EFI Developer Kit (EDK)

Getting Started Guide

Version 0.4 (draft)

December 30th, 2004

Revision History

Revision	Revision History	Date
0..4	Had to start some where.	12/30/04

Contents

Figures

Tables

1 Introduction

1.1 Summary

This document provides detailed instructions using the EFI Developer kit (hereafter referred to as "EDK").

It is intended to be a reference for those people just starting EDK development. However, it also provides details on some of the more esoteric tasks relating to development and debug.

1.2 Overview

This document does not try to explain what the EDK is all about. For that information you should look at the FAQ and other material at www.TianoCore.org. This document is about downloading the code, building it and running it.

1.3 EDK Build Tips

1.3.1 NT32 - Edk\Sample\Platform\Nt32\Build

The NT32 build tip is the Framework (Tiano) ported to a platform that is a Windows application. In essence the platform specific code abstracts Windows* APIs. This is a powerful environment for debugging and exploring the core parts of Tiano as you can walk through the code with a source level debugger. In this document we will describe how to use Microsoft® Visual Studio® to build and debug the EDK.

1.3.2 IPF - Edk\Sample\Platform\IPF\Build

The IPF build tip exists to allow the common components to build compiled with a 64-bit compiler. There is not currently enough platform code or any emulator code in the EDK project for an Itanium™ Processor Family build tip. This build tip is still useful since just compiling with a 64-bit compiler helps find non portable code. It also ensures that any assembly language components have a corresponding Itanium build option. It also helps ensure that non portable extensions such as `_asm {}` are not used in common code.

We expect to add build tips for other processor architectures in the future.

1.4 Glossary

The following table defines several terms that are used throughout this document.

Table 1-1. Definitions of Terms

Term	Description
ASL	ACPI Source Language.
BDS	Boot Device Selection.
Build tip	An EDK build target whose build results in one or more firmware volumes.
CIS	Core Interface Specification.
Component	An EDK group of source files that provide a particular firmware function, and are typically located in the same subdirectory under the EDK_SOURCE directory structure. A component's INF file specifies the source files making up the component, and multiple components are used to build the final firmware volume.
CSM	Compatibility Support Module.
DDK	Driver Development Kit.
DLL	Dynamic-link library.
DSC	Description file.
DXE	Driver Execution Environment.
DXE CIS	*Intel® Platform Innovation EDK for EFI Driver Execution Environment Core Interface Specification.*
EBC	EFI Byte Code.
EDK	EFI Developer Kit
EFI	Extensible Firmware Interface.
EDK_SOURCE	An environmental variable that specifies the base directory where the EFI source files are located. Typically this directory will be C:\TianoCore\Edk.
FFS	Firmware File System.
FV	Firmware volume.
GUID	Globally Unique Identifier.
HOB	Hand-Off Block.
IA-32	32-bit Intel® architecture.
IPL	Initial Program Load.
MASM	Microsoft* Macro Assembler.
PE	Portable Executable.
PEI	Pre-EFI Initialization.
PEI CIS	*Intel® Platform Innovation EDK for EFI Pre-EFI Initialization Core Interface Specification.*
PEIM	Pre-EFI Initialization Module.
PPI	PEIM-to-PEIM Interface.
SEC	Security.
SMM	System Management Mode.

1.5 Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

1.5.1 Pseudo-Code Conventions

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

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

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

1.5.2 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.
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.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
`Plain Monospace`	In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

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

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

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

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

2 Development System Setup

This section describes the steps that are necessary to initially set up the local system in preparation for building a platform (build tip) in the EDK source tree.

In general we tried to make the project as compiler neutral as possible. We would like to add support for more compiler types and also support a Linux hosted development environment in the future, but we need your help.

2.1 Install Tools

Several Microsoft* tools are used to build the EDK source tree. The following tools must be installed on the development system.

- Microsoft Windows* 2000 or Microsoft Windows XP™ operating system
- For 32-bit Intel® architecture (IA-32) platform development:
  - Microsoft® Visual Studio® .NET 2003 Enterprise (7.1)
- For Itanium® processor family development:
  - Microsoft Windows Server* 2003 Driver Development Kit (DDK), build 3790

The EDK supports an **`EDK_TOOLS_PATH`** that can be used to optionally point to the location of the development tools. In most of this document we assume this tools path is not set. The **`EDK_TOOLS_PATH`** is very useful if you have your development tools checked into your source base. Since we are just getting started we will assume you just have the tools installed in the default locations.

3 Quick Start Guide

This quick start guide assumes you have a copy of Microsoft® Visual Studio® .NET 2003 Enterprise (7.1) installed on your system.

The first step is down load the source code from www.TianoCore.org. At this time you need to register and get a log in to have access the source. Don't worry every one is welcome to join.

Directions for downloading source (using link):

- Log on to www.TianoCore.org.
- Once logged in, navigate to the following URL; http://edk.tianocore.org/servlets/ProjectDocumentList?folderID=5&expandFolder=5&folderID=0. You will be taken to the EDK Documents & files: Development Snapshot page.
- Under Development Snapshots pick the one with the latest date and click on it. It's a ZIP file so it should automatically launch a decompress program (I used WinZip).
- Extract the Zip file to a location on your system. For this example I extracted to C:\TianoCore.

Directions for downloading source (using web site):

- Click the Projects tab near the upper right left hand corner of the page.
- Click on edk in Name column of the Projects folder
- Under Projects >> edk there is a Project tools box (upper left of the page). In this box click on – Documents & files.
- Under edk Documents & Files click on Releases. Two subdirectories should show up: Development Snapshots and Official Releases.
- Click Development Snapshots.
- Under Development Snapshots pick the one with the latest date and click on it. It's a ZIP file so it should automatically launch a decompress program (I used WinZip).
- Extract the Zip file to a location on your system. For this example I extracted to C:\TianoCore.

For those of you who are more risk averse you try an official release from the URL: http://edk.tianocore.org/servlets/ProjectDocumentList?folderID=6&expandFolder=6&folderID=5.

In this example we will assume the zip file of the source was extracted to C:\TianoCore. You can choose any location you like.

3.1 Running the Nt32 Emulation

The makefile for the **`Nt32`** emulation build tip has a run target that can be used to run the **`Nt32`** emulation. The Nt32 build tip is located at Edk\Sample\Platform\Nt32\Build. However, before running the emulation, the following steps must be performed. Type in the commands in **`this`** highlight, the rest of the text is just explanation.

- Launch a "Visual Studio .Net 2003 Command Prompt". The "Visual Studio .Net 2003 Command Prompt" can be launched from [Start]->[All Programs]->[Microsoft Visual

Studio .Net 2003]->[Visual Studio .Net Tools]->[Visual Studio .Net 2003 Command Prompt].

  - If you don't have this option on your system you can do this manually by performing the following steps.
  - Open Windows command line shell (**`cmd.exe`**) and perform the following operations. Click [Start]->[Run] and type in **`cmd.exe`**. You can also usually find it under [Start]->[All Programs]->[Accessories]->[Command Prompt].
  - Before you can build the EDK you need to make sure the VC++ compiler is in the path.
    - VC++ has a batch file called vcvars32.bat that you can execute
    - A lot of people just make sure the VC++ tools are in their path
      - Right click on My Computer icon
      - Select Properties
      - Select the Advanced tab
      - Click the Environment Variables button near the bottom of the tab
      - Under System variables double click Path. You may have to scroll down to find Path
      - Edit "Variable value:" to include the path for VC++
        - C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin; is the value on my system.

- **`Set EDK_SOURCE=C:\TianoCore\Edk`** or where ever you extracted the zip. EDK_SOURCE is used by the build infrastructure to know where the source tree is located. Make sure you don't put spaces around the =.
- **`cd \TianoCore\Edk\Sample\Platform\Nt32\Build`**
- Type **`nmake`** at the command line to build the source.
- **`System.cmd`** runs a shell script to set environment variables to configure the environment for the emulator. This allows you to add are removed emulated devices. This is optional as defaults will be used if the system.cmd shell command is not executed.
- **`nmake run`**. Execute the emulator. This command should result in a Boot Device Selection (BDS) prompt to select the boot device. If no action is taken the boot prompt will time out and launch the **`EFI Shell`**. No action should put you at the EFI Shell prompt. To exit the simulation, enter **`reset`.** To exit the shell type `exit` and you will get back to the BDS prompt.

You can type help at the EFI Shell prompt to get help on the EFI shell commands. For more info see EFI Shell documentation [link TBD].

3.1.1 Modify System.cmd to match your configuration

Note: This is an optional step. The build will use the default settings for these variables if this step is not performed.

Here is a list of the Windows environment variables that can be used to configure the NT32 environment.

- **`EFI_WIN_NT_SERIAL`**[1]– Maps a physical serial port through Windows into the emulator. Windows reserves the following names COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, and COM9 for the serial ports.
- **`EFI_WIN_NT_UGA`**[1] – Creates a Windows window that emulates an EFI UGA device. Each element represents the name of the window.
- **`EFI_WIN_NT_FILE_SYSTEM`**[1] – Maps a Windows directory path into the emulator. Multiple paths are supported. It's possible to access the named directory and all it's subdirectories from the emulator.
- **`EFI_FIRMWARE_VOLUMES`**[1] – Windows path for a Firmware Device. The default is the output from the nmake command. Multiple Firmware Devices can be mapped into the emulator.
- **`EFI_MEMORY_SIZE`**[1] – List of decimal numbers that represents megabytes of memory in the emulator. It's not possible to specify the address of the memory as the emulator must do this dynamically.
- **`EFI_WIN_NT_CONSOLE`** – Maps a Windows text based console into the emulator. This is option has been made obsolete by **`EFI_WIN_NT_UGA`**.
- **`EFI_BOOT_MODE`** – Decimal representation for the Tiano boot mode
- **`EFI_CPU_MODEL`** – String for the CPU model. Used in a UI.
- **`EFI_CPU_SPEED`** – String for the CPU speed. Used in a UI.
- **`EFI_WIN_NT_VIRTUAL_DISKS`**[1] – Maps a file into the emulator as disk device.
    - <F | R><O | W>;<block count>;<block size>[!...]
        - Fixed – Fixed disk, like a hard disk
        - Removable – Removable disk like a floppy or CD-ROM
        - Read Only – Write protected device
        - Read Write – Read/Write device
        - <block count> - decimal number of blocks a device supports.
        - <block size> - decimal number of bytes per block. Disks are usually 512 and a CD-ROM 2048.
- **`EFI_WIN_NT_PHYSICAL_DISKS`**[1] – Maps a physical device managed by Windows into the emulator. You can directly map CD-ROMs, hard drives, floppies, and USB disks

directly into the emulator. Care should be taken with this option as you could damage you system if you are not careful.

  - <drive letter>:<F | R><O | W>;<block count>;<block size>[!...]

[1]These options support the '!' as a separator. The '!' allows multiple instances of a feature to be supported with a single variable.

You can look at the comments in the system.cmd file to see example and current settings for all the variables.

3.1.2 The Easy Way to Build from the Command Shell

Lots of developers make cmd.exe an Icon on their desktop. It's possible to make the icon run a shell script before it opens and it's very convenient to have this shell script set up the environment variables and paths you need to do an EDK build.

Go to the Command Prompt icon right click and select Properties. From the Properties windows select the Shortcut tab. Under Target: in the short cut tab type "%SystemRoot%\system32\cmd.exe /kc:\TianoCore\build.cmd c:\TianoCore\Edk NT32" (no quotes). It's also a good idea to give the icon on your desktop a descriptive name. Now when you click on the icon it opens a command shell window and starts up a build. See 3.2.1 for a listing on Build.cmd.

3.2 Using VC++ IDE

It's very convenient to build the EDK from within Microsoft Visual Studio as you can double click on build errors in the build window and the IDE will open the file with the cursor on the error.

The following section was tested on Microsoft Development Environment 2003 Version 7.1.3088. It is also assumed that the EDK has been placed in C:\TianoCore\Edk. The following instructions allow you to build the EDK NT32 build target from VC++ that is located at C:\TianoCore\Edk\Sample\Platform\Nt32\Build.

The first step is to identify a place where VC++ can store its files. This can be anywhere, but we recommend you don't use the EDK source tree. For this example we will use C:\TianoCore\VC++. The following steps assume a batch file called Build.cmd that is located at C:\TianoCore\ Build.cmd (see 3.2.1 for the contents of the batch file):

- Start Visual Studio.
- If a solution is opened, select [File]->[Close Solution] to close the current solution.
- Select [File]->[New]->[Project], which brings up the "New Project" dialog. In this dialog:
  - In the "Project Types:" pane, select the [Visual C++ Projects] folder
  - In the "Templates:" pane, select the [Makefile Project] icon
  - In the "Name:" text box, type **`NT32`**
  - In the "Location:" textbox, type **`C:\TianoCore\VC++`**

- Click [OK]

- The “Makefile Application Wizard – NT32” dialog then pops up. In this dialog:
  - Select [Application Settings]
  - For "Build command line:", enter **`c:\TianoCore\build.cmd c:\TianoCore\Edk NT32`**
  - For “Output:”, enter **`c:\TianoCore\Edk\Sample\Platform\Nt32\Build\Ia32\SecMain.exe`**. Replace any existing entries.
  - For "Clean commands:", enter **`c:\TianoCore\build.cmd c:\TianoCore\Edk NT32 clean`**
  - For "Rebuild command line:", enter **`c:\TianoCore\build.cmd c:\TianoCore\Edk NT32`**
    - Note: The “Output:” box will default to NT32.exe and this will not work. Theoretically you should be able to type **`c:\TianoCore\build.cmd c:\TianoCore\Edk NT32 clean`**, but Visual Studio requires that this entry be an executable image. If you know how to solve this problem please let us know!
  - Click [Finish]

- This should bring you back to the main Visual Studio interface. In the "Solution Explorer" pane, right-click the [NT32] folder and select [Build]. This should cause the NT32 project to be built. If something fails, view the contents of the [Output] tabbed page and correct the problem. The most common problem is incorrect build commands. To diagnose, in the "Solution Explorer" pane, right-click the [NT32] folder and select [Properties]. Select [Configuration Properties] folder, then the [Nmake] folder and check the build command settings are correct. Note that "NT32" is case-sensitive.
  - Depending on your current configuration build icons that may be located on the tool bar. You can also build from the build menu
  - A solution is just a set of projects associated together.

- If a build fails, then you can double-click on the error message (or press F8 which sometimes works) in the [Output] tabbed page to jump to the source line where the error occurred.

- Before running the emulator, you will need to set the working directory or execution will fail..
  - On the Solution Explorer right click on NT32 and select Properties
  - In the NT32 Property Pages pop up go to Configuration Properties and select Debugging
  - Under Action click on Working Directory. Enter **`C:\TianoCore\Edk\Sample\Platform\Nt32\Build\IA32`**

    - Select OK to exit and complete the changes

- Press F5 (or [Debug]->[Start]) to run the emulator.
- The only way to support the system.cmd shell command using VC++ is to execute system.cmd in a command prompt and then launch VC++ from that command prompt. **`devenv.exe`** will launch VC++ IDE from a command prompt window.

3.2.1 Build.cmd Source

```
REM @echo off
REM ***************************************************************************
REM
REM    File: Build.cmd
REM
REM    Usage:  Build.cmd [EDK directory] [Platform name]
REM
REM    Abstract:
REM      This batch file is used to initiate an EDK build from within Visual
REM      Studio. To use it, create a Visual Studio makefile project, and
REM      for the project build commands invoke this batch file with the
REM      first argument being the path to the EDK source tree, and the
REM      second argument the platform to build (typically "NT32").
REM
REM  ***************************************************************************

REM Only used for error messages from this file
SET THIS_FILE=c:\TianoCore\build.cmd

if .%2. == .. goto Usage
set EDK_SOURCE=%1

REM Call the batch file that the Visual Studio install created to set environmental
REM variables if required.

REM call "C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools\vsvars32.bat"

if %2 == NT32 goto Build_NT32

REM  This error occurs when your Visual Studio project's build command is incorrect.
REM  In Visual Studio, right-click the project folder (typically NT32) in the Solution
REM  Explorer pane and then select [Properties], then [Configuration Properties],
REM  then [NMake] to check your commands. The first argument to this command file
REM  is the EDK directory, and the second is the platform to build (case sensitive).
REM
echo.
echo %THIS_FILE%(36) : error 0000 : %2 : platform not supported by this batch file
echo.
goto Usage

:Build_NT32
REM This changes to the right driver letter
%EDK_SOURCE:~0,2%
cd %EDK_SOURCE%\Sample\Platform\Nt32\build
nmake %3 %4 %5 %6
goto Done


:Usage
echo.
echo Usage: Build [EDK directory] [NT32 more...]
echo.
goto Done

:Done

REM ***  END Build.cmd  ***
```

3.3 Debug Tips

There are few tricks we would like to share with you to get started playing with the EDK. Microsoft Visual Studio (VC++ IDE) is a very powerful tool for doing software development and this document assumes you will learn how to use it from some other source.

- When the NT32 environment is running you can attach the debugger to the running process. Select [Tools] and from the pull down menu select [Debug Processes…]. The Processes dialog box will pop up. In the Available Processes window find the process called SecMain.exe (the Title is commonly UGA Window 2") and double click it. If you get a dialog box click O.K.
    - Note: There is more than one thread running so you may have go to [Debug]->[Widows]->[Threads]. The main thread of the emulator is running at priority Highest. The other threads are for child threads of the emulation, for example each UGA window is supported via a thread.
- Insert the macro **`EFI_BREAKPOINT ();`** in your code. If you do an nmake run when the code hits the **`EFI_BREAKPOINT ()`** macro it will launch the VC++ IDE and open the file and be sitting right on the breakpoint. You can do standard source level debugging from this point.
    - This is a really good way to watch your code run.
- Use the **`ASSERT (FALSE)`** macro. The argument to ASSERT is a Boolean expression and if it evaluates to FALSE the ASSERT is triggered. ASSERT is very useful for finding the system in a corrupted state or to catch some one doing something you know the code will not handle. The ASSERT macro is only included in a debug build.
    - The Standard Error Window (commonly the EDK shell window where you typed nmake run) will have a print out of filename, line number, and contents of the expression. The following is an example
        - ASSERT c:\TianoCore\Edk\Foundation\Core\Dxe\Event\Event.c(247): ((BOOLEAN) 0 == 1)
        - A breakpoint will also be triggered.
- Use the DEBUG () macro to print out information to Standard Error. The Debug macro supports different error levels that can be set dynamically. Errors of type EFI_D_ERROR are always displayed so this is a simple debug print for debug builds. The simple version is:

```
DEBUG ((EFI_D_ERROR, "String to print\n"));
```

- Getting rid of the annoying question every time you hit a break point or unhandled exception if you code is native or not.
    - Go to the Solution Explorer and right click on the project name (NT32 in our example)
    - Under Configuration Properties select Debugging
    - Under Debugging pain in the Debuggers section click on Debugger Type until it shows Native Only

- Hit OK to exit.

4 Building the EDK

This section describes the unique steps that are necessary to set up and build the EDK **Nt32** platform. There are other platforms available, but their setup instructions are not described here; the steps are expected to be similar, however.

4.1 Build Configuration Files

One important feature of the EDK source tree is that three configuration files primarily are used to describe the local system for building the various build tips. Two of these files are located in the $(**EDK_SOURCE)\Sample** directory, while one is in the build tip. Table 4-1 lists the configuration files that can be used. How these files get used is described in a later section.

Table 4-1. Build Configuration Files

Type of Configuration File	Description
PlatformTools.env	Defines platform-specific requirements. It's located in the platform build directory.
$(EDK_SOURCE)\Sample\CommonTools.env	Defines generic build options and common definitions.
$(EDK_SOURCE)\Sample\LocalTools.env	This file is expected to be customized by each user. This file defines the paths to the build tools for each build tip.

4.2 Environmental Variables

4.2.1 EDK Platform Builds

The only environmental variable that must be defined for building the various EDK build tips is **EDK_SOURCE**. **EDK_SOURCE** must be defined to point to the base of the EDK source tree, (e.g. **set EDK_SOURCE=C:\TianoCore\Edk**).

If no **EDK_TOOLS_PATH** is set the compiler and other build tools are executed using the path variable. The **EDK_TOOLS_PATH** is used if you have all the binary build tools in a single directory path. Having the build tools grouped under a specific path is common practice if the build tools are checked into source control.

4.2.2 Microsoft Tools Environment Variables

In order to build any of the EDK platform tips, there are specific environment variables that must be set for Visual C++ to work properly. **VSVARS32.BAT** must be run for the C compiler to build the selected tip properly. If the Visual C++ installation was not allowed to update environmental variables, an EDK build may terminate because the C compiler is not in the path. If this scenario occurs, then **VSVARS32.BAT**, which was created when Visual C++ was installed, must be manually run to set up environmental variables for using Visual C++. This file is typically located in the **`\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools`** directory.

See section 3-1 for an example of how to set a global path variable that includes the Visual C++ tools so you do not need to run **`VSVARS32.BAT`**.

4.3 Typical Build Flow

This section provides a brief flow of how a build progresses. Many EDK-specific build tools are used when building a firmware volume. However, details of their functionality are beyond the scope of this document. From a higher perspective, a build progresses as indicated in the following subsections.

4.3.1 Build Overview

The build is primarily driven using the platform makefile and build description (DSC) file. For example, in the **`Nt32`** emulation build tip, these files would be the following:

- **`$(EDK_SOURCE)\Sample\Platform\Nt32\Build\Makefile`**
- **`$(EDK_SOURCE)\Sample\Platform\Nt32\Build\Nt32.dsc`**

The makefile will typically call another makefile to build the EDK build tools and then call the **`ProcessDsc`** tool to process the platform DSC file. The platform DSC file will **`#include`** other common DSC files, which are listed in Table 4-2.

Table 4-2. DSC Files Included by Platform DSC File

File	Description
$(EDK_SOURCE)\Sample\Platform\Common.dsc	A common file that contains the build description pieces that are independent of platform or processor architecture.
$(EDK_SOURCE)\Sample\Platform\CommonIa32.dsc	Processor-specific build description files for IA-32 processors.
$(EDK_SOURCE)\Sample\Platform\CommonIpf.dsc	Processor-specific build description files for the Itanium processor family.

The following is the output of **`ProcessDsc`**:

- A makefile for each component and library that is being built
- An additional **`makefile.out`** that will call each of the component and library makefiles that it created

4.3.2 Invoke nmake

To begin the build process, invoke the **`nmake`** utility, which by default parses the default makefile **`makefile`**. This step is the only manual step that is required to complete a build.

4.3.3 Include PlatformTools.env

The makefile first includes the platform's **`PlatformTools.env`** file, which includes **`$(EDK_SOURCE)\CommonTools.env`**. This **`CommonTools.env`** file in turn includes **`$(EDK_SOURCE)\LocalTools.env`**. The functionality of these tools was described in section 4.1. Once these tools have been parsed, the paths to the build tools and default build options have been defined and can be used to compile and link the source files that make up the EDK source tree. Control then returns to the main makefile.

4.3.4 Build the "build_tools" Target

The next target to be built by the makefile is **`build_tools`**. The source to these EDK-specific tools is included with the EDK source tree and is located under the following directory:

```
$(EDK_SOURCE)\Sample\Tools
```

4.3.5 Build the "makefiles" Target

The `makefiles` target runs the `ProcessDsc` utility to parse the platform DSC file and generate the following:

- An output build tree (where binaries are built)
- The numerous component makefiles
- `Makefile.out` that calls all the component makefiles

Via `#include` statements, the DSC file is a concatenation of the following:

- Common build descriptions
- Processor architecture common build descriptions (one of `$(EDK_SOURCE)\Platform\CommonIpf.dsc` or `$(EDK_SOURCE)\Platform\CommonIa32.dsc`)
- Platform-specific build descriptions

4.3.6 Build the "builds" Target

The `ProcessDsc` utility generates a makefile for each component that comprises the build tip. It also creates another makefile called `makefile.out`, which calls each of these individual component makefiles. The `builds` target in the main build tip's makefile is expected to recursively call `nmake` for this makefile, which builds the components.

4.3.7 Build the "fds" Target

Once the individual components have been built, the target `fds` can invoke `makefile.out` to package the individual files into firmware device images. In the simplest case, this step would complete the build process for a build tip.

4.3.8 Custom Build Steps

It is often required to add custom build steps at different points in the build process. This addition is acceptable and is typically accomplished via changes to the main build tip makefile. However, whenever possible, custom build steps should be avoided.

4.4 Build Inputs and Outputs

The preceding sections described the flow of the build process chronologically. Figure 4-1 provides a very high-level pictorial overview of the build process for the **Nt32** emulation build tip.

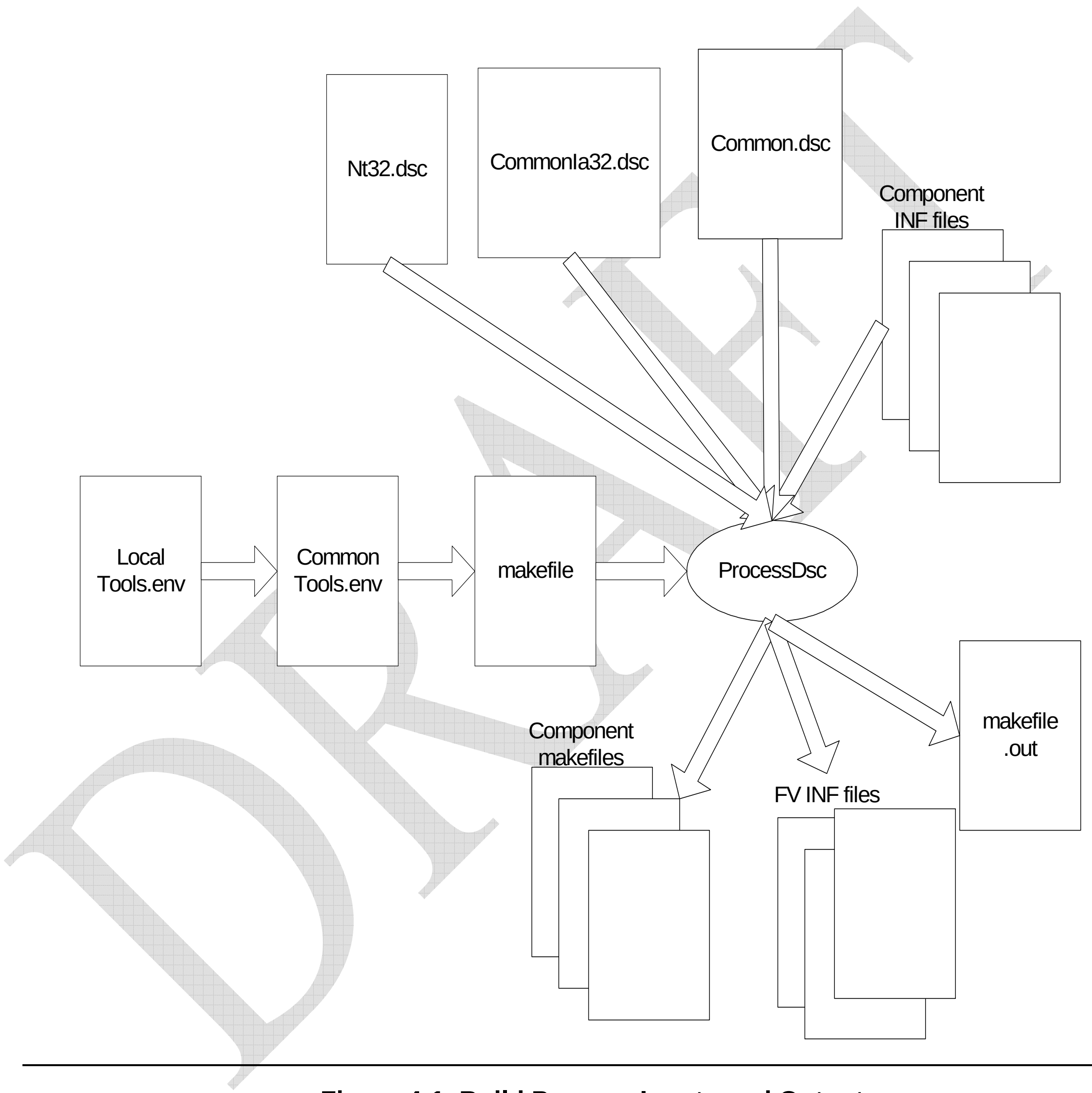

Figure 4-1. Build Process Inputs and Outputs

In essence, the makefile includes the two tool configuration files and then invokes the **ProcessDsc** build utility. This utility then takes as input the build tip description file **Nt32.dsc**, which specifies a list of component information files that are also parsed by the utility. **ProcessDsc** then creates the following:

- A makefile for each component INF file
- A **makefile.out** that calls **nmake** for each component makefile
- One or more firmware volume INF files that are used by the **GenFvImage** utility to build the desired firmware volume(s)

4.5 Build Dependencies

For the most part, the EDK build process makes use of incremental builds. As a result, if a build is invoked immediately after a previous build completes, the second build performs minimal compiling and linking. The **MakeDeps** utility is used to provide this functionality by creating source dependency files that can be included by the component makefiles. **MakeDeps** scans source files for included files and emits dependencies in a form that is compatible with makefiles. For example, if source file **foo.c** includes **foo1.h** and **foo1.h** further includes **foo2.h**, then **MakeDeps** would emit the following after processing **foo.c**:

```
foo.obj : foo.c
foo.obj : foo1.h
foo.obj : foo2.h
```

A separate dependency file should be generated for each source file.

4.5.1 Build Dependency Limitations

There are some limitations to the dependency support in the EDK build process. EDK developers need to be aware of these limitations to prevent debugging of build "problems" that may be encountered. These limitations include the following:

- To reduce build times, the dependency include files are generated only the first time a build is run on a build tip, or if the corresponding source file has changed. As a result, if **foo1.h** above was modified to include **foo3.h**, the dependency would be inaccurate and **foo.c** would not be recompiled if **foo3.h** changed.
- There are no dependencies on the build tools themselves. As a result, if a build tool is being developed, the developer must be aware of the affected built files and manually delete them such that they get rebuilt as the tool changes are made.

MakeDeps will typically be set up to ignore when include files are not found. As a result, if a broken built tip is checked out and built, **MakeDeps** may generate incomplete include dependency files if the paths to include files are insufficient to locate all the **#include** files.

4.6 Common Build Issues

Some of the most common build issues with package file generation include the following:

- A build stops with an error message similar to the following:

  ```
  "NMAKE : fatal error U1073: don't know how to make
  'C:\efi_path\build_path\MyNewLibrary.lib'"
  ```

  This error is usually caused by the component INF for **`MyNewLibrary`** not being added to the list of **`[libraries]`** in the DSC file. This omission causes the link to fail for components that call out the library in their INF files.
- **`ProcessDsc`** fails to find **`[package.xxx.yyy]`** for a component. For this error, do the following:
  - Make sure that section **`[package.xxx.yyy]`** exists in the platform-specific generated DSC file.
  - Examine the component INF file to ensure that **`COMPONENT_TYPE=xxx`**.
  - Examine the platform DSC file for the component in question to ensure that **`PACKAGE=yyy`** for that component.

  `PACKAGE` definitions on the component line take precedence over global **`PACKAGE`** definitions. If there is a problem in the generated DSC file, it can be traced to the originating DSC file by parsing backwards until a file header (containing the copyright and file name) is found.
- A build will not run. This error is typically caused by using a default package file for a component that requires a special package file. Typical components that require special package file formats include the following:
  - **`PeiCore`**
  - **`DxeMain`**
  - *A priori* lists
  - ACPI tables
- **`GenFfsFile`** results in a build error. This error typically occurs because **`PACKAGE_FILENAME`** was not passed through to the component makefile via the **`[makefile.common]`** section in the platform DSC file. If this scenario is not the cause, then verify that a backward slash "**`\`**" appears before the opening bracket "**`[.]`**" line in the package section for the component. Otherwise the package file will simply contain "**`PACKAGE.INF`**" because **`ProcessDsc`** will think the "**`[.]`**" is the start of a new section in the file.

5 Miscellaneous Operations

This section describes how to do various other common operations that do not fit in any other section or documents.

5.1 Setting up the Console

The Boot Maintenance Manager utility allows the user to specify the devices that are used for console output, input, and error (console redirection). By default, the console output and input are connected to the emulated UGA windows. Error output is sent to the shell command window. The utility allows other COM ports and/or PCI devices (video cards, USB and PS/2*) to be selected for output, input, and error. Multiple devices may be selected for console redirection. To invoke the utility, press the space bar when the system is first powered on and displays a start-up progress bar. The EDK's FrontPage user interface is displayed. Select the utility from the menu that is displayed.

5.2 EFI Shell Drive Mapping

The EFI Shell supports fixed file system mapping. The mapping order will never change as long as the configuration of the storage devices is unchanged (no storage devices are added or removed and no storage devices in the system are changed or reconfigured). If a change occurs, the file system mapping may change.

The file system mapping may be discontinuous because some mappings may be reserved for the removable storage devices or unrecognized file systems.

5.3 Cleaning a Build Tip

The makefiles for all the major build tips include a clean target that completely removes the binaries from previous builds. So, to completely remove all built binaries, simply enter **`nmake clean`** from the build tip directory.

5.4 Adding a New Component to a Build Tip

One of the most common developer operations is to add a component to a build tip. Adding a component can be accomplished by performing the following steps:

1. Create a component INF file in the source directory for the component. Typically a similar component's INF file will be copied and modified.
2. Add the component to the list of components in the build tip's DSC file. For example, if the **`DiskIo`** component is to be added to the **`Nt32`** build tip, then you must edit the **`Nt32.dsc`** file, find the **`[components]`** section, and add the INF file name to the list of components.
3. Execute a build to build the component.

The resultant firmware file will then go into the default firmware volume in the final image.

5.5 Building a Driver for EBC

To build a driver as an EBC image, the component should first be added as a standard component described earlier. Then the component line can be modified to specify the target processor as EBC. For example, to build the **DiskIo** driver as an EBC driver, modify the DSC component line for it to be something like the following:

```
Universal\Disk\DiskIo\Dxe\DiskIo.inf  PROCESSOR=EBC
```

This example assumes the following:

- The EBC compiler has been installed in the default location.
- EBC versions of the libraries have been added to the **[libraries]** section of the DSC file.

If the build fails, the **LocalTools.env** file may need to be modified to specify the path to the EBC compiler and linker.

5.6 Specifying Destination Firmware Volumes for a Component

The build process will put each component in the **[components]** section of the DSC file in the firmware volume or firmware volumes specified by variable **$(FV)**. For example, assume the platform DSC file has the firmware volume defined as shown (no spaces on either side of the comma):

```
[components]
DEFINE FV=fv0001,fv0002
```

For this example, all the components following this definition will be put in both **fv0001.fv** and **fv0002.fv**. However, a component line can specify a different value of FV, for example:

```
Universal\Disk\DiskIo\Dxe\DiskIo.inf  FV=fv0001
```

For this case, the **DiskIo** component will be included only in **fv0001.fv**. Because this definition is a local definition (which only applies to the current component) of the firmware volume, any components that follow it in the DSC file will revert back to the default firmware volume value.

6
Directory Structure of the Release Package

6.1 Description of Directory Structure

The top-level directory structure for the EDK release package is shown below and described in further sections. The root directory of the package contains directories for foundational code, sample implementation, and other add-on drivers, tools and applications.

```
EDK\
     Foundation\
     Sample\
     Other\
```

6.1.1 \Foundation

This directory contains the foundational source code, including PEI and DXE core, libraries, essential definitions for both EFI and *Intel® Platform Innovation Framework for EFI*, as well as definitions of protocols, GUIDs, and PPIs. This directory contains 9 subdirectories, which are shown below and described in further sections.

```
EDK\
     Foundation\
          Core\
          Cpu\
          Efi\
          Framework\
          Guid\
          Include\
          Library\
          Ppi\
          Protocol\
```

6.1.1.1 \Foundation\Core

This directory contains the source code for PEI core and the DXE core.

```
EDK\
    Foundation\
        Core\
            Dxe\
            Pei\
```

6.1.1.2 \Foundation\Cpu

This directory contains library for frequently used CPU instructions of IA32 architecture, such as HLT, WBINVD, CPUID, and etc. The library provides a standard procedure interface for these instructions, so that they can be invoked as encapsulated functions.

```
EDK\
    Foundation\
        Cpu\
            Pentium\
```

6.1.1.3 \Foundation\Efi

This directory contains definitions for protocols, GUIDs, and other items that are defined in *Extensive Firmware Interface Specification*, which can be found at http://developer.intel.com/technology/efi/. A Protocol/GUID component usually contains a source .h file that defines the Protocol/GUID, and a source .c file that defines a global variable for the Protocol/GUID. No PPI definition occurs here since the EFI specification does not define any PPIs.

```
EDK\
    Foundation\
        Efi\
            Guid\
            Include\
            Protocol\
```

6.1.1.4 \Foundation\Framework

This directory contains definition for protocols, GUIDs, PPIs and other items that are defined in the specifications of *Intel® Platform Innovation Framework for EFI*. These specifications can be found at http://developer.intel.com/technology/framework/. Each Protocol/GUID/PPI component usually contains a source .h file that defines the Protocol/GUID/PPI, and a source .c file that defines a global variable for the Protocol/GUID/PPI.

```
EDK\
    Foundation\
        Framework\
            Guid\
            Include\
            Ppi\
            Protocol\
```

6.1.1.5 \Foundation\Guid

This directory contains foundational GUIDs that are not defined in EFI or Framework specifications. Each GUID component usually contains a source .h file that defines the GUID, and a source .c file that defines a global variable for the GUID. Files for each GUID component usually constitute a separate subdirectory.

Files for each GUID usually constitute a separate subdirectory. The subdirectory structure is omitted here.

6.1.1.6 \Foundation\Include

This directory contains header files for generic definitions, including common macros, type definitions for various architectures, and definitions for industry standards. The subdirectory structure is shown below.

```
EDK\
    Foundation\
        Include\
            Ebc\
            Ia32\
            IndustryStandard\
            Ipf\
            Pei\
```

6.1.1.7 \Foundation\Library

This directory contains header files and source files for functions that are commonly used, including PEI library for PEI phase, DXE library for DXE phase, Runtime-DXE library for both runtime phase and DXE phase, and the common library for various phases. The subdirectory structure is shown below.

```
EDK\
    Foundation\
        Library\
            Dxe\
            EfiCommonLib\
            Pei\
            RuntimeDxe\
```

6.1.1.8 \Foundation\Ppi

This directory contains foundational PPIs that are not defined in the Framework specification (No PPI is defined in EFI specification, as discussed above). Each PPI component usually contains a source .h file that defines the PPI, and a source .c file that defines a global variable for the PPI.

Files for each PPI usually constitute a separate subdirectory. The subdirectory structure is omitted here.

6.1.1.9 \Foundation\Protocol

This directory contains foundational protocols that are not defined in EFI or Framework specifications. Each Protocol component usually contains a source .h file that defines the protocol, and a source .c file that defines a global variable for the protocol.

Files for each protocol usually constitute a separate subdirectory. The subdirectory structure is omitted here.

6.1.2 \Sample

This directory contains sample implementations for a set of drivers, libraries, tools as well as related definitions. The NT32 emulation platform is also included in this directory. This directory contains 8 subdirectories, which are shown below and described in further sections.

```
EDK\
      Sample\
            Bus\
            Chipset\
            Cpu\
            Include\
            Library\
            Platform\
            Tools\
            Universal\
```

6.1.2.1 \Sample\Bus

This directory contains sample bus driver implementation for industry standard buses, such as PCI, SCSI, and USB. It also contains drivers for devices on these industry standard buses, such as drivers for USB keyboard, SCSI disk, and etc. In order to support NT32 emulation platform, a subdirectory named "WinNtThunk" is created to contain a set of simulated buses for the NT32 emulation tip.

```
EDK\
      Sample\
            Bus\
                  Pci\
                  Scsi\
                  Usb\
                  WinNtThunk\
```

6.1.2.2 \Sample\Chipset

This directory contains only one subdirectory named WinNtThunk, which contains software emulated chipset drivers to support NT32 emulation platform..

```
EDK\
      Sample\
            Chipset\
                  WinNtThunk\
```

6.1.2.3 \Sample\Cpu

This directory contains the CPU related modules. The subdirectory "WinNtThunk" contains software emulated CPU driver to support NT32 emulation platform; The subdirectory "DebugSupport" includes some debug support facilities.

```
EDK\
      Sample\
            Cpu\
                  DebugSupport\
                  WinNtThunk\
```

6.1.2.4 \Sample\Include

This directory contains header files for sample implementations. The header files defines items such as common macros, type definitions, and preprocessor directives controlling behavior of compiler. No subdirectory exists here.

6.1.2.5 \Sample\Library

This directory contains functions that are commonly used in sample implementations. The subdirectory structure is shown below.

```
EDK\
      Sample\
            Library\
                  Dxe\
```

6.1.2.6 \Sample\Platform

This directory contains PEIMs, DXE drivers, as well as other components for supporting some specific platforms. Some of the modules are generic enough to be used for multiple platforms thus they are gathered in subdirectory “Generic.” For other modules, they are grouped by the corresponding platform names. For example, all the components (drivers, build configuration files, etc) that are specific for NT32 platform reside in subdirectory “NT32”; all components that are specific for IPF reference tip reside in subdirectory “IPF”.

```
EDK\
      Sample\
            Platform\
                  Generic\
                  Nt32\
                  IPF\
```

6.1.2.7 \Sample\Tools

This directory contains source code and makefiles for the tools that are used to build EDK.

```
EDK\
      Sample\
            Tools\
                  Source\
```

6.1.2.8 \Sample\Universal

This directory contains components that can be used on various platforms by simply recompiling them. These components typically consume one set of interfaces to generate another set of interfaces, without touching hardware directly, ensuring their portability. The PEIMs use only PEI services and PPIs, and the DXE drivers depend only upon EFI services and protocol interfaces to interact with hardware

```
 EDK\
      Sample\
           Universal\
                Console\
                DataHub\
                Debugger\
                Disk\
                DxeIpl\
                Ebc\
                FirmwareVolume\
                GenericMemoryTest\
                MonotonicCounter\
                Network\
                Runtime\
                Security\
                UserInterface\
                Variable\
                WatchdogTimer\
```

6.1.3 \Other

This directory contains other add-on drivers, tools and applications. These components are classified according to whether they are maintained or not. Those which are maintained reside in the subdirectory “Maintained.” On the other hand, those which are no longer maintained would be placed in subdirectory “NonMaintained.” Since currently all add-on drivers, tools and applications in the \Other directory are maintained, the subdirectory “NonMaintained” does not exist.

```
 EDK\
      Other\
           Maintained\
```

6.1.3.1 \Other\Maintained

This directory contains add-on drivers, tools and applications that are maintained, including Shell binaries, third-party tools, and etc. The subdirectory structure is shown below.

```
 EDK\
      Other\
           Maintained\
                Application\
                Tools\
```

6.1.3.2 \Other\NonMaintained (not existing)

For the moment, all add-on drivers, tools and applications which should appear in the \Other directory are maintained, so this directory does not exist. In future, if there is some not-maintained component should be under \Other directory, this directory would be created.

6.2 Rules

The following rules apply when creating a directory structure for an EDK implementation:

1. Create processor-specific directories as a subdirectory to a component to contain processor specific code for that component. The processor types that are currently supported are IA-32, Itanium processor family, and EBC. They will use the following processor-specific directory names:
   - `Ia32`
   - `Ipf`
   - `Ebc`
2. A DXE driver must have one and only one "`\Dxe\`" name in its path.
3. A runtime DXE driver must have one and only one "`\RuntimeDxe\`" name in its path.
4. A PEIM must have one and only one "`\Pei\`" name in its path
5. An SEC component must have one and only one "`\Sec\`" name in its path
6. A tool component must have one and only one "`\Tools\`" name in its path.
7. A PPI definition must have one and only one "`\Ppi\`" name in its path.
8. A protocol definition must have one and only one "`\Protocol\`" name in its path.
9. A DXE Architectural Protocol definition must have one and only one "`\ArchProtocol\`" name in its path.
10. A GUID definition must have one and only one "`\Guid\`" name in it path.
11. It is legal to have a "`\Shared\`" directory as a peer to one or more component directories.
12. No assembly files are allowed at the component level since assembly code is processor specific.
13. Both C and assembly files are allowed in a processor-type-specific subdirectory.
14. It is legal to have both `\Dxe` and `\RuntimeDxe` directories present at the same level.

Appendix A
File Name Extensions

Table 6-1 below describes the different file name extensions that are used when building an EDK implementation.

Table 6-1. File Name Extensions

File Extension	Description
.APP	An intermediate DXE application program file that is produced by the **GenFfsFile** build tool from .PKG, .DPX, .SST, .PEI, and .PE32 files.
.ASM	Assembly source code file.
.BIN	A binary file that is produced by the **Pe2Bin** build tool from .EXE files and also produced by other build tools.
.BIN1	An intermediate file that is produced by the **SplitFile** build tool from a .BIN file. This file is not used.
.BIN2	An intermediate file that is produced by the **SplitFile** build tool from a .BIN file.
.C	C source code file.
.COD	A listing of the assembly code that is generated.
.COM	An executable file that is produced by **LINK** using the **/TINY** switch.
.DEP	Makefile dependency file that is generated by the **MakeDeps** utility.
.DLL	Windows dynamic-link library (DLL) that is produced by **LIB** and **LINK**.
.DPE	An intermediate file that is produced by the **GenDepex** build tool from a .TMP1 file.
.DPX	A binary dependency file that is produced by the **GenSection** build tool from a .TMP2 or .DPE file.
.DSC	A build description text file, which defines the components, build rules and commands, FV definitions, and package file definitions for a build tip.
.DXE	An intermediate DXE driver or DXE Foundation file that is produced by the **GenFfsFile** build tool from .PKG .DPX, .SST, .PEI, and .PE32 files.
.DXS	A dependency text source file.
.EFI	An intermediate file that is produced by the **FwImage** build tool from a .DLL file.
.EXE	An executable program that is produced by **LIB** and **LINK**.
.EXP	An export file. Exports a function from a program to allow other programs to call the function. Produced by **LINK**.
.FFS	An intermediate file that is produced by the **GenFfsFile** build tool from .PKG and other files.
.FV	A partial flash volume file that is produced by the **GenFvImage** build tool.
.H	C source header File.
.HDR	The makefile.HDR file that is produced by the **GenMake** build tool from a make.INF file.

File Extension	Description
.I	An Intel® Itanium® assembler include file.
.INC	An assembly include file.
.INF	Input file that specifies information for various build tools.
.LIB	An intermediate library file that is produced by **LIB** and **LINK**.
.MAP	A text file that contains information about the program being linked, including the groups in the program and a list of public symbols. **LINK** names the map file with the base name of the program and the file name extension .MAP.
.OBJ	A file containing object code or data generated by a compiler or an assembler from the source code of an application.
.PDB	Platform-Dependent Driver. A file used by the Platform Builder build tools to store information about a user's application. A .PDB file speeds linking during the debugging phase of development by keeping the debugging information separate from the object files. Produced by **LINK**.
.PE32	Portable Executable (PE) 32 (bit) file format that is produced by the **GenSection** build tool from an .EFI file.
.PEI	An intermediate file that is produced by the **GenFfsFile** build tool from .PKG, .PEIM, and .PIC files.
.PEIM	An intermediate file that is produced by the **GenSection** build tool from .INF, .MAP, .BIN, and .TMP files.
.PKG	A package file that is used by the **GenFfsFile** build tool. This text source file designates which files the **GenFfsFile** build tool should use along with other information.
.PRO	An intermediate file that is produced by the Intel® Itanium® compiler from .S files. This file is used by the Intel Itanium assembler to make an .OBJ file.
.RAW	An intermediate file that is produced by the **GenBsfImage** build tool from .INF, .BIN, and .SYM files.
.S	An Intel® Itanium® assembler source code file.
.SBR	Input files for **BSCMAKE**. The compiler creates a .SBR file for each object file (.OBJ) that it compiles. **BSCMAKE** builds a browse information file describing classes, functions, data, macros, and types in your program.
.SST	A terminator section file that is produced by the **GenSection** build tool.
.SYM	The symbols file that is produced by the **Pe2Sym** build tool from .PDB files and that is also produced by other build tools.
.TMP	An intermediate file that is produced by the **Pe2Bin** build tool from a .DLL file.
.TMP1	An intermediate file that is produced by the precompiler from a .DXS file.
.TMP2	An intermediate file that is produced by the **GenDepex** build tool from a .TMP1 file.