|  |  |  |
| --- | --- | --- |
| EDK II Build Data Viewer | | May 2013  Rev 1.0.1.2 |
| The EDK II Build Data Viewer tool provides a view of what files, GUIDs, and PCDs were used in a build. It parses the build log file to determine source code and INF files used in the build and the workspace directory. Additional build information can be created by using the –Y and –y switches to generate a build report. |

Contents

[EDK II Build Data Viewer user guide 2](#_Toc328858125)

[Main dialog UI information 2](#_Toc328858126)

[Tool version information 2](#_Toc328858127)

[Loading build log and report files 3](#_Toc328858128)

[Editing files 4](#_Toc328858129)

[View File/Data Types 4](#_Toc328858130)

[Search 6](#_Toc328858131)

[Search Items for each File/Data Type 6](#_Toc328858132)

[EDK2 Build Data Viewer developer design guide and notes 7](#_Toc328858133)

[Build Spec revision and location 7](#_Toc328858134)

[Registry settings 7](#_Toc328858135)

[Resource editing 7](#_Toc328858136)

[UI controls and forcing clicks of many controls to execute the same code 7](#_Toc328858137)

[Search data 8](#_Toc328858138)

[Lists and Trees for each File/Data Type, but displayed at the same location 8](#_Toc328858139)

[HTML dialogs and their .htm file 8](#_Toc328858140)

[Ensure all backslashes (‘\’) are converted to slash (‘/’) 9](#_Toc328858141)

[Loading build log file 9](#_Toc328858142)

[Loading build report file 9](#_Toc328858143)

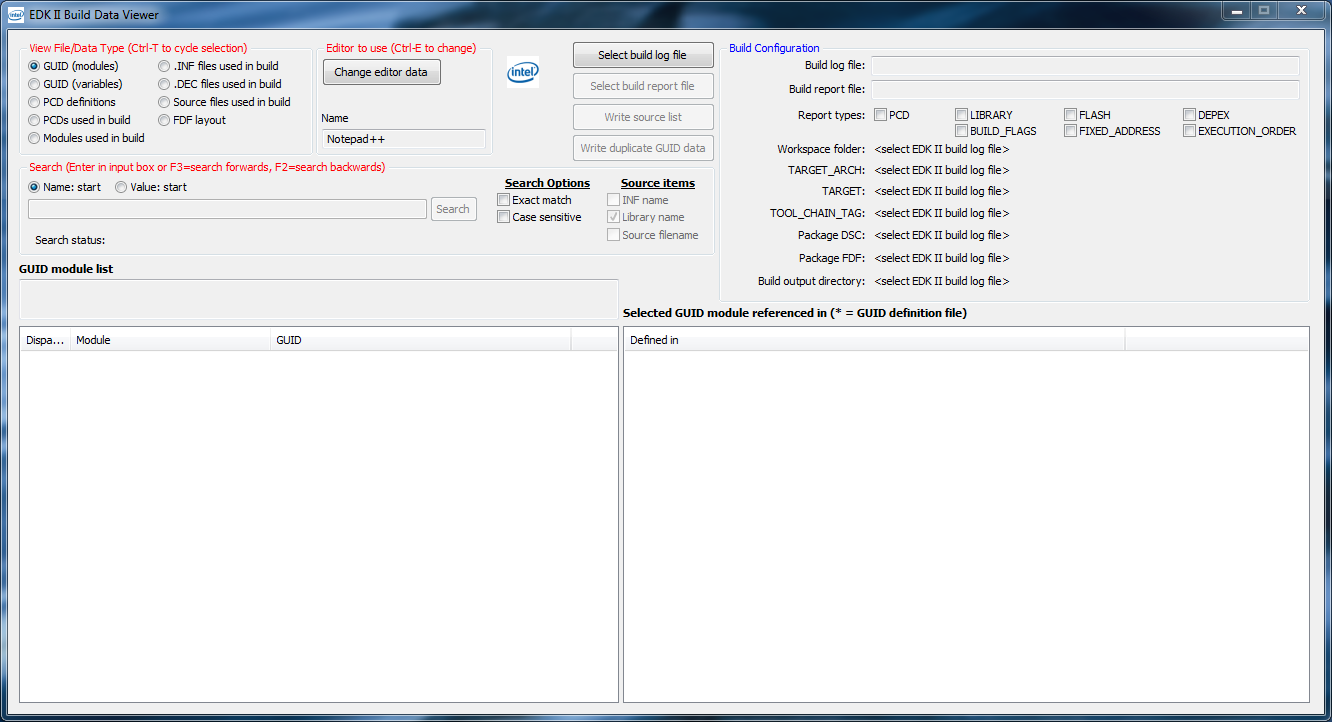
[Data types for internal structures 10](#_Toc328858144)

[Recognition 11](#_Toc328858145)

# EDK II Build Data Viewer user guide

## Main dialog UI information

When you run the tool, you are presented with a lot of UI to provide you with quick views of various build data. UI functions are grouped and arranged to easily navigate the options.

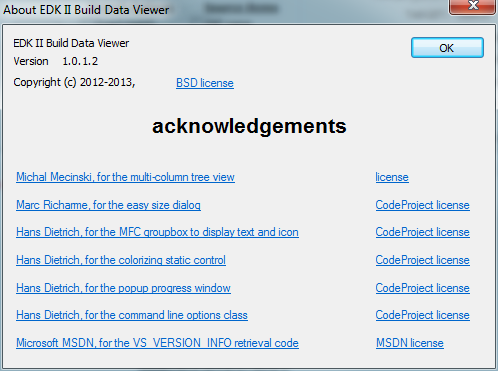


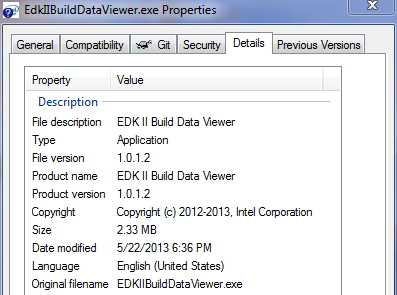
**NOTE:** The dialog has a minimum size, but can be sized wider or taller by the normal Windows resize method of grabbing a border and dragging.

|  |  |
| --- | --- |
|  | |
| The text under the **Search** group (Type Info) communicates information specific to the **File/Data Type** that is selected. | The text under the **Build Configuration** group (Type Actions) communicates actions for the **File/Data Type** selected, such as double-clicking a column to edit the file, double-click a column to automatically select and find that item in another **File/Data Type**, or to copy that item’s data to the clipboard *<currently not implemented, code is broken>*. |

## Tool version information

To find the tool version, click the Windows system icon (upper left corner) and choose About, or right-click on Edk2BuildDataViewer.exe and choose the Details tab.



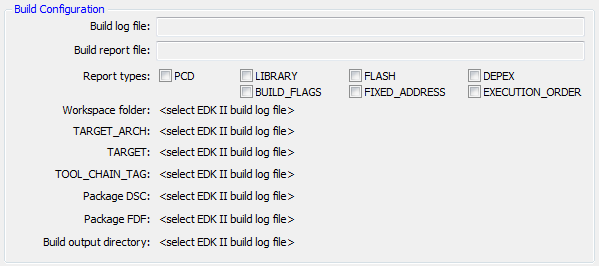


## Loading build log and report files

There are 2 build files that generate the build data that can be parsed by this tool, as specified by the Build Spec.

The build log file is generated by running build.exe, while the build report is generated by running build.exe with the additional –Y and –y switches. EdkIIBuildDataViewer asks you to select these files in the UI with the load buttons. It forces you to load the build log first, as the build report is additional data added to the build log.

Build configuration data found in the build log and report is displayed in the **Build Configuration** group.



|  |  |  |
| --- | --- | --- |
|  | On the left are the 2 build load buttons. The build log must be loaded before the build report can be loaded.  On the right the build log has been loaded, which allows the build report to be loaded. Once it is loaded, the Build Configuration tells you which –Y options it found in the build report. |  |
|  |  | |

## Write source list

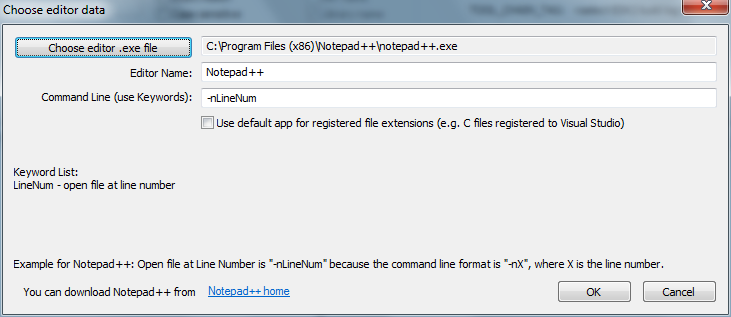
An EDKII BIOS tree may contain files that aren’t used in the build. When the build log is loaded, it parses the [Sources] section of every .INF file listed in the build log. You can then write this source list to a file. One use of the source list is as input to [doxygen](http://www.stack.nl/~dimitri/doxygen/index.html), a freeware tool that creates call graphs and extracts documentation from code, so that you are examining the exact source list used by your BIOS build. This is achieved by specifying **@INCLUDE = $(INPUT)** in the doxygen configuration file, where INPUT is an environment variable set to the path of the source list.

|  |  |  |
| --- | --- | --- |
|  | On the left is the build log load button. The build log must be loaded before the source list can be created.  On the right the build log has been loaded, which allows the source list to be created. |  |

## Editing files

|  |  |
| --- | --- |
|  | The **Editor to use** option is to allow the user to specify a desired text editor to launch when a source file is double-clicked. If your editor supports a switch to begin at a line number in the file, then you can input this data in the editor configuration dialog. Notepad++ is the default editor if it is found. Editor options are saved to the Windows registry so the data is input once. |

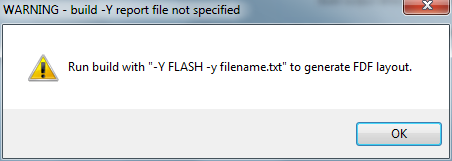
If you check **Use default app for registered file extensions**, then the selected file will be opened with the app registered for the file extension. Double-clicking a file in Windows Explorer takes the same action.



## View File/Data Types

|  |  |
| --- | --- |
|  | The build types are presented either by File Type or Data Type. These groupings are loaded with data from the build log file or the build report. For types that require data from the build report, you must load the build report after loading the build log. |

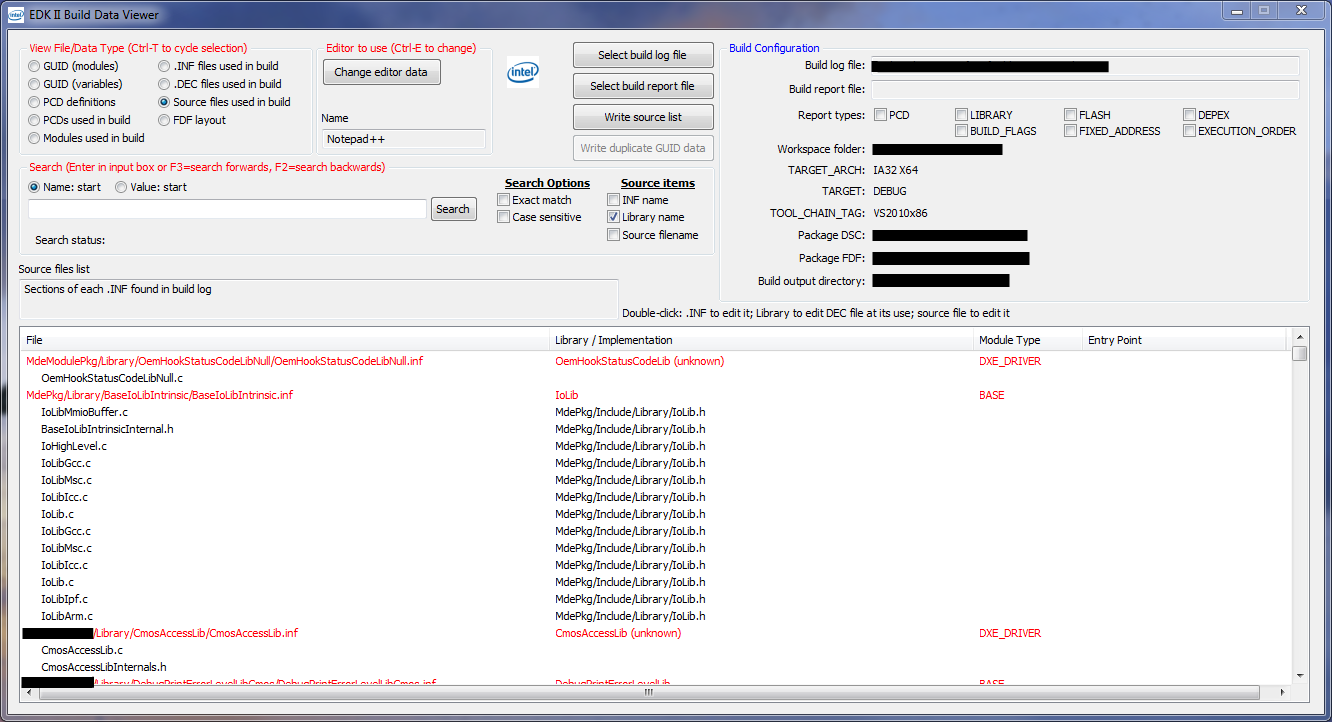
If you select a type that requires data from the build report, you will see a dialog similar to this:



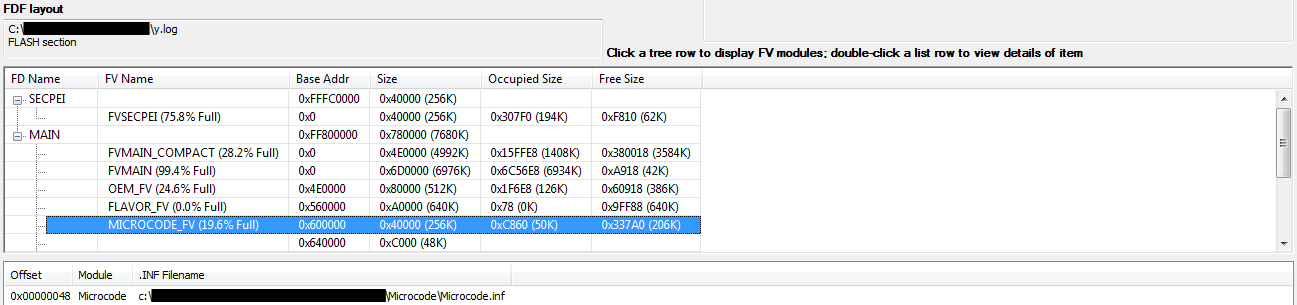
The dialog title bar informs you that you did not load the –Y report file, and the dialog body tells you how to generate the data for the selected type.

**NOTE:** If you plan on using this tool, then I suggest you run your build with the –Y switch and all of the possible options so that the data will be available to the tool. The build report takes a fair amount of time to generate, so be mindful of additional build time. I configure my build –Y to run at night when I’m not using my build system.

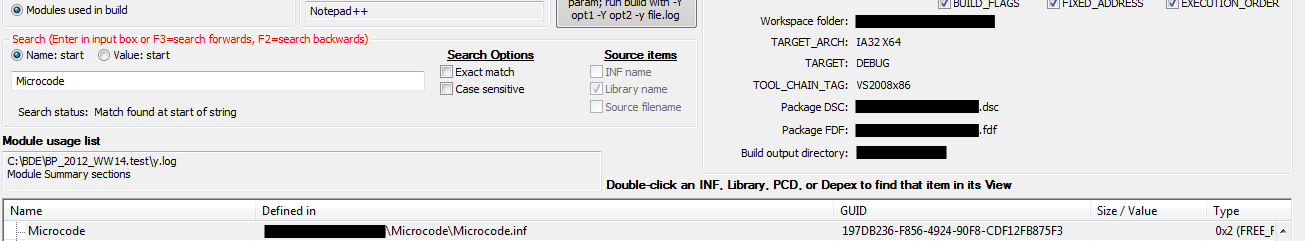
Here is an example of a build log and report loaded in Edk2BuildDataViewer.exe. The **Source files used in build** is selected by default after loading the build log. The UI area below the buttons and **Build Configuration** has changed from what was there before the build log was loaded. You will notice that this UI area (the **File/Data Type presentation** area) changes depending on which **File/Data Type** is selected. This is done because each type has data that is unique to it, and thus requires different controls (list vs tree) and different number of columns to display all data.



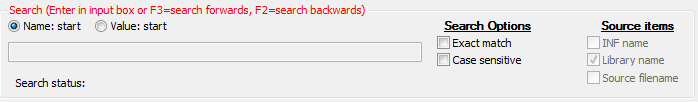
For example: the **FDF Layout** type uses a tree view to organize the data more efficiently:



When you select an FV in the FDF, data specific to that FV is displayed in the list below the tree. Note the Type Actions states that you can double-click a list row to view details about the item. When you do this, the **Modules used in build Type** is selected, and that item is selected in the Type Info UI.



## Search



Every **File/Data Type** selection can be searched. The **Name: start** field means to search for a string starting at the beginning. The **Value: start** field means to search for a value. This tool could be improved to tell the user which column will be searched for each search type.

### Search Items for each File/Data Type

|  |  |  |
| --- | --- | --- |
| **File/Data Type** | **Name: start search column #** | **Value: start search column #** |
| GUID (modules) | 1 Module | 2 GUID |
| GUID (variables) | 0 GUID | Searches as Name |
| PCD definitions | 1 PCD Name | 3 Value |
| PCDs used in build | 0 PCD Name | 3 Value |
| Modules used in build | 0 Module Name | Searches as Name |
| .INF files used in build | Not supported | Not supported |
| .DEC files used in build | 0 .DEC File | Searches as Name |
| Source files used in build | Depends on selected **Source items** | Searches as Name |
| FDF layout | Not supported | Not supported |

# EDK2 Build Data Viewer developer design guide and notes

The following design guide and notes assumes the developer is very familiar with Windows MFC development. There aren’t any exotic uses of MFC, thus a good MFC developer should be able to understand the code.

## Build Spec revision and location

This code is written per the Build File spec as of July 2012 (revision 1.22 Errata B). See build -? for details, and the Build Spec located at <http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK_II_Specifications>. As the build file formats change, so must this tool change so that it can correctly parse the files.

This tool has virtually all of the code in EdkIIBuildDataViewer.c. An obvious improvement would be to modularize the code so it’s easier to find code. There wasn’t a good reason to avoid this design point.

## Registry settings

When the main dialog constructor is executed, ReadSettingsFromRegistry() is called to read registry values.

When the main dialog destructor is executed, WriteSettingsToRegistry() is called to write registry values.

## Resource editing

* If you are editing the main dialog IDD\_EDKIIBUILDDATAVIEWER\_DIALOG in the Visual Studio Resource View, it will appear as if a lot of the edit boxes and static text items are missing. This is not the case, as they are overdrawn by the Group Box control that is associated with the custom class XGroupBox defined in XGroupBox.h. This custom Group Box control requires you to edit the .RC file and replace GROUPBOX with LTEXT. This causes Visual Studio to draw the group box outline as a static control as defined by the control dimensions.

To see the Group Box outline, you can either edit the .RC file and change the Group Box items from LTEXT to GROUPBOX (don’t forget to change them back before building), or click on the Group Box titles to cause the outline to be displayed.

* Lists and trees for each **File/Data Type** are displayed at the same location. You can’t size them in the Visual Studio Resource Editor at the same location, otherwise you wouldn’t be able to see any but the top item, which makes it difficult to manipulate the items. The 2 lists that are large are anchors, meaning they determine the upper-left corner for where all lists and trees will be displayed. The rest of the lists and trees are sized very small so you can see each individual one. **You should not move the anchor lists, as the rest of the lists and trees and positioned and sized according to the anchor list position and size in code. The non-anchor lists and trees are then hidden in code.**

## UI controls and forcing clicks of many controls to execute the same code

To avoid code duplication and to make use of code reuse for control groups executing their ON\_BN\_CLICKED handler, I modified several control group ON\_BN\_CLICKED handler in the MESSAGE\_MAP block.

* All **File/Data Type** radio buttons are associated with m\_radioView by DoDataExchange() , and all radio button IDs are associated with OnBnClickedRadioFileData() in the MESSAGE\_MAP block. This function calls UpdateData(TRUE) to set the current radio button selection in m\_radioView. m\_radioView is then compared to enums that are defined for the **File/Data Type** choices. OnBnClickedRadioFileData() uses the same code to show/hide controls based on the selected type and the control IDs in an array, and groups the Type Info and Type Action static control updates into 1 function for easier maintenance.
* All Build Report options call OnBnClickedCheckYLog() when they are clicked so that it immediately returns. This was done so the checks could be set based on options found in the Build Report. Setting the Disabled property on the checks grays it out and doesn’t look as nice, so I opted to leave them enabled but ignore the clicks.
* List and tree control double-click events are funneled into OnNMDblclkLaunchEditor(). This function uses m\_radioView to determine which control to read from, and thus what text to read based on the double-click event’s iSubItem (column) value. A parameter string is built, and then ShellExecuteEx() is called to invoke the defined editor with the parameters as defined in the Editor dialog.
* PreTranslateMessage() checks hotkey presses. The search hotkeys are defined and checked here. If a search hotkey is detected, the search function is called regardless of the selected **File/Data Type**.
* Double-clicking items in lists and trees to jump to that item in a different **File/Data Type** is done in the double-click handlers for **Modules used in build** via OnNMDblclkTreeModule() and for **FDF layout** and **GUID (modules)** via OnNmDblclkList\_ShowModuleTree(). Both of these functions set m\_radioView to select a new **File/Data Type** because the visible Type control is changing, and Search() is called to find the double-clicked item. This enables the search code to be used for multiple purposes, which is efficient code reuse.

## Search data

Search() is called when a search hotkey is detected in PreTranslateMessage(). In order to compare to a string in a list or tree control, the appropriate function for each control type must be called to retrieve the text, compare the text, and set the visible and selected control item to the match index. I encapsulated the retrieve/compare/index code for lists in ListCompareItem() and trees in TreeWndCompareItem(). Each compares based on the set search options, and return -1 if no match is found or the control’s row number for the matched item. *S*ee source code in EdkIIBuildDataViewerDlg.c Search() function.

## Lists and Trees for each File/Data Type, but displayed at the same location

OnInitDialog() positions and sizes all lists and trees according to the anchor lists, and then hides them. As each **File/Data Type** is selected, the visible controls for the current selected type are hidden by the control IDs, m\_RadioView value is updated, and then the controls for the new current selected type are shown by the control IDs. This design point allows for code reuse, placing controls at the same location while leaving it easy to see them in the Resource Editor to enable efficient screen use, and easy show/hide of controls based on a radio selection.

## HTML dialogs and their .htm file

There are 2 .htm files in the solution: EdkIIBuildDataViewerDlg.htm and ChooseEditor.htm. If you examine EdkIIBuildDataViewerDlg.h file, you will see that CEdkIIBuildDataViewerDlg class is derived from CDialog, and its IDD= statement isn’t followed by an IDH= statement. The main dialog used to be an HTML dialog, but I removed it because it presented some development problems that I can’t recall. Examination of ChooseEditor.h shows that CChooseEditor class is derived from CDHtmlDialog, and thus its IDD= statement is followed by an IDH= statement. If you examine the definition of the resource ID after IDH=, you find that the resource ID is set to ChooseEditor.htm. This is how the build process knows how to parse an HTM file associated with a CDHtmlDialog.

## Ensure all backslashes (‘\’) are converted to slash (‘/’)

Hundreds, possibly thousands, of files can comprise a BIOS build project. In order for code to be able to parse file locations to extract path and filename, I decided it was easier to replace backslashes with slash instead of searching for both characters while tokenizing file locations. You will see this code in many places

// ensure all backslashes are slashes

fileStr.Replace(\_T('\\'), \_T('/'));

## Loading build log file

OnBnClickedSelectBuildLog() is called when a build log file is loaded. I tried to put in as much error checking as possible for expectation of start and end tags, but there is only so much one can do for file formats. The following **File/Data Type** selections are populated with data from the build report:

* **GUID (variables)**
* **PCD definitions**
* **.INF files used in build**
* **.DEC files used in build**
* **Source files used in build**

The order of operations for build log read:

1. Read some build configuration data that appears at the start of the build log.
2. Read and store .INF files used in internal data structure.
3. Parse each found .INF for the following sections:
   1. [defines] to determine INF info such as GUID, Module Type, and Entry Point. If Module Type is a value that requires a LIBRARY\_CLASS, then the LIBRARY\_CLASS value is saved so the library implementation can be found in a .DEC file, which will provide the .h file for the library class.
   2. [sources] to determine source files used by the .INF
   3. [packages] to determine .DEC files used by the .INF
4. Read and parse .DSC file for .INFs used in build; mark .INFs found in step 2 with .DSC line number.
5. Call ParseDecFiles() to read and parse .DEC files:
   1. Parse GUID name/value pairs
   2. Parse PCD name/value pairs
   3. If [libraryclasses] section is found, match .INFs with LIBRARY\_CLASS to their .h implementation.

## Loading build report file

OnBnClickedSelectYLog() is called when a build report file is loaded. I tried to put in as much error checking as possible for expectation of start and end tags, but there is only so much one can do for file formats. The following **File/Data Type** selections are populated with data from the build report:

* **GUID (modules)**
* **PCDs used in build**
* **Modules used in build**
* **FDF layout**

The order of operations for build report read depends on the order of the report tags in the build report. I use boolean section flag variables named bSectionXXX and parsing flag variables named bProcessXXX, where XXX represents the section detected or item to process (e.g. FV, FD, Module). Each section and item have begin and end tags in the log file so the file parser knows when it can start and stop processing an item. A line is read from the log file, and then all parsing flags are checked; only 1 flag at a time is true. For the flag that is found to be true, the new line read is parsed according to that item’s properties.

## Data types for internal structures

Each **File/Data Type** has unique data associated with it, and thus requires its own data structure. These data structures are defined in EdkIIBuildDataViewer.h. Depending on the type, either a CArray, CStringArray, or vector data type is used to store the type’s data. I chose each data type for how easy it was to code set/get/search functionality.

## Connecting log file, data type, and UI

There are 2 build log files and 2 corresponding load file functions, multiple data structures, and multiple UI items. Some data structures are completely populated by reading a block of data from a log file; these are the easiest to follow in code. Other data structures are partially populated with data from a log file, with a remaining amount populated by searching other data structures. Some UI items are completely populated with data from a single data structure, while other UI items are populated with data from multiple data structures.

The 2 load log file functions OnBnClickedSelectBuildLog() and OnBnClickedSelectYLog() are represented by text.

Data structures and parsing steps are represented by rectangles. UI items are represented by ovals.

**-Y log FLASH section**

OnBnClickedSelectYLog()

Save FD data. If FDRegion tag found, set bProcessFV=true and bProcessFVModules=true. When FD end detected, save to vector in FDF.

FD section found, set bSectionFD=true. t\_FD structure used.

FD found, set bProcessFV=true. t\_FV structure used.

OnBnClickedSelectYLog()

Save FV data. If FVModule tag found, set bProcessFV=false and bProcessFVModules=true. When FV end detected, save to vector in FD.

FV found, set bProcessFV=true. t\_FV structure used.

OnBnClickedSelectYLog()

Save FV data. If FVModule tag found, set bProcessFV=false and bProcessFVModules=true. When FV end detected, save to vector in FD.

FVModule found, set bProcessFVModules=true. t\_FVModule structure used.

OnBnClickedSelectYLog()

Save to vector in FV.

# Acknowledgements

For reducing reinventing the wheel, or “use open source custom Windows control code when possible” …

Without the generosity of people who enjoy creating and sharing custom Windows controls, it would not be possible to write user friendly Windows apps in a timely manner. I must acknowledge the efforts of the following individuals for their excellent and easy to integrate code:

* Michal Mecinski, for the multi-column tree view

<http://www.mimec.org/components/mfc>

<http://doc.mimec.org/articles/mfc/mctree/index.html>

* Marc Richarme, for the easy size dialog

<http://www.codeproject.com/Articles/1657/EasySize-Dialog-resizing-in-no-time>

<http://www.codeproject.com/info/EULA.aspx>

* Hans Dietrich, for the MFC groupbox to display text and icon

<http://www.codeproject.com/Articles/29016/XGroupBox-an-MFC-groupbox-control-to-display-text>

<http://www.codeproject.com/info/EULA.aspx>

* Hans Dietrich, for the colorizing static control

<http://www.codeproject.com/Articles/5242/XColorStatic-a-colorizing-static-control>

<http://www.codeproject.com/info/EULA.aspx>

* Hans Dietrich, for the popup progress window

<http://www.codeproject.com/KB/miscctrl/XProgressWnd.aspx>

<http://www.codeproject.com/info/EULA.aspx>

* Hans Dietrich, for the popup progress window

<http://www.codeproject.com/Articles/1940/XGetopt-A-Unix-compatible-getopt-for-MFC-and-Win32>

<http://www.codeproject.com/info/EULA.aspx>

* Microsoft MSDN, for the VS\_VERSION\_INFO retrieval code

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms646985%28v=vs.85%29.aspx>

<http://msdn.microsoft.com/en-us/cc300389.aspx#D>