EFI Application Toolkit Release Notes

Release 2.0.0.1

1.0 Introduction

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 is compatible with UEFI Specification 2.0, EFI specifications 1.10 and 1.02 and has been tested with Release 1.10.14.62 of the EFI Sample Implementation and platforms containing the EDK versions found on Tianocore.org. The portability libraries were derived from the FreeBSD 3.2 source distribution.

These release notes contain the following information:

- System requirements
- Package contents
- Installation
- Build environment
- Directory structure overview
- Constructing a target makefile
- Network setup and configuration
- Known limitations for this release
- Changes history

Migration notes to Tianocore.org

This version of the toolkit has been updated for the X64 build environment which will allow the toolkit to be run on systems compliant with the UEFI 2.0 specification. This package has also been migrated from the Intel EFI website http://developer.intel.com/technology/efi/toolkit_overview.htm to the Tianocore.org site to allow open participation by other developers interested in expanding the tools and utilities available for UEFI systems. The EFI shell program was moved out of the toolkit and is now a separate project on Tianocore.org.

2.0 System Requirements

This release of the EFI Application Toolkit requires the following software on an IA-32 based system and will cross compile to the desired target platform designated by the build directory.

Software Requirements:

- Microsoft* Windows XP* operating system
- Microsoft* Visual C++* Version 6.0 or 7.0 or 7.1 (Microsoft Visual Studio .NET 2003 recommended)
- Microsoft* Windows Server 2003 SP1 DDK
  (found at http://www.microsoft.com/whdc/DevTools/ddk)
- 120MB of free disk space for EFI Application Toolkit source
- For the IA-32 environment, 140MB of free disk space for build binaries and intermediate files.
- For the Itanium™ environment, an Itanium™ compiler is required along with 200MB for build binaries and intermediate files. The compiler included in the Windows Server 2003 SP1 DDK 3790.1830 is recommended.
- For Intel 64(X64) UEFI environments the Microsoft Windows Server 2003 SP1 DDK is required

3.0 Package Contents

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

- Full screen Unicode/ASCII text editor
- Full screen file/memory hex editor
- Python scripting language interpreter
- Authentication Protocol
- 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
- IPMI protocol driver
- Multi-processor test support protocol driver (Itanium only)
- 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 Release notes (this document)
- Unix style manual pages for the compatibility libraries and commands
- Design documentation for TCP/IPv4, IPMI, locale, MP, authentication protocols.
- Design documentation for SMBIOS library.

- EFI build tools (binaries only)
- Binaries for each of the toolkit target build environments:
  IA32, EM64T(X64), Itanium™ or IPF (SAL64) and NT32

4.0 Installation

Release 2.0.0.1 of the EFI Application Toolkit is distributed as a single zip file. When unpacked, the source and documentation occupy approximately 40MB of disk space. Depending on the processor target, binaries built from the source (both intermediate and final targets) require an additional 140MB to 340MB of disk space.

The toolkit has been built and tested in the IA-32, Intel 64(X64) and Itanium™ environments using the following compilers:

- IA32 - Microsoft* Visual Studio* .NET 2003 (7.0 and 7.1)
- Itanium – Microsoft Windows* Server DDK SP1 compiler version 14.00.40310.39 for IPF from build 3790.1830
- Intel 64(X64/EM64T) - Microsoft Windows* Server DDK SP1 compiler version 14.00.40310.41 build 3790.1830

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 platforms must use NMAKE.EXE and LINK.EXE and LIB.EXE from the Microsoft Windows* Server SP1 DDK.

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.

5.0 Building EFI Application Toolkit Release 2.0.0.1

The EFI Application Toolkit supports 4 build environments for different targets. Each is given a hosting directory within the tree. These are:

- build\nt32 - The NT EFI emulator environment
- build\bios32 - IA-32 EFI Standard BIOS environment
- build\sal64 - Itanium™ EFI environment
- build\em64T - Intel 64(X64) EFI environment

The nt32 environment is provided as a debug and learning tool which is available from Tianocore.org. All toolkit binaries built under the bios32 environment are suitable to run in any IA-32 EFI/UEFI environment. X64/EM64T binaries will also run on EM64T/X64 UEFI/EFI system or on any system booted from the UOL boot image (systems booted of f of the UEFI image found on Tianocore.org).

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 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. The following variables must be set in an NT command window environment before invoking a toolkit component makefile.

SDK_BUILD_ENV	- Set to the target build environment (em64t, bios32, nt32, or sal64)
SDK_INSTALL_DIR	- Set to the root of the EFI Application Toolkit source tree i.e. c:\toolkit\EFI_Toolkit_2.0 if unzipped to c:\toolkit
EFI_APPLICATION_COMPATABILITY	- Set to EFI_APP_MULTIMODAL for both EFI 1.10 and 1.02 compatability - Set to EFI_APP_110 for EFI 1.10 compatability -Set to EFI_APP_102 for EFI 1.02 compatability
EFI_DEBUG	-Set to YES for compilation with and error console output on or -Set to NO for no debug or error console output

Due to the available Itanium™ compilers, you must review and edit the sdk.env file in the build/sal64 directory if you are building for that environment. There are two variables that point to the root tools directory for either the Microsoft* or Intel® compilers. These are **MSSdk** and **_IA64SDK_DIR** respectively. The sdk.env is setup by default for the Windows DDK SP1 Itanium compiler tools with debug turned on and optimization turned off. To enable optimization for either compilers will require editing of the compiler variables which are commented out. The Intel compiler options are also commented out.

Before building the toolkit the path must be set to the Microsoft Visual Studio tools (ie executing vsvars32.bat/vcvars32.bat or the build must be run from a Visual Studio command prompt.

The following macros are also required but are setup in the sdk.env file. If the default locations do not meet your needs, edit sdk.env and set them to the appropriate values.

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 Studio .Net 2003 installation where the toolkit source was placed in C:\Toolkit and the build environment was for em64t:

Vsvars32.bat (copy this from your visual studio or run it from the visual studio command prompt)

```
SDK_BUILD_ENV=em64t
SDK_INSTALL_DIR=C:\Toolkit\EFI_Toolkit_2.0

Edit and update MSSDK in sdk.env in the build target dir
In this case for em64t sdk.env would be at
C:\Toolkit\EFI_Toolkit_2.0\build\em64t\sdk.env
If the DDK sp1 was installed at c:\winddk MSSDK would be:

MSSdk        = C:\WINDDK\3790.1830\bin\win64\x86\amd64

Nmake        to build all binaries for the em64t target
```

Default values for SDK_BUILD_DIR and SDK_BIN_DIR are:

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

Building only part of the toolkit

If it is desired to only build part of the toolkit (ie cmds or protocols) then the 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
```

or at a parent level which will invoke all sub-component makefiles:

```
cd cmds
nmake –f cmds.mak
```

Invoking nmake –f sdk.mak at the root of the toolkit source tree, will do a complete build of the EFI Application Toolkit.

6.0 EFI Application Toolkit Directory Structure

This section provides an overview of the toolkit directory structure. The root of the toolkit source tree contains the following primary directories. Each of these will be covered in more detail below:

apps	EFI sample applications.
binaries	Pre-compiled binary executables for each target build environment
build	Build environment directories
cmds	Ports of FreeBSD commands and utilities
doc	Documentation for the EFI Application Toolkit
etc	**example free bsd lan stack setup and config files**
include	Common include files
lib	Common libraries both EFI and Posix
protocols	Toolkit supplied EFI protocols and drivers

Apps directory structure

Applications in this directory provide working examples for using native EFI services and the EFI support library.

exitboot	ExitBootServices test
NtFloppy	NT emulation utility to enable/disable access to the A: device under emulation
osloader	An OS loader example
pktsnoop	Example EFI Simple Network Interface example that reads packets
pktxmit	Example EFI Simple Network Interface example that sends packets
rtdriver	Example of driver unload interface
rtunload	Example for requesting a driver unload
salpaltest	Example for calling sal or pal calls (Itanium only)
scripts	Test scripts
test	Example for locating and printing EFI Loaded Image protocol data
test2	Example of finding the block device of filesystem that contained 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
testva	Example of using virtual address call
testvrt	Example of using virtual address call under the runtime environment

Build directory structure

The build directory provides environment specific configuration files used during the build process and the source for support tools 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.

em64t	**Contains the sdk.env file for EM64T/X64 build environment**
bios32	Contains the sdk.env file for a non-emulated EFI environment
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™ environment
tools	Contains build support tools. Currently, the only tool required is *fwimage*.

Cmds directory structure

The programs in this directory provide working examples for using the Posix C portability libraries of the toolkit. Some of these 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.

ed	Unix command line editor (ASCII only)
edit	**EFI full screen text editor**
ftp	Ftp client
hexdump	Dump files in varying display formats
hostname	Set the network name for the system
ifconfig	Configures network interfaces
loadarg	**shell program used to pass arguments to an EFI driver**
mkramdisk	Configures and installs RAM disk
mptest	Test program for multi-processor test protocol
nunload	**unload EFI program from handle database (if supported)**
ping	ping network diagnostic tool for tcpip network stack
python	Object oriented scripting language interpreter
route	Set network gateway for the tcpip lan stack
which	Identifies which file would have been executed had its argument been given as a command

Doc directory structure

This directory contains the documentation provided for this release. It is not as well organized and integrated as we hope to make it in future releases. Note that complete Python documentation in contained in cmds/python/documentation.

.	The root of the directory contains a copy of these release notes, EFI Developers Guide, and EFI Library 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. These man pages are included as reference from the original Freebsd port and have not been modified for the EFI implementation of the toolkit.

These 2 guides were included as historical reference for EFI 1.02 and EFI 1.10

EFI_DG	**Original EFI 1.10 developers guide for writing EFI applications and EFI os loader**

EFI_LIB	**Original EFI 1.10 library guide for library found in lib\efilib**
Relnote	**This release note**

EfiShell

The original toolkit contained the stand alone EFI shell program. This has since been updated and replaced by the Tianoshell project up on Tianocore.org. For applications that are meant to run in the EFI shell or Tiano shell environment the include and lib directory still use the original EFI shell definitions.

Include directory structure

This is the root of all common includes files that an application may use.

bsd	Includes from FreeBSD distribution. Associated with libc, libdb, libm, libsmbios, libsocket, libtty, and libz
efi	base EFI definitions based as well as those associated with libefi
efi110	**base EFI definitions when build for EFI 1.10 target**
efishell	Include files associated with libefishell

Lib directory structure

The lib directory contains the source for all common libraries that an application may link to. 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.

libc	Standard C library. Both ANSI and FreeBSD specific routines as well as Unix system call emulation.
libdb	Database access method library from Freebsd
libefi	General support library for EFI services
libefishell	Support library for shell applications
libm	Standard math library. 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

Protocols directory structure

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.

dhclient	DHCP client protocol use with EFI tcpipv4 lan stack
ipmi	EFI protocol driver for accessing the IPMI 1.0 system management bus
locale	Loadable locale protocols used with libc
mp	Multi-processor support protocol
pppd	PPP serial connection management driver
ramdisk	RAM disk protocol
tcpipv4	EFI protocol implementing a complete TCP/Ipv4 network stack

7.0 Creating an EFI Application Toolkit Makefile

The following 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 specified in *sdk.env*. This should be the first executed line of 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 it will be run with.

```
#
# 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 your application needs to link against. 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 it's 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
```

8.0 Network Setup and Configuration

Starting with the original EFI Sample Implementation support has been provided for Ethernet cards utilizing 16 bit UNDI options 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. The protocol stack in the toolkit only supports the EFI 1.10 and 1.02 stack and does not support the UEFI 2.0 lan stack.

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 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.

Name Resolution Configuration Files

The socket library will do name resolution either through a *hosts* 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>
```

where *<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 filesystem 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 filesystems, you may duplicate the */etc* directory and associated files to those filesystems.

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
```

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 for the toolkit EFI stack.

The DHCP client constructs configuration scripts and runs them when it is first loaded. Temporary scripts are placed in the root of the filesystem 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 protocols*./dhclient/client/scripts* source directory.

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 "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 be 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
```

and then run

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

to produce a version of dhclient.efi that runs in the same mode as previous releases.

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.

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 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 utilize PPP support.

9.0 Known Limitations, Issues, Caveats, and Bugs

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 5 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) 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 since the shell included with BIOSes based on release 0.99.12.29 will not disable the watchdog timer. (Note *InitializeLibC()* and *InitializeShellApplication()* are automatically called through the *LIBC_Start[AU]* and *LIBC_Start_ShellApp_[AU]* entry points.)

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

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

Additionally, binaries built with this release of the Toolkit and 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 wish to remove the features described above, you may modify *lib/libc/efi/init.c:InitializeLibc(), lib/libefishell/init.c:InitializeShellApplication(), and efishell/nshell/init.c:InitializeShell()* to remove the code to disable the watchdog timer.

EFI Shell Environment

This has been updated to the Tianoshell on tianocore.org. This is left here for comapatability with older EFI shells.

Starting with EFI Application Toolkit release 0.80, several enhancements have been made to the EFI Shell that are exploited by components of the Toolkit. Therefore, to get the full benefit, you must be sure to run the shell provided in release 0.80 or higher.

The EFI Shell provides most of its functionality through the *shellenv.efi* protocol. Once this protocol is loaded, any new instances of *nshell.efi* will use it. For this reason, you can not simply boot a system with an older shell then execute *nshell.efi* built from an updated release. In this case the old *shellenv.efi* will still be providing shell services.

The easiest way to prevent this is to make a boot options entry that will invoke the *nshell.efi* from this release. When started, *nshell.efi* will search, in order, the following directories for *shellenv.efi:* \, \efi, and \efi\tools. It must find a *shellenv.efi* binary that was built from this release to supply the services required.

Note that once a shell has been started, exiting back to the boot manager menu will not remove the *shellenv.efi* protocol. The next time *nshell.efi* is invoked from the boot manager, it will find that *shellenv.efi* is already loaded and will not attempt to load it from disk. This behavior means that

systems which boot directly to the internal EFI Shell can not use the new shell services. If this is the case for you, you should build components of this release with the *sdk.env* macro **OLD_SHELL** to 1.

Programs that use the _LIBC_Start_[AU] entry point will not receive quoted command line strings as a single argument in the argv array of main(). You can work around this bug by escaping the quotes with the shell escape character as follows:

```
ed ^"argument botch^"
```

This bug will be addressed in future releases.

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 (ie. –I include/bsd), are made dependencies of the target instead of just the files 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.

The *efishell, lib/libefishell* and *apps* source directories contain a file called make.inf. This is an artifact of when toolkit components were built using the EFI Sample Implementation tool suite. They are not used in the EFI Application Toolkit build environment but are retained for those environments that integrate sources from these directories into the EFI Sample Implementation source tree.

The Itanium™ build 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
```

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

In addition, the Intel® C/C++ Itanium™ compiler 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 is an intrinsic function of the Microsoft* compiler and is out of the control of the EFI Application Toolkit build environment. The only know 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.

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 EFI Application Toolkit. The libc.html, libsocket.html, libm.html, libdb.html, and libtty.html indexes in *doc/man* accurately list all supported interfaces.

EFI Developers Guide

The EFI Developers Guide (Efi_dg.doc) is the original guide from the EFI Sample Implementation. As such, the procedures for constructing a make.inf file do not apply. Refer to section 7.0 Creating an EFI Application Toolkit Makefile for the proper procedures.

Libc

The standard C library attempts to provide a familiar Posix 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 libc is based on. The following enumerates some of those differences.

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

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:

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 filesystem the application was loaded from

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 filesystem name (i.e. fs0:, fs1:) to the EFI device path for that filesystem. 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:

```
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, refer to the _LIBC_MapProtocol and _LIBC_MapDevice libc manual pages.

Memory allocation – Libc will automatically close all open files descriptors and free all memory 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.

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

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

will not produce the intended result. The following two lines would:

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

getenv/putenv – Along the ASCII vs 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 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.

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 fields filled in with valid information are:

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 filesystem 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* receptively.

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 location for these files is /etc/localtime and the /usr/share/zoneinfo directory. Although technically supported in the libc implementation, the toolkit does not supply the FreeBSD tools required to build these time database files. In the absence of these files, libc will assume the system is running on Universal Coordinated Time (UTC). This is an acceptable default since 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.

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 filesystem. 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 (i.e. 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 for with read-only access (O_RDONLY or "r") and not closed before an unlink operation is performed, unlink will fail.

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 one position and delete the preceding character from the line buffer. However, the deleted character is not removed from the video display.

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.

Python

Python searches for library modules in the current directory and then in directories specified in the environment variable **PYTHONPATH**. If this variable is not set or the file was not found, it will search the default search path which is ".;/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 pre-compiled, 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 prevent the overhead of first-time compilation.

LOAD command

EFI protocols and drivers that are dynamically loaded with the shell *load* command must make a copy of the system table before exiting. The 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 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 pathname of the protocol/driver and current working directory in the EFI_LOADED_IMAGE protocol *LoadOptions* field. A protocol/driver that wishes to take advantage of this information must copy it before returning from the EFI entry point.

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)
```

10.0 Changes from Release 2.0.0.1 to Release 1.02.12.38

Build Environment

- EM64T(X64) build tip was added so EFI toolkit binaries could run on X64 platforms
- Python was upgraded from 1.5 to 2.4.2 so more features could be supported for this interpreter
- IPMI KCS interface driver from 1.02.12.38 was added back to the code tree

Moved source to new website

- Migrated EFI toolkit package from Intel website to Tianocore.org

11.0

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.
- Modified the buildsal64\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.
- 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.

Cmd

- Added a full-screen text editor (edit).
- Added a **loadarg** command to the source tree to load EFI drivers with parameters.
- Removed the **passwd** command from the source tree.
- Fixed a bug where **loadarg.efi** did not handle parameter correctly.
- 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.

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**.
- Removed C++ support library (**libcpp**) from the source tree.

- Removed IPMI 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.

Networking

- Removed the FTP server (**\protocols\ftpserv**) from the source tree.
- 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.
- 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 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 a crash in EFI 1.10.

Debugger

- Removed the **Debugger** directory from the source tree.

Binaries

- Added binaries to the package, starting from this release. The compiled ***.efi** files from each tip are placed under **Binaries\em64t**, **\Binaries\Bios32**, **\Binaries\NT32**, and **\Binaries\Sal64**.

12.0 Changes from Release 1.02 to Release 1.02.12.38

Build Environment

- The LINK option /opt:REF was added to all the environments. This significantly reduces the size of the binaries.

EFI Include Files

- Added #pragme pack(1) to SMBIOS.h
- Updated the EFI_FIRMWARE_MINOR_REVISION from 33 to 38

EFI Shell Enhancements and Fixes

The following bug fixes and enhancements have been made to the EFI shell:

- Fixed the EFI Shell command CP to work with read only source files.
- Fixed the -r option of the EFI Shell command CP.
- Fixed error messages from the shell to be consistent when and attempt is made to write to write-protected media.
- Fixed the EFI Shell Command parser to work with absolute file paths starting with a '\'.

- Fixed the EFI Shell batch processing of command line arguments. If a command line argument of the form %n was referenced, but it was not passed in, the line of the batch script was truncated at the point where the argument was referenced.
- The EFI Shell Command LS would show the contents of the wrong disk after media for the current drive was removed. This was fixed.
- Fixed the help description of the EFI Shell Command GUID.
- Fixed some spelling errors in the EFI Shell Command ERR.
- Enhanced the EFI Shell Command DBLK to support zero blocks, pattern blocks, and verify blocks.
- Fixed the EFI Shell Command COMP to not generate an ASSERT() if file1 or file2 does not exist.
- Fixed the EFI Shell Command LOAD to return the correct return status of the load operation.

libc

- Fixed a bug in the JoinFilePaths() macro.

Networking

- Added driver unload support to the TCP/IPv4 driver.
- Fixed timer related bug when the EFI timer tick frequency was increased on some platforms.

Python

- Fixed 64 bit pointer issues that was affecting IPF platforms with memory above 4GB.

13.0 Changes from Release 0.90

Authentication Protocol

A simple authentication protocol *(auth.efi)* and a password demonstration program *(password.efi)* have been included in the Toolkit. The authentication protocol includes an interface for creating and deleting authentication credentials as well as validating an arbitrary ID/key sequence against a previously created credential. Refer the design document *doc/design/Authentication EPS.doc* for further information.

Build Environment

The INCLUDE environment variable is no longer required to build the EFI Application Toolkit.

The SDK_64_INCLUDE_DIR environment variable is required when building the debugger support components for the Itanium environment. This environment variable is not required for any other components of the Toolkit. The SDK_64_INCLUDE_DIR environment variable should be set to the root of the Visual C++* Itanium include directory currently provided in the Microsoft* Platform SDK.

Debugger Support

Support for remote debugging of an Itanium™ EFI application or protocol using the Intel® EDB debugger has been added to the Toolkit. The support includes three components: an EFI debug agent *(debug.efi)*, a debug support protocol *(debugsupport.efi)* used by the debug agent, and a proxy agent *(edbproxy.exe)* that is used to communicate between the EDB debugger and the debug agent. Refer the usage document *doc/design/debugusage.doc* for further information.

Note : The debugger is only supported with Intel EDB debugger and the Intel SDK 2.0 Compiler and has only undergone limited testing. It may not work with all EFI drivers and applications.

Note : that this support is considered preliminary and will not be finalized until the EFI 1.1 Specification is available. As such, the interfaces and code provided for remote debugging may change in subsequent releases.

EFI Shell Enhancements and Fixes

The following bug fixes and enhancements have been made to the EFI shell:

- EFI shell batch file processing endlessly loops if the last line of the script file does not end with a <LF>.
- Shell environment function to expand meta arguments destroys input string.
- Shell environment function to expand meta arguments does not correctly handle pathnames consisting only of a volume name.

General Utilities

The “which” utility has been added to the Toolkit. The utility *(which.efi)* searches the path looking for the specified file. The command also searches the path for a file with a “.efi” or “.nsh” extension added to the specified file name.

libc

Support to map block devices has been added to the file I/O subsystem. These mappings use the EFI shell names such as *blk0:*, *blk1:*, etc. Support for this feature can be turned off by removing the definition of MAP_BLOCKIO_DEVICES from the file *lib/libc/libc.mak*.

The following bug fixes have been made to libc:

- Argv[argc] not set to NULL as defined by ANSI C.
- ResolveFileName() does not correctly construct file names.
- ssize_t not defined for C++ applications.

libtty

The libtty serial protocol timeout period has been increased from the time to send two characters to the time to send one character plus 10 usecs. This change was required to provide reliable performance in environments with 2 msec timer ticks.

Networking

The following bug fixes have been made to the TCP/IPv4 network stack, related network libraries, and applications:

- Race condition in the connect() and accept() functions which could prevent connect() and accept() from returning once the associated events completed.
- Condition where an application would always get an errno value ENOBUFS when trying to send data.

PPP Authentication Support

Clear text password support (PAP) is now provided by default in PPP and does not require the use of libmd.lib and libcrypt.lib. However, CHAP support still requires the use of these libraries. CHAP support can be enabled by setting the makefile macro **LIBMD_SUPPORT** in *protocols/pppd/pppd.mak* to **YES**. Refer to the files *lib/libmd/readme.txt* and *lib/libcrypt/readme.txt* for information on how to obtain the source for these libraries.

PAP/CHAP secrets can be passed to the PPP driver. The user can be passed via *auth-user* and the secret can be passed via *auth-secret*.

The following bug fixes have been made to the PPP support:

- Assert in *pppd.efi* when calling core EFI GetTime() service.

Python

Python makefile *(cmds/python.mak)* has been enhanced so Python library targets are always rebuilt. This change has the side effect that Python will always be relinked even if no source code is updated.

RAM Disk Support

Support for a RAM disk has been added to the Toolkit. The RAM disk support consists of two major components: a RAM disk configuration program *(mkramdisk.efi)* and a driver which provides the actual RAM disk functionality *(ramdisk.efi)*. Additionally, a manual page has been included to describe the configuration settings for the RAM disk.

14.0 Changes from Release 0.80

Build Environment

The old style *make.env* files in the build directories (*build/bios32*, *build/nt32*, and *build/sal64*) have been removed.

The file *lib/libefishell/make.ini* was missing in release 0.80. This caused an EFI Sample Implementation build to fail when it was configured with a built-in version of the shell and shell applications. This file has been restore with release 0.90 of the Toolkit.

EFI Shell Enhancements and Fixes

A bug in the EFI shell would cause the redirection directive (> or >>) along with the filename to be passed as program arguments to libc programs that used main() as the entry point. This bug has been fixed.

A new *unload* internal command has been added to the EFI shell. This new command allows EFI protocols and drivers to be stopped and unloaded from the shell prompt. Note that an EFI protocol or driver must register their ability to be unloaded through the EFI **LOADED_IMAGE** protocol. If they have not made that registration, the unload command will report an "Unsupported" error message. Refer to *doc/efi_dg.doc* for usage details of this command.

Full Screen Editors

Many bug fixes have been made to both the Unicode/ASCII and hex editors (edit and hexedit). In addition, both now assume a file is in the current working directory when an explicit path is not specified.

PPP Support

PPP network support has been added to the Toolkit. Supporting PPP required adding additional code into the TCP/IPv4 protocol and providing a driver that manages the serial port and negotiates PPP connection parameters with the far end.

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 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 utilize PPP support.

Networking

Several bug fixes have been made to the TCP/IPv4 network stack and network related libraries and applications. They include:

- Applications written for libc would crash if loaded and run through PXE.
- The network stack could not be used if EFI booted through the PXE device.
- Closing a socket that was configured for asynchronous notification could cause the system to crash.
- The select(2) call could return a false indication that an event had occurred on a socket.

The dhclient.efi protocol is now unloadable.

Python

Python has been enhanced to support dynamic loading of modules. Past releases required any desired modules in *cmds/python/modules* to be compiled into the Python binary image. These modules, or any custom modules, can now be compiled as stand-alone binaries that will be dynamically loaded on demand.

The benefits to dynamic loading are lower overall memory foot print (when the full complement of modules are not needed) and the ability to write and add custom Python modules without integrating them in to the Python source tree.

The *cmds/python/modules/unimplemented/xxmodule.c* provides an example of hooks needed to write a loadable module. The toolkit distribution builds Python for loadable modules by default. However, static module linking is still possible. To do a static link, comment out the **DYNAMIC_PYTHON_MODULES** macro definition in *cmds/python/modules/Modules.mak*.

Dynamic Python modules are built with a .pym extension and are placed in the *build/<env>/bin* directory by default. They must be placed on the EFI target system so that they can be found using the same search path criteria as Python library scripts.

libc

The **F_SETFL** operation of fcntl(2) is now supported in libc.

libtty

Libtty provides partial tty and termios emulation for EFI serial devices. Refer to the manual page index *doc/man/libtty.html* for details.

IPMI Protocol Driver

The KCS protocol of the IPMI driver has been updated per the *Addenda, Errata, and Clarifications IPMI Specification v1.0, rev 1.1, Addendum Document rev 3, June 6, 2000*. This update may not be backward compatible with the previous KCS protocol. Use the IPMI driver from previous Toolkit releases if this is an issue in your environment.

The IPMI Protocol Driver is now unloadable.

libsmbios

The following fixes and enhancements have been made to the SMBIOS library:

- Fixed an Itanium alignment problems.
- Added support for the Rev 3 addendum to the IPMI spec for type 38.
- A change was made to the type 0 structure to support a variable number of extension bytes.
- Added the SMBIO_FreeStructure() call to be used to free all structures returned by the SMBIO_GetStructure() and SMBIOS_GetRawStructure() calls.
- Removed all libc dependencies of libsmbios. The only library dependency is libefi.
- Updated *doc/design/EfiIpmi EPS.doc* to reflect the above enhancements.

NtFloppy

An NT emulation utility has been added called NtFloppy to the *apps* directory. This utility allows applications running under NT emulation to access the floppy drive. The NT emulation environment can not supply true filesystem semantics for the drives it maps to the *bin* directory. However, the NT emulation environment will provide true semantics through the floppy drive making this utility useful in those situations.

To enable access to the floppy, build the application under the nt32 build environment. Bring up the NT emulation environment buy running nt32.exe. Boot the EFI Shell and invoke the command:

```
NtFloppy on
```

Kill the nt32 program, place a floppy in the A: driver, and restart nt32.exe. When EFI boots, it will find and map the floppy drive. This operation is persistent and will remain in force through subsequent runs of nt32.exe until floppy access is turned off. Floppy access can be turned off by running the command:

```
NtFloppy off
```

Floppy access will not be turned off until nt32.exe is stopped and restarted.

Note that once floppy access has been turned on, nt32.exe will not start successfully unless a floppy is in the drive.

11.0 Changes from Release 0.72

Build Environment

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.

The build process no longer leaves a .idb and .pdb file in the source directory after a build.

All make.inf files have been removed with the exception of those in the efishell and apps directories.

Manual Pages

The HTML manual pages generated from FreeBSD nroff source are now faithfully reproduced. However, they still have not been edited to remove features or interfaces that are not supported.

Libc

The following functions have been added to libc. Refer to the associated manual page for usage information.

_LIBC_Cleanup	globfree	sl_find	wfopen
_LIBC_Start_A	LoadImage	sl_free	wmkdir
_LIBC_Start_U	localeconv	sl_init	wopen
_LIBC_GetArgsFromLoadOptions	longjmp	StartImage	wrmdir
catclose	rename	system	wstat
catgets	setjmp	UnloadImage	
catopen	setlocale	utime	
glob	sl_add	utimes	

In additions to these functions, most ANSI wide character functions have been added to libc. Refer to the *doc/man/libc.html* index under the WCHAR.H heading for a complete listing.

The source code and manual pages for *btowc*, *fwide*, *mbrlen*, *mbrtowc*, *mbsinit*, *wcrtomb*, and *wctob* have been included but are not completely implemented. For this reason, they are not linked into the libc binary or included in the libc.html index.

Libdb

The Unix style database access method library has been ported to the Toolkit. Refer to the *doc/man/libdb.html* index for a complete listing.

Libz

The file compression/decompression library libz has been ported to the Toolkit. Refer to the *include/bsd/zlib.h* and the *doc/man/libz.html* index for a complete listing.

Locale Support

As a start to supporting internationalization (i18n) support to the Toolkit, a Locale Protocol has been defined that allows multiple locals to be dynamically loaded as EFI protocols. The simplified Chinese locale has been provided as a sample implementation. Refer to the design document "libci18n.doc" located in doc/design for further details.

Multi-processor Test Support

To facilitate testing and diagnostics in an Itanium™ multiprocessor system, an MP protocol has been defined and implemented. The protocol allows an EFI application to start and stop non-EFI based programs on specific non-boot processors. **This facility is not meant to provide a general multi-processor environment**. Refer the design document *doc/design/MPSpec0.9.doc* for further information.

New Utilities and Services

An FTP client and FTP server have been added to the toolkit. These are ports of the FreeBSD implementations. Refer to the *doc/man/libc.html* index for a pointer to their perspective manual pages.

Beta versions of two new full screen editors have been added. The directories *efishell/edit* and *efishell/hexedit* contain the source for these utilities. The *Edit* utility provides a full screen Unicode and ASCII text editor. The *Hexedit* utility provides a full screen editor for both files and memory. Use of these editors is fairly straightforward and there is currently no documentation available.

A DHCP client has been implemented as an EFI protocol. This is a port of the DHCP client supplied with the FreeBSD distribution. The implementation requires the use of the EFI Shell supplied in this release of the EFI Application Toolkit (0.80). Refer to section 8.0 of this document and the *doc/man/libc.html* index for further information on configuration and use of this new protocol.

Python

The standard Python modules *dbmodule*, *pcremodule*, and *zlibmodule* have been added to the Python implementation. In addition, the os.system method is now supported.

Protocols - general

All Toolkit protocols now return the EFI error code **EFI_ALREADY_STARTED** when attempt to load an existing protocol is made. This can be treated as a non-fatal error that can be distinguished from other fatal errors.

EFI Shell and Utilities

Virtually all EFI Shell utilities that can produce more than one screen full of information support a –b flag that will display a screens worth of information and then prompt for a carriage return to continue.

The *set* and *alias* commands now supports a –v option to define a volatile variable or alias that will be lost on the next system reboot. If the –v option is not spec, variables and alias are persistent and will survive system reboot.

Batch files may now contain blank lines and lines may be terminated with a new line (0x0a) or a DOS style carriage return – newline (0x0d 0x0a). Previous version of the EFI Shell would terminate a batch script when a blank line was encountered and could get confused when a carriage return (0x0d) was encountered at the end of a line.

12.0 Changes from Release 0.71

New source files

The following source file directories were added:

- apps/osloader/ia64
- build/sal64
- efishell/ver
- include/efi/ia64
- lib/libefi/ia64

Apps

The *file* command directory has been removed.

Includes

The following files have been moved:

efiipmi.h: from *protocols/impi* to *include/bsd*
smbios.h: from *lib/libsmbios* to *include/bsd*

13.0 Changes from Release 0.7

Libc

Current working directory – In release 0.7, all file path specifications were assumed to be relative to the root of the filesystem. Therefore, /etc/hosts and etc/hosts were equivalent. This restriction has been removed. An application started from the shell will inherit the current working directory of the shell. Any file path reference made in a C library call will be assumed relative to the current working directory unless it begins with a / or \ or, it contains a device name. Any file path that contains a device name (i.e. fs1:) is assumed to be a fully qualified path and the current working directory will not be applied.

The C library functions *getcwd(), wgetcwd(),chdir(),* and *wchdir()* are now supported. The *getcwd()* and *wgetcwd()* calls return the ASCII and UNICODE paths of the current working directory respectively. The *chdir()* and *wchhdir()* calls allow an application to change the notion of current working directory for the calling application. Refer to the C library manual pages for the semantics of these calls.

Filesystem name mappings – In release 0.7, the C library limited filesystem access to the filesystem that contained the application image (i.e. the EFI device path associated with the EFI_LOADED_IMAGE protocol for the application in question). Therefore, if the application was stored on the filesystem the EFI shell mapped to fs0:, only files on that disk can be addressed through libc. To further complicate matters, libc was unaware of shell name mappings and internally maps that filesystem to the name “default:”. If a device specification is not found in a file path, libc assumes it is default:. Therefore, default:/etc/hosts is equivalent to /etc/hosts and fs0:/etc/hosts would be unrecognized.

This restriction no longer applies. The C library now recognizes all EFI shell device mappings within a file path including the current working directory. If the environment the application is run from does not contain files system mappings, the file access behavior is identical to the way it was in release 0.7 where the default device is labeled “default:” and references the filesystem of the device the application was

loaded from. This situation can occur when an application is directly run from the EFI boot manager menu.

Libsocket

There was a bug in the *socket()* call that could result in an otherwise successful call to randomly return a failure indication (-1). This bug has been fixed.