LAUTERBACH
DEVELOPMENT TOOLS

UEFI Awareness Manual
TianoCore

MANUAL

UEFI Awareness Manual TianoCore

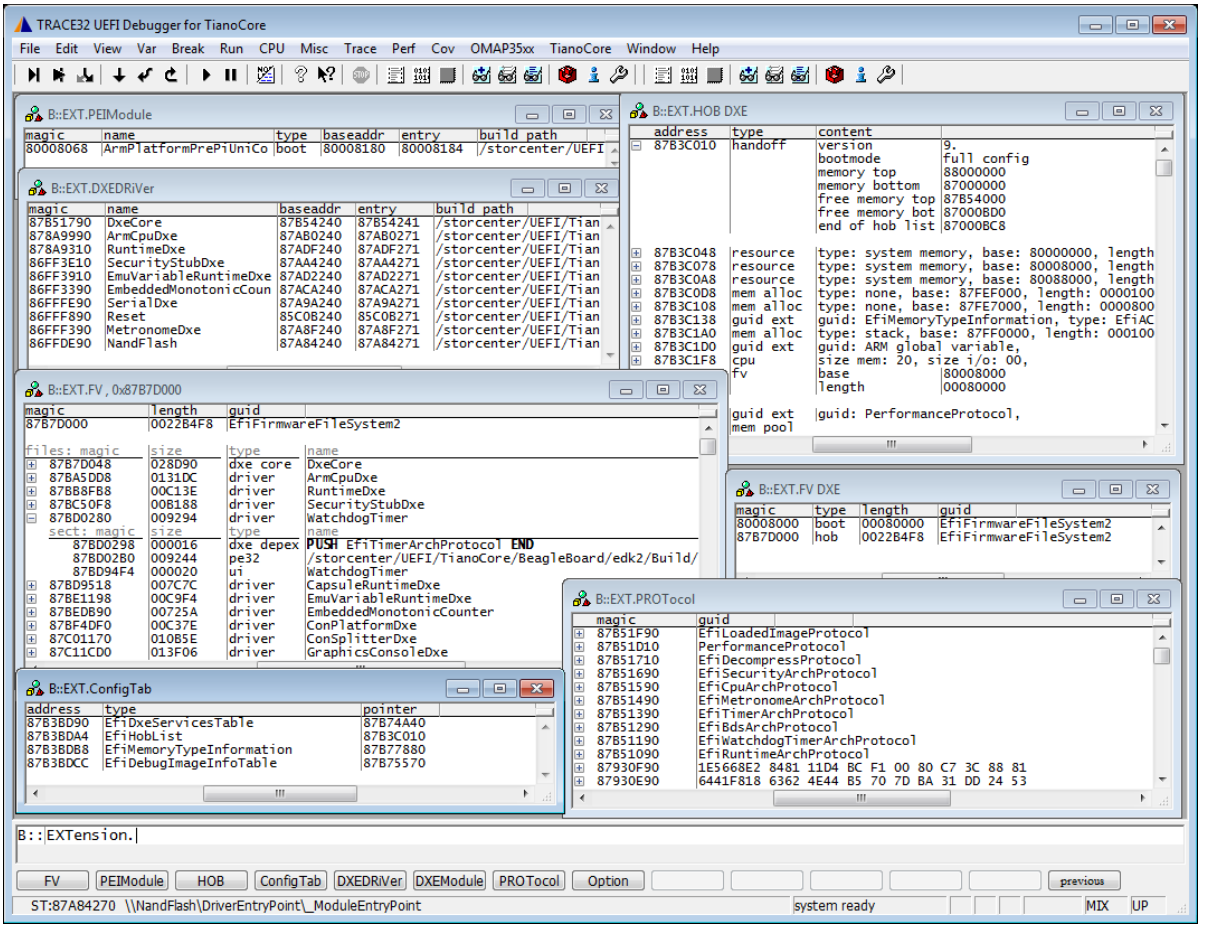
UEFI Awareness Manual TianoCore

Version 06-Jun-2024

History

28-Aug-18 The title of the manual was changed from “UEFI *<x>* Debugger” to “UEFI Awareness Manual *<x>*”.

Overview



The UEFI Awareness for TianoCore contains special extensions to the TRACE32 Debugger. This chapter describes the additional features, such as additional commands and debugging approaches.

Architecture-independent information:

- **"Training Basic Debugging"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **"General Commands"** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **"OS Awareness Manuals"** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.
- **"UEFI Awareness Manuals"** (uefi_<*x*>.pdf): TRACE32 PowerView can be extended for UEFI-aware debugging. The appropriate UEFI manual informs you how to enable the UEFI-aware debugging.

Supported Versions

Currently TianoCore is supported for the following versions:

- TianoCore on ARM32 and ARM64 architectures

Configuration

The UEFI Awareness for TianoCore is configured by loading an extension definition file called “tiano.t32” from the demo directory with the **EXTension.CONFIG** command. The command takes two parameters that specify the memory base address and size of the UEFI package. See the file *<board>*Pkg/*<board>*Pkg.dsc of your UEFI implementation. “PcdSystemMemoryBase” rsp. “PcdSystemMemorySize” are the needed values.

Additionally, load the “tiano.men” menu file (see “**TianoCore specific Menu**”) and configure the **Symbol Autoloader**.

ARM 32-Bit

A full configuration for ARM **32-bit** can look like this (the path prefix ~~ expands to the system directory of TRACE32.):

```
; Specify the memory base address and size,
; see <board>Pkg/<board>Pkg.dsc:
; PcdSystemMemoryBase = 0x80000000
; PcdSystemMemorySize = 0x08000000

; Load the TianoCore Awareness:
EXTension.CONFIG ~~/demo/arm/bootloader/uefi/tiano/tiano.t32 \
    0x80000000 0x08000000

; In a TrustZone/Hypervisor environment, you may need to
; specify the access class where the UEFI BIOS runs.
; E.g. if TianoCore runs in hypervisor zone:
EXTension.ACCESS H:

; Load the additional menu:
MENU.ReProgram ~~/demo/arm/bootloader/uefi/tiano/tiano.men

; Configure symbol autoloader:
sYmbol.AutoLOAD.CHECKUEFI "do ~~/demo/arm/bootloader/uefi/tiano/autoload "
```

See also the example scripts in ~~/demo/arm/bootloader/uefi/tiano

ARM 64-Bit

A full configuration for ARM **64-bit** can look like this (the path prefix ~~ expands to the system directory of TRACE32.):

```
; Specify the memory base address and size,
; see <board>Pkg/<board>Pkg.dsc:
; PcdSystemMemoryBase = 0x80000000
; PcdSystemMemorySize = 0x08000000

; Load the TianoCore Awareness:
EXTension.CONFIG ~~/demo/arm/bootloader/uefi/tiano/tiano.t32 \
    0x80000000 0x08000000

; In a TrustZone/Hypervisor environment, you may need to
; specify the access class where the UEFI BIOS runs.
; E.g. if TianoCore runs in hypervisor zone:
EXTension.ACCESS H:

; Load the additional menu:
MENU.ReProgram ~~/demo/arm/bootloader/uefi/tiano/tiano.men

; Configure symbol autoloader:
sYmbol.AutoLOAD.CHECKUEFI "do ~~/demo/arm/bootloader/uefi/tiano/autoload "
```

See also the example scripts in ~~/demo/arm/bootloader/uefi/tiano

Hooks & Internals in TianoCore

IMPORTANT:

When using GCC on ARM:

The ELF->COFF converter (GenFw) may spoil the debug information when using several text/data sections (check with "-v"). The linker must combine all sections into one text section and one data section. The edk2/BaseTools/Scripts directory contains a suitable linker script (GccBase.lds or previously gcc4.4-ld-script). Please ensure that this script is used when linking a module, e.g. by adding it to the linker flags in the edk2/BaseTools/Conf/tools_def.template:

```
--script=$(EDK_TOOLS_PATH)/Scripts/GccBase.lds
```

Features

The UEFI Awareness for TianoCore supports the following features.

Display of UEFI Resources

The extension defines new commands to display various UEFI resources. Information on the following UEFI components can be displayed:

PEI phase:

EXTension.FV PEI	PEI firmware volumes
EXTension.PEIModule	PEI modules in FVs
EXTension.HOB PEI	PEI HOBs

DXE phase:

EXTension.FV DXE	DXE firmware volumes
EXTension.DXEModule	DXE modules in FVs
EXTension.DXEDRiVer	Loaded DXE drivers
EXTension.HOB DXE	DXE HOBs
EXTension.PROTocol DXE	Installed DXE protocols
EXTension.ConfigTab	DXE configuration table

For a description of the commands, refer to chapter "**TianoCore Commands**".

If you want to display the UEFI objects "On The Fly" while the target is running, you need to have access to memory while the target is running. Enable **SYStem.MemAccess** or **SYStem.CpuAccess** (CPU dependent), but be aware of the limitations (no cache reading, real-time intrusion).

Symbol Autoloader

The UEFI code is provided by the boot FLASH, but debugging becomes more comfortable when debug symbols are available.

TRACE32 contains an “Autoloader”, which can be set up for automatic loading of symbol files. The Autoloader maintains a list of address ranges, corresponding UEFI components and the appropriate load command. Whenever the user accesses an address within an address range known to the Autoloader, the debugger invokes the load associated command. The command is usually a call to a PRACTICE script, that handles loading the symbol file.

The TRACE32 Autoloader has to be set up. This includes the following steps:

1. Autoloader configuration.
2. Scan of the UEFI module table to the Autoloader table.
3. Display of the Autoloader table.

Autoloader Configuration

The command **sYmbol.AutoLOAD.CHECKUEFI** <*load_command*> specifies the command that is automatically used by the Autoloader to load the symbol information. Typically the script `autoload.cmm` provided by Lauterbach is called.

The command **sYmbol.AutoLOAD.CHECKUEFI** implicitly also defines the parameters that TRACE32 uses internally for the Autoloader.

The script is provided in the TRACE32 demo directory:

- 32-bit: ~~/`demo/arm/bootloader/uefi/tiano/autoload.cmm`.
- 64-bit: ~~/`demo/arm/bootloader/uefi/tiano/autoload.cmm`.

Example:

```
; Configure symbol Autoloader for 32-bit TianoCore
sYmbol.AutoLOAD.CHECKUEFI "DO ~~/demo/arm/bootloader/uefi/tiano/autoload.cmm"
```

Scan the UEFI Module Table

When the Autoloader is configured, the command **sYmbol.AutoLOAD.CHECK** can be used to scan the UEFI module table into the Autoloader table and to activate the Autoloader.

Since the UEFI module table is updated by UEFI a re-scan might be necessary.

The point of time at which the UEFI module table is re-scanned can be set very flexibly:

sYmbol.AutoLOAD.CHECK [**ON** | **OFF** | **ONGO**]

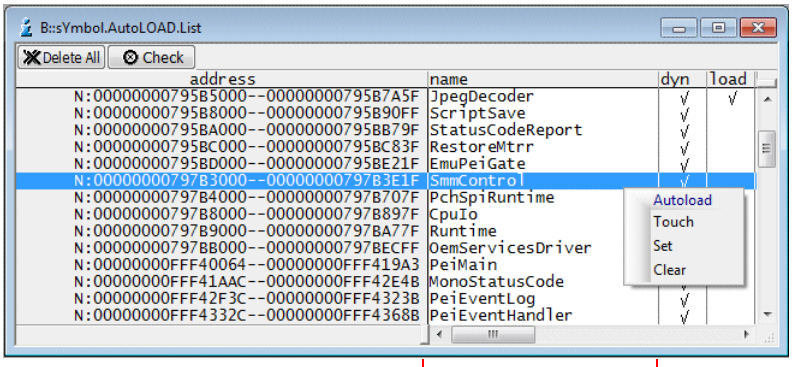
The default setting is **sYmbol.AutoLOAD CHECK OFF.** With this setting TRACE32 re-scans the UEFI module table only on request by using the **sYmbol.AutoLOAD.CHECK** command.

With **sYmbol.AutoLOAD.CHECK ON**, TRACE32 re-scans the UEFI module table after every single step and whenever the program execution is stopped. This significantly slows down the speed of TRACE32.

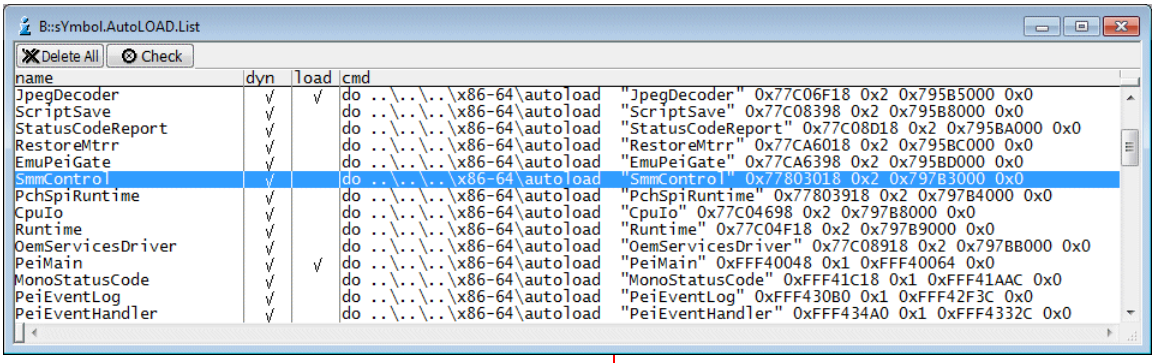
With **sYmbol.AutoLOAD.CHECK ONGO**, TRACE32 re-scans the UEFI module table whenever the program execution is stopped.

NOTE:	The Autoloader can load the symbol information for the SecCore, the PeiCore, all PEI modules and the DXE core as soon as the memory mode (e.g. 32-bit protected mode) used by UEFI is activated. The Autoloader can only load symbol information for DXE modules that are already loaded.

The command "**sYmbol.AutoLOAD.List**" shows a list of all known address ranges/components and their symbol load commands.



Module address range | Module name | Module status
dyn: (no meaning)
load: symbols for module are loaded



Load command | Parameters for load command

Autoload context menu	
Touch	Advise TRACE32 to load the symbols for the selected module now.
Set	Mark selected module as loaded.
Clear	Delete symbols for the selected module in TRACE32.

TianoCore Specific Menu

The menu file “tiano.men” contains a menu with TianoCore specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **TianoCore**.

- Use the **PEI** submenu to launch windows displaying PEI specific resources.
- Use the **DXE** submenu to launch windows displaying DXE specific resources.
- Use the **Symbol Autoloader** submenu to configure the symbol autoloader.
  See also chapter “**Symbol Autoloader**”.
  - **List Components** opens a **sYmbol.AutoLOAD.List** window showing all components currently active in the autoloader.
  - **Check Now!** performs a **sYmbol.AutoLOAD.CHECK** and reloads the autoloader list.
  - **Set Loader Script** allows you to specify the script that is called when a symbol file load is required. You may also set the automatic autoloader check.

Debugging UEFI Phases of TianoCore

UEFI runs in several “phases”. It starts with the “Security” (SEC) phase which immediately switches to the “Pre-EFI Initialization Environment” (PEI) phase. After this phase ended, control is given to the “Driver Execution Environment” (DXE) phase. Shortly, before the OS is booted, the “Boot Device Selection” (BDS) phase is running.

Each of this phases needs a different debugging environment. See below for a detailed description of each phase.

Debugging from Reset Vector

TRACE32 is a JTAG-based debugging tool and, as such, allows the user to start debugging their system right from the reset vector. It is possible to walk through the very first steps of the start-up to detect FLASH problems or faulty reset behavior.

Shortly after reset, the system switches into the SEC phase.

SEC Phase

TianoCore itself does not provide an SEC phase. It is up to the developer to implement a custom SEC phase, and as such out of the scope of this document

PEI Phase

If you want to debug the PEI phase right from the start, halt the system at the reset vector. Then load the symbols of your PEI core module with the symbol autoloader, and go until the desired entry point, e.g:

```
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.Touch "ArmPlatformPrePiUniCore"
Go PrePiMain
```

Inspect the PEI resources with the menu items in the “PEI” submenu.

DXE Phase

After PEI phase completed, it hands off control to the DXE core. To debug the DxeCore from start, load the symbols of “DxeCore” just before PEI jumps into the DxeCore and set a breakpoint at “DxeMain”. DxeMain then starts the DXE dispatcher.

For debugging a DXE driver from its entry point, a special script “go_dxedrv” is available in the ~~/demo directory. Call this script with the name of the DXE module *before* the module is started. E.g. to debug the DXE driver “Metronome”:

```
DO go_dxedrv Metronome
```

This script sets a breakpoint in the DXE core code and waits until the specified DXE module is loaded. Then it sets a breakpoint onto the module entry point and halts there. You can then start debugging the module from scratch.

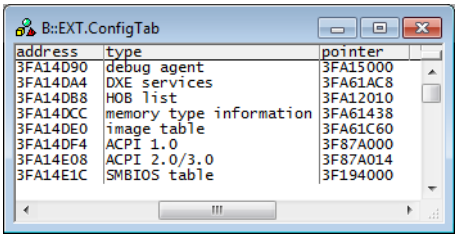
BDS Phase

TianoCore implements the BDS phase as DXE driver. To debug the BDS phase, debug the “ArmPlatformBds” module like shown in “**DXE Phase**”.

EXTension.ConfigTab — Display DXE configuration table

Format: **EXTension.ConfigTab**

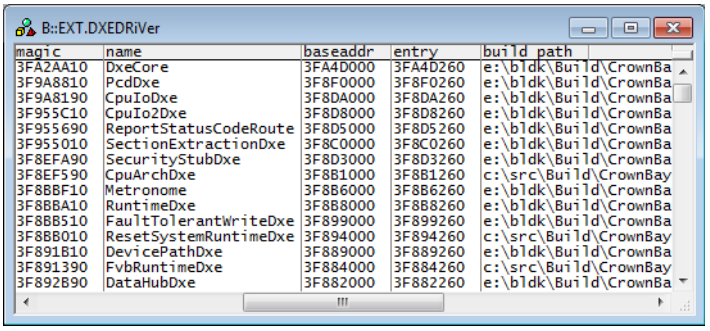
Displays the DXE configuration table.



EXTension.DXEDRiVer — Display loaded DXE drivers

Format: **EXTension.DXEDRiVer**

Displays a table with all DXE drivers that DxeCore already loaded into the system.

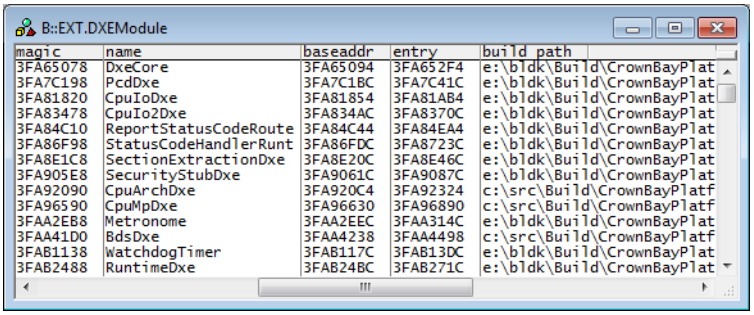


You can sort the window to the entries of a column by clicking on the column header.

“magic” is a unique ID, used by the UEFI Awareness to identify a specific driver.

Format:	**EXTension.DXEModule**

Displays a table with all DXE modules found in the system (firmware volumes or HOBs).



magic	name	baseaddr	entry	build_path
3FA65078	DxeCore	3FA65094	3FA652F4	e:\bldk\Build\CrownBayPlat
3FA7C198	PcdDxe	3FA7C1BC	3FA7C41C	e:\bldk\Build\CrownBayPlat
3FA81820	CpuIoDxe	3FA81854	3FA81AB4	e:\bldk\Build\CrownBayPlat
3FA83478	CpuIo2Dxe	3FA834AC	3FA8370C	e:\bldk\Build\CrownBayPlat
3FA84C10	ReportStatusCodeRoute	3FA84C44	3FA84EA4	e:\bldk\Build\CrownBayPlat
3FA86F98	StatusCodeHandlerRunt	3FA86FDC	3FA8723C	e:\bldk\Build\CrownBayPlat
3FA8E1C8	SectionExtractionDxe	3FA8E20C	3FA8E46C	e:\bldk\Build\CrownBayPlat
3FA905E8	SecurityStubDxe	3FA9061C	3FA9087C	e:\bldk\Build\CrownBayPlat
3FA92090	CpuArchDxe	3FA920C4	3FA92324	c:\src\Build\CrownBayPlatf
3FA96590	CpuMpDxe	3FA96630	3FA96890	c:\src\Build\CrownBayPlatf
3FAA2EB8	Metronome	3FAA2EEC	3FAA314C	e:\bldk\Build\CrownBayPlat
3FAA41D0	BdsDxe	3FAA4238	3FAA4498	c:\src\Build\CrownBayPlatf
3FAB1138	WatchdogTimer	3FAB117C	3FAB13DC	e:\bldk\Build\CrownBayPlat
3FAB2488	RuntimeDxe	3FAB24BC	3FAB271C	e:\bldk\Build\CrownBayPlat

You can sort the window to the entries of a column by clicking on the column header.

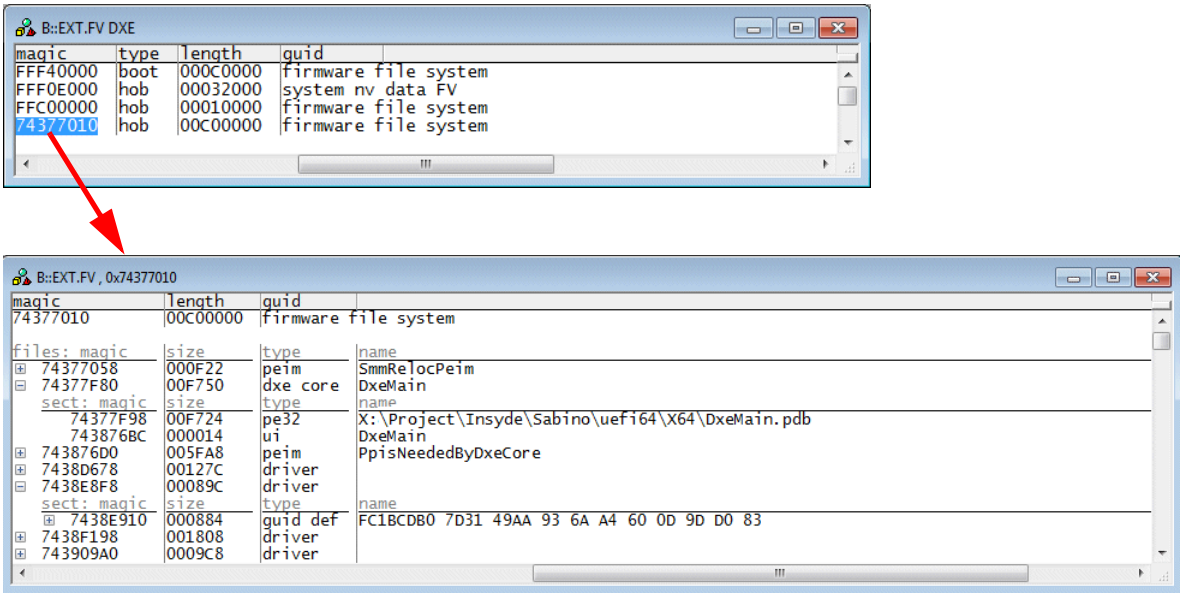
“magic” is a unique ID, used by the UEFI Awareness to identify a specific module.

The “magic” fields are mouse sensitive. Right-click on them to get a local menu. Double-clicking on them opens appropriate windows.

Format: **EXTension.FV** [**PEI** | **DXE** [<*fv_address*>]]

Displays a table with the firmware volumes of the PEI or DXE phase.

If an address of a firmware volume is specified, the command displays the contents of this FV.



“magic” is a unique ID used by the UEFI Debugger to identify a specific firmware volume or file.

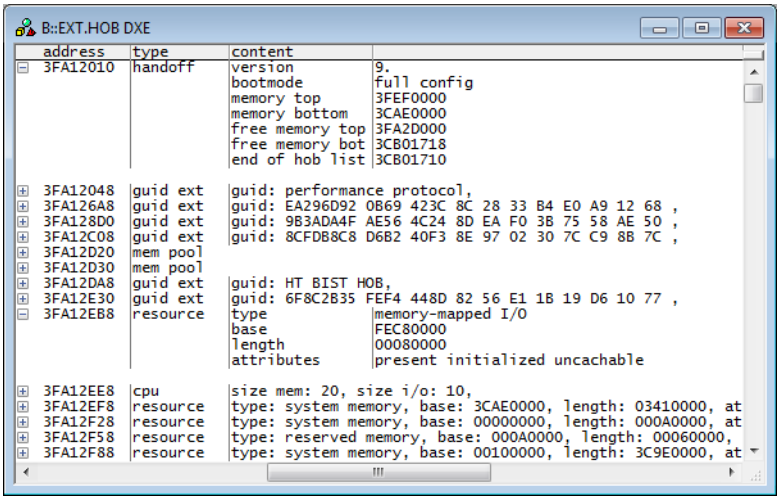
The “magic” fields are mouse sensitive, double clicking on them opens appropriate windows. Right-clicking on them will show a context menu.

The debugger tries to detect the address of the boot firmware volume automatically. If this fails, specify the address of the boot FV manually with the **EXTension.Option BOOTFV** command.

EXTension.HOB Display HOBs

Format: **EXTension.HOB** [**PEI** | **DXE**]

Displays a table with the hand off blocks of the PEI or DXE phase.



The “address” fields are mouse sensitive, double-clicking them opens appropriate windows. Right-clicking on them will show a context menu.

EXTension.Option — Set awareness options

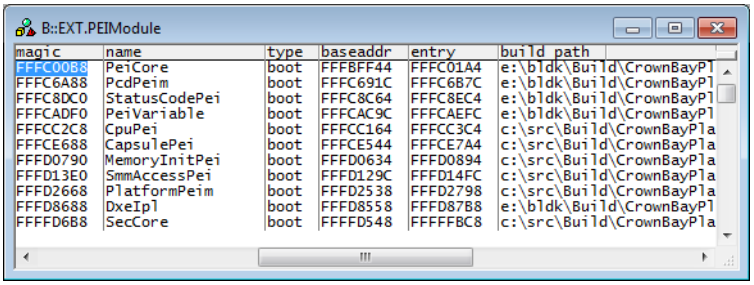
Format:	**EXTension.Option** *<option>*
<option>:	**BOOTFV** *<address>* **PEIHOBS** *<address>* **SYSTABLE** *<address>* **UCODE** *<address>*

Sets various options to the awareness.

BOOTFV	Set the base address of the boot firmware volume.
PEIHOBS	Set the base address of the HOB list in PEI phase.
SYSTABLE	Set the base address of the EFI System Table
UCODE	Set the base address of the microcode table.

Format: **EXTension.PEIModule**

Displays a table with all PEI modules found in the system.



You can sort the window to the entries of a column by clicking on the column header.

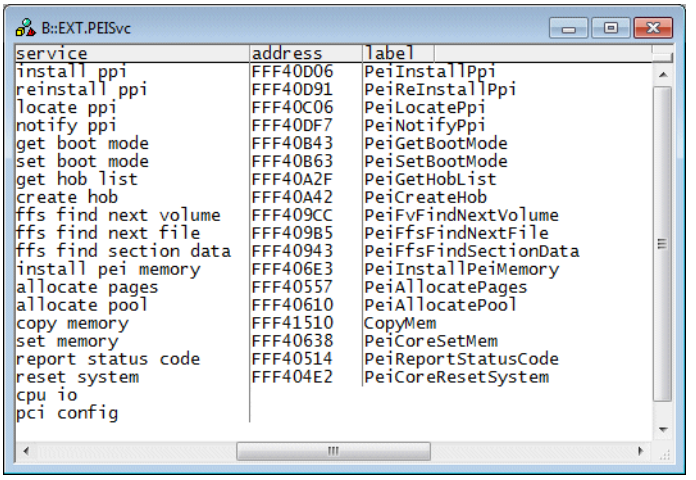
“magic” is a unique ID, used by the UEFI Awareness to identify a specific module.

The “magic” fields are mouse sensitive. Right-click on them to get a local menu. Double-clicking on them opens appropriate windows.

EXTension.PEISvc Display PEI services

Format: **EXTension.PEISvc**

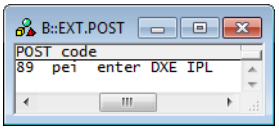
Displays a table with all available PEI services.



Format: **EXTension.POST**

(Only available on x86/x64 targets.)

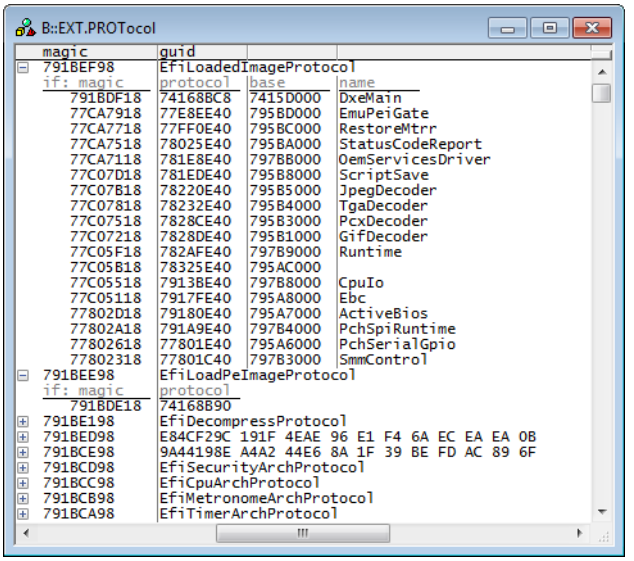
Displays the Power-On Self-Test code.



EXTension.PROTocol Display installed protocols

Format: **EXTension.PROTocol**

Displays the list of installed DXE protocols.



TianoCore PRACTICE Functions

There are special definitions for TianoCore specific PRACTICE functions.

EXT.DXEDRV.ENTRY() Entry address for DXE driver

Syntax: **EXT.DXEDRV.ENTRY(***<dxedrv_magic>***)**

Returns the entry address for the specified DXE driver.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

EXT.DXEDRV.MAGIC() Magic of DXE driver

Syntax: **EXT.DXEDRV.MAGIC("***<dxedrv_name>***")**

Returns the “magic” of the specified loaded DXE driver.

Parameter Type: String (*with* quotation marks).

Return Value Type: Hex value.

EXT.DXEDRV.PATH() Build path for DXE driver

Syntax: **EXT.DXEDRV.PATH(***<dxedrv_magic>***)**

Returns the build path for the specified DXE driver.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

EXT.DXEFILE.PATH() Build path for DXE module

Syntax: **EXT.DXEFILE.PATH(**<*dxem_magic*>**)**

Returns the build path for the specified DXE module.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

EXT.PEIM.ENTRY() Entry address for PEI module

Syntax: **EXT.PEIM.ENTRY(**<*peim_magic*>**)**

Returns the entry address for the specified PEI module.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

EXT.PEIM.MAGIC() Magic of PEI module

Syntax: **EXT.PEIM.MAGIC("**<*peim_name*>**")**

Returns the “magic” of the specified PEI module.

Parameter Type: String (*with* quotation marks).

Return Value Type: Hex value.

EXT.PEIM.PATH() Build path for PEI module

Syntax: **EXT.PEIM.PATH(**<*peim_magic*>**)**

Returns the build path for the specified PEI module.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.