $USB_FEATURE $_AS_BUILT_ = 1 $_DEFAULT_ = 1, ‘‘Enable USB?’’,
     ‘‘Enables input and output USB devices.’’
     ‘‘Feature is always in Flash’’
  $TOUCH_SCREEN_FEATURE $_AS_BUILT_ = 0 $_DEFAULT_ = 0,
     ‘‘Enable Touch Screen Input?’’,
```

Note: *“Feature is not included in the flash part if it is disabled”*

```
EndFeature

StructDef
    Find ''Begin''
    $Var1               1 byte $_AS_BUILT_ = 0x0F
    $Var2               4 bits $_AS_BUILT_ = 0b1001
    $USB_FEATURE        1 bit   $_AS_BUILT_ = 0b1
    ALIGN
      $Var3             20 bytes $_AS_BUILT_ = 0x00, 0x01, 0x02, 0x03 \
                             0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A \
                             0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11 \
                              0x12, 0x00

    Find ''Next''
    $Var7               1 byte $_AS_BUILT_ = 0x10
    $Var8               32 bits $_AS_BUILT_ = 0x0000C000
EndStruct
```

In this example, the USB feature was enabled during the build stage, but may be disabled in the final firmware image.

2.4.5 Structure

This section shows the layout, within the firmware, of binary configurable settings. Multiple configuration sections are permitted within the definition. Additionally, the specification permits a profile specified in the global data section to be used for setting all values in this area. By selecting a profile, the tools will automatically use these settings rather than query the user for values.

Filters specified in the GlobalDataDef section may also be listed. Filters are used to control display-able binary configuration settings. Some settings may require a priori knowledge of the values, and are therefore not something a general user might need to modify. For example, a setting that controls a clock may have several possible settings, but for normal usage, a single value may be all that is required. Making this setting viewable for modification by expert users may be desirable, but not required for production operators on the factory floor.

Note: *The string values specified for Find and Find_Ptr_Ref elements must be unique and must not be a subset of another name, such as 'Begin' and 'MyBeginStr', as tools will search the binary image for a match and 'Begin' is a subset of 'MyBeginStr'.*

2.4.5.1 Find

The Find statement requires that tools search the binary data file for a specified string and sets the current location to the byte following the string. If the string is not found, the tools should stop loading the file and an error message should be reported. The structure definition section can support multiple find statements which allows a file to support binary files with multiple data segments. The syntax for the find command is as follows:

```
  Find signature [, Offset]
```

Example:

```
StructDef
  Find ‘‘Begin’’
  $Var1 1 byte
EndStruct
```

In this example, tools will search the data file for “Begin” and set the current location in the file to the byte after begin.

The following is an example of the structure in an ‘As Built’ BSF file.

```
StructDef
  Find ‘‘Begin’’
  $Var1 1 byte $_AS_BUILT_ = 0xFF
EndStruct
```

2.4.5.2 Find_Ptr_Ref

The Find_Ptr_Ref statement requires that tools searches the data file for a specified string or number value and returns the location at the beginning of that item. If the string is found, all pointer variables defined after it will add that location to their data pointer. If the string is not found, the location will be set to the beginning of the file. This command is very similar to the Find command, except it only affects pointers, whereas find affects all variables. The syntax for the Find_Ptr_Ref element is as follows:

To search for a string:

```
Find_Ptr_Ref string
```

To search for a number:

```
Find_Ptr_Ref number [ , number ]
```

Note: *Leading zero's are ignored.*

Example:

```
StructDef
  Find_Ptr_Ref ‘‘Begin’’
  $DataPtr1 2 bytes
  $Table1 , $DataPtr1 , 10 bytes
EndStruct
```

In this example, BMP will search the file for “Begin” and add that location to all pointers defined after the command.

```
StructDef
  Find_Ptr_Ref 0x31 , AAh
  $DataPtr1 2 bytes
  $Table1 , $DataPtr1 , 10 bytes
EndStruct
```

In this example, BMP will search the file for “31 AA” and add that location to all pointers defined after the command.

Find_Ptr_Ref using the number format is not valid for EDK II VPD PCDs.

2.4.5.3 Basic Variables

The BSF language uses variables to represent locations in the data file. All variables need to be defined in the structure definition section. The basic variable definition follows the following format:

```
  $string number size [ FilterID ]* [ Label = value ]
```

Example:

```
  StructDef
    Find ‘‘PCIVERSION’’
      $Var1 10 bytes %SAFE %INTERMEDIATE $_DEAULT_ = 0x00020000
  $USER1= 0x0080000000
  EndStruct
```

In this example, we are defining a 10 byte variable called '$Var1', the variable has a mask, %SAFE logically or'd with %INTERMEDIATE applied (this is a bit mask,) and the profile, ‘$USER1’ has a predefined value. If the profile is selected, then the value specified by the label, “$USER1” is required to be placed in the firmware image - regardless of any bit mask filter.

If the user selects a filter (defined in the GlobalDataDef section) that includes the bit defined as %SAFE or the bit defined as %INTERMEDIATE, then the user will be able to see question fields (from a PageDef) that permit modifying this value.

For EDK II the string may consist of the dollar ‘$’ sign, PCD Token Space GUID C Name, a dot separator and the PCD Token C Name.

The label, $_DEFAULT_ is a label that specifies the platform provider’s recommended value for the variable. Tools should never modify these labels.

The label, $_AS_BUILT_ is reserved for use by tools, and represents the value selected by the user, even if the user does not change the value, the entry must exist for the existing value. This label is added by tools (or added by a user) prior to patching a binary firmware image file.

The following is an example of the structure in an ‘As Built’ BSF file.

```
StructDef
    Find ‘‘PCIVERSION’’
    $MyVariable 10 bytes %SAFE %INTERMEDIATE \
      $USER1 = 0x0080000000 \
      $_AS_BUILT_ = 0x50, 0x43, 0x49, 0x52, 0x20, \
          0x76, 0x31, 0x2E, 0x30, 0x0D \
  EndStruct
```

In the above example, the profile, $USER1, was selected, however the user also chose to override the $USER1 value with a different value.

Note: *All variables are case sensitive*

2.4.5.4 Pointer Variables

The BSF language uses pointers to represent a pointer to the location of the variable's data in the binary data file. All pointer variables need to be defined in the structure definition section. Pointer variables follow the basic format of:

```
 Variable Name , Data Pointer Name , Size , [Offset] [FilterID]*
```

The 'Variable Name' is the unique name representing the pointer variable. The 'Data Pointer Name' is the variable storing the location of the pointer's address. This variable must be previously defined in the file or an error will occur. The 'Size' is the size of the data and consists of a number and a size word qualifier. The optional 'Offset' is the address offset for the pointer data. The size of the pointer can be either a static size, which means the size is defined in the BSF file, or a dynamic size, which means the size is represented by another variable.

The format for a static size pointer is:

```
  $PtrString , $DataString , number size , offset number size \ [ FilterID ]*
```

The format for a dynamic size pointer is:

```
  $PtrString , $DataString , $SizeString , offset number size \ [ FilterID ]*
```

Example:

```
  StructDef
    $PtrData 4 bytes
    $MyPointer, $PtrData, 10 bytes, Offset 2 bytes
  EndStruct
```

In this example, we define a 10 byte pointer variable named 'MyPointer' whose address is stored in the variable 'PtrData'. The data location is then defined as the address contained in 'PtrData' + 2 bytes (from the optional Offset parameter). If 'PtrData' is not defined before 'MyPointer', an error should be generated similar to the following message:

```
''Error: Missing Symbol $PtrData; Unable to find symbol.''
```

The following is an example of the structure in an 'As Built' BSF file.

```
StructDef
   $PtrData 4 bytes $_AS_BUILT_ = 00h, 00h, C0h, 00h
   $MyPointer, $PtrData, 10 bytes, Offset 2 bytes $_AS_BUILT_ = \
        0x50, 0x43, 0x49, 0x52, 0x20, \
            0x76, 0x31, 0x2E, 0x30, 0x0D \
  EndStruct
```

Example:

```
  StructDef
    $PtrData 4 bytes
    $PtrSize 3 bytes
    $MyPointer, $PtrData, $PtrSize
  EndStruct
```

In this example, we define a pointer variable named 'MyPointer' whose address is stored in the variable 'PtrData' and whose size is stored in the variable 'PtrSize'. If 'PtrData' or 'PtrSize' are not defined before 'MyPointer', an error will be generated.

The following is an example of the structure in an 'As Built' BSF file.

```
StructDef
  $PtrData 4 bytes $_AS_BUILT_ = 00h, 00h, C0h, 00h
  $PtrSize 1 byte $_AS_BUILT_ = 0x04
  $MyPointer, $PtrData, $PtrSize $_AS_BUILT_ = 0x02, 0x0F, \
       0xFF, 0x0E
EndStruct
```

2.4.5.5 Skip

The ‘SKIP’ element is used to prevent modification of binary settings within the configuration data area. The syntax for the SKIP command is as follows:

```
  SKIP number size
```

Example:

```
  StructDef
    $Var1 1 byte
    SKIP 3 bytes
    $Var2 1 byte
  EndStruct
```

In this example, Var1 is located at byte 0. After the skip, Var2 would be located at byte 4 and the data in the 1st, 2nd and 3rd bytes should not be modified.

The following is an example of the structure in an ‘As Built’ BSF file.

```
  StructDef
    $Var1 1 byte $_AS_BUILT_ = 0Fh SKIP      3 bytes
    $Var2 1 byte $_AS_BUILT_ = 0x02
  EndStruct
```

2.4.5.6 Align

To help facilitate dealing with bit structures, the BSF language provides the keyword ALIGN. Align forces the next variable to begin at the next BYTE aligned address.

To skip to the next byte:

```
  ALIGN
```

To skip to the next N bytes for an element that must be aligned on a machine’s natural boundary, where ‘number’ is the natural address boundary (it is assumed that the start of the section, i.e., the signature, is naturally aligned) the specification permits the following:

```
  ALIGN number
```

Example:

```
  StructDef
    $Var1 10 bits
    ALIGN
    $Var2 2 bytes
  EndStruct
```

In this example 'Var1' would occupy the first 10 bits of the data file. Since the ALIGN command is used before the definition of 'Var2', 'Var2' begins at bit 16 (third byte), instead of the 11th bit.

The following is an example of the structure in an ‘As Built’ BSF file.

```
StructDef
  $Var1 10 bits $_AS_BUILT_ = 0b1000110011
 ALIGN
  $Var2 2 bytes $_AS_BUILT_ = 0x01, 0x02
EndStruct
```

2.4.6 Lists

This section defines different sets of valid values that can be used for configurable binary settings. The list definition section is used to define lists for “drop-down combo box” controls. This section is only required if there are combo box elements defined in the page definition section. Unlike the structure definition section, the entire section is not bounded by keywords, but rather each list is individually bounded. Therefore, there can be one or more list definitions in the list definition section. Each list is identified by a unique list name which must be prefixed with the ampersand ‘&’ character. A list may be used more than one time, for multiple combo box elements. The format for entries within the list sections is:

```
‘‘Selection’’ value , UserInterfaceIdentifier [ FilterID ]* \
[ , Help string]
```

Example:

```
List &BooleanList
 Selection 0x1 , True
  Selection 0x0 , False
EndList

List &ThreeList
  Selection 0b01, ‘‘One’’
  Selection 0b10, ‘‘Two’’ %ADVANCED
  Selection 0b11, ‘‘Three’’
EndList
```

In this example, our list definition sections contains two lists; “&BooleanList” and “&ThreeList”. For list &ThreeList, the second selection is only available during binary configuration if the %ADVANCED filter has been selected.

2.4.7 InfoBlock

This section provides the user with information about the version of the BSF file being used, as well as where checksum information for this image that is located within the firmware itself. An additional, optional entry for EDK II platforms is the location, specified by the keyword, “VpdRegion” followed by a HEX value which is the offset from the beginning of the FD file to the first character of the VpdRegion. This entry is only valid for EDK II code-base platforms.

The following is the syntax for the BSF Info Block:

```
BeginInfoBlock
    version
    [image]
    [Description]
EndInfoBlock
```

Example:

```
BeginInfoBlock
  PPVer 2
  Image 0 Thru 20 At 10
EndInfoBlock
```

In this example, our BSF Info Block contains a version number of 2 and a checksum of 0 through 20 at location 10.

2.4.8 RelationshipDef

This section is used to define error conditions for specific variables or features. When an inconsistency is established, this tells the BSF processor that the expression which is defined should be evaluated to ensure an error condition is not encountered. If the expression evaluates to true, this means that an inconsistency condition has been met and the BSF processor should flag this as an error condition. Since variable names cannot be used before they have been defined, this section should be located after the InfoBlock section.

The following is the syntax for the BSF RelationshipDef section:

```
RelationshipDef
  Inconsistency = LogicalExpression , UserInterfaceName \
         [, Attribute]
  OneOf =          $Var2, $Var3
EndRelationship
```

Example:

```
RelationshipDef
  Inconsistency = ($Com1Address == $Com2Address) ,
                 ''The two serial ports must have different addresses'' ,\
                 LATE_CHECK
  Inconsistency = (($Day >= 29) AND ($Month == 2) AND
                 ($YEAR != 2008) OR ($YEAR != 2012) OR \
             ($YEAR != 2016) OR ($YEAR != 2020)) ,\
                 ''February only has 28 days!''
EndRelationship
```

One of selection relationships are used to associate multiple items as a set of choices of which only one of them can be picked. The valid values for OneOf include Feature names which are evaluated as Boolean data types.

2.4.9 Pages

This section is used to define the look and feel of a user interface that can enable or disable features and modify settings within a firmware image. The Page information can be described using the following. Each Page is identified by a unique name and an optional 'user interface' name. The 'Page Name' is a quoted string; leading and trailing whitespace characters within the quotes should be ignored.

The user interface should display EditNum, Combo, EditText, MultiText, Table or StringTable elements which use or set variables that are not masked by a filter selection.

It is recommended that a single word be used for the 'Page Name' rather than using multiple words with space characters within the string.

The following in a generic overview of a page definition:

```
Page ''Page Name'' [ , UserInterfaceName ]
   Element Definitions (1 or more)
  Page1 definitions (0 or more)
EndPage
```

Example:

```
Page ''Page1'' , ''This is Page 1''
  TitleB ''This is the parent page''
  EditText $Var1 , ''Name''
   Page ''Page1A'' , ''This is Page 1A''
    Title ''This is the child page''
   EndPage
EndPage
```

In this example, we define a page named “Page1”. Page1 has a bold title (TitleB) with the text “This is the parent page” and an edit text control named “Name”. Page1 also contains a child page “Page1A”. This page has a non- bolded title with the text “This is the child page”. Both page definitions provide human readable 'user interface' names that can be used by tools.

2.4.9.1 Title

The title element is used to place a static text label on the page. It can be used to separate elements or place a title at the top of the page. The following is the syntax of the title element:

```
Title string [ , Help string ]
```

2.4.9.1.1 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display 'pop-up tool tip' data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
Page ''IntroPg''
   Title ''Introduction''
   .
   .
   .
EndPage
```

1.Page definitions are pages defined in the same syntax as given here. They will be child pages of the current page. The page definitions must always be defined after the elements.

2.4.9.2 TitleB

The TitleB element is used to place a static bold text label on the page. It is identical to the title element except it will appear bold in the UI. The following is the syntax of the TitleB element:

```
TitleB string [ , Help string ]
```

2.4.9.2.1 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display ‘pop-up tool tip’ data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
Page ‘‘IntroPg’’
    TitleB ‘‘A Bold Introduction’’
    .
    .
    .
 EndPage
```

2.4.9.3 Combo

A combo box element is a standard ‘drop-down list box’. This element is used to allow the user to select one and only one value from a list of valid values instead of having to enter the data manually. The selection list must be defined in the list definition section.

The value’s data type specified in the list definition must match the data type specified for the variable that will be modified using this combo entry.

If the value stored in the binary data file does not match one of the selections, the selection that closest matches the value will be selected when the combo box is created. The ‘variable’ must be defined in the StructDef section.

The following is the syntax for a combo box element:

```
  Combo variable , string , list [ , Help string ]
```

2.4.9.3.1 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display ‘pop-up tool tip’ data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
List &Boolean
  Selection 0x1 , ‘‘True’’
 Selection 0x0 , ‘‘False’’
EndList

Page ‘‘Page1’’
   Combo $Var1 , ‘‘Enabled’’ , &Boolean
EndPage
```

In this example, we define a combo box named “Enabled” whose data is stored in the variable “$Var1”. The combo box has a drop-down list with two selections; “True” and “False”.

2.4.9.4 EditNum

The editnum element is an edit box with a label that allows users to change data displayed in hexadecimal, end-hexadecimal, decimal, or binary format. Numbers are expressed in hex (with or without a ‘0x’ prefix or with or without the ‘h’ suffix,) decimal (integer values) or binary (prefixed with ‘0b’ characters.) The user must enter the information in the same format as the displayed data. The following are the syntaxes for the five different number edit elements:

For a hexadecimal display:
```
EditNum variable , string , HEX [ , Help string ]
```

For an end-hexadecimal display:
```
EditNum variable , string , EHEX [ , Help string ]
```

For a decimal display:
```
EditNum variable , string , DEC [ , Help string ]
```

For a binary display:
```
EditNum variable , string , BIN [ , Help string ]
```

For an end-binary display:
```
EditNum variable , string , EBIN [ , Help string ]
```

2.4.9.4.1 Number Checking

Numeric values can be entered into a numeric element in either hexadecimal, end-hexadecimal, binary, end-binary or decimal format. These numbers must match the format specified in the element definition. If a number is not entered in format specified, the tools should provide an error message, and the value for the variable must not be changed until the correct formatted number has been entered.

2.4.9.4.2 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display ‘pop-up tool tip’ data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
Page ‘‘Page1’’
  EditNum $Var1 , ‘‘size’’
  , HEX
  , Help ‘‘Value must in HEX’’
EndPage
```

In this example, we define a number edit control called “size”. This edit control's data comes from the data defined by the variable $Var1 and is displayed in hexadecimal format.

2.4.9.5 EditText

The edit text element is used to create an editable text box with a label that allows the user to edit text.

The following is the syntax of an edit text element:

```
  EditText variable , string [ , Help string ]
```

2.4.9.5.1 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display ‘pop-up tool tip’ data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
Page ‘‘Page1’’
  EditText $Var1 ‘‘,’’ ‘‘Name’’ ‘‘,’’ Help ‘‘Your company name is?’’
EndPage
```

In this example, we create an edit text element named “Name”, whose data is stored in the variable “$Var1”.

2.4.9.6 Multi-Text

A multi-line text element is a used to define multiple line editable text box. This element is used for longer text strings or text strings that require carriage returns.

The following is the syntax of a multi-line edit text element:

```
  MultiText variable ‘‘,’’ string [ , Help string ]
```

2.4.9.6.1 Carriage Returns

To place a new line in the text of the control, use the escape sequence \n\r.

2.4.9.6.2 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display ‘pop-up tool tip’ data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
 Page ‘‘Page1’’
   MultiText $Var1 , ‘‘Address’’ , Help ‘‘Your address is?’’
EndPage
```

In this example, we define a multi-line edit text control named “Address” whose data is stored in the variable $Var1.

2.4.9.7 Table

The table element is used to create a spreadsheet control on a page. Table elements must be the last element specified for any given page. For example, you can have a page that has an editnum element, then a multi-text element and then a table element, but you cannot have a page that has an editnum element, a table element and then a multi-text element. Like the editnum element, the table element can display its data in hexadecimal, end- hexadecimal, decimal or binary format. The table element is unique in that the table data isn’t stored in a variable, but rather a pointer variable. This variable contains a pointer to a location in the binary file that contains the data of the table.

The following is the syntax for a table element:

```
  Table PtrVar , columns [ , Help string ]
```

2.4.9.7.1 Number Checking

Numeric values can be entered into a numeric element in either hexadecimal, end-hexadecimal, binary, end-binary or decimal format. These numbers must match the format specified in the element definition when the element is updated. If a number is not entered in format specified, the tools should provide an error message, and the value for the variable must not be changed until the correct formatted number has been entered.

2.4.9.7.2 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display ‘pop-up tool tip’ data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
  Page ‘‘Page1’’
    Table $PtrVar1 ‘‘Table’’ , Column ‘‘Time’’ 2 bytes , DEC
                               Column ‘‘Price’’ 2 bytes , HEX
  EndPage
```

In this example, we create a table called “Table” whose data is stored at “$PtrVar1”. The table has two columns; “time” and “price”. “Time” is a two byte column whose data is displayed (and must be entered) in decimal format, while “price” is a two byte column whose data is displayed (and must be entered) in hexadecimal format.

2.4.9.8 StringTable

The string table element allows the user to view strings in a spreadsheet control. The data for the table comes from a variable with a pre-defined size. However, each string, as defined in the string items list, does not have a size, but rather is sized based on a terminating null character. In other words, we define a string table with a total size, such as ten bytes. Then we say the table has a defined number of strings, for example two. Now, if the first string has a length of four (including the null-terminator), then the second string would only be allowed a length of six. An error should be generated if the length of all of the all of the defined strings exceeds the total size allocated for the string table. The null terminator must be counted in determining the length of each string.

The following is the syntax for a string table element:
```
  StringTable variable , items [ , Help string ]
```

2.4.9.8.1 Labels and Titles

It is recommended that labels and titles be kept to short strings with the longer definitions saved for the help message. Help messages are intended for use by tools that can display 'pop-up tool tip' data or, if tools do not support pop-up tool tips, the help text can be displayed along with the text.

Example:

```
  Page ''Page1''
    StringTable $Var1 ''StringTable'' , String ''String1''
                                     String ''String2''
  EndPage
```

In this example we create a string table whose data is stored in the variable "$Var1". This table has two strings with the headers "String1" and "String2".

2.4.9.9 Link

The link element may be used to provide a hypertext link to another page using a button. When the button is clicked, the page specified in the link element would be opened and the current page will be closed. Leading and trailing space characters within the quoted 'PageName' should be ignored.

The following is the syntax for the link element:
```
  Link ButtonText , PageName [ , Help string ]
```

The button text is the string that will appear on the button and the PageName is the label of the page to jump to. The page is defined like a DOS directory structure. For example, to find a child page all you need is the name of the page. To find a page at the same level as the current page, the name would need to have either a ".\" or "./" in front of it. To go up to the parent page, the page would be specified as ".." or any page can be specified from the root directory by the path: either "\parent\child" or "/parent/child".

Example:

```
  Page ''Page1''
    EditText $Var1 , ''Name''
  EndPage

  Page ''Page2''
     Link ''Back'' , ''\Page1''
  EndPage
```

In this example, we define two pages; "Page1" and "Page2". In page two, we specify the link "Back", which when clicked will take us to "Page1".

```
Page ‘‘Page1’’
  EditText $Var1 , ‘‘Name’’
  Link ‘‘Table’’ , ‘‘Page2’’
  Page ‘‘Page2’’
   Table $Table1 ‘‘Table’’ , Column ‘‘Time’’ 1 byte Column ‘‘Size’’
1 byte
   Link ‘‘Page1’’ , ‘‘..’’
  EndPage
EndPage
```

In this example, we define a Page, “Page1” that has a link to the embedded page, “Page2”. “Page2” has a link back up to “Page1”, but instead of specifying the entire path, we use the shortcut “..” to jump to the parent of “Page2”.

However, we could have specified the link as “Page1”.

3 Boot Setting File Specification

This section uses EBNF to define the data structures, format and content of the Boot Setting File. The BSF should be encoded as either ASCII, ISO-8859-1 or UTF-8.

Note: *The early BMP tools required that the file be DOS format, while newer tools must be able to process either DOS, OS/X or Unix formatted files.*

The top level format for the file is:

```
  <BsfFormat>     ::=      [<GlobalDef>]
                             [<FeatureDef>]
                             [<StructDef>]{0,1}
                             [<ListDef>]{0,}
                             <InfoBlock>
                             [<RelationshipDef>]{0,}
                             [<PageDef>]{0,}
```

If the PageDef sections are missing from the BSF file, the file cannot be used to display any user configurable binary settings.

The StructDef section is optional for BSF files that will not permit changing settings (where only users are only allowed to enable features prior to a build.)

One and only one StructDef section may be valid for a given instance of the BSF file. Directives may be used to encapsulate different StructDef sections, but the directive usage must provide a unique StructDef during processing.

Sections of a BSF file have a start tag and an end tag, such as ‘Page’ and ‘EndPage.’ Refer to table 1 for a list of section tags.

Table 1. Section Tag Keywords

Beginning Tag	Ending Tag	Description
GlobalDataDef	EndGlobalData	Values that can be used throughout the file
FeatureDef	EndFeature	Define Available Features
StructDef	EndStruct	Defines the data structure layout within the
List	EndList	Defines sets of valid values
BeginInfoBlock	EndInfoBlock	Contains information about the image
RelationshipDef	EndRelationship	Contains relationship parameters for specified elements
Page	EndPage	Display format (Page sections can be nested within other Page sections)

Entries in the BSF file consist of a keyword followed by content. In order to support multiple line entries, an entry is terminated by another entry keyword or a section's end tag. Refer to table 2 for a list of keywords.

Table 2. Begin Entry Keywords

Align	EditNum	Image	Prompt	StringTable
CategoryID	EditText	Inconsistency	Selection	Table
Column	Find	Link	Skip	Title
Combo	Find_Ptr_Ref	MultiText	SKUID	TitleB
DefaultID	Help	PPVer	String	VpdRegion

Entries may also be terminated by an entry termination keyword (these key words are part of the entry.) Refer to table 3 for a list of these entry termination keywords

Table 3. Entry Termination Keywords

bit	bits	byte
bytes	BIN	DEC
EBIN	EHEX	HEX

Reserved words have pre-defined meanings or values (that cannot be redefined.) Refer to table 4 for a list of reserved words and their values.

Table 4. Reserved Words

Reserved Word	Value
Enable | ENABLE | enable	1
Disable | DISABLE | disable	0
True | TRUE | true	1
False | FALSE | false	0
One | ONE | one	1
Zero | ZERO | zero	0
Thru	n/a
At	n/a
NULL	n/a
EOF	n/a
$SKUID	n/a
$_AS_BUILT_	n/a
$_DEFAULT_	n/a
#IF|#if	n/a
#elif|#ELIF|#elseif|#ELSEIF	n/a
#else|#ELSE	n/a
LATE_CHECK	n/a

3.1 Common Elements

All valid content within the file is case-sensitive.

Space and tab characters at the beginning and end of a line should be ignored by tools processing the file.

A back slash character at the end of a line can be used to continue a line. This character must be the last character on a line, must be preceded by a space character and must not be followed by a space character. It may not be used to extend a comment. Use of the back slash character within individual value elements (such as a directory path or a value that should not be split such as a quoted string) is not permitted.

The following elements may be used anywhere within the document.

3.1.1 Common Definitions

The following EBNF describes content used throughout this specification.

```
<Quote>             ::=    ‘"’
<Word>              ::=    (a-zA-Z)(a-zA-Z0-9_){0,}
<Number>            ::=    <HexNumber> <Int>
<HexDigit>          ::=    (a-fA-F0-9)
<Hex>               ::=    [‘0x’] <HexDigit>{1,}
<EndHex>            ::=    <HexDigit>{1,} [‘h’]
<HexNumber>         ::=    {<Hex>} {<EndHex>}
<Int>               ::=    (0-9){1,}
<Binary>            ::=    ‘0b’(0-1){1,}
<EndBinary>         ::=    (0-1){1,}‘b’
<BinaryNumber>      ::=    {<Binary>} {<EndBinary>}
<Size>              ::=    {‘byte’} {‘bytes’} {‘bit’}      {‘bits’}
<eol>               ::=    [‘0x0d’] ‘0x0a’
<text>              ::=    <Word>
                           [<WhiteSpaces>{1,} <Word> [<Punctuation>]]{0,}
<QuotedString>      ::=    <Quote> <text> <Quote>
<QuotedText>        ::=    <Quote> [<text> [<eol>]{0,}]{1,} <Quote>
<WhiteSpaces>       ::=    {‘0x20’} {‘0x09’}
<Punctuation>       ::=    {‘.’} {‘,’} {‘-’} {‘_’} {‘:’} {‘;’} {‘?’}
                           {‘!’} [<Symbols>]
<Symbols>           ::=    {‘~’} {‘@’} {‘#’} {‘$’} {‘%’} {‘^’} {‘&’}
                           {‘*’} {‘(’} {‘)’} {‘+’} {‘=’} {‘{’} {‘[’}
                           {‘}’} {‘|’} {‘\’} {‘<’} {‘>’} {‘/’}
```

3.1.2 Comments

Comments are statements in the file that serve no purpose other than to make the document more readable.

Comments may appear anywhere within the document.

Single line comments may use either a semi-colon (‘;’) character, or the C++ style comment line, using double forward slash (‘//’) characters.

Single line comments within a line terminate the processing of the line.

Comments do NOT terminate the processing of an element (a comment may be positioned at the end of a line, but the element's content may be continued on the following line.

Comments are permitted within an entry provided they are not in the middle of an entry field. Comments within entries must be positioned at the end of the line.

Multi-line comments or comment blocks can use the C style comment, where the comment block begins with a forward slash - star ('/*') character string and terminated by the star - forward slash ('*/') character string.

The following EBNF defines the comment.

```
  <Comments>             ::= {<SingleLineComments>} {<CommentBlock>}
  <SingleLineComments> ::= {';'} {'//'} <text> <eol>
  <CommentBlock>       ::= '/*' [<text>] [<eol>]
                            [[<text>]{0,} <eol>]{0,}
                           '*/' <eol>
```

3.1.3 Directives

Directives provide conditional control of content within the file. Directives use the C pre-processor format of #if, #elif, #elseif, #else and #endif. Additionally, these directives may be written in upper case: #IF, #ELIF, #ELSEIF, #ELSE and #ENDIF.

Directives must be the only statement on a line. Directives may be nested within directive sections. String evaluations are not permitted.

The expression immediately following the #if statement controls whether the content after the line is processed or not. TRUE is any non-zero and/or non-null value other than zero.

Each #if within the source code must be matched with a closing #endif.

Zero or more #elif statements are permitted, and may be nested, however one and only one #else statement is permitted for an #if statement.

The file must be processed top to bottom, with the expressions being processed left to right using in-fix notation.

Refer to section 2.4.1 for a detailed discussion of directive statement usage, as well as the description for BSF sections, below.

The following EBNF defines the directives.

```
<Directives>          ::=    {'#if'} {'#IF'} <expression> <eol>
                           <entryelement>
                           [<elseif>]{0,}
                           [<else>]
                           {'#endif'} {'#ENDIF'} <eol>
<entryelement>        ::=    One or more lines of Valid Content
<elseif>              ::=    <elseifkeyword> <expression> <eol>
                           <statements>
<elseifkeyword>       ::=    {'#elif'} {'#ELIF'} {'#elseif'}
                           {'#ELSEIF'}
<else>                ::=    {'#else'} {'#ELSE'} <eol>
                           <statements>
<expression>          ::=    A valid C expression
```

3.1.3.1 Related Definitions

`Valid Content`

Valid content is limited to elements within a StructDef or the List, FeatureDef and Page sections and around elements within these sections. Directives may NOT be used around GlobalDataDef sections.

`PcdName`

This is entry is a formatted entry to specify EDK II Platform Configuration Data entries. The PcdName consists of the C variable name of the Token Space GUID (a name space for PCDs) followed by a dot '.' character, then the C variable name of the PCD that represents a token number that is unique to the token space.

`Variable`

Variable values must be numeric or Boolean in nature. Tools will cast all variable values to unsigned long long (UINTN) when performing evaluations on values that are of different datum types. Variables used in directives must declared before they can be used.

`Expression`

C-style expression using C relational, equality and logical numeric and bitwise operators and/or arithmetic and bitwise operators that evaluate to a value of either TRUE or FALSE. Precedence and associativity follow C standards. Along with absolute values, macro names and PCDs may be used within an expression. For both macro names and PCDs, the element must be previously defined before it can be used. The Feature PCD is a Boolean PCD that is set to either True (1) or False (0).

3.1.4 Symbols

The dollar sign ('$') symbol is used to represent a variable; the ampersand ('&') characters is used to denote a list. The comma (',') character is used to separate items.

3.2 GlobalDataDef Definitions

This optional section is used to define filters, profiles and SKUs that can be used by tools and associates different variables with the profiles.

Filters may be used similar to directives, however they do not modify the content or layout of the flash area. They used as a bitwise mask. They are used to mask questions for different structure elements from the user. Filters may not be used in directive statements.

Profiles (identified using the keyword, DefaultID) are used to specify binary configuration values for elements (variables) define in the structure section. It is not permissible to have a data element in the StructDef section that has the same name as the profile. Additionally, profiles may not be used in directive statements. The profile identifier can be used as a label in the FeatureDef and StructDef sections.

SKUs are typically associated with a hardware's available features. SKUs are typically used in directives, however they may also be used to specify settings with the structure section.

The ViewID mask is used in two ways. Tools may filter what is displayed based on a selected ViewID, and within the StructDef and List sections, the ViewID is used to define the binary bit mask.

Note: *The ViewID labels are used as a bit mask for displaying page content. A maximum of 32 bits are available for use by ViewID values.*

Note: *The DefaultID profile identifier must be unique to this BSF file, i.e., it may not be defined more than once in the GlobalDataDef section.*

Note: *In order to use SKUID statements in directives, the SKUID must be defined, i.e., a directive containing '#if SKUID == 0x03' is only permitted in the BSF file if the 'SKUID = 0x03, "somename"' line is defined in this global data section.*

This is an optional section.

3.2.1 Global Definitions Section Format

```
<GlobalDef>     ::=  'GlobalDataDef' <eol>
                     [<LockView>]
                     [<Categories>]{0,}
                     [<Views>]{0,}
                     [<Profiles>]{0,}
                     [<Skus>]{0,}
                     'EndGlobalData' <eol>
<LockView>      ::=    'UserView' '=' <FilterName> <eol>

<Categories>    ::=    'CategoryID' '=' <CategoryName> ','
                           <BitMask> ',' <UiName>

<Views>         ::=    'ViewID' '=' <FilterName> ',' <BitMask>
                           ',' <UiName>
<FilterName>    ::=    '%' <Word> [<AsBuilt>]
<CategoryName>  ::=    '%' <Word>
<BitMask>       ::=    '0x'<HexDigit>{8} || <BinaryNumber>{32}'b'
                             [',' <Help>]
<Profiles>      ::=    'DefaultID' '='       <DefaultId> ',' <UiName>
<DefaultId>     ::=    '$' <Word> [<AsBuilt>] [',' <Help>]
<UiName>        ::=    <QuotedString> <eol>
<Skus>          ::=    'SKUID' '=' <SkuId> [<AsBuilt>] ',' <UiName>
<SkuId>         ::=    <Hex>
<AsBuilt>       ::=    '$_AS_BUILT' '=' {'1'} {'0'}
```

Related Definitions

`LockView`

If present, this gives an indication to a BSF parser that the user view is locked to the provided Filter Name's value.

`Categories`

A mask which allows a series of variables to be associated with a particular category. These categories are logical groupings which enable a tool to coalesce information into meaningful organization. The Bitmask associated with the filter provides a mechanism which one can create selections which reference multiple items at once.

Views

A mask that associates a variable with a particular group.   This mask is intended to be used so that it can enable or limit the visibility of variables based on a current user's Since this is a mask, it provides the ability for a hierarchical view of accessibility such that one mask has greater visibility than others.

UiName

The string for a user interface used to represent the variable.

DefaultID

A keyword used to specify default settings for a given profile; typically used during manufacturing, test, or production.

SkuId

A number used to specify groups of features or settings; typically used during manufacturing.

BitMask

A hex value that is used as a bit mask for displaying content in a user interface.

Examples Input BSF

```
GlobalDataDef
              ViewID   = %ADVANCED,   0xffffffff , ‘‘Advanced View’’
              ViewID   = %INTERMEDIATE, 0x80000008, ‘‘Intermediate View’’
              ViewID   = %SAFE,              0x80000004 , ‘‘Safe View’’
              CategoryID      = %PCI, 0x000000001 , ‘‘PCI’’
              CategoryID      = %USB, 0x000000002 , ‘‘USB’’ DefaultID
              = $STANDARD , ‘‘Standard Defaults’’ DefaultID =
              $MANUF , ‘‘Manufacturing Defaults’’ DefaultID =
              $USER1 , ‘‘Mike’s Preferred Defaults’’ SKUID =
              0x00 , ‘‘Menlow’’
              SKUID = 0x01 , ‘‘Crown
              Beach’’ SKUID = 0x02 ,
              ‘‘XYZ Platform’’
            EndGlobalData

            StructDef Find
              ‘‘Begin’’
              #if SKUID == 0x01
                $Var1 1 byte $MANUF= 0x08 $USER1= 0x05  // Always available
                                                        // for modification
                #elif SKUID == 0x02
                $Var1 1 byte $MANUF= 0x08 $USER1= 0x03  // Always available
                                                        // for modification
                #else
                $Var1 1 byte                            // Always available
                                                        // for modification
                #endif

                $Var2 1 byte %ADVANCED                  // Can only be modified if
                                                        // ADVANCED is selected Filter

                $Var3 1 byte %INTERMEDIATE              // Can only be modified if
                                                        // ADVANCE or INTERMEDIATE
                                                        // filter selected, but not
                                                        // SAFE or EVERYONE

                $Var4 1 byte %SAFE %USB                 // Can only be modified if
                                                        // ADVANCED or SAFE filter
                                                        // selected but not
INTERMEDIATE
                                                        // or EVERYONE. Also member of
                                                       // USB category.

                $Var5 1 byte %INTERMEDIATE %SAFE        // Access if any one of the
                                                        // three filter values is
                                                        // selected
                $Var6 1 byte                            // Always available for
                                                        // modification
  EndStruct
```

Example ‘As Built’ BSF

```
GlobalDataDef
  DefaultID    = $STANDARD , ‘‘Standard Defaults’’
   DefaultID   = $MANUF    , ‘‘Manufacturing Defaults’’
   DefaultID   = $USER1    , ‘‘Mike’s Preferred Defaults’’
   SKUID       = 0x00      , ‘‘Menlow’’
   SKUID       = 0x01 $_AS_BUILT_ = 1 , ‘‘Crown Beach’’
   SKUID       = 0x02 , ‘‘XYZ Platform’’
EndGlobalData

StructDef Find
  ‘‘Begin’’
  #if SKUID == 0x01
    $Var1 1 byte $_AS_BUILT_ = 0x0F $MANUF= 0x08 $USER1= 0x05
                                            // Var1 Always available
                                            // for modification
  #elif SKUID == 0x02
  $Var1 1 byte $MANUF= 0x08 $USER1= 0x03    // Always available
                                            // for modification
  #else
  $Var1 1 byte                              // Always available
                                            // for modification
  #endif

  $Var2 1 byte %ADVANCED                    // Can only be modified if
                                            // ADVANCED is selected Filter

  $Var3 1 byte %INTERMEDIATE                // Can only be modified if
                                            // ADVANCE or INTERMEDIATE
                                            // filter selected, but not
                                            // SAFE or EVERYONE

  $Var4 1 byte %SAFE   // Can only be modified if
                                            // ADVANCED or SAFE filter
                                            // selected but not INTERMEDIATE
                                            // or EVERYONE

  $Var5 1 byte %INTERMEDIATE %SAFE // Access if any one of the
                                            // three filter values is
                                            // selected
  $Var6 1 byte                              // Always available for
                                            // modification
EndStruct
```

Note: *One and only one ViewID may be selected by the user.*

Note: *One and only one DefaultID may be selected. It is not possible to select more than one set of pre-defined values for binary configuration settings.*

Note: *One and only one SKUID may be set in the ‘As Built’ file.*

3.3 FeatureDef Definitions

This optional section is used to define Features that may be enabled or disabled during the build. Some features may also be used post build to support features that are in a binary image, but that can be disabled using a mechanism that is code-base specific. One and only one FeatureDef section is permitted within the BSF file.

The feature name is typically used in directives around structure elements, pages or elements within a page. It may also be a StructDef variable element.

Directives are permitted within the FeatureDef section. Arguments used in the expression of a directive within the FeatureDef section are limited to SKUID values defined in the GlobalDef section, as well as using the name of feature a previously defined feature element within this section. Directives may only encapsulate an entire feature entry.

The BSF file may also specify a default value of either ‘Enabled’ or ‘Disabled’ for the features defined in this section.

The ‘As Built’ file requires the ‘$_AS_BUILT_ = 1’ entry for every feature that was selected by the user.

If a feature does not include the $_DEFAULT_ entry, tools should assume that the feature is disabled.

This is an optional section.

3.3.1 Feature Definitions Section Format

```
<FeatureDef>        ::=   ‘FeatureDef’ <eol>
                          [<Features>]{1,}
                          ‘EndFeature’ <eol>
<Feature>           ::=   {<Edk2Features>} {<CefdkFeatures>}

<Features>          ::=   <Name><Entry> <eol>

<PcdName>           ::=   ‘$’ <TokenSpaceName> ‘.’ <PcdCName>

<Name>              ::=   ‘$’ <Word>

<Entry>             ::=   [‘,’ [<Filter>] [<Labels>]] ‘,’ <UiInfo>

<UiInfo>            ::=   <Prompt> [‘,’ <Help>] <eol>

<Filter>            ::=   <MaskName> [<MaskName>]{0,}

<MaskName>          ::=   <Name>

<Default>           ::=   ‘$_DEFAULT_’ ‘=’ {‘1’} {‘0’}

<TokenSpaceName>    ::=   <Word>

<PcdCName>          ::=   <Word>

<Labels>            ::=   [<Default>] [<AsBuilt>] [<Label>]{0,}

<Label>             ::=   <Name> ‘=’ {‘1’} {‘0’}

<AsBuilt>           ::=   ‘$_AS_BUILT_’ ‘=’ {‘1’} {‘0’}

<Prompt>            ::=   <QuotedString>

<Help>              ::=   {<Strings>} {<QuotedText>}
<Strings>           ::=   <QuoteString> [<QuotedString>]{0,} <eol>
```

3.3.1.1 Related Definitions

`Name`

This is an entry which is formatted as a $Variable reference and should be interpreted as a BOOLEAN type, meaning that it is either Enabled or Disabled.

`MaskName`

Mask names are used by tools to generate a bit mask using the value of the mask name defined in the FilterID entry of the GlobalDataDef section. The values are logically or'd together to create the bit mask.

3.3.1.2 Examples Input BSF

```
FeatureDef
  $USB_FEATURE, $_DEFAULT_ = 1, ‘‘Enable USB?’’ ,
      ‘‘Enables input and output USB devices.\nFeature is always in Flash’’
  #if $USB_FEATURE == 1
      $USB_KEYBOARD, $_DEFAULT_ = 1, ‘‘Enable USB Keyboard?’’,
         ‘‘Enable external USB Keyboard’’
      $USB_MOUSE, $_DEFAULT_ = 1, ‘‘Enable USB Mouse?’’ , ‘‘Enable external USB Mouse’’
  #endif
  #if SKUID == 0x01
      $TOUCH_SCREEN_FEATURE, $_DEFAULT_ = 0 $MANUF = 1,
        ‘‘Enable Touch Screen Input?’’ ,
        ‘‘Feature is not included in the flash part if it is disabled’’
  #endif
EndFeature


StructDef Find
  ‘‘Begin’’
  $Var1               1 byte
  $Var2               4 bits
  $USB_FEATURE        1 bit
  ALIGN
  #if $USB_FEATURE
    $Var3             20 bytes
  #else
    Skip              20 bytes
  #endif

  #if (SKUID == 0x01) && $TOUCH_SCREEN_FEATURE
    Find ‘‘TSFBegin’’
      $Var4           1 bit
      $Var5           1 bit ALIGN
      $Var6           1 byte
  #endif

  Find ‘‘Next’’
  $Var7               1 byte
  $Var8               32 bits
EndStruct
```

3.3.1.3 Examples 'As Built' BSF

```
GlobalDataDef
  SKUID = 0x00 $_AS_BUILT_ = 1 , ''Menlow''
  SKUID = 0x01 , ''MyHardware''
EndGlobalData

FeatureDef
    $USB_FEATURE $_DEFAULT_ = 1 $_AS_BUILT_ = 1, ''Enable USB?'' ,
      ''Enables input and output USB devices.\nFeature is always in Flash''
    #if USB_FEATURE
      $USB_KEYBOARD $_AS_BUILT_ = 0, ''Enable USB Keyboard?'', ''Enable external USB
Keyboard''
      $USB_MOUSE $_AS_BUILT_ == 0, ''Enable USB Mouse?'' , ''Enable external USB Mouse''
    #endif
    #if SKUID == 0x01
      $TOUCH_SCREEN_FEATURE $_AS_BUILT_ = 0 $MANUF = 1,
         ''Enable Touch Screen Input?'' ,
         ''Feature is not included in the flash part if it is disabled''
    #endif
EndFeature

StructDef Find
  ''Begin''
  $Var1               1 byte $_AS_BUILT_ = 0x0f
  $Var2               4 bits $_AS_BUILT_ = 0b1110
  $USB_FEATURE        1 bit   $_AS_BUILT_ = 0b1
  ALIGN
  #if $USB_FEATURE
  $Var3               20 bytes $_AS_BUILT_ = \
    0x55, 0x45, 0x46, 0x49, 0x20, 0x31, 0x2e, 0x30, 0x0D, 0x0A, \
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  #else
    SKIP 20 bytes
  #endif

  #if (SKUID == 0x01) && $TOUCH_SCREEN_FEATURE
    Find ''TSFBegin''
      $Var4           1 bit
      $Var5           1 bit ALIGN
      $Var6           1 byte
  #endif Find
  ''Next''
  $Var7               1 byte $_AS_BUILT_ = 0xFF
  $Var8               32 bits $_AS_BUILT_ = 0x8000C000
EndStruct
```

3.4 StructureDef Definitions

The structure definition section is used to define the data areas of the binary file. It maps the layout of the binary image data to variables used by the page features to display data. The order that the variables are defined in this section does not necessarily match the order that they appear in the binary data file.

The size of the data section defined by the StructDef is fixed at build time.

Directives used in this section are typically for determining what data may or may not be included or configured.

The directive statements may use SKUID names from the GlobalDataDef section, feature names from the FeatureDef section and/or variable names that have been previously defined within the StructDef section in the evaluation portion of the directive statement.

Filters (defined in the GlobalDataDef section,) may be specified. One or more filter names may be specified for any given data element in this section. The filter names' values are logically or'd to create the bit mask. A valid data element (not excluded by directives) that does not have a filter name listed in the entry may be presented to the user for modification.

A profile identifier (defined in the GlobalDataDef section,) can be used as a label which can be appended to an entry. Labels will be used for specifying pre- determined values.

This is an optional section.

Note: *If this StructDef section is not specified in the BSF file, no binary configuration settings can be modified. Only pre-build feature selection is possible.*

3.4.1 Structure Definition Section Format

```
<StructDef>             ::=    'StructDef' <eol>
                               [<Structures>]{1,}
                          'EndStruct' <eol>
<Structures>::=         {<FindStruct>} {<FindPtrStruct>}
<FindStruct>            ::=    'Find' <Sig> <eol> [<VarStatements>]{1,}


<Sig>                   ::=    <Quote> <Signature> <Quote>

<Signature>             ::=    [{<Symbols>}{<Punctuation>}{<Letter>}]{1,}

<Letter>                ::=    (a-zA-Z0-9)

<FindPtrStruct>         ::=    'Find_Ptr_Ref' <FindWhat> <eol> [<VarStatements>]{1,}


<FindWhat>              ::=    {<Quote> <Sig> <Quote>} {<Data>}

<Data>                  ::=    <Number> [',' <Number>]

<VarStmt>                      {<VarStatements> <eol>} {<Special> <eol>}

<VarStatements>         ::=    {<Variable>} {<PointerVar>} [<Qualifier>]

<Qualifier>             ::=    [<Filter>] [<LabelData>]

<Variable>              ::=    '$' <Word>           <Number>      <Size>
```

```
<Size>                 ::=    {‘bit’} {‘bits’} {‘byte’} {‘bytes’}

<Filter>               ::=    <MaskName> [<MaskName>]{0,}

<MaskName>             ::=    ‘$’ <Word>

<PointerVar>           ::=    <PtrStr> ‘,’ <DataStr> ‘,’ <PointerOffset>

<PtrStr>               ::=    ‘$’ <Word>

<DataStr>              ::=    ‘$’ <Word>


<PointerOffset>        ::=    <PtrSize> [<Offset>]

<Offset>               ::=    ‘,’ ‘Offset’ <Number> <Size>

<PtrSize>              ::=    {<Number> <Size>} {<PtrVar>} ‘$’ <Word>
                              [<Label>] [<Default>] [<AsBuilt>]
<PtrVar>               ::=
                              <LabelName> ‘=’ <LabelValue> ‘$’ <Word>
<LabelData>            ::=    {<NumberValue>} {<QuotedStr>}

<Label>                ::=

<LabelName>            ::=

<LabelValue>           ::=



                              {<UnicodeStr>} {<NumberArray>}

<NumberValue>          ::=    {<Number>} {<Binary>} {<EndBinary>}

<NumberArray>          ::=    {<CStyleArray>} {<CsvArray>}

<CStyleArray>          ::=    ‘{’ <NumberValue>
                              [‘,’ <NumberValue>]{1,} ‘}’


<CsvArray>             ::=    <NumberValue> [‘,’ <NumberValue>]{1,}


<UnicodeStr>           ::=    ‘L’<QuotedStr>
```

```
<Default>             ::=    '$_DEFAULT_' '=' <AnyValue>

<AnyValue>            ::=    {<Numeric>} {<Strings>} {'L'<Strings>}
                             {<Array>}

<Numeric>             ::=    {<HexNumber>} {<BinaryNumber>} {<Int>}

<Array>               ::=    {<CArray>} {<CvArray>}

CvArray>              ::=    <Numeric> [ ',' <Numeric> ]{1,}

<CArray>              ::=    '{' <CvArray> [ <CArray> ]{1,} '}'

<EOL>                 ::=    {['0x0d'] '0x0a'} {['\r'] '\n'}

<Strings>             ::=    <Quote> [<text> [<EOL>]{0,} ]{1,} <Quote>

<AsBuilt>             ::=    '$_AS_BUILT_' '=' <LabelValue>

<Special>             ::=    {<SkipStmt>} {<Align>}

<SkipStmt>            ::=    {'Skip'} {'SKIP'} <Number> <Size>

<Align>               ::=    'ALIGN' [<AlignNumber>]

<AlignNumber>         ::=    {'1'} {'2'} {'4'} {'8'} {'16'} {'32'}
                             {'64'} {'128'} {'256'} {'512'} ...
```

Related Definitions

Signature

Each signature must be a unique value in this BSF file (they may not be used more than one time.) Additionally, the values should not be shorter parts of other signatures, i.e., do not use '$Begin', 'Begin' and/or 'BeginPtr' within the same BSF file. Also, since there is no order to these Find sections, it is best to make sure that the signature is not subset of another signature, like: 'Begin' and 'MyBegin', where 'Begin' could be found twice within the data file.

Find

The Find command causes a search of the data file for a specified string and sets the current location to the byte following the string. If the string is not found, the file stops loading and an error message must be reported to the help view. The structure definition section can support multiple find statements which allows a BSF file to support binary files with multiple data segments

Find_Ptr_Ref

The Find_Ptr_Ref command causes a search of the data file for a specified string or number value and returns the location at the beginning of that item. If the string is found, all pointer variables defined after it will add that location to their data pointer. If the string is not found, the location will be set to the beginning of the file. This command is very similar to the Find command, except it only affects pointers, whereas Find affects all variables.

Skip

The Skip command allows the user to skip a specified size of data in the data file. When the skip command is called, the current address pointer is moved forward by the specified size.

Align

To help facilitate dealing with bit structures, the BSF language provides the keyword ALIGN. If no number is given, the Align forces the next variable to begin at the next BYTE aligned address. If a number is given, the Align forces the next variable to begin at the byte at the (# of byte) alignment which must be a power of two, e.g., 2^0, 2^1,..., 2^n.

LabelName

A profile identifier must be defined in the GlobalDataDef section in order to be used as a label in this section, i.e., you may not use a label name that does not exist in the GlobalDataDef section. The only exceptions to this rule are for the '$_DEFAULT_ and '$_AS_BUILT_' labels, used by tools to specify default or actual values for features and variables.

MaskName

Mask names are used by tools to generate a bit mask using the value of the mask name defined in the FilterID entry of the GlobalDataDef section. The values are logically or'd together to create the bit mask.

Example

```
/*****************************************************************
Structure definition section
*****************************************************************/
  StructDef           //mark beginning of structure def section
    Find "$Begin"     //find statement

  //conditional variables
    $Com1             1      byte   //variable definition
    $TableHelp        1      byte   //variable definition
    $CheckSum         1      byte
    SKIP              9      bytes  //Skip 10 bytes

  //Edit Text Controls variables
    $EText            30     bytes  //variable definition
    $ETextH           30     bytes  //variable definition
    SKIP              10     bytes  //skip 10 bytes

  //Multiline edit controls
    $MLEdit           100    bytes  //variable definition
    $MLEditH          100    bytes  //variable definition
    SKIP              10     bytes  //skip 10 bytes

 //Edit Controls
    $ENumHex          2      bytes  //variable     definition
    $ENumHexH         1      bytes  //variable     definition
    $ENum             2      bytes  //variable     definition
    $ENumH            1      byte   //variable     definition
    $ENumDec          2      bytes  //variable     definition
    $ENumDecH         1      byte   //variable     definition
    $ENumBin          2      bytes  //variable     definition
    $ENumBinH         1      byte   //variable     definition
    $ENumEBin         2      bytes  //variable     definition
    $ENumEBinH        1      byte   //variable     definition
    SKIP              7 bytes //skip 10 bytes

//directive- $Com1 equals 1
  #if $Com1 == 1
    $C1ComboD 2         bytes //variable      definition
    $C1ComboH 1  bytes //variable     definition
    $C1ComboE 2  byte  //variable     definition
    $C1ComboB 1  byte  //variable     definition

  //directive- $Com1 less than 1
  #elif $Com1 < 1
    $C1ComboD 1         bytes  //variable      definition
    $C1ComboH 2  bytes  //variable    definition
    $C1ComboE 2  byte   //variable    definition
    $C1ComboB 1  byte   //variable    definition
  //directive- $Com1 greater than 1
  #else
    $C3ComboD         2      bytes  //variable     definition
    $C3ComboH         1      byte   //variable     definition
    $C3ComboE         2      bytes  //variable     definition
    $C3ComboB         1      byte   //variable     definition
  //End directive statement
  #endif

  $ComboDH            1      byte
  $ComboHH            1      byte   //variable     definition
  $ComboEH            1      byte   //variable     definition
  $ComboBH            1      byte   //variable     definition

  SKIP                10 bytes      //skip 10 bytes
  ALIGN                      //align to next byte

  //TABLES
  //string tables
  $STbl               50 bytes      //variable definition
  $STblH              50 bytes      //variable definition

  //dynamic size table with a byte offset
  $TDynOffPtr         2 bytes //variable definition
  $TDynOffSize        1 byte  //variable definition
  //pointer variable definition
  $TDynOff , $TDynOffPtr , $TDynOffSize , Offset 10 bytes

  //dynamic size table with a bit offset
  $TDynOffbitPtr      2 bytes //variable definition
  $TDynOffbitSize 1 byte            //variable definition
  //pointer variable definition
  $TDynOffBit , $TDynOffbitPtr , $TDynOffbitSize , Offset 6 bits

  //dynamic size table without an offset
  $TDynPtr            2 bytes //variable definition
  $TDynSize 1 byte                 //variable definition
  $TDyn , $TDynPtr , $TDynSize      //pointer variable definition

  Find_Ptr_Ref "SecondBegin"

  //static size table with a byte offset
  $TStatOffPtr        2 bytes //variable definition
  //pointer variable definition
  $TStatOff , $TStatOffPtr , 16 bytes , Offset 10 bytes

  //static size table with a bit offset
  $TStatOffbitPtr     2 bytes //variable definition
  //pointer variable definition
  $TStatOffBit , $TStatOffbitPtr , 16 bytes , Offset 10 bits

  //static size table with no offset
  $TStatPtr           2 bytes //variable definition
  $TStat , $TStatPtr , 16 bytes     //pointer variable definition
EndStruct
```

3.5 List Definitions

This section defines the format for lists. Lists are used by tools to present the user with a choice of valid values. Each list is uniquely identified by a variable name. More than one List section can be specified. The tools will display the quoted string value for each selection, allowing the user to choose one of the values - the value following the selection keyword is the value that will be use to complete the data structure specified in the page definition referencing the list. More than one of the structure variables can use the same list, e.g., list variables are mapped 1 to many (structure variables.)

The directive statements are permitted around List sections, list Selection entries and the UiName portion of the Selection entry. The directive expression may use labels (DefaultId profile identifier and

SKUID values) from the GlobalDataDef section, feature names (or PcdNames) from the FeatureDef section and/or variable names that have been defined within the StructDef section.

This is an optional section.

3.5.1 List Section Definition Format

```
<ListDefs>          ::=    'List' <ListName> <eol>
                                [<Selections> <eol>]{2,}
                                'EndList' <eol>
<ListName>          ::=    '&' (a-zA-Z0-9){1,}
<Selections>        ::=    'Selection' <Value> ',' <UiName> [<Filters>]
<UiName>            ::=    <QuotedString> [<QuotedString>]{0,}
<Value>             ::=    {<Number>} {<Binary>} {<QuotedString>}
<Filters>           ::=    <MaskName> [<MaskName>]{0,} [ ',' <Help>]
<MaskName>          ::=    '$' <Word>
<Help>              ::=    <QuotedString> [<QuotedString>]{0,}
```

3.5.1.1 Related Definitions

UiName

The string for a user interface, used to represent the value.

MaskName

Mask names are used by tools to generate a bit mask using the value of the mask name defined in the FilterID entry of the GlobalDataDef section. The values are logically or'd together to create the bit mask.

3.5.1.2 Examples

```
  List &BooleanList
      Selection 1h , ''True''
      Selection 0h , ''False''
  EndList

  List &ThreeList
      Selection 0b01, "One"
      Selection 0b10,
      #if $Label == 2
          "Two - default"
      #else
          ''Two''
      #endif
  Selection 0b11, "Three"
EndList
```

3.6 InfoBlock Definitions

Info blocks are used to record data about the BSF or data file. There is only one piece of required information, a version section. The checksum is calculated by adding the byte values starting with the BeginAddr and ending one byte short of the EndAddr, storing the sum at Location. The EndAddr must be greater than the BeginAddr.

If the all parameters in the IMAGE command (BeginAddr, EndAddr and Location) are EOF, then the checksum result is automatically placed in the VBT (VideoBiosTable) Header Checksum Field at offset 0x1A (from the detected start of the VBT data region).

The checksum is calculated by adding the values starting at the BeginAddr and through the end of the configuration element located at the EndAddr and storing the sum at Location. The algorithms for the one byte checksum calculations follows.

Directives are permitted within the InfoBlock section. One and only one valid InfoBlock section is permitted within the BSF file.

In addition, the InfoBlock will optionally contain a field, which contains a string that contains a description of the BSF file itself.

```
BYTE
CalcCheckSum (                 // Calculate a checksum
  DWORD BeginAddr,
  DWORD EndAddr,
  DWORD Location
  )
{
  unsigned char sum = 0;       // Var used in the summation
  unsigned char checksum = 0; // Var used to store the checksum
// Loop thru the data
  for (int i = BeginAddr; i < Endaddr; i++) {
    sum += data[i];    // Sum the entire data range
    }
  checksum = (-sum);   // SUMMATION & CHECKSUM are inverts
    return checksum;
}

TestCheckSum()         // Verifying a checksum
{
  unsigned char sum = 0;       // Var used in the summation

  // Loop thru the data
  for (int i = startOffset; i < endOffset; i++) {
    sum += data[i];    // Sum the entire data range
    }
  if (sum == 0)        // Checksum is OK if summation == 0
  {
    return TRUE;
  } else {
    return FALSE;
  }
```

This is a required section.

3.6.1 Info Block Section Format

```
<InfoBlock>            ::=    'BeginInfoBlock' <eol>
                              'PPVer' <Version>
                               [<Image>]{0,1}
                              [<Description>]

                              'EndInfoBlock' <eol>

<Version>              ::=    {<Integer>} {<QuotedString>} <eol>

<Checks>               ::=    {<CefdkImage>} {<Edk2Checksum>}

<Image>                ::=    'Image' <Checksum>

<Checksum>             ::=    <BeginAddr> 'Thru' <EndAddr> 'At' <Location>

<BeginAddr>            ::=    {<Int>} {<Variable>} {'EOF'}

<EndAddr>              ::=    {<Int>} {<Variable>} {'EOF'}

<Location>             ::=    {<Int>} {<Variable>} {'EOF'} <eol>

<Variable>             ::=    '$' <Word>

<Description>          ::= 'Description' <QuotedString> \
                              [<QuotedString>]{0,}<eol>
```

Related Definitions

Version

The version of this BSF file. This version number or name is informational only, and is not part of the flash image.

BeginAddr

The offset into the rom image or byte aligned variable where the checksum calculation must start.

EndAddr

The offset into the rom image or byte aligned variable where the checksum calculation must end.

Location

The offset or variable where the checksum will be placed.

Example

```
/*****************************************************************
BMP Info Block
****************************************************************
/
BeginInfoBlock
  PPVer 2
  Image $FirstSetting Thru $LastSetting At $Checksum
  Description ''A highly optimized Configuration for Calpella''
EndInfoBlock
```

In this example, our BSF Info Block contains a version number of 2. The checksum range would include all data starting at the variable value specified by $FirstSetting and continue through the end of the configuration element located at the variable value of $LastSetting. The resulting checksum would then be stored in the variable $Checksum.

3.7 RelationshipDef Definitions

This section is used to describe error conditions for specific variables or features as well as providing a means to establish one-of relationships between selectable items. An inconsistency is expressed as relationships between variables or features with other selectable items and/or numeric values. This does not presume to do string-based comparisons. When an inconsistency is established, this tells the BSF processor that the expression which is defined should be evaluated to ensure an error condition is not encountered. If the expression evaluates to true, this means that an inconsistency condition has been met and the BSF processor should flag this as an error condition.

One-of selection relationships are used to associate multiple items as a one-of set of choices. Common scenarios where this would show up is where a user would be presented with several possible choices and only one of those choices is allowed to be active at a time. This also provides a hint to a BSF parser that such a relationship could also be displayed as a radio selection.

This section should be located after the InfoBlock section.

This is an optional section.

3.7.1 RelationshipDef Section Definition Format

```
<RelationshipDef> ::= ‘‘RelationshipDefDef’’ <EOL>
                      [<Inconsistency>]{0,}
                      [<OneOf>]{0,}
                      ‘‘EndRelationship’’ <EOL>
<Inconsistency>   ::= ‘Inconsistency’ ‘=’ <Expression> ‘,’
                      <UiName> [‘,’ <LateCheck>]

<LateCheck>       ::= ‘LATE_CHECK’

<OneOf>           ::= ‘OneOf’ ‘=’ <VarName> [‘,’ <VarName>]{1,}
```

Related Definitions

Inconsistency

An association of certain configuration settings with other values which create an error condition. This is typically used to express conditions which involve relationships between configuration settings or relationship between certain configuration settings and specific values.

Expression

C-style expression using C relational, equality and logical numeric and bitwise operators and/or arithmetic and bitwise operators that evaluate to a value of either TRUE or FALSE. Precedence and associativity follow C standards. Along with absolute values, macro names and PCDs may be used within an expression. At least one variable name listed in the StructDef section or one feature name list in the FeatureDef section must appear in the expression.

LateCheck

If an inconsistency definition has a LATE_CHECK attribute, this is to be used as a hint to the parser that the error condition should not immediately cause an error to occur. However this would not enable the parser to save bad values to the target. This is primarily used when some configuration items have exclusive relationships that make it an error condition for them to be the same value as each other. In this situation there are cases where a user might need to temporarily set one value to be the same as another while in the midst of swapping the item values. An example is if one

UART has an address set to 0x2F8 and UART2 is set to 0x3F8. It is illegal to have these values be the same. The only way to switch their values is to temporarily switch the UART value to 0x3F8 (an error while UART2 is that value) and then switch UART2 to 0x2F8.

OneOf

Oneof selection relationships are used to associate multiple items as a set of choices which one of them can be picked. Common scenarios where this would show up is where a user would be presented with several possible choices and only one of those choices is allowed to be active at a time. This also provides a hint to a BSF parser that such a relationship could also be displayed as a radio selection.

Examples Input BSF

```
RelationshipDef
  Inconsistency = ($Var1 == $Var2) , ‘‘Error Condition’’,
                  LATE_CHECK

  Inconsistency = ($Var3 == 23) || ($Var2 == 21) , ‘‘Error!’’
    OneOf       = $Var2, $Var3
EndRelationship
```

Example ‘As Built’ BSF

```
  RelationshipDef
    Inconsistency = ($Var1 == $Var2) , ‘‘Error Condition’’,
                  LATE_CHECK

    Inconsistency = ($Var3 == 23) || ($Var2 == 21) , ‘‘Error!’’
    OneOf             = $Var2,   $Var3
  EndRelationship
```

3.8 PageDef Definitions

This section defines the format of the page definitions. A page is a viewable entity within a parsing tool, with only one page being displayed at a time. Pages may be embedded inside other pages. A page index may be provided by the tools. Each page definition defines a type of entry, a variable that binds this entry to a variable in the StructDef (previously defined,) prompt information and type of entry. There are three basic forms for entry, a Combo, which uses a named list of valid values, numeric entry (either free form or as part of a table) and text entry (single-line string without EOL characters, multi-line string with embedded EOL characters or as part of a string table.)

A Combo entry variable’s data type (defined in the StructDef) must match the data type for the list values. For example, if the variable’s data type is HEX, the list must have only HEX values, not EHEX, BIN, EBIN or DEC.

Directives may be used to encapsulate page sections, encapsulate entries within a page section and/or any portion of an entry. The directive statement expressions may use labels (DefaultId profile identifier and SKUID values) from the GlobalDataDef section, feature names (or PcdNames) from the FeatureDef section and/or variable names that have been previously defined within the StructDef section.

This is an optional section for BSF files (if omitted, then the BSF does not permit users to modify binary setting values.) Users that will not run a graphical tool to modify values must specify the $_AS_BUILT_ label in the structure definition section.

3.8.1 Page Definition Section Format

```
<PageDef>             ::    'Page' <PageName> [',' <UiName>] <eol>
                             <PageDefs>{1,}
                             <PageDef>{0,}
                             'EndPage' <eol>

<PageName>            ::=    <Quote> <Word> <Quote>

<UiName>              ::=    <Quote> <String> <Quote>

<PageDefs>            ::=    ['Title' <Title> <eol>]
                             ['TitleB' <Title> <eol>]
                             {<CefdkEntries>}

<CefdkEntries>       ::=     ['Link' <Link> <eol>]
                             ['EditNum' <NumberEntry> <eol>]
                             ['EditText' <StringEntry> <eol>]
                             ['MultiText' <TextEntry> <eol>]
                             ['Combo' <ComboBox> <eol>]
                             ['Table' <TableData> <eol>]
                             ['StringTable' <StringData> <eol>]
<Title>               ::=    <QuotedString> [<QuotedString>]{0,} [<help>]
<Link>                ::=    <ButtonText> ',' <PageRef> [<help>]
<ButtonText>          ::=    <QuotedString>
<PageRef>             ::=    {<PageName>} {<RelOrAbs>} {<Local>}
<RelOrAbs>            ::=    {'''..'''} {[''''/' <String> '''']{1,}}
<Local>               ::=    {'''.\' <String> ''''} {'''./' <String> ''''}
<help>                ::=    ',' 'Help' <HelpString>
<HelpString>          ::=    {<Strings>} {<QuotedText>}
<Strings>             ::=    <QuotedString> [<QuotedString>]{0,}
<NumberEntry>         ::=    <Variable> ',' <Prompt> ',' <NType> [<help>]
<EdkNumEntry>          ::=   <PcdName> ',' <Prompt> ',' <NumType> [<help>]
<PcdName>              ::=   '$' <TokenSpaceCName> '.' <PcdCName>
<NumType>             ::=   {'UINTN'} {'UINT8'} {'UINT16'} {'UINT32'}
                             {'UINT64'}

<Prompt>              ::=    <QuotedString>
<NType>               ::=    {'BIN'}{'DEC'} {'EBIN'} {'EHEX'} {'HEX'}
<StringEntry>         ::=    <Variable> ',' <Prompt> [<help>]
<EdkStrEntry>         ::=    <PcdName> ',' <Prompt> [<help>]
<TextEntry>           ::=    <Variable> ',' <Prompt> [<help>]
<Combo>               ::=    <Variable> ',' <Prompt> ',' <List> [<help>]
<EdkCombo>            ::=    <PcdName> ',' <Prompt> ',' <List> [<help>]
<List>                ::=    '&' <Word>
<TableData>           ::=    <PtrVar> <TableName> ',' <ColInfo> [<help>]
<ColInfo>             ::=    'Column' <ColName> ',' <IntSize> <Col2>
                             ['Column' <ColName> ',' <IntSize> <Col2>]{1,}
<IntSize>             ::=    (0-9){1,} {'bit'} {'bits'} {'byte'} {'bytes'}
<Col2>                ::=    ',' <NType> <eol>
<PtrVar>              ::=    '$' <Word>

<TableName>           ::=    <QuotedString>
<ColName>             ::=    <QuotedString>
<StringData>          ::=   <Variable> [<TableName>] ','
                             <StrInfo> [<help>]
<StrInfo>             ::=    'String' <QuotedString> <eol>
                             ['String' <QuotedString> <eol>]{1,}
```

Related Definitions

`Combo`

This specifies that the value must come from a list of valid values, rather than a range or free form input. The data type specified for the Combo's variable in the StructDef must match the data type used in the list.

`Table`

Refers to a two dimensional array of numeric values where the number entered must be in the format specified and bounded by the number of bits or bytes specified for the column.

`StringTable`

Refers to a list of strings (without new-line characters,) rather than a two dimensional array of strings.

`EditText`

A string of characters that cannot include new-line characters.

`Multi-Line Text`

A string of characters where new-line characters are permitted.

`EditNumber`

Permits entering a number in the format specified (HEX, EHEX, EBIN, BIN or DEC,) and bounded only by the number of bytes or bits defined for the variable. All numbers are considered unsigned.

`PcdName`

This is entry is a formatted entry to specify EDK II Platform Configuration Data entries. The PcdName consists of the C variable name of the Token Space GUID (a name space for PCDs) followed by a dot '.' character, then the C variable name of the PCD that represents a token number that is unique to the token space.

Example

```
/*****************************************************************
           Page Definition Section
*****************************************************************/
Page "ConditionalVariables"   // page definition
  // EditNum definitions
  EditNum $Com1, "Combo Boxes" , DEC ,
    Help "Three checks : <1, ==1, >=1)"
  EditNum $TableHelp, "Show help with Tables" , DEC , Help "Two Check (0, 1)"
EndPage               // end of page "Conditional Variables"

Page "BasicEditControls"      // page definition with sub-pages
  // TitleB definition
  TitleB "Select a type of edit control to view"

  // Link definitions
  Link "Text Controls" , "EditTextControls"
  Link "Num Controls" , "EditNumberControls"

  Page "EditTextControls" // Sub-page definition with sub-pages
    // Multi-line title definition
    Title "Edit Text Controls" "(EditText $BIOSID, string, help)"
    // EditText defintions
    EditText $EText, "Edit Text w/o Help (20)"
    EditText $ETextH , "Edit Text w/ Help (20)",
      Help "Edit text features are convienently "
      "used for editing strings."
    // Link definitions
    Link "MultiLine" , "MultiLineEditControls"
    Link "Back to Edit Controls", ".."
    Page "MultiLineEditControls"     // Sub-page definition
      // Title definition
      Title "Multi-line edit controls"
      // MultiText edit definitions
      MultiText $MLEdit , "Multi-line w/o Help (100)"
      MultiText $MLEditH , "Multi-line w/ Help (100)" ,
        Help "When saving multiline features "
          "to the settings file, all the return "
          "characters are converted to \\r\\n. "
          "In order to specify a return when "
          "modifying the settings file in a "
          "text editor, put in the characters "
          "\\r\\n and they will be converted to " "a carraige return"
      // Link definition
      Link "Back to Text Controls" , ".."
      Link ‘‘Back to Basic Edit Controls’’ , ‘‘/BasicEditControls’’
    EndPage           // end of page "MultiLineEditControls"
EndPage               // end of page "EditTextControls"

Page "EditNumberControls"     // Sub-page definition
  // EditNum definitions
  EditNum $ENumHex , "Edit Hex w/o Help (2 bytes)" , HEX
  EditNum $ENumHexH, "Edit Hex w/ Help (1 byte)", HEX,
    Help "Data should be in the form: 0x12"
  EditNum $ENumEHex , "Edit End Hex w/o Help (2 bytes)" , EHEX
  EditNum $ENumEHexH , "Edit End Hex w/ Help (1 byte)", EHEX ,
    Help "Data should be in the form: 1234h"
  EditNum $ENumDec , "Edit Dec w/o Help (2 bytes)" , DEC
  EditNum $ENumDecH , "Edit Dec w Help (1 byte)", DEC,
    Help "Data should be in the form: 1234"
  EditNum $ENumBin , "Edit Bin w/o Help (2 bytes)", BIN
  EditNum $ENumBinH , "Edit Bin w Help (1 bytes)", BIN,
    Help "Data should be in the form: 0b1001"
  EditNum $ENumEBin , "Edit EBin w/o Help (2 bytes)", EBIN
  EditNum $ENumEBinH , "Edit EBin w Help (1 bytes)", EBIN ,
    Help "Data should be in the form: 0001b"
    // Link definition
    Link "Return to Edit Controls" , ".."
  EndPage                 // end of page "EditNumberControls"
EndPage                    // end of page ‘‘BasicEditControls"
//directive -- $Com1 is equal to 1
#if $Com1 == 1
  Page "ComboBoxes_Com1_Equal_1"     // page definition
    // Combo box definitions
    Combo $C1ComboD, "Dec Combo w/o Help (2 bytes)" , &Dec
    Combo $C1ComboH, "Hex Combo w/o Help (1 byte)" , &Hex
    Combo $C1ComboE, "EndHex Combo w/o Help (2 bytes)" , &EndHex
    Combo $C1ComboB, "Bin Combo w/o Help (1 byte)" , &Bin
    Combo $ComboDH , "Dec Combo w/ Help (1 byte)" , &Dec,
      Help "Pick a dec value from the list"
    Combo $ComboHH , "Hex Combo w/ Help (1 byte)" , &Hex ,
      Help "Pick a hex value from the list"
    Combo $ComboEH , "EndHex Combo w/ Help (1 byte)" , &EndHex,
      Help "Pick an end hex value from the list"
    Combo $ComboBH , "Bin Combo w/ Help (1 byte)", &Bin,
      Help "Pick a bin value from the list"
EndPage              // end of page "ComboBoxes_Com1_Equal_1"
// directive -- $Com1 is less than 1
#elif $Com1 < 1
  Page "ComboBoxes_Com1_LessThan_1"  // page definition
    // Combo box definitions
    Combo $C2ComboD , "Dec Combo w/o Help (2 bytes)" , &Dec
    Combo $C2ComboH , "Hex Combo w/o Help (1 byte)" , &Hex
    Combo $C2ComboE , "EndHex Combo w/o Help (2 bytes)", &EndHex
    Combo $C2ComboB , "Bin Combo w/o Help (1 byte)" , &Bin
    Combo $ComboDH , "Dec Combo w/ Help (1 byte)" , &Dec,
      Help "Pick a dec value from the list"
    Combo $ComboHH , "Hex Combo w/ Help (1 byte)" , &Hex,
      Help "Pick a hex value from the list"
    Combo $ComboEH , "EndHex Combo w/ Help (1 byte)" , &EndHex,
      Help "Pick an end hex value from the list"
    Combo $ComboBH , "Bin Combo w/ Help (1 byte)" , &Bin,
      Help "Pick a bin value from the list"
  EndPage             // end of page "ComboBoxes_Com1_LessThan_1"

//directive -- $Com1 is greater than 1
#else
  Page "ComboBoxes_Com1_GreaterThan_1"        // page definition
    // Combo box definitions
    Combo $C3ComboD , "Dec Combo w/o Help (2 bytes)" , &Dec
    Combo $C3ComboH , "Hex Combo w/o Help (1 byte)" , &Hex
    Combo $C3ComboE , "EndHex Combo w/o Help (2 bytes)" , &EndHex
    Combo $C3ComboB , "Bin         Combo w/o Help (1 byte)" , &Bin
    Combo $ComboDH , "Dec Combo w/ Help (1 byte)" , &Dec,
      Help "Pick a dec value from the list"
    Combo $ComboHH , "Hex Combo w/ Help (1 byte)" , &Hex,
      Help "Pick a hex value from the list"
    Combo $ComboEH , "EndHex Combo w/ Help (1 byte)" , &EndHex,
      Help "Pick an end hex value from the list"
    Combo $ComboBH , "Bin Combo w/ Help (1 byte)" , &Bin,
      Help "Pick a bin value from the list"
  EndPage              // end of page "ComboBoxes_Com1_GreaterThan_1"
//end of directive statement
#endif


//tables with help
//directive -- TableHelp is not 1
#if !$TableHelp
  Page "StringTable"   // page definition
    // String table definition
    StringTable $STbl , String "First String"
                        String "Second String"
                        String "Third String"
  EndPage                         //end of page "StringTable"
  Page "DynamicTables" // page definition w/sub-pages
    Title "Table with Dynamic Sizes" //title definition
    // Link definitions
    Link "Byte offset" , "DynamicWithByteOffset"
    Link "Bit offset" , "DynamicWithBitOffset"
    Link "No offset" , "DynamicWithNoOffset’’
    Page "DynamicWithByteOffset" // Sub-page definition
       // EditNum definitions
       EditNum $TDynOffSize , "Table Size" , DEC
       EditNum $TDynOffPtr, "Ptr", HEX
       // Link definition
       Link ‘‘Bit offset’’ , ‘‘./DynamicWithBitOffset’’
       Link "Return" , ".."
       // Table definition
       Table $TDynOff "Dyn with byte off" , Column "Dec (1)" ,
         1 byte, DEC
         Column "Bin (1)" , 1 byte , BIN
         Column "Hex (1)" , 1 byte , HEX
         Column "EHex (1)" , 1 byte , EHEX
  EndPage              // end of page "DynamicWithByteOffset"

  Page "DynamicWithBitOffset" // page definition
    // EditNum definitions
    EditNum $TDynOffbitSize, "Table Size" , DEC
    EditNum $TDynOffbitPtr, "Ptr", HEX
    // Link definition
    Link ‘‘Byte offset’’ , ‘‘.\DynamicWithByteOffset’’
    Link "Return" , "DynamicTables"
    // Table definition
    Table $TDynOffBit "Dyn with bit off", Column "Dec (1)" ,
      1 byte, DEC
      Column "Bin (1)" , 1 byte , BIN
      Column "Hex (1)" , 1 byte , HEX
      Column "EHex (1)" , 1 byte , EHEX
  EndPage              // end of page "DynamicWithBitOffset"

  Page "DynamicWithNoOffset"  // page definition
    // EditNum definition
    EditNum $TDynSize , "Table Size" , DEC
    EditNum $TDynPtr, "Ptr", HEX
    // Link definition Link "Return" , ".."
    // Table definition
    Table $TDyn "Dyn with no off", Column "Dec (1)",
      1 byte, DEC
      Column "Bin (1)" , 1 byte , BIN
      Column "Hex (1)" , 1 byte , HEX
      Column "EHex (1)" , 1 byte , EHEX
   EndPage           //end of page "DynamicWithNoOffset"
```

```
 EndPage              //end of page "DynamicTables’’

 Page "StaticTables"   // page definition
  // Title definition
  Title "Tables with Static Sizes"
  // Link definitions
  Link "Byte offset" , "StaticWithByteOffset"
  Link "Bit offset" , "StaticWithBitOffset’’
  Link "No offset" , "StaticWithNoOffset"

  Page "StaticWithByteOffset" // Sub-page definition
    // EditNum definition
    EditNum $TStatOffPtr, "Ptr", HEX
    // Link definition Link "Return" , ".."
  // Table definition
  Table $TStatOff "Static with byte off", Column "Dec (1)"
    , 1 byte, DEC
    Column "Bin (1)" , 1 byte , BIN
    Column "Hex (1)" , 1 byte , HEX
    Column "EHex (1)" , 1 byte , EHEX
  EndPage             // end of page "StaticWithByteOffset"
  Page "StaticWithBitOffset"   // Sub-page definition
    // EditNum definition
    EditNum $TStatOffbitPtr, "Ptr", HEX
    // Link definition Link "Return" , ".."
    // Table definition
   Table $TStatOffBit "Static with bit off", Column "Dec (1)"
     , 1 byte, DEC
     Column "Bin (1)" , 1 byte , BIN
     Column "Hex (1)" , 1 byte , HEX
     Column "EHex (1)" , 1 byte , EHEX
 EndPage              //end of page "StaticWithBitOffset"
 Page "StaticWithoutOffset"   // Sub-page definition
   EditNum $TStatPtr, "Ptr", HEX      // EditNum definition
   Link "Return" , ".."               // Link definition
   // Table definition
   Table $TStat "Static with no offset",
      Column "Dec (1)" , 1 byte, DEC
      Column "Bin (1)" , 1 byte , BIN
      Column "Hex (1)" , 1 byte , HEX
      Column "EHex (1)" , 1 byte , EHEX
    EndPage  // end of page "StaticWithoutOffset"
  EndPage             // end of page "StaticTables"

//tables with help
//directive-- $TableHelp is 1
#else

  Page "StringTable"   // Page definition
    // String table definition
    StringTable $STbl ,String "First String"
      String "Second String"
      String "Third String" ,
      Help "This is a 50 byte string "
      "table with three strings."
  EndPage             // end of page ‘StringTable’

  Page "DynamicTables" // Page definition
    Title "TableWithDynamicSizes"           ; Title definition
    ; Link definitions
    Link "Byte offset" , "DynamicWithByteOffset"
    Link "Bit offset" , "DynamicWithBitOffset’’
    Link "No offset" , "DynamicWithNoOffset"

    Page "DynamicWithByteOffset" ; Sub-page definition
      // EditNum definition
      EditNum $TDynOffSize , "Table Size" , DEC
      EditNum $TDynOffPtr , "Ptr", HEX
      Link "Return" , ".."    ; Link definition
      ; Table definition
      Table $TDynOff "Dyn with byte off" , Column "Dec (1)"
        , 1 byte, DEC
        Column "Bin (1)" , 1 byte , BIN
        Column "Hex (1)" , 1 byte , HEX
        Column "EHex (1)" , 1 byte , EHEX ,
        Help "Dynamic size table with byte offset"
    EndPage // end of page ‘DynamicWithByteOffset’

Page "DynamicWithBitOffset"
  EditNum $TDynOffbitSize, "Table Size" , DEC
  EditNum $TDynOffbitPtr , "Ptr" , HEX
  Link "Return" , ".."
  Table $TDynOffBit "Dyn with bit off" , Column "Dec (1)" ,
    1 byte, DEC
    Column "Bin (1)" , 1 byte , BIN
    Column "Hex (1)" , 1 byte , HEX
    Column "EHex (1)" , 1 byte , EHEX ,
    Help "Dynamic size table with bit offset"
EndPage               // end of page ‘DynamicWithBitOffset’

Page "DynamicWithNoOffset"
  EditNum $TDynSize , "Table Size" , DEC
  EditNum $TDynPtr , "Ptr" , HEX     // EditNum definition
  Link "Return" , ".." //   Link definition
  Table $TDyn "Dyn with no off" , Column "Dec (1)" , 1 byte ,
    DEC
    Column "Bin (1)" , 1 byte , BIN
    Column "Hex (1)" , 1 byte , HEX
    Column "EHex (1)" , 1 byte , EHEX ,
    Help "Dynamic size table with no offset"
  EndPage  // end of page ‘DynamicWithNoOffset’
EndPage               // end of page ‘DynamicTables’

Page "StaticTables"
Title "Tables with Static Sizes"
Link "Byte offset" , "StaticWithByteOffset"
Link "Bit offset" , "StaticWithBitOffset"
Link "No offset" , "StaticWithoutOffset"

Page "StaticWithByteOffset" EditNum
    $TStatOffPtr, "Ptr" , HEX Link
    "Return" , ".."
    Table $TStatOff "Static with byte off" , Column "Dec (1)" ,
      1 byte, DEC
      Column "Bin (1)" , 1 byte , BIN
      Column "Hex (1)" , 1 byte , HEX
      Column "EHex (1)" , 1 byte , EHEX ,
      Help "Static size table with byte offset"
  EndPage

  Page "StaticWithBitOffset’’
    EditNum $TStatOffbitPtr , "Ptr" , HEX
    Link "Return" , "..’’
    Table $TStatOffBit "Static with bit off" , Column "Dec (1)"
```

```
      , 1 byte, DEC
      Column "Bin (1)" , 1 byte , BIN
      Column "Hex (1)" , 1 byte , HEX
      Column "EHex (1)" , 1 byte , EHEX ,
      Help "Static size table with bit offset"
  EndPage

  Page "StaticWithoutOffset’’
    EditNum $TStatPtr , "ptr" , HEX
    Link "Return" , ".."
    Table $TStat "Static with no off" , Column "Dec (1)" ,
      1 byte, DEC
      Column "Bin (1)" , 1 byte , BIN
      Column "Hex (1)" , 1 byte , HEX
      Column "EHex (1)" , 1 byte , EHEX ,
      Help "Static size table with no offset"
  EndPage
EndPage§
```