intel

EFI Application Toolkit
Version 1.10.14.62

Product Release Notes

Version 1.10.14.62

January 16, 2004

Revision History

Revision	Revision History	Date
0.71	Draft release.	6/7/01
1.02.12.38	First release matching 1.02.12.38 Sample Implementation release.	1/5/02
1.10.14.62	Second release matching 1.10.14.62 Sample Implementation release.	1/16/04

intel

Contents

Tables

1 Introduction

1.1 Summary

The EFI Application Toolkit contains source code and documentation that enables rapid development of EFI-based applications, protocols, device drivers, EFI Shells, and OS loaders. It provides libraries and samples for using native EFI services, as well as portability libraries that make it easy to port or write Unix*/POSIX* style programs.

This release of the Toolkit has been tested with version 1.10.14.62 of the EFI Sample Implementation and is compatible with the following versions of the *Extensible Firmware Interface Specification:*

- The final draft of the *Extensible Firmware Interface Specification*, version 1.10 (hereafter referred to as the "EFI 1.10 Specification")
- *EFI 1.10 Specification Update*, version -001

These two specifications and the EFI Sample Implementation are both available from the EFI web site at:
http://developer.intel.com/technology/efi/index.htm

The portability libraries were derived from the FreeBSD* 3.2 source distribution.

These release notes contain the following information:

- System requirements
- Package contents
- Installing the EFI Application Toolkit
- Building the EFI Application Toolkit
- An overview of the Toolkit's directory structure
- Constructing a target makefile
- Setting up and configuring a network
- Known limitations for this release
- Changes history

NOTE

These release notes contain valuable information for successful use of the EFI Application Toolkit. Although new and important information has been highlighted in yellow, please take a few minutes to review the entire document.

1.2 System Requirements

This release of the EFI Application Toolkit requires the following software on an IA-32-based system.

Software Requirements

- Microsoft* Windows* 2000 or Windows XP operating system
- One of the following versions of Microsoft Visual Studio* or Visual C++*:
  - Microsoft Visual Studio .NET 2003 or 2002. These versions do not require service packs. It is recommended to use the Visual Studio .NET versions instead of Visual C++ 6.0.
  - Microsoft Visual C++ 6.0 with the Visual Studio 6.0 Service Pack 5 or later and the Visual C++ Processor Pack Download. These two service packs are available for download from:
    http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/default.aspx
    http://msdn.microsoft.com/vstudio/downloads/tools/ppack/download.aspx
- 50 MB of free disk space for the EFI Application Toolkit source
- For the IA-32 environment, 140 MB of free disk space for build binaries and intermediate files

For the Itanium® architecture environment, an Itanium® compiler is required along with 200 MB for build binaries and intermediate files.

1.3 Package Contents

The EFI Application Toolkit provides source code for the following components:

- Full-screen Unicode/ASCII text editor
- Python scripting language interpreter
- TCP/IPv4 network stack with PPP support
- Network stack configuration utilities and applications:
  - ifconfig
  - route
  - hostname
  - ping
  - FTP client
  - DHCP client
  - PPP connection management driver
- Multiprocessor test support protocol driver
- Internationalization support through loadable locale protocols
- RAM disk protocol
- Sample applications
- Support libraries:
  - General EFI library (**`libefi`**)
  - EFI Shell interface library (**`libefishell`**)
  - C library (**`libc`**)

  - Berkeley sockets library (**libsocket**)
  - Math library (**libm**)
  - SMBIOS support library (**libsmbios**)
  - Database access methods library (**libdb**)
  - File compression/decompression library (**libz**)
  - Serial port TTY emulation library (**libtty**)
- Design/user documentation:
  - *EFI Developer's Guide*
  - *EFI Library Specification*
  - *EFI Application Toolkit Version 1.10.14.62 Product Release Notes* (this document)
  - Unix style manual pages for the compatibility libraries and commands
  - Design documentation for TCP/IPv4, locale, MP
  - Design documentation for SMBIOS library
- EFI build tools (binaries only)
- EFI toolkit binaries:
  - Toolkit binaries for IA-32 EFI systems
  - Toolkit binaries for Itanium®-based EFI systems
  - Toolkit binaries for **Nt32** (Windows emulation) environment

2 Installing and Building the EFI Application Toolkit

2.1 Installing the Toolkit

Release 1.10.14.62 of the EFI Application Toolkit is distributed as a single zip file. When unzipped, the source, documentation, and binaries occupy approximately 50 MB of disk space. Depending on the processor target, binaries that are built from the source (both intermediate and final targets) require an additional 140 MB to 340 MB of disk space.

The Toolkit has been built and tested in the IA-32 and Itanium architecture environments using the following compilers:

- IA-32: Microsoft Visual C++ 6.0 and Microsoft Visual Studio .NET 2002 and 2003

  If you chose to use the Microsoft* tools, it is recommended that that you move any IA-32 EFI Toolkit applications to the latest Microsoft Visual Studio .NET 2002 or 2003 tools.
- Itanium architecture:
  - Intel® C++ Compiler 7.1 for Windows*
  - Microsoft* C/C++ Optimizing Compiler Version 13.10.2240.8 for IA-64 (available in the Microsoft Windows Server* 2003 Driver Development Kit (DDK), Build 3790).

  The Itanium compilers are required only if building any executables for Itanium architecture in the **`\sal64\bin`** directory

The following information assumes that one or more of these compilers have been installed on the development platform. In addition to the compiler, the Toolkit build environment also uses **`NMAKE.EXE`** from Visual C++. Itanium-based platforms must use **`NMAKE.EXE`**, **`LINK.EXE`**, and **`LIB.EXE`** from the Microsoft Windows Server 2003 DDK (Build 3790).

To install the source, simply unzip the contents of this distribution into an **empty** directory. Do not unzip the contents into a previous release of the EFI Application Toolkit.

To use the Microsoft C/C++ Optimizing Compiler Version 13.10.2240.8 for IA-64 in the Microsoft Windows Server 2003 DDK (Build 3790), copy the Itanium compiler and linker from **`\WINDDK\3790\bin\WIN64\x86`** to **`C:\IPFTools\Microsoft`**. If you want to use the Intel C++ Compiler 7.1 for Windows*, you should copy it into **`C:\IPFTools\Intel71`**..

2.2 Building the Toolkit

The EFI Application Toolkit supports three build environments. Each is given a hosting directory within the tree. Table 2-1 lists these build environments.

Table 2-1. Build Environments in EFI 1.10.14.62 Source Tree

Build Tip	Description
build\Nt32	The NT EFI emulator environment.
build\bios32	The IA-32 EFI standard BIOS environment.
build\sal64	The EFI environment for Itanium architecture.

The simplest way to build the Toolkit is to use the following commands:

1. **`cd \efi_toolkit_1.10.14.62`**
2. **`build bios32`** (or "**`build nt32`**" or "**`build sal64`**")
3. **`nmake`**

The EFI Sample Implementation supports these environments plus an **`ia-32emb`** environment. All Toolkit binaries built under the **`bios32`** environment are suitable to run in the **`ia-32emb`** environment as well.

Users can use **`EFI_APPLICATION_COMPATIBILITY`** to control the compatibility mode for their applications. It is defined in **`\efi_toolkit_1.10.14.62\build.cmd`**. Table 2-2 lists the modes that are available in **`EFI_APPLICATION_COMPATIBILITY`**. See section 4.3 for backward compatibility between EFI 1.02 and EFI 1.10.

Table 2-2. Compatibility Modes for EFI Applications

EFI_APPLICATION_COMPATIBILITY	Notes
EFI_APP_102	The application can run in the EFI 1.02 and EFI 1.10 environments but can use only EFI 1.02 services and protocols. This setting is the default in **`build.cmd`** for maximum compatibility of the EFI application. However, this setting will limit the features in the application to EFI 1.02–level protocols and services.
EFI_APP_110	The application can run only in the EFI 1.10 environment and can use all the EFI 1.10 services and protocols. If the application is known to ship only on EFI 1.10 systems or higher, then the developer should select this setting in the **`build.cmd`** environment before building the EFI application.
EFI_APP_MULTIMODAL	The application can run in the EFI 1.02 and EFI 1.10 environments and can use all the EFI 1.10 services and protocols. However, it is the application developer's responsibility to determine the current running environment and which service/protocol to use.

There is a single **`master.mak`** file located in the root of the build directory that applies to all build environments. The **`master.mak`** file is "included" in all Toolkit component makefiles, providing common rules and final macros that are required for the build environment.

Each build environment directory contains an **`sdk.env`** file. The **`sdk.env`** file is also "included" by all Toolkit component makefiles. Its primary purpose is to define the set of macros that locate

build tools, source files, and output directories. Table 2-3 lists the variables that must be set in an NT command window environment before invoking a Toolkit component makefile.

Table 2-3. Variables to Set in NT Command Window Environment

Variable	Notes
SDK_BUILD_ENV	Set to the target build environment (**bios32**, **nt32**, or **sal64**).
SDK_INSTALL_DIR	Set to the root of the EFI Application Toolkit source tree.
SDK_64_INCLUDE_DIR	Set to the root of the Visual C++ include directory for Itanium® processors. Required only for building debugger components.

Due to the number of possible build environments for Itanium architecture, you must review and edit the **sdk.env** file in the **build/sal64** directory if you are building for that environment. There are two macros that point to the root tools directory for either the Microsoft* or Intel® compilers. These macros are **MSSdk** and **_IA64SDK_DIR** respectively.

The macros listed in Table 2-4 are also required. However, they are set to default values that are constructed from the macros above. If the default locations do not meet your needs, edit **sdk.env** and set them to the appropriate values.

Table 2-4. Required Macros

Macro	Notes
SDK_BUILD_DIR	Set to the root of the intermediate binary output tree.
SDK_BIN_DIR	Set to the directory for final binary targets.

The following is an example for a Visual C++ 6.0 installation where the Toolkit source was placed in **C:\Toolkit** and the build environment was for **bios32**:

```
SDK_BUILD_ENV=bios32
SDK_INSTALL_DIR=C:\Toolkit
```

The following are the default values for **SDK_BUILD_DIR** and **SDK_BIN_DIR**:

```
SDK_BUILD_DIR=$(SDK_INSTALL_DIR)\build\$(SDK_BUILD_ENV)\output
SDK_BIN_DIR=$(SDK_INSTALL_DIR)\build\$(SDK_BUILD_ENV)\bin
```

The final step is to invoke the desired EFI Application Toolkit target makefile. Target makefiles use the naming convention **<component>.mak** where **<component>** is generally equal to the containing directory name. Invoking a makefile can be done at the individual Toolkit component level:

```
cd cmds\ed
nmake –f ed.mak
```

You can also invoke a makefile at a parent level, which will invoke all subcomponent makefiles:

```
cd cmds
nmake –f cmds.mak
```

Invoking **nmake** at the root of the Toolkit source tree, will do a complete build of the EFI Application Toolkit.

3 EFI Application Toolkit Directory Structure

3.1 Top Directory Level

This section provides an overview of the Toolkit directory structure. The root of the Toolkit source tree contains the primary directories that are listed in Table 3-1. Each of these will be covered in more detail in section 3.2.

Table 3-1. Contents in Root Directory of Toolkit Source Tree

Primary Directories	Notes
apps	EFI sample applications.
binaries	Executable binaries built from the source.
build	Build environment directories.
cmds	Ports of FreeBSD commands and utilities.
doc	Documentation for the EFI Application Toolkit.
include	Common include files.
lib	Common libraries.
protocols	Toolkit-supplied EFI protocols and drivers.

3.2 Subdirectories

3.2.1 \Apps

Applications in the **`\apps`** directory provide working examples for using native EFI services and the EFI support library. Table 3-2 lists the contents of the **`\apps`** directory.

Table 3-2. Contents of `\apps` Directory

Directory	Notes
exitboot	**`ExitBootServices()`** test.
NtFloppy	NT emulation utility to enable/disable access to the A: device under emulation.
osloader	An OS loader example.
pktsnoop	Example of an EFI Simple Network interface that reads packets.
pktxmit	Example of an EFI Simple Network interface that sends packets.
rtdriver	Example of a driver unload interface.
rtunload	Example for requesting a driver unload.
salpaltest	Tests some sample SAL/PAL procedures using library functions.
scripts	Test scripts.

continued

Table 3-2. Contents of `\apps` Directory (continued)

Directory	Notes
test	Example for locating and printing EFI Loaded Image protocol data.
test2	Example of finding the block device of a file system that contains the test image.
test3	Another example of using EFI Loaded Image protocol data.
test4	Example of using EFI timers.
TestBoxDraw	Example of using Unicode Box Draw characters.
testfpswa	Tests virtual address calls by making a one-to-one transition.
testva	Example of using a virtual address call.
testvrt	Example of using a virtual address call under the runtime environment.

3.2.2 \Binaries

The **`\binaries`** directory contains executable files that were built from each of the build environments, targeted for a different EFI system. In this version of the Toolkit, the **`.efi`** executables are prebuilt for system integrators to use and do not require Toolkit users to build the executables from the provided source.

Table 3-3. Contents of `\binaries` Directory

Directory	Notes
bios32	**`.efi`** executables compiled from the **`\EFI_Toolkit_1.10.14.62\build\bios32\bin`** directory. These IA-32 executables are designed to work on EFI-compliant IA-32 systems. They will also work on systems compiled with the **`Ia-32emb`** build tip from the EFI 1.10 Sample Implementation (see the EFI Web site under Tools→Sample Implementation).
nt32	**`.efi`** executables compiled from the **`\EFI_Toolkit_1.10.14.62\build\nt32\bin`** directory. These **`.efi`** executables are designed to be used in the **`Nt32`** emulation environment from the EFI 1.10.14.62 Sample Implementation.
sal64	**`.efi`** executables compiled from the **`\EFI_Toolkit_1.10.14.62\build\sal\bin`** directory. These **`unoptimized.efi`** executables are designed to be used on EFI-compliant Itanium-based systems.

3.2.3 \Build

The **\build** directory provides the following:

- Environment-specific configuration files that are used during the build process
- The source for support tools that are needed in the build process.

It is also the default location for intermediate and target binaries. By default, intermediate files will be stored in the target environment directory under the directory **\output**. Final target binaries are stored in the **\bin** directory. Table 3-4 lists the contents of the **\build** directory.

Table 3-4. Contents of \build Directory

Directory	Notes
bios32	Contains the **sdk.env** file for IA-32 EFI environments (**bios32** and **Ia-32emb**).
nt32	Contains the **sdk.env** file for building EFI binaries for the NT emulation environment.
sal64	Contains the **sdk.env** file for building EFI binaries for the Itanium architecture environment.
tools	Contains build support tools. Currently, the only tool required is **fwimage**.

3.2.4 \Cmds

The programs in the **\cmds** directory provide working examples for using the portability libraries of the Toolkit. Some of these programs were chosen as a way of validating our port of the libraries while others, such as Python and the network configuration utilities, provide real value. Table 3-5 lists the contents of the **\cmds** directory.

Table 3-5. Contents of \cmds Directory

Directory	Notes
Ed	Unix command line editor (ASCII only).
Edit	Full-screen Unicode/ASCII text editor.
ftp	FTP client.
Hexdump	Dump files in varying display formats.
Hostname	Sets the network name for the system.
Ifconfig	Configures network interfaces.
Loadarg	Loads an EFI file with parameters to pass onto another driver such as Dhclient (see section 7.9 for TCP/IP limitations).
mkramdisk	Configures and installs RAM disk.
Mptest	Test program for multiprocessor test protocol. (The default in the source code is set to four processors in the system.)

continued

Table 3-5. Contents of \cmds Directory (continued)

Directory	Notes
Nunload	Removes and stops execution of any instances of **Dhclient** and **Tcpipv4** (network unload) drivers in the system.
Ping	"The" network diagnostic tool.
Python	Object-oriented scripting language interpreter.
Route	Sets the network gateway.
Which	Identifies which file would have been executed had its argument been given as a command.

3.2.5 \Doc

The **\doc** directory contains the documentation that is provided for this release. Note that complete Python documentation is contained in **\cmds\python\documentation**. Table 3-6 lists the contents of the **\doc** directory.

Table 3-6. Contents of \doc Directory

Directory	Notes
.	The root of the directory contains a copy of these release notes, the *EFI Developer's Guide,* and the *EFI Library Specification* for reference.
Design	Contains API design documents for various Toolkit components.
Man	Contains Unix-style man pages for all programs in the **\cmds** directory as well as **libc**, **libm**, **libsocket**, **libdb**, **libz**, and **libtty**. Man pages are further subdivided into HTML and original NROFF format.

3.2.6 \Include

The **\include** directory is the root of all common includes files that an application may use. Table 3-7 lists the contents of the **\include** directory.

Table 3-7. Contents of \include Directory

Directory	Notes
Bsd	Include files from FreeBSD distribution. Associated with **libc**, **libdb**, **libm**, **libsmbios**, **libsocket**, **libtty**, and **libz**.
Efi	Include files distributed with the 1.02.12.38 EFI Sample Implementation, as well as those associated with **libefi**.
efi110	Include files distributed with the 1.10.14.62 EFI Sample Implementation, as well as those associated with **libefi**.
efishell	Include files associated with **libefishell**.

3.2.7 \Lib

The **\lib** directory contains the source for all common libraries to which an application may link. By default, the build process will place the library in the output directory of the same name. For example, **build\bios32\output\lib\libc\libc.lib**.

Due to export and possible licensing restrictions, the source for **libcrypt** and **libmd** are not distributed with this release.

Table 3-8 lists the contents of the **\lib** directory.

Table 3-8. Contents of \lib Directory

Directory	Notes
Libc	Standard C library. Contains both ANSI- and FreeBSD-specific routines as well as Unix system call emulation.
Libdb	Database access method library.
libefi	General support library for EFI services.
libefishell	Support library for EFI Shell applications.
Libm	Standard math library. Contains both ANSI- and FreeBSD-specific routines
libsmbios	Library routines for parsing and retrieving SMBIOS tables.
libsocket	Standard Berkeley sockets library.
Libtty	Serial port TTY emulation library.
Libz	File compression/decompression library

3.2.8 \Protocols

The **\protocols** directory contains the source for EFI Application Toolkit–supplied protocols and drivers. Use the EFI Shell **load** command to make them available for use by an application. Table 3-9 lists the contents of the **\protocols** directory.

Table 3-9. Contents of \protocols Directory

Directory	Notes
Dhclient	DHCP client protocol.
Locale	Loadable locale protocols that are used with **libc**.
Mp	Multiprocessor support protocol.
Pppd	PPP serial connection management driver.
Ramdisk	RAM disk protocol.
tcpipv4	EFI protocol implementing a complete TCP/Ipv4 network stack.

4
Creating a Simple EFI Application with the Toolkit

This section describes how to create a simple EFI application with the EFI Application Toolkit. It is recommended that all EFI applications that use the EFI Library be placed in the **\apps** directory and all EFI applications that use **libc** be placed in the **\cmds** directory. This recommendation is not strictly required. However, it does provide a convenient build environment. To add a new application to the build environment, a new subdirectory needs to be created, and a makefile (**test.mak**) along with the source code needs to be placed in that subdirectory. For this example, the files for a simple test application are placed in the directory **\apps\test**, and their contents are listed in the following sections.

4.1 Makefile for a Simple EFI Application

The makefile **test.mak** specifies the following:

- A list of source files
- The path to the include directories
- The path to the library directories
- The entry point for the application
- The name of the executable EFI application image

The contents of **test.mak** are listed below and a detailed explanation of the makefile is in the next section, section 5.

```
!include $(SDK_INSTALL_DIR)\build\$(SDK_BUILD_ENV)\sdk.env
#
# Set the base output name and entry point
#

BASE_NAME         = test
IMAGE_ENTRY_POINT = InitializeTestApplication

#
# Globals needed by master.mak
#

TARGET_APP = $(BASE_NAME)
SOURCE_DIR = $(SDK_INSTALL_DIR)\apps\$(BASE_NAME)
BUILD_DIR  = $(SDK_BUILD_DIR)\apps\$(BASE_NAME)

#
# Include paths
#
```

```
!include $(SDK_INSTALL_DIR)\include\$(EFI_INC_DIR)\makefile.hdr
INC = -I $(SDK_INSTALL_DIR)\include\$(EFI_INC_DIR) \
      -I $(SDK_INSTALL_DIR)\include\$(EFI_INC_DIR)\$(PROCESSOR) $(INC)

#
# Libraries
#

LIBS = $(LIBS) $(SDK_BUILD_DIR)\lib\libefi\libefi.lib

#
# Default target
#

all : dirs $(LIBS) $(OBJECTS)

#
# Program object files
#

OBJECTS = $(OBJECTS) $(BUILD_DIR)\$(BASE_NAME).obj

#
# Source file dependencies
#

$(BUILD_DIR)\$(BASE_NAME).obj : $(*B).c $(INC_DEPS)

#
# Handoff to master.mak
#

!include $(SDK_INSTALL_DIR)\build\master.mak
```

4.2 The “C” File for a Simple EFI Application

The test application uses the *ImageHandle* and *SystemTable* that are passed into the entry point to initialize the EFI Library. After it is initialized, the test application can use Boot Services, Runtime Services, and the EFI System Table. The test application then uses the Boot Service **HandleProtocol()** to get the **EFI_LOADED_IMAGE** protocol interface and print the information from that interface. The application then waits for a keystroke from the user on the console input device, using the **WaitForSingleEvent()** service with the *WaitForKey* event in the **SIMPLE_INPUT_INTERFACE** protocol. This test application will wait until a key is pressed and then the application will exit.

```
#include "efi.h"
#include "efilib.h"

VOID
PrintLoadedImageInfo (
    IN  EFI_LOADED_IMAGE    *LoadedImage
    );
```

```
EFI_STATUS
InitializeTestApplication (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
    )
{
    EFI_LOADED_IMAGE    *LoadedImage;
    EFI_INPUT_KEY       Key;

    InitializeLib (ImageHandle, SystemTable);

    Print(L"Test application started\n");

    BS->HandleProtocol (ImageHandle, &LoadedImageProtocol,
(VOID*)&LoadedImage);
    PrintLoadedImageInfo (LoadedImage);

    Print(L"\n%EHit any key to exit this image%N");
    WaitForSingleEvent (ST->ConIn->WaitForKey, 0);

    ST->ConIn->ReadKeyStroke(ST->ConIn,&Key);

    ST->ConOut->OutputString (ST->ConOut, L"\n\n");

    return EFI_SUCCESS;
}

UINT16 *MemoryType[] = {
            L"reserved  ",
            L"LoaderCode",
            L"LoaderData",
            L"BS_code   ",
            L"BS_data   ",
            L"RT_code   ",
            L"RT_data   ",
            L"available ",
            L"Unusable  ",
            L"ACPI_recl ",
            L"ACPI_NVS  ",
            L"MemMapIO  ",
                    L"MemPortIO ",
                    L"PAL_code  "
            L"BUG:BUG: MaxMemoryType"
};

VOID
PrintLoadedImageInfo (
    IN  EFI_LOADED_IMAGE    *LoadedImage
    )
{
    EFI_STATUS          Status;
    EFI_DEVICE_PATH     *DevicePath;

    Print (L"\n%HImage was loaded from file %N%s\n", DevicePathToStr (LoadedImage-
>FilePath));

    BS->HandleProtocol (LoadedImage->DeviceHandle, &DevicePathProtocol,
(VOID*)&DevicePath);
    if (DevicePath) {
```

```
        Print (L"%HImage was loaded from this device %N%s\n", DevicePathToStr
(DevicePath));

    }

    Print (L"\n Image Base is %X", LoadedImage->ImageBase);
    Print (L"\n Image Size is %X", LoadedImage->ImageSize);
    Print (L"\n  Image Code Type %s", MemoryType[LoadedImage->ImageCodeType]);
    Print (L"\n  Image Data Type %s", MemoryType[LoadedImage->ImageDataType]);
    Print (L"\n %d Bytes of Options passed to this Image\n", LoadedImage-
>LoadOptionsSize);

    if (LoadedImage->ParentHandle) {
        Status = BS->HandleProtocol (LoadedImage->ParentHandle,
&DevicePathProtocol, (VOID*)&DevicePath);
        if (Status == EFI_SUCCESS && DevicePath) {
           Print (L"Images parent is %s\n\n",DevicePathToStr (DevicePath));
        }
    }
}
```

4.3 Backward Compatibility to EFI 1.02 and EFI 1.10

A backward-compatible EFI application can run both in EFI 1.02 and EFI 1.10. Developers can use the Toolkit to write backward-compatible applications. The Toolkit supports three kinds of applications:

- EFI 1.02 applications
- EFI 1.10 applications
- Multimodal applications

They differ in the services and protocols they use and the EFI environment they can run.

EFI 1.02 applications use only EFI 1.02 services and protocols. Because EFI 1.10 services and protocols are a superset of those in EFI 1.02, EFI 1.02 applications can run both in EFI 1.02 and EFI 1.10. In contrast, EFI 1.10 applications use EFI 1.10 services and protocols; as a result, they can run only in EFI 1.10. When running in EFI 1.02, EFI 1.10 applications will show a friendly message, indicating that a higher version of EFI is required, and exit. To some degree, multimodal applications are a combination of EFI 1.02 and EFI 1.10 applications. Like EFI 1.10 applications, multimodal applications use EFI 1.10 services and protocols. However, like EFI 1.02 applications, multimodal applications can run both in EFI 1.02 and EFI 1.10. It is the programmer's responsibility to determine the current running environment and which service/protocol to use in multimodal applications.

An environment variable **`EFI_APPLICATION_COMPATIBILITY`** is introduced to specify the application type. Two sets of include files are put in the Toolkit source directory to support backward compatibility. EFI 1.02 applications use the directory **`\inc\efi`**. EFI 1.10 applications and multimodal applications use the directory **`\inc\efi110`**.

To write an EFI 1.02 application, **`EFI_APPLICATION_COMPATIBILITY`** needs to be set to **`EFI_APP_102`**. Also, the application should use only EFI 1.02 services and protocols; otherwise, it will cause a compiling error.

To write an EFI 1.10 application, **EFI_APPLICATION_COMPATIBILITY** needs to be set to **EFI_APP_110**. The application can use all the EFI 1.10 services and protocols. The Toolkit will check the EFI version to guarantee that the application runs only in EFI 1.10 or a higher version.

To write a multimodal application, **EFI_APPLICATION_COMPATIBILITY** needs to be set to **EFI_APP_MULTIMODAL**. The multimodal application is a little more complex than EFI 1.02 and EFI 1.10 applications, in that multimodal applications should take care of the EFI version by itself. At runtime, the application should retrieve the EFI version and behave accordingly. For example, if the running environment is EFI 1.02, the application will not use a service that is available only in EFI 1.10. The Toolkit cannot guarantee the correctness of application. It is the application's responsibility to take the correct execution path.

The Toolkit provides a lightweight Boot Services library and some functions to facilitate developing multimodal applications. The Boot Services function in the library will automatically determine whether the underlying service is available in the EFI environment. If available, the function will call the service; otherwise, it will return an error code. The functions **IsEfi102()**, **IsEfi110()**, and **GetEFIVersion()** can help to determine the EFI version.

The following code fragment shows an example for writing a multimodal application.

```
if( IsEFI110() == TRUE ){
      //
      // It is in EFI 1.10, So use EFI 1.10 boot service
      //
      Status = BS->OpenProtocol(
                  ImageHandle,
                  &gEfiSimpleNetworkProtocolGuid,
                  &pInterface,
                  Private->Handle,
                  EFI_OPEN_PROTOCOL_BY_DRIVER );
      ...
}
else {
      //
      // It is in EFI 1.02, use EFI 1.02 boot service
      //
      Status = BS->HandleProtocol(
                  ImageHandle,
                  &gEfiSimpleNetworkProtocolGuid,
                  &pInterface );
      ...
}
```

Another simple way to write a multimodal application is to use the following multimodal functions when **EFI_APPLICATION_COMPATIBILITY** is set to **EFI_APP_MULTIMODAL**:

- **MultimodalAllocatePages()**
- **MultimodalAllocatePool()**
- **MultimodalCheckEvent()**
- **MultimodalCloseEvent()**
- **MultimodalCreateEvent()**
- **MultimodalExit()**

- **MultimodalExitBootService()**
- **MultimodalFreePages()**
- **MultimodalFreePool()**
- **MultimodalGetMemoryMap()**
- **MultimodalGetNextMonotonicCount()**
- **MultimodalHandleProtocol()**
- **MultimodalInstallConfigurationTable()**
- **MultimodalInstallProtocolInterface()**
- **MultimodalLoadImage()**
- **MultimodalLocateDevicePath()**
- **MultimodalLocateHandle()**
- **MultimodalRaiseTPL()**
- **MultimodalRegisterProtocolNotify()**
- **MultimodalReinstallProtocolInterface()**
- **MultimodalRestoreTPL()**
- **MultimodalSetTimer()**
- **MultimodalSetWatchdogTimer()**
- **MultimodalSignalEvent()**
- **MultimodalStall()**
- **MultimodalStartImage()**
- **MultimodalUnloadImage()**
- **MultimodalWaitForEvent()**
- **MultimodalCopyMem()**
- **MultimodalSetMem()**
- **MultimodalCalculateCrc32()**
- **MultimodalOpenProtocol()**
- **MultimodalCloseProtocol()**
- **MultimodalOpenProtocolInformation()**
- **MultimodalConnectControler()**
- **MultimodalDisconnectControler()**
- **MultimodalProtocolPerHandle()**
- **MultimodalLocateHandleBuffer()**
- **MultimodalLocateProtocol()**
- **MultimodalInstallMultipleProtocolInterfaces()**
- **MultimodalUninstallMultipleProtocolInterfaces()**

5
Creating an EFI Application Toolkit Makefile

This section describes how to construct a makefile that can use the default inference rules, macros, and directory structure of the EFI Application Toolkit. However, there is no requirement to construct makefiles as outlined below. For example, one may chose to collect libraries and include files outside of the distributed Toolkit directory structure and have output binaries remain in a subdirectory of the source directory. A makefile to support such an environment would be more traditional and should be straightforward to construct.

Perhaps the easiest way to construct an application, protocol, or library makefile is to start by copying an existing makefile of the appropriate type from the Toolkit. All Toolkit makefiles use a consistent format. The following paragraphs explain the purpose of the relevant makefile sections. The following discussion uses the **\cmds\ifconfig\ifconfig.mak** file as an example.

The first section includes the build environment–specific macros that are specified in **sdk.env**. This line should be the first one executed in the makefile.

```
#
# Include sdk.env environment
#
!include $(SDK_INSTALL_DIR)\build\$(SDK_BUILD_ENV)\sdk.env
```

The next section defines the **BASE_NAME** macro needed by **master.mak** and the remaining sections of the makefile. This defines the name of the output directory for the application, library, or protocol being built.

```
#
# Set the base output name and type for this makefile
#
BASE_NAME  = ifconfig
```

The next section defines the EFI entry point of the program. All programs need to have their entry point defined through the **IMAGE_ENTRY_POINT**. This is the label of the EFI image entry point that conforms to the following prototype definition:

```
typedef
(EFIAPI *EFI_IMAGE_ENTRY_POINT) (
     IN EFI_HANDLE         ImageHandle,
     IN EFI_SYSTEM_TABLE   *SystemTable
     );
```

If you are using the **libc** portability library and using **main()** as your program starting point, **IMAGE_ENTRY_POINT** should be set to either **_LIBCStart_Shellapp_A** or **_LIBC_Start_A** depending on which version of the EFI Shell with which it will be run.

```
#
# Set entry point
#
!IFDEF OLD_SHELL
IMAGE_ENTRY_POINT = _LIBC_Start_Shellapp_A
!ELSE
IMAGE_ENTRY_POINT = _LIBC_Start_A
!ENDIF
```

The next section defines the macros needed by **master.mak**. Depending on whether your program is an application, library, or a protocol/driver, you must either set the **TARGET_APP**, **TARGET_LIB**, or **TARGET_BS_DRIVER** macro respectively. These macros define the name of the final EFI binary image and the way they will be tagged by the **fwimage** tool. In most cases, this macro can be set to the same value as **$(BASE_NAME)**. The **SOURCE_DIR** and **BUILD_DIR** macros should be set to reflect the path of the makefile's directory relative to the root of the installation.

```
#
# Globals needed by master.mak
#
TARGET_APP = $(BASE_NAME)
SOURCE_DIR = $(SDK_INSTALL_DIR)\cmds\$(BASE_NAME)
BUILD_DIR  = $(SDK_BUILD_DIR)\cmds\$(BASE_NAME)
```

The next section builds the include search paths and includes predefined header file dependencies. These should be tailored to your application's needs.

```
#
# Include paths
#
!include $(SDK_INSTALL_DIR)\include\efi\makefile.hdr
INC = -I $(SDK_INSTALL_DIR)\include\efi \
      -I $(SDK_INSTALL_DIR)\include\efi\$(PROCESSOR) $(INC)

!include $(SDK_INSTALL_DIR)\include\bsd\makefile.hdr
INC = -I $(SDK_INSTALL_DIR)\include\bsd $(INC)
```

The next section specifies the libraries against which your application needs to link. Note that some libraries may have dependencies on other libraries. For example, **libefishell** requires **libefi**.

```
#
# Libraries
#
LIBS = $(LIBS) \
       $(SDK_BUILD_DIR)\lib\libc\libc.lib \
       $(SDK_BUILD_DIR)\lib\libsocket\libsocket.lib \
!IFDEF OLD_SHELL
       $(SDK_BUILD_DIR)\lib\libefi\libefi.lib \
       $(SDK_BUILD_DIR)\lib\libefishell\libefishell.lib \
!ENDIF
```

The next section declares the default build target and its dependencies. In most cases, the default build target will be **all:** and the dependencies **dirs**, **$(LIBS)**, and **$(OBJECTS)**. The **master.mak** file contains the **dir:** target and targets for all libraries distributed with the Toolkit.

```
#
# Default target
#
all : dirs $(LIBS) $(OBJECTS)
```

The next section can specify your local include dependencies that would apply to the makefile as a whole. You could also add specific include file dependencies on the individual source file target lines.

```
#
# Local include dependencies
#
INC_DEPS = $(INC_DEPS) \
     ifconfig.h
```

Finally, we get to the section that specifies the list of all object files in the output target. Individual object file targets that specify the source file and any include file dependencies generally follow this. If the object file has no dependencies, other than the input source file, it can be omitted as a target and let the default inference rules build the object.

```
#
# Program object files
#
OBJECTS = $(OBJECTS) \
    $(BUILD_DIR)\ifconfig.obj \
    $(BUILD_DIR)\ifmedia.obj \

# Source file dependencies
$(BUILD_DIR)\ifconfig.obj : $(*B).c $(INC_DEPS)
$(BUILD_DIR)\ifmedia.obj  : $(*B).c $(INC_DEPS)
```

The last line of the makefile includes the **master.mak** file, which contains all of the default targets and inference rules.

```
#
# Handoff to master.mak
#
!include $(SDK_INSTALL_DIR)\build\master.mak
```

6
Network Setup and Configuration

Starting with the EFI 0.92.10.8 Sample Implementation, support has been provided for Ethernet cards utilizing 16-bit UNDI option ROMS through the **EFI_SIMPLE_NETWORK** protocol interface. The EFI Application Toolkit provides a complete TCP/IPv4 network stack and configuration tools that take advantage of this protocol.

There are two ways to configure the network stack. One is manual configuration. The other is through the use of a DHCP server and client. The following sections outline the network configuration files that are used by the socket library and the two configuration procedures. Network stack configuration commands need to be executed after booting to the Shell. It is suggested that these commands be grouped together in an EFI batch script to make setting up the network a simple one-line command.

6.1 Name Resolution Configuration Files

The socket library will do name resolution either through a *host*'s file and/or a DNS. The *host* file is an ASCII file with the following format:

```
<ip address>    <hostname> <alias1> <alias2>...
```

An example host file might look as follows:

```
127.0.0.1       localhost
10.8.198.178    kalama1
10.8.162.21     wolfen-c
```

To use the DNS, you must have a **resolv.conf** file. The **resolv.conf** file is also an ASCII file. If you are using the DHCP client to configure the network, it will automatically recreate this file each time the client protocol is loaded. The format of this file is as follows:

```
domain      <default domain>
nameserver  <DNS ip address>
```

In the format above, *<default domain>* is the default domain name of the network and *<DNS ip address>* is the IP address of the DNS server. For example:

```
domain      my.company.com
nameserver  10.8.2.17
```

There is currently a restriction on the location of these files. The socket library will look for both of these files in the **/etc** directory on the file system of the current working directory. That is, if you run **ping** while the current working directory is on **fs0:**, the **hosts** and **resolv.conf** files must be available in **fs0:/etc**. To work around the situation where networking commands need to be run when the current working directory can be one of several EFI file systems, you may duplicate the **/etc** directory and associated files to those file systems.

6.2 Manual Network Configuration

The first operation is to load the TCP/IP protocol. This is done with the EFI **load** command. The **load** command does not use the search path to locate protocols, so the path must be explicitly given if it is not in the current working directory.

Next, the network interfaces need to be configured. The TCP/IP stack contains a loopback interface **lo0** which can be optionally configured along with the Ethernet interface **sni0** if a compatible UNDI Ethernet card is installed. Configuration is performed with the **ifconfig** command. The simple form of the command is:

```
ifconfig lo0 inet <ip address> up
```

where *<ip address>* is the address assigned to the system. If the system is connected to a network that employs subnetting, a subnet mask would also need to be specified as follows:

```
ifconfig sni0 inet <ip address> netmask <netmask> up
```

where *<netmask>* is the network mask appropriate for your network.

Finally, if you wish to do networking across multiple networks or subnetworks, you must set a gateway address for the appropriate gateway(s) attached to your network. This is done with the **route** command as follows:

```
route add <destination> <gateway ip address>
```

where *<destination>* specifies the target network or host and *<gateway ip address>* specifies the address for the gateway attached to your network responsible for routing data to the destination. Using *default* for *<destination>* will set a default route.

The following is a series of commands that might be found in a network configuration batch file. All IP addresses (except the loopback address) are fictional and would need to be changed to reflect your network configuration:

```
load fs0:\efi\tools\tcpipv4.efi
ifconfig lo0 inet 127.0.0.1 up
ifconfig sni0 inet 10.8.198.178 netmask 255.255.255.0 up
route add default 10.8.198.251
```

6.3 DHCP Client Network Configuration

If a DHCP server is accessible for the EFI system, the DHCP client protocol can be loaded to configure the network interface. **The DHCP client protocol can be used only if run with the EFI Shell provided in EFI Application Toolkit release 0.80 or higher. Early BIOS releases for Itanium processors containing 0.99 EFI implementations do not contain the required EFI Shell. See section 7 for possible workarounds.**

The DHCP client constructs configuration scripts and runs them when it is first loaded. Temporary scripts are placed in the root of the file system and removed when configuration has completed. The temporary scripts will set some volatile Shell variables and invoke **`\etc\dhcp-script.nsh`** to perform network configuration based on the information returned by a DHCP server. There is little reason to modify this script.

The DHCP client uses the configuration file **`\etc\dhclient.conf`** to determine how it will interact with the DHCP server. This file may be edited to specify the host name of the EFI system. Some DHCP servers will use this information to register the host with the network DNS server.

The **`dhcp-script.nsh`** and **`dhclient.conf`** files can be found in the **`\protocols\dhclient\client\scripts`** source directory.

To execute **`dhcp-script.nsh`** when loading **`dhclient.efi`**, the **`nshell.efi`** from the EFI 1.10 Sample Implementation is required in the same directory as **`dhclient.efi`**.

A simple Shell batch script with the following lines is all that is required to configure the network stack when DHCP services are available:

```
# The following assumes the network stack and DHCP client
# protocols are in the current working directory.
load tcpipv4.efi
load dhclient.efi
```

Starting with the 1.0 Toolkit release, the DHCP client can maintain a "lease file" (**`\etc\dhclient.leases`**) that allows the client to request the same IP address as in previous boots. This is a compiler option that is on by default. If this mode of operation is not desired, edit the file **`\protocols\dhclient\dhclient.mak`** and comment out the following **`C_FLAGS`** line:

```
#
# Uncomment the next line if the DHCP client should support
# maintaining lease files.  This allows the client to request
# the same IP address it used the last time it ran.
#
C_FLAGS = $(C_FLAGS) -D LEASE_FILE_SUPPORT
```

Then, run the following to produce a version of **dhclient.efi** that runs in the same mode as previous releases:

```
nmake -f dhclient.mak clean
nmake -f dhclient.mak
```

NOTE

Lease file support requires the **\protocols\dhclient\client\scripts\dhclient-script.nsh** *file distributed with the 1.0 or later release of the EFI Application Toolkit.*

6.4 PPP Support

PPP network support is available in the Toolkit network stack. If PPP support is not desired, it can be conditionally compiled out of the TCP/IPv4 network stack by setting the makefile macro **PPP_SUPPORT** in **\protocols\tcpipv4\tcpipv4.mak** to **NO**.

The driver that manages the serial port can be found in **\protocols\pppd**. It is a port of the FreeBSD daemon PPPD. It is implemented as an EFI driver that must be loaded after the tcpipv4.efi protocol is loaded. Refer to the PPPD manual page, which can be found through the **libc.html** index, for usage information.

NOTE

There is a bug in the **SERIAL_IO** *protocol in the pre-0.99.12.29 version of the EFI Sample Implementation that will prevent PPP from operating successfully. You must use EFI release 0.99.12.29 or higher to use PPP support.*

7 Known Limitations, Issues, Caveats, and Bugs

7.1 EFI Watchdog Timer Interaction

The EFI watchdog timer was enabled starting with EFI Sample Implementation release 0.99.12.29. Platform BIOSes based on release 0.99.12.29 may enable the EFI watchdog timer whenever EFI **`BootServices->StartImage()`** is called. For these systems, the watchdog timer is *enabled each time an EFI image is started*. Platform BIOSes based on the EFI Sample Implementation release 0.99.12.31 and later enable the watchdog timer *only prior to starting an image from the EFI Boot Manager*. In either case, the watchdog timer must be disabled within five minutes, or EFI will reset the system.

This release includes support to disable the watchdog timer when **`InitializeLibC()`** is called or an EFI Shell (built with this Toolkit) is invoked. For compatibility with platform BIOSes based on release 0.99.12.29, this Toolkit release also disables the watchdog timer when **`InitializeShellApplication()`** is called. This additional support is provided because the Shell included with BIOSes based on release 0.99.12.29 will not disable the watchdog timer. (Note that **`InitializeLibC()`** and **`InitializeShellApplication()`** are automatically called through the **`LIBC_Start[AU]`** and **`LIBC_Start_ShellApp_[AU]`** entry points.)

To run without fear of the system resetting on systems with platform BIOSes based on release 0.99.12.29, applications or protocols/drivers only need to be relinked with the libraries that are built from this Toolkit. However, if you build applications that do not make any of these calls, you must disable the watchdog timer yourself. The semantics for disabling the watchdog are as follows:

```
BS->SetWatchdogTimer(0x0000, 0x0000, 0x0000, NULL);
```

Additionally, binaries built with this release of the Toolkit and that run in the **`bios32`** environment with EFI versions earlier than 0.99.12.29 will cause an illegal opcode violation.

If you wish to run this release of the Toolkit with a version of EFI that does not support the watchdog timer or you wish to remove the features described above, you may modify the following to remove the code to disable the watchdog timer:

- **`\lib\libc\efi\init.c:InitializeLibc()`**
- **`\lib\libefishell\init.c:InitializeShellApplication()`**

7.2 Build Environment

The makefiles distributed with the Toolkit take a “big hammer” approach to include file dependencies. Basically, all files in a target’s include search path (e.g., **`-I include/bsd`**) are made dependencies of the target instead of just the files that the target, or the individual objects of the target, actually includes. Therefore, changing an include file will cause all targets that reference the containing directory to recompile.

A build for Itanium processors may produce the following warning, which can be safely ignored:

```
lib/libefi/runtime/lock.c
#pragma RUNTIME_CODE(RtAcquireLock)  warning #255: text segment
already specified

lib/libm/common_sources/ert.c
   if (ax < 1.0e-308)  warning #239: floating point underflow
```

You can also ignore the following:

- All warning LNK4001 messages
- All warning LNK4221 messages
- All warning A3201 messages

In addition, the Intel C++ Compiler 7.1 for Windows* will warn about certain **`printf`** format string options. These warning can be safely ignored.

Starting with EFI Application Toolkit release 0.80, applications linking to **`libc`** no longer are required to link to **`libefi`** and **`libefishell`**. As a result, applications can no longer assume that the **`libefi`** initialization routine **`InitializeLib()`** or the **`libefishell`** initialization routine **`InitializeShellApplication()`** will be made by **`libc`**. If required, the application must explicitly initialize these libraries.

If your program uses large amounts of automatic variable storage, the unresolved symbol **`__chkstk`** may be generated. This function is intrinsic to the Microsoft compiler and is out of the control of the EFI Application Toolkit build environment. The only known way of preventing the compiler intrinsic is to either declare large automatic storage as static or change the program to dynamically allocate and release the needed storage.

7.3 Manual Pages

HTML manual pages are generated from the original FreeBSD NROFF source. In many cases, manual pages document features and interfaces that have not been implemented by the EFI Application Toolkit. The **`libc.html`**, **`libsocket.html`**, **`libm.html`**, **`libdb.html`**, and **`libtty.html`** indexes in **`\doc\man`** accurately list all supported interfaces.

7.4 Libc

The standard C library attempts to provide a familiar programming environment that enables rapid porting of existing tools and applications. In most cases, it does a very good job of meeting this goal. However, there is not always a one-to-one mapping between EFI and the FreeBSD services on which **`libc`** is based. The following enumerates some of those differences.

7.4.1 Math Library

Because **SDK_LIBPATH** is removed from the build environment, the Toolkit uses its own math functions in the new math library (**\lib\libc\i386\math**, **\lib\libc\ia64\maths**). So when writing applications using the Toolkit, the corresponding math operation must be done by the new math functions. The added math functions are:

Table 7-1. Math Functions Added in EFI Toolkit

Function	Description
LIBC_LShiftU64	Left shift 64 bits by 32 bits and get a 64-bit result.
LIBC_RShiftU64	Right shift 64 bits by 32 bits and get a 64-bit result.
LIBC_MultU64x32	Multiply unsigned 64 bits by 32 bits and get a 64-bit result.
LIBC_DivU64x32	Divide unsigned 64 bits by 32 bits and get a 64-bit result.
LIBC_MulU64x64	Multiply two unsigned 64-bit values.
LIBC_DivU64x64	Divide unsigned 64 bits by 64 bits and get a 64-bit result.
LIBC_DivS64x64	Divide signed 64 bits by 64 bits and get a 64-bit result.

7.4.2 File Paths

File path separators can be either '**/**,' '****,' or mixture of the two. The C library will convert all separators to the'****' convention required by the **SIMPLE_FILE_SYSTEM** protocol.

7.4.3 Predefined Device Names

In the C library environment, access to any device must be performed through a file descriptor, which is obtained through a successful call to **open()**. To facilitate this, **libc** maps the following device names:

Table 7-2. Math Functions Added in EFI Toolkit

Device Name	Description
consolein	Maps to **EFI_SYSTEM_TABLE.ConIn**.
consoleout	Maps to **EFI_SYSTEM_TABLE.ConOut**.
consoleerr	Maps to **EFI_SYSTEM_TABLE.StdErr**.
default	Maps to the file system from which the application was loaded.

In addition, the C library will provide a mapping for all devices identified in the EFI variable store with the vendor GUID:

```
{ 0x47c7b225, 0xc42a, 0x11d2, 0x8e, 0x57, 0x0, 0xa0, 0xc9, 0x69,
0x72, 0x3b }
```

This is the GUID used by the EFI Shell to map the file system name (e.g., **fs0:**, **fs1:**) to the EFI device path for that file system. The data portion of the variable is assumed to be an appropriate binary device path.

During initialization, **libc** creates the standard file descriptors and FILE streams for **stdin**, **stdout**, and **stderr**. The **stdin** device is "**consolein:**." The **stdout** device is "**consoleout:** ." The **stderr** device defaults to "**consoleout:**." Sending both **stderr** and **stdout** to the same EFI console output device insures that POSIX programs, which routinely write to the **stderr** device, would get the desired result when the EFI configuration has these output streams directed to different devices such as the video display and serial port.

You can force **libc** to map **stderr** to "**consoleerr:**" by creating an EFI environment variable named **USE_STDERR_DEV** and setting it to *Y* or *y*. One can also gain explicit access to the EFI **StdErr** device with the following:

```
open( "consoleerr:", O_WRONLY );
```

The C library provides a mechanism for an application to define and hook additional device mappings into the general file descriptor mechanism. For more information, see the **libc** manual pages for **_LIBC_MapProtocol** and **_LIBC_MapDevice**.

7.4.4 Memory Allocation

Libc will automatically close all open file descriptors and free all memory that was allocated through **libc** allocation routines when an application calls **exit()** or returns from **main()** if it uses the **_LIBC_Start _[AU]** or **_LIBC_Start_Shellapp_[AU]** image entry points. If an application needs to allocate memory that must remain allocated after returning to the Shell, it should directly call EFI memory allocation services.

The family of **malloc()** routines are implemented as a thin veneer over EFI memory allocation services. As such, the manual page regarding **malloc()** is largely incorrect. None of the environment variables controlling **malloc()** behavior in FreeBSD are supported.

7.4.5 ASCII vs. Unicode

EFI is a Unicode-based system. Traditionally, most **libc** interfaces and ported programs assume ASCII. **Libc** attempts to hide this difference from the user by converting console output to Unicode and console input to ASCII. However, it is possible for an application to inadvertently sidestep these efforts. The most common case is using the **write()** or **fwrite()** calls on the **stdout** file descriptor and using **read()** or **fread()** on the **stdin** file descriptor. In these cases, the steam is treated as a regular file and no translations are attempted. Therefore, doing the following will not produce the intended result:

```
fwrite( "hello world", 1, 11, stdout );
```

The following two lines, however, would produce the intended result:

```
fwrite( L"hello world", 2, 11, stdout );
fputs( "hello world", stdout );
```

7.4.6 Getenv/putenv

Along the ASCII versus Unicode lines, the C library supplies **`getenv()`** and **`putenv()`**. At **`libc`** initialization, all EFI variables are enumerated. If the data portion of the variable appears to be either a well-formed Unicode or ASCII string, both the variable name and data are converted to ASCII and stored in an environment table that is local to **`libc`**. If there are two EFI variables with the same name but different vendor GUIDs, from a **`libc`** point of view, these will be seen as duplicates and the most recently found variable will overwrite any existing value.

In addition, Unix does not allow a child application to export an environment variable into the parent's environment. This implementation retains this behavior. Therefore, if an application adds or changes an environment variable, it will not be seen by the spawning application, which is typically the EFI Shell.

7.4.7 Stat/fstat/lstat

The **`stat`** system calls fill in the fields of a **`struct stat`**. In the EFI Application Toolkit implementation, only some of the fields in the **`stat`** structure are filled in with valid information. The following fields are filled in with valid information:

- **`st_mode`**
- **`st_nlink`**
- **`st_size`**
- **`st_blksize`**
- **`st_atimespec`**
- **`st_mtimespec`**
- **`st_ctimespec`**

All other fields in the **`stat`** structure are set to zero by the **`stat`** call, and their values are not valid.

Because the EFI operation to get the file system block size can be very slow, two undocumented calls have been added that return a compile-time fixed block size instead of actual block size. These calls are **`_faststat`** and **`_fastlstat`**. The semantics of these calls are equivalent to **`stat`** and **`lstat`** respectively.

7.4.8 Time

FreeBSD time functions use a variety of database files that allow a system to provide detailed information about time attributes for its geographical location. The default locations for these files are the **`\etc\localtime`** and **`\usr\share\zoneinfo`** directories. Although technically supported in the **`libc`** implementation, the Toolkit does not supply the FreeBSD tools that are required to build these time database files. In the absence of these files, **`libc`** will assume that the system is running on Universal Coordinated Time (UTC). This is an acceptable default because it will not apply an offset to the time reported by EFI, which is generally set to local time. In addition, time will not be adjusted during daylight savings time periods.

7.4.9 Unlink

On Unix systems, a program may unlink (remove) an open file yet continue to perform read and write operations on the open file descriptor. When the last file descriptor is closed to an unlinked file, it is removed from the file system. This typically is used on temporary files where the applications wants the file removed when it exits. The **libc** implementation of unlink retains this behavior. However, if an EFI program unlinks an open file but does not exit gracefully (e.g., crashes or does not use the **_LIBC_Start_[AU]** or **_LIBC_Start_Shellapp_[AU]** entry points and terminates with open file descriptors without calling **exit()**), the unlinked file will not be removed.

NOTE

*If a file is opened with read-only access (**O_RDONLY** or "**r**") and is not closed before an unlink operation is performed, unlink will fail.*

7.4.10 Line Input

Currently, **libc** only provides buffered line-oriented input. Characters will not be returned until the enter key has been struck. The backspace key will move the cursor to the left by one position and delete the preceding character from the line buffer. However, the deleted character is not removed from the video display.

7.5 Libm

A call to **InitializeLibM()** may be made before using any **libm** routines. Currently, this function sets the FPU mode to IEEE 754 format in the **bios32** environment so the math functions will perform the same in the **bios32** and **nt32** environments. (NT uses IEEE 754.) However, future releases may add other initialization actions.

The manual page for **libm** is missing for this release. The prototype is defined in **\bsd\include\atk_libm.h** as follows:

```
int
InitializeLibM(
EFI_HANDLE          ImageHandle,
EFI_SYSTEM_TABLE    *SystemTable
);
```

The values for *ImageHandle* and *SystemTable* are ignored and may be **NULL**.

7.6 Python

Python searches for library modules in the current directory and then in directories that are specified in the environment variable **PYTHONPATH**. If this variable is not set or if the file was not found, it will search the default search path, as follows:

```
.;/Toolkit/Python1.5;/Toolkit/Python1.5/lib.
```

Python will "compile" library scripts the first time they are referenced. This will produce a **.pyc** file from the associated **.py** file. The compile process is done at runtime and can take quite a while. Python does not present an indication to the user that a compile is taking place, which may lead the user to believe that Python has hung the system. Please provide plenty of time for the compile process to complete before assuming the system has hung. If all referenced library modules have been precompiled, Python will provide a prompt within a few seconds. Note that Python **.pyc** files are processor independent and are portable across platforms. Therefore, once compiled, they may be moved from system to system to reduce the overhead of first-time compilation.

7.7 LOAD command

EFI protocols and drivers that are dynamically loaded with the EFI Shell **load** command must make a copy of the EFI System Table before exiting. The EFI System Table allocated by the **load** command is freed once it exits back to the EFI shell. When that memory is eventually reused, it will result in the protocol referencing corrupted EFI System Table pointers. The following fragment provides a possible workaround:

```
EFI_STATUS EFIAPI
DriverInitialization(
  EFI_HANDLE        *ImageHandle,
  EFI_SYSTEM_TABLE  *SystemTable
  )
{
      //
      //  Workaround loaded protocol bug
      //
      static  EFI_SYSTEM_TABLE  BackupSystemTable = *SystemTable;
      SystemTable = &BackupSystemTable;

      ...
      ...
}
```

The **LOAD** command has been augmented to pass the full path name of the protocol/driver and current working directory in the *LoadOptions* field of the **EFI_LOADED_IMAGE** protocol. A protocol/driver that wishes to take advantage of this information must copy it before returning from the EFI entry point.

7.8 FTP Client

There is a known issue with some FTP servers that will not let the same socket number be used for the data connection from the server back to the client for 240 seconds. In some environments, the FTP client may be restarted in less time, and the FTP client will fail to function and will return FTP error 425. To avoid this issue, the FTP client must be started with the **-p** switch:

```
ftp -p <ftp host>
```

If the FTP client is used from a Python script, then the following lines can be used in a Python script file to work around this same issue:

```
ftp_connection = FTP(<ftp_path>)
ftp_connection.set_pasv(1)
```

7.9 TCP/IP Stack

There is a limitation when using the TCP/IP stack. When a user wants to use the TCP/IP stack to develop applications that run on a TPL higher than **TPL_APPLICATION** and use the blocking mode to receive and send data, the user must guarantee that the piece of code that does the blocking mode receive and send data is not re-entrant.

The TCP/IP stack that is currently in this release does not match the full *EFI 1.10 Specification* in its binding to the SNP3264 protocol interface (follows the BSD driver model). Consequently, executing the **reconnect** or other commands that assume that the TCP/IP driver will stop when the UNDI and SNP3264 driver are stopped may cause unpredictable behavior because the TCP/IP and **Dhclient** drivers continue to run (the system hangs). You can work around this problem by either manually locating the handles and disconnecting the TCP/IP and **Dhclient** drivers or by running the **Nunload** command that was added to the Toolkit. This command has been added to ensure that the TCIP/IP and **Dhclient** drivers are disconnected and stopped. This command can be scripted or executed manually from the EFI Shell before doing a **reconnect -r** command or booting to an operating system that requires the LAN to be in a quiescent state.

Running TCP/IP Stack and Dhclient on a Multi-NIC System

To be able to bind **Dhclient** and TCP/IP to any network connection or card in the system, the **loadarg** command was added to be able to pass in alternate network interfaces for **Dhclient** and TCP/IP to bind to. **Dhclient** was altered to allow the passing-in arguments to choose which network interface (UNDI and SNP3264 interface) in the system to use. The reason for allowing them to choose was to avoid some network interfaces that were dedicated to system server management or console redirection network ports.

8
Changes from Release 1.02.12.38 to Release 1.10.14.62

Table 8-1 lists the changes that were made to the EFI Application Toolkit from release 1.02.12.38 to release 1.10.14.62.

Table 8-1. Changes to Toolkit from Release 1.02.12.38 to Release 1.10.14.62

Area of Toolkit	Changes
Build Environment	• Added **EFI_APPLICATION_COMPATIBILITY** to **build.cmd** to support multiple compatibility modes for applications. • Removed the Microsoft* library path variable **SDK_LIBPATH** from the build environment, which means you can now use any version of Visual Studio without modifying makefiles. • Modified **build\sal64\sdk.env** to use the Microsoft C/C++ Optimizing Compiler Version 13.10.2240.8 for IA-64 (available in the Windows Server 2003 DDK, Build 3790). • Modified the **build\sal64\sdk.env** to use the Intel C++ Compiler 7.1 for Windows*. • Removed all the **make.inf** files from the source tree. • To support the EFI Toolkit 1.02 compatibility, building the Toolkit from the build tip directory (**build\nt32**, **build\bios32**, **build\sal64**) is no longer supported.
EFI Include Files	• Added the latest EFI 1.10 Sample Implementation (version 1.10.14.62) header files to **\include\efi110**. EFI Toolkit applications can now use the full EFI 1.10 protocols and services. However, read section 4.3 on compatibility and set the **EFI_APPLICATION_COMPATIBILITY** variable in **build.cmd** before building your application. • Added **\lib\libefi\systable.c** into the source tree to be compatible with the EFI 1.10.14.62 Sample Implementation.
EFI Shell	• Removed the **EfiShell** directory from the source tree as of this release. The EFI Shell is available in the EFI Sample Implementation.

continued

Table 8-1. Changes to Toolkit from Release 1.02.12.38 to Release 1.10.14.62 (continued)

Area of Toolkit	Changes
Cmd	• Added a full-screen text editor (edit). • Added a **loadarg** command to the source tree to load EFI drivers with parameters. • Added a **Nunload** command to turn off and unload the **Tcpipv4** and **Dhclient** LAN stack drivers. If these drivers were left on, these non–EFI 1.10–compliant drivers may hang the EFI 1.10 system if **reconnect** is executed or the system is booted to an OS. Run **Nunload** before running **reconnect -r** or booting to an OS. • Removed the **passwd** command from the source tree. • Fixed a bug where **loadarg.efi** did not handle parameter correctly. • Fixed a bug in **Mptest** (Itanium architecture only) that did not test all associate processors. The default code assumes a maximum of four processors in a system. • Fixed a bug where **which.efi** did not handle parameter correctly. • Fixed a bug where **Mkramdisk.efi** needed **ramdisk.efi** to be located in the root directory. **Ramdisk.efi** now will accommodate RAM disk sizes down to 2 MB (the previous version would allow minimum RAM disk sizes down only to 5 MB). The maximum RAM disk size (assuming memory is available) is 512 MB.
Libc	• Removed RSA Message Digest library (**\lib\libmd**) from the source tree. • Fixed a bug where **Fileio.c** did not use **RootDir->Close()** to close the **RootDir**. • Fixed memory leaks in **Libc** initialization. • Removed C++ support library (**libcpp**) from the source tree. • Removed authentication protocol from the source tree. • Removed **\lib\libc\msft** from the source tree. The Toolkit now uses its own math library for math operations.

continued

Table 8-1. Changes to Toolkit from Release 1.02.12.38 to Release 1.10.14.62 (continued)

Area of Toolkit	Changes
Networking	• Removed the FTP server (**\protocols\ftpserv**) from the source tree. • Added a **Nunload** command to turn off and unload the **Tcpipv4** and **Dhclient** LAN stack drivers. If these drivers were left on, these non–EFI 1.10–compliant drivers may hang the EFI 1.10 system if **reconnect** is executed or the system is booted to an OS. Run **Nunload** before running **reconnect -r** or booting to an OS. • Fixed a bug where **Tcpipv4.efi** caused a system hang on Intel® Server Platform SR870BN4 in the EFI 1.10 environment. • Fixed a bug where **Pktxmit.efi** would cause other network application to never work. • Fixed a bug where **Tcpipv4.efi** caused a system hang in the EFI 1.02 environment (**Ia-32Emb**) • Fixed a bug where loading **Dhclient.efi** twice would cause a system crash. • **Tcpipv4** and **Dhclient** now support unload. • Fixed a bug where **ping.efi** displayed the round-trip time incorrectly. • Fixed a bug in the socket library where **sendto()** could not work on a connected DGRAM socket. • Fixed a bug in **Tcpiv4** that affected FTP file transfers where the data transfer rate would decrease continuously as more data was transferred. • Fixed a bug where the FTP client application took "**fs0:**" as part of the file name.
Python	• Fixed a memory leak bug. • Fixed a bug where **python.efi** caused memory corruption. If any files were compiled with the old Python (**.pyc**) files, they should be recompiled with this new version.
Protocols	• Fixed a bug in the MP Protocol (Itanium architecture only) that did not allow all associate processors to be registered with the MP Protocol. This fix also requires that the SAL implementation correctly implements (has) the SAL rendezvous procedure call and that it works for all associate processors. • Removed IPMI from the source tree.
Debugger	• Removed the **Debugger** directory from the source tree. The **Debugger** in the Toolkit is no longer supported.
Binaries	• Added binaries to the package, starting from this release. The compiled ***.efi** files from each tip are placed under **\Binaries\Bios32**, **\Binaries\NT32**, and **\Binaries\Sal64**.