LAUTERBACH
DEVELOPMENT TOOLS

UEFI Awareness Manual BLDK

MANUAL

UEFI Awareness Manual BLDK

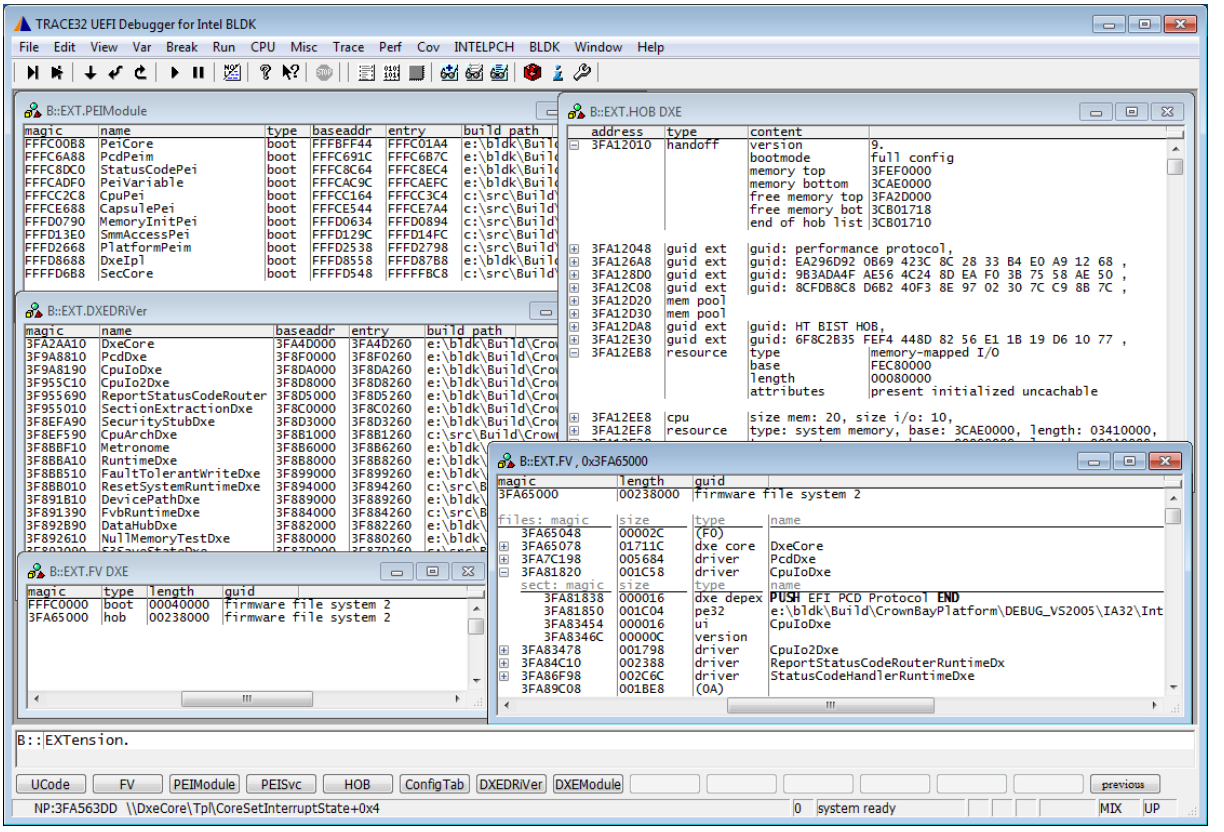
UEFI Awareness Manual BLDK

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 Intel® BLDK contains special extensions to the TRACE32 Debugger. This chapter describes the additional features, such as additional commands and debugging approaches.

Brief Overview of Documents for New Users

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 Intel® BLDK is supported for the following versions:

- Intel® BLDK core 2.x on x86 and x64 architectures
- Intel® UDK 2010 to 2015

Configuration

The UEFI Awareness for Intel® BLDK is configured by loading an extension definition file called “bldk.t32” from the demo directory with the **EXTension.CONFIG** command. Additionally, load the “bldk.men” menu file (see “**BLDK specific Menu**”) and configure the **Symbol Autoloader**.

x86 32-Bit

A full configuration for x86 **32-bit** can look like this (the path prefix ~~ expands to the system directory of TRACE32.):

```
; Load the Intel® BLDK Awareness:
EXTension.CONFIG ~~/demo/x86/bootloader/uefi/bldk/bldk.t32

; Load the additional menu:
MENU.ReProgram ~~/demo/x86/bootloader/uefi/bldk/bldk.men

; Configure symbol autoloader:
sYmbol.AutoLOAD.CHECKUEFI "do ~~/demo/x86/bootloader/uefi/bldk/autoload "
```

See also the example scripts in ~~/demo/x86/bootloader/uefi/bldk

x64 64-Bit

A full configuration for x64 **64-bit** can look like this (the path prefix ~~ expands to the system directory of TRACE32.):

```
; Load the Intel® BLDK Awareness:
EXTension.CONFIG ~~/demo/x64/bootloader/uefi/bldk/bldk.t32

; Load the additional menu:
MENU.ReProgram ~~/demo/x64/bootloader/uefi/bldk/bldk.men

; Configure symbol autoloader:
sYmbol.AutoLOAD.CHECKUEFI "do ~~/demo/x64/bootloader/uefi/bldk/autoload "
```

See also the example scripts in ~~/demo/x64/bootloader/uefi/bldk

When using the debug build of the Intel® BLDK (which is recommended), the build system automatically inserts breakpoints into the images when loading new modules. TRACE32 does **not** use these breakpoints. Either remove them from the debug build, or simply continue when halting there.

The UEFI Awareness needs some basic pointers:

- For PEI phase, it needs the address of the boot firmware volume (BootFV). Usually, the UEFI image holds a pointer to the BootFV at address 0xFFFFFFFC. If, for some reason, this pointer is not available, please ask your system integrator where the BootFV resides. Report this address to the UEFI Awareness with the command **EXTension.Option BOOTFV**.
- For DXE phase, it needs the address of the EFI System Table. It uses a global symbol of the DxeCore to get this address. If the DxeCore symbols are not loaded, or if the symbol is not available, the UEFI Awareness searches for the system table signature within the current RAM area. If the system table is found, **EXTension.ConfigTab** shows its address.
  This search may run over address ranges, that are not mapped and that may cause the target system to crash. In this case, either mask out all memory areas, that should not be touched, with **MAP.DenyAccess**, or set the address explicitly with the command **EXTension.Option SYSTABLE**. To get the address of the EFI System Table in your running UEFI BIOS, you can also execute the “mem” command in the EFI Shell (if available).

Features

The UEFI Awareness for Intel® BLDK supports the following features.

Display of UEFI Resources

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

EXTension.POST	POST code

SEC phase:

EXTension.UCode	Available microcodes

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 “**Intel BLDK Commands**”.

Since the x86/x64/Atom architecture does not allow to read memory while the program execution is running, the information can only be displayed, if the program execution is stopped.

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/x86/bootloader/uefi/bldk/autoload.cmm`.
- 64-bit: ~~/`demo/x64/bootloader/uefi/bldk/autoload.cmm`.

Example:

```
; Configure symbol Autoloader for 32-bit Intel® BLDK
sYmbol.AutoLOAD.CHECKUEFI "DO ~~/demo/x86/bootloader/uefi/bldk/autoload.cmm"
```

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**]

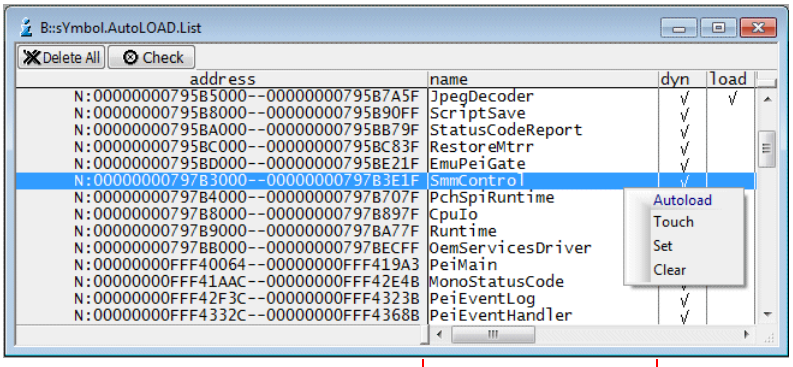
The default setting is **sYmbol.AutoLOAD CHECK OFF.** With this setting TRACE32 re-scans the UEFI module table only on request by using the **sYmbol.AutoLOAD.CHECK** command.

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.

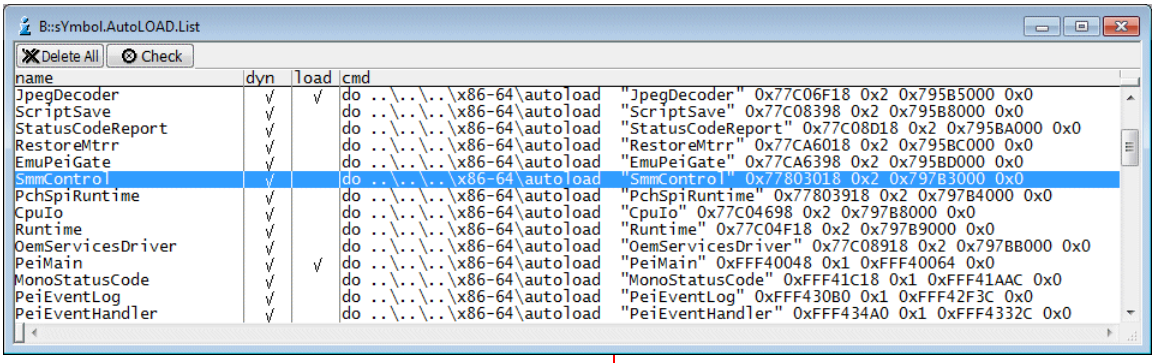
With **sYmbol.AutoLOAD.CHECK ONGO**, TRACE32 re-scans the UEFI module table whenever the program execution is stopped.

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.

Intel BLDK Specific Menu

The menu file “bldk.men” allows to add a menu with Intel® BLDK specific menu items. Load this menu with the **MENU.ReProgram** command.

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

- Use the **PEI** submenu to launch windows displaying PEI specific resources.
- Use the **DXE** submenu to launch windows displaying DXE specific resources.
- The **Display POST Code** submenu allows to display the current POST (Power On SelfTest) code.
- **Load Current Symbols** calls a script that tries to find the image header of the code at the current program counter location. If the header is found, it tries to load the according symbol file.
- 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 Intel BLDK

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 Atom/x86 system right from the reset vector (normally at BP:0xF000:0xFFF0). 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

The Intel® BLDK does not provide symbol information for the SEC phase, so we cannot debug this phase in source code. However, the debugger has access to the boot firmware volume. During SEC phase, use **EXTenstion.FV PEI** to inspect the boot FV.

PEI Phase

NOTE:	Debugging within the PEI phase requires access to the boot firmware volume. The UEFI Awareness tries to find the address of the boot firmware volume automatically, see **Hooks & Internals** if this fails.

If you want to debug the PEI phase right from the start, halt the system while in SEC phase. Then load the symbols of the PEI core module (“PeiCore”) with the symbol autoloader, and go until “PeiCore”:

```
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.Touch "PeiCore"
Go PeiCore
```

Intel® BLDK starts the PeiCore several times with different settings. The first time after the SEC phase, code runs from Flash and data is in internal memory. PeiCore then initializes external RAM and calls itself, starting PeiCore a second time, now with code in Flash and data in external RAM. Now the PeiCore module will be copied into RAM for faster execution. Check and touch the module in the symbol autoloader again, to trigger a reload of the PeiCore symbols, now to RAM address.

Inspect the PEI resources with the menu items in the **PEI** submenu.

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

```
DO go_peimdyn PcdPeim
```

This script sets a breakpoint in the PEI code and waits until the specified PEI 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.

DXE Phase

NOTE:	Debugging within the DXE phase requires access to the EFI System Table. The UEFI Awareness tries to find the address of the system table automatically, see **Hooks & Internals** if this fails.

After PEI phase completed, it hands off control to the DXE phase, starting with the DxeCore Module.

To debug the DxeCore right from start, run the PEI until it jumps into the DxeCore (the function HandOffToDxeCore() is a good place to stop). Load the symbols of “DxeCore” (by executing sYmbol.AutoLOAD.CHECK and sYmbol.AutoLOAD.TOUCH "DxeCore"). Then set a breakpoint at “DxeMain”. DxeMain then starts the DXE dispatcher. If you do not want to debug the DxeCore startup, simply load the symbols of DxeCore after DXE phase came up.

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.

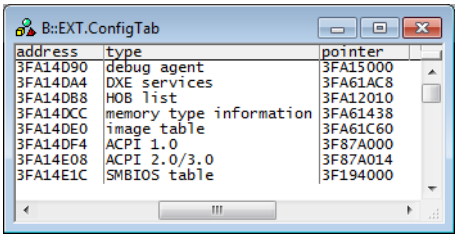
BDS Phase

Intel® BLDK implements the BDS phase as DXE driver. To debug the BDS phase, debug the “BdsDxe” module like shown in “**DXE Phase**”.

EXTension.ConfigTab — Display DXE configuration table

Format: **EXTension.ConfigTab**

Displays the DXE configuration table.

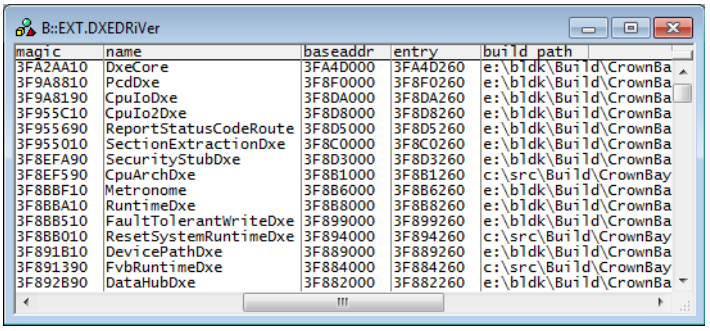


address	type	pointer
3FA14D90	debug agent	3FA15000
3FA14DA4	DXE services	3FA61AC8
3FA14DB8	HOB list	3FA12010
3FA14DCC	memory type information	3FA61438
3FA14DE0	image table	3FA61C60
3FA14DF4	ACPI 1.0	3F87A000
3FA14E08	ACPI 2.0/3.0	3F87A014
3FA14E1C	SMBIOS table	3F194000

EXTension.DXEDRiVer — Display loaded DXE drivers

Format: **EXTension.DXEDRiVer**

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



magic	name	baseaddr	entry	build path
3FA2AA10	DxeCore	3FA4D000	3FA4D260	e:\bldk\Build\CrownBa
3F9A8810	PcdDxe	3F8F0000	3F8F0260	e:\bldk\Build\CrownBa
3F9A8190	CpuIoDxe	3F8DA000	3F8DA260	e:\bldk\Build\CrownBa
3F955C10	CpuIo2Dxe	3F8D8000	3F8D8260	e:\bldk\Build\CrownBa
3F955690	ReportStatusCodeRoute	3F8D5000	3F8D5260	e:\bldk\Build\CrownBa
3F955010	SectionExtractionDxe	3F8C0000	3F8C0260	e:\bldk\Build\CrownBa
3F8EFA90	SecurityStubDxe	3F8D3000	3F8D3260	e:\bldk\Build\CrownBa
3F8EF590	CpuArchDxe	3F8B1000	3F8B1260	c:\src\Build\CrownBay
3F8BBF10	Metronome	3F8B6000	3F8B6260	e:\bldk\Build\CrownBa
3F8BBA10	RuntimeDxe	3F8B8000	3F8B8260	e:\bldk\Build\CrownBa
3F8BB510	FaultTolerantWriteDxe	3F899000	3F899260	e:\bldk\Build\CrownBa
3F8BB010	ResetSystemRuntimeDxe	3F894000	3F894260	c:\src\Build\CrownBay
3F891B10	DevicePathDxe	3F889000	3F889260	e:\bldk\Build\CrownBa
3F891390	FvbRuntimeDxe	3F884000	3F884260	c:\src\Build\CrownBay
3F892B90	DataHubDxe	3F882000	3F882260	e:\bldk\Build\CrownBa

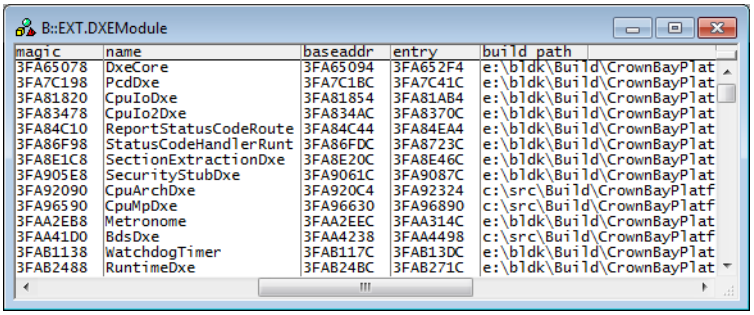
You can sort the window to the entries of a column by clicking on the column header.

“magic” is a unique ID, used by the UEFI Awareness to identify a specific driver (address of the EFI debug image structure).

Right clicking on the “magic” opens a local window. It allows to load the symbols of the selected DXE driver.

Format:	**EXTension.DXEModule**

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



B::EXT.DXEModule

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.

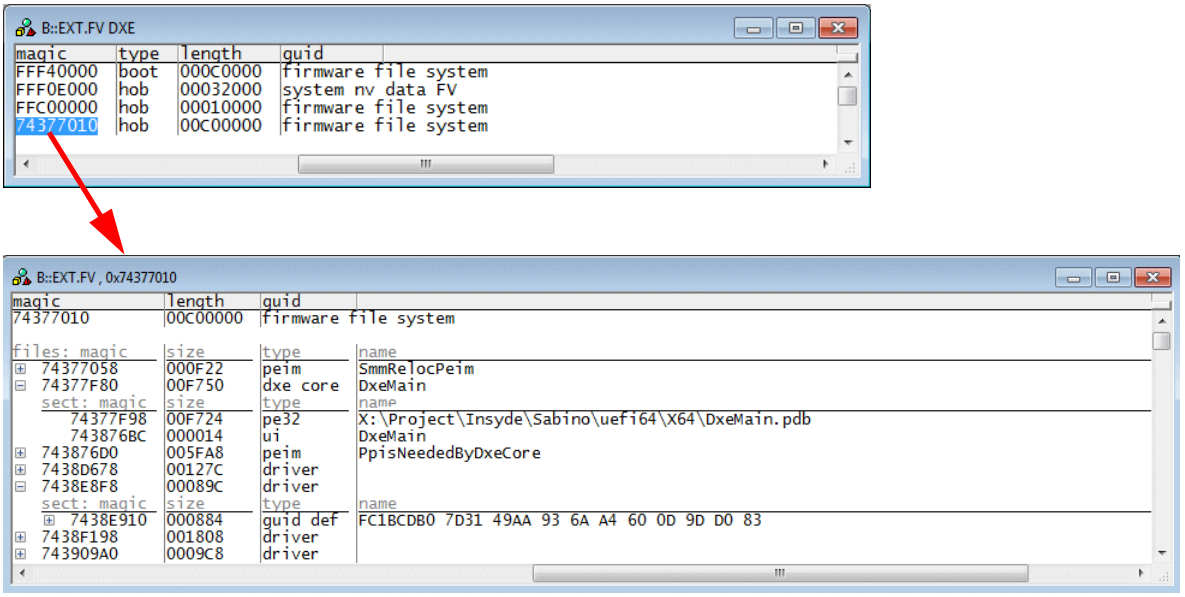
“magic” is a unique ID, used by the UEFI Awareness to identify a specific module.

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.



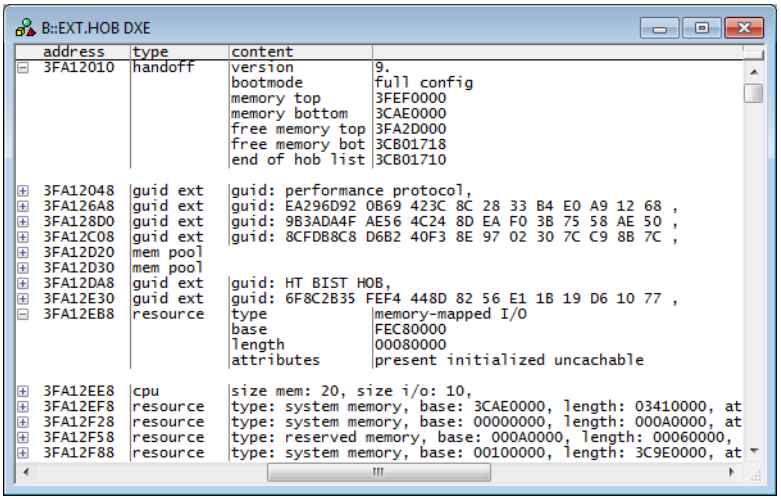
“magic” is a unique ID used by the UEFI Debugger to identify a specific firmware volume or file.

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.

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 on them opens appropriate windows. Right clicking on them will show a local 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.

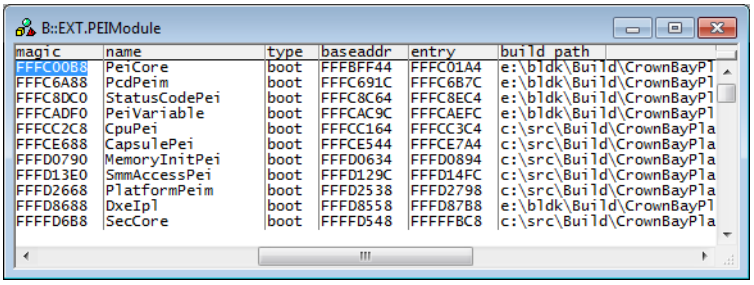
PEIHOBS Set the base address of the HOB list in PEI phase.

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.



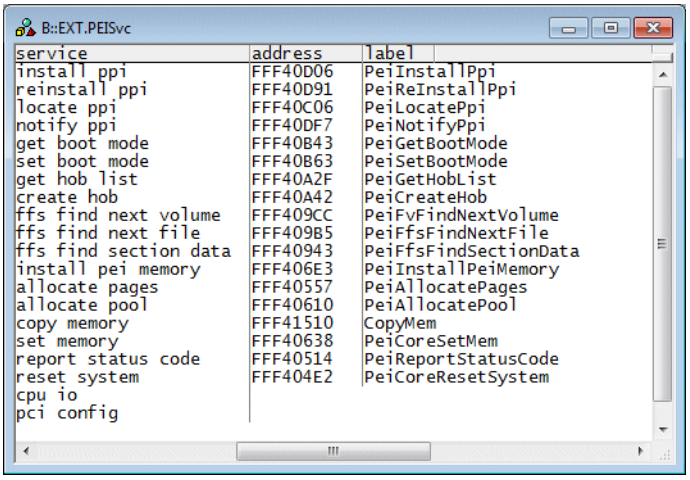
You can sort the window to the entries of a column by clicking on the column header.

“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.

Format: **EXTension.PEISvc**

Displays a table with all available PEI services.

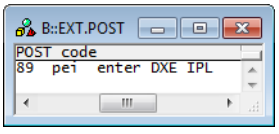


EXTension.POST Display POST code

Format: **EXTension.POST**

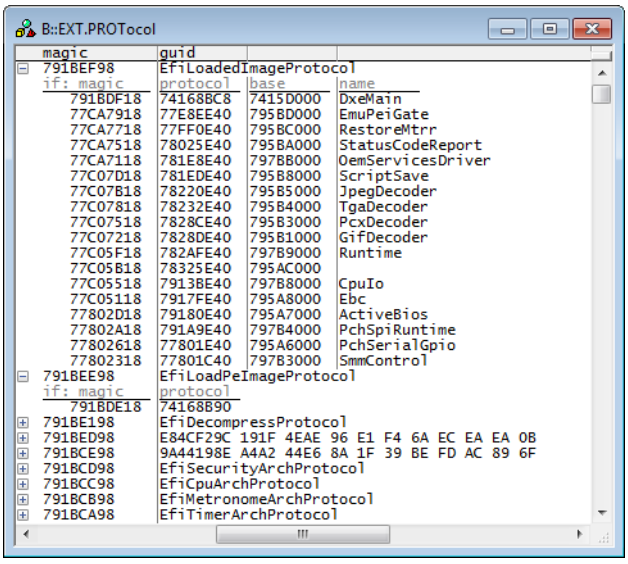
(Only available on x86/x64 targets.)

Displays the Power-On Self-Test code.



Format: **EXTension.PROTocol**

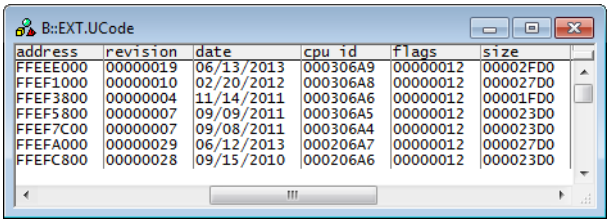
Displays the list of installed DXE protocols.



EXTension.UCode Display microcodes

Format: **EXTension.UCode**

Displays the list of available microcodes.



The debugger tries to detect the address of the microcode list automatically. If this fails, specify the address of the first microcode manually with the command **EXTension.Option UCODE**.

Intel BLDK PRACTICE Functions

There are special definitions for Intel® BLDK 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() DXE driver magic number

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

Returns the driver magic number 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.MACHINE() Machine type for DXE module

Syntax: **EXT.DXEFILE.MACHINE(***<file_address>***)**

Returns the machine type for the specified DXE module.

Parameter Type: Decimal or hex or binary value.

Return Value Type: Hex value.

Return Value and Description:

0	0 = 32-bit
1	1 = 64-bit

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

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

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.