LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual Windows Standard

Release 02.2025

MANUAL

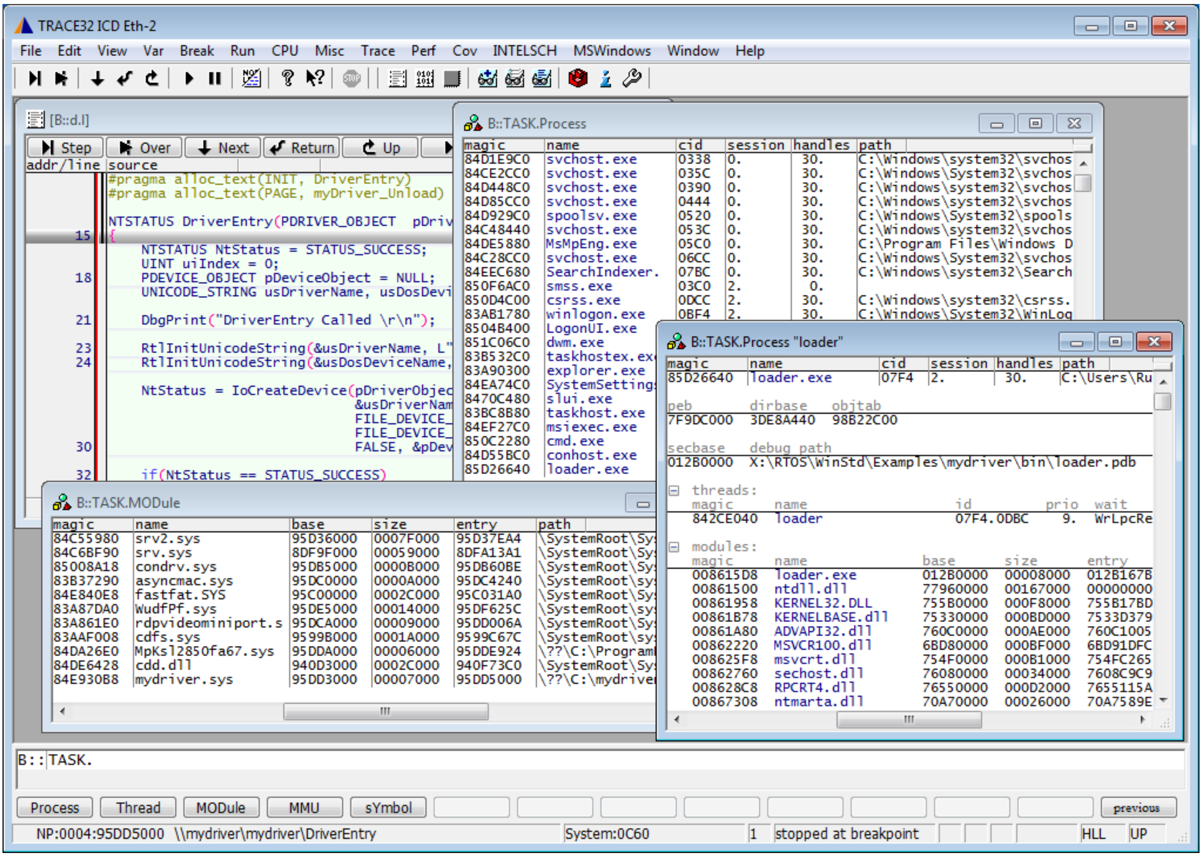
OS Awareness Manual Windows Standard

Version 13-Feb-2025

History

30-Dec-20	Add description for commands and functions relative to debug of UEFI runtime service modules.

Overview



The OS Awareness for Windows Standard contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistic evaluations.

Terminology

Windows uses the terms “processes” and “threads”. If not otherwise specified, the TRACE32 term “task” corresponds to Windows threads.

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Debugger Tutorial"** (debugger_tutorial.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"General Commands"** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.
- **"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.

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**.

Supported Versions

Currently Windows Standard awareness is supported for Intel® x86/x64, ARM and ARM64 architectures.

The following table summarize the supported targets

Windows version	Bit-ness	Supported Architectures
Windows XP	32	x86, x64
Windows Vista	32	x86, x64
Windows 7	32	x86, x64
Windows 8	32	x86, x64
Windows 10	32	x86, x64, ARM, ARM64
Windows 7	64	x64
Windows 8	64	x64
Windows 10	64	x64, ARM64

Configuration

The **TASK.CONFIG** command loads an extension definition file. Depending on the target architecture, and the Windows bit-ness, the corresponding extension file need to be configured:

Format:	**TASK.CONFIG** *<config_file>*
<config_file>:	*select the file appropriate for your target architecture and bit-ness:* **~~/demo/x86/kernel/windows/win32.t32** **~~/demo/x64/kernel/windows/win64.t32** **~~/demo/arm/kernel/windows/win32.t32** **~~/demo/arm/kernel/windows/win64.t32**

For x64 targets running 32bit Windows versions, the extension "~~/demo/x86/kernel/windows/win32.t32" needs to be used. And for ARM64 targets running 32bit Windows versions the extension "~~/demo/arm/kernel/windows/win32.t32" needs to be used.

After loading the extension definition file, the extension needs to load the Windows kernel symbols using the command **TASK.sYmbol.LOADNT**. This is necessary for the proper operation of the Windows awareness.

Quick Configuration Guide

To access all features of the OS Awareness you should follow the following road map:

1. Carefully read the PRACTICE demo start-up scripts (~~/demo/*<arch>*/kernel/windows/board/)
2. Make a copy of the PRACTICE script file "windows.cmm". Modify the file according to your application.
3. Run the modified version in your application. This should allow you to display the kernel resources and use the trace functions (if available).

In case of any problems, please carefully read the previous Configuration chapter.

Hooks & Internals in Windows

No hooks are used in the kernel.

For retrieving the kernel data structures, the OS Awareness uses the global kernel pointers. For some features, also the symbols and structure definitions of the kernel are necessary (ntkr*.pdb). The debugger needs access to the kernel symbol file or needs the ability to download the symbol file from the Microsoft Symbol Server.

Features

The OS Awareness for Windows Standard supports the following features.

Display of Kernel Resources

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

TASK.Process	Processes
TASK.Thread	Threads
TASK.MODule	Kernel modules / drivers
TASK.UefiMODule	UEFI runtime service modules

For a description of the commands, refer to chapter “**Windows Commands**”.

If your hardware allows memory access while the target is running, these resources can be displayed “On The Fly”, i.e. while the application is running, without any intrusion to the application.

Without this capability, the information will only be displayed if the target application is stopped.

Task-Related Breakpoints

Any breakpoint set in the debugger can be restricted to fire only if a specific task hits that breakpoint. This is especially useful when debugging code which is shared between several tasks. To set a task-related breakpoint, use the command:

Break.Set *<address>*|*<range>* [**/***<option>*] **/TASK** *<task>*	Set task-related breakpoint.

- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **“What to know about the Task Parameters”** (general_ref_t.pdf).
- For a general description of the **Break.Set** command, please see its documentation.

By default, the task-related breakpoint will be implemented by a conditional breakpoint inside the debugger. This means that the target will *always* halt at that breakpoint, but the debugger immediately resumes execution if the current running task is not equal to the specified task.

NOTE:	Task-related breakpoints impact the real-time behavior of the application.

On some architectures, however, it is possible to set a task-related breakpoint with *on-chip* debug logic that is less intrusive. To do this, include the option **/Onchip** in the **Break.Set** command. The debugger then uses the on-chip resources to reduce the number of breaks to the minimum by pre-filtering the tasks.

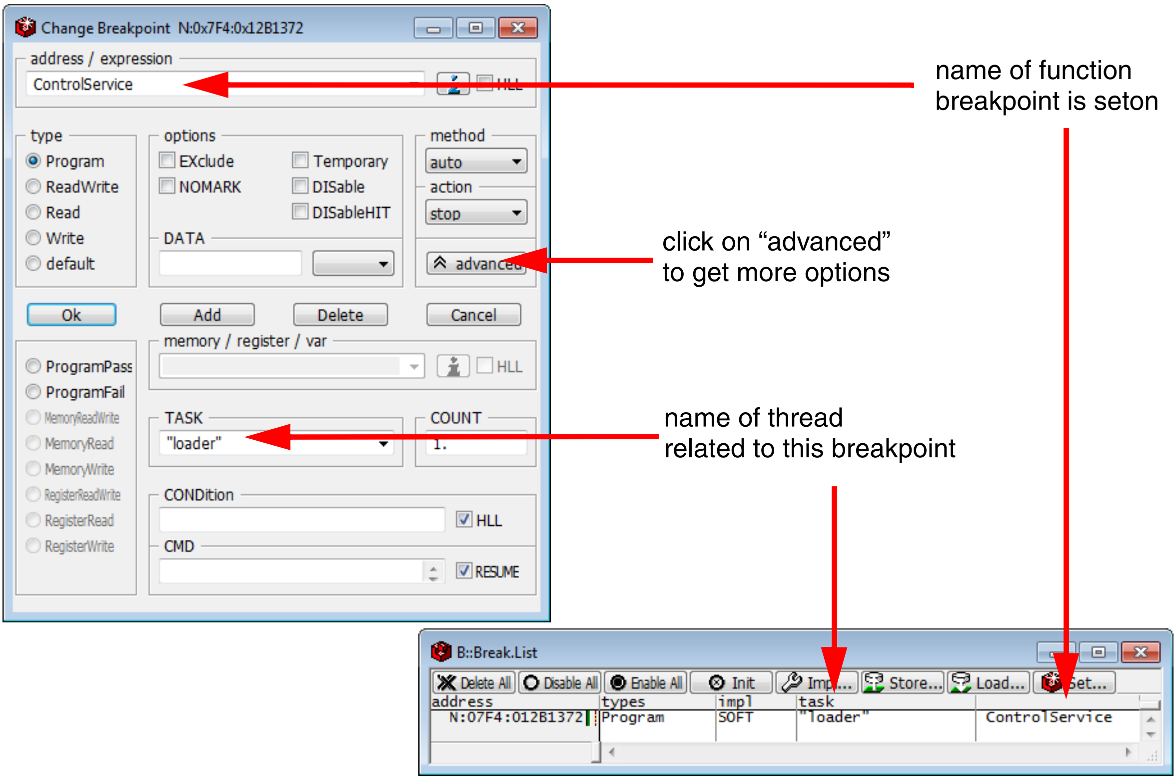
For example, on ARM architectures: *If* the RTOS serves the Context ID register at task switches, and *if* the debug logic provides the Context ID comparison, you may use Context ID register for less intrusive task-related breakpoints:

Break.CONFIG.UseContextID ON	Enables the comparison to the whole Context ID register.
Break.CONFIG.MatchASID ON	Enables the comparison to the ASID part only.
TASK.List.tasks	If **TASK.List.tasks** provides a trace ID (**traceid** column), the debugger will use this ID for comparison. Without the trace ID, it uses the magic number (**magic** column) for comparison.

When single stepping, the debugger halts at the next instruction, regardless of which task hits this breakpoint. When debugging shared code, stepping over an OS function may cause a task switch and coming back to the same place - but with a different task. If you want to restrict debugging to the current task, you can set up the debugger with **SETUP.StepWithinTask ON** to use task-related breakpoints for single stepping. In this case, single stepping will always stay within the current task. Other tasks using the same code will not be halted on these breakpoints.

If you want to halt program execution as soon as a specific task is scheduled to run by the OS, you can use the **Break.SetTask** command.

Example for a task-related breakpoint, equivalent to the **Break.Set <function> /TASK <task>** command:



Task Context Display

You can switch the whole viewing context to a task that is currently not being executed. This means that all register and stack-related information displayed, e.g. in **Register**, **List.auto**, **Frame** etc. windows, will refer to this task. Be aware that this is only for displaying information. When you continue debugging the application (**Step** or **Go**), the debugger will switch back to the current context.

To display a specific task context, use the command:

Frame.TASK [*<task>*]	Display task context.

- Use a magic number, task ID, or task name for *<task>*. For information about the parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf).
- To switch back to the current context, omit all parameters.

To display the call stack of a specific task, use the following command:

Frame /Task *<task>*	Display call stack of a task.

If you'd like to see the application code where the task was preempted, then take these steps:

1. Open the **Frame /Caller /Task** *<task>* window.
2. Double-click the line showing the OS service call.

MMU Support

To provide full debugging possibilities, the Debugger has to know, how virtual addresses are translated to physical addresses and vice versa. All MMU and TRANSlation commands refer to this necessity.

Because of the "On Demand Paging" mechanism of Windows, when single stepping the code, the instruction pointer could jump to a not yet loaded page. The debugger will not be able to display the assembly code and could not single step the current instruction. See "**On Demand Paging**" for details and workaround.

Space IDs

Under Windows different processes may use identical virtual addresses. To distinguish between those addresses, the debugger uses an additional identifier, the so-called space ID (memory space ID) that specifies, which virtual memory space an address refers to. The command **SYStem.Option.MMUSPACES ON** enables the use of the space ID. The space ID is zero for the kernel. For processes using their own address space, the space ID equals the "cid" of the process. Threads of a particular process use the memory space of the invoking parent process. Consequently threads have the same space ID as the parent process.

MMU Declaration

To access the virtual and physical addresses correctly, the debugger needs to know the format of the MMU tables in the target. When loading the extension file (see chapter Configuration), the debugger already declares the MMU format automatically.

Just for reference, the following code contains the setup done for x86/x64 Windows 32bit:

```
MMU.FORMAT PAE
TRANSlation.COMMON 0x80000000--0xFFFFFFFF
TRANSlation.TableWalk ON
TRANSlation.ON
```

The set up done for x64 Windows 64bit versions is the following:

```
MMU.FORMAT PAE64
TRANSlation.COMMON 0xFFFF080000000000--0xFFFFFFFFFFFFFFFF
TRANSlation.TableWalk ON
TRANSlation.ON
```

The set up done for ARM and ARM64 for Windows 32bit is the following:

```
TRANSlation.COMMON 0x80000000--0xFFFFFFFF
TRANSlation.TableWalk ON
TRANSlation.ON
```

The set up done for ARM64 for Windows 64bit is the following:

```
TRANSlation.COMMON 0xFFFFF80000000000--0xFFFFFFFFFFFFFFFF
TRANSlation.TableWalk ON
TRANSlation.ON
```

Symbol Autoloader

The OS Awareness for Windows Standard installs a so-called symbol autoloader, which automatically loads symbol files corresponding to executed processes, modules or libraries. The autoloader maintains a list of address ranges, corresponding to Windows components and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader (e.g. via **List.auto**), the debugger invokes the command necessary to load the corresponding symbols to the appropriate addresses (including relocation). This is usually done via a PRACTICE script.

In order to load symbol files, the debugger needs to be aware of the currently loaded components. This information is available in the kernel data structures and can be interpreted by the debugger. The command **sYmbol.AutoLOAD.CHECK** defines, *when* these kernel data structures are read by the debugger (only on demand or after each program execution).

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

The loaded components can change over time, when processes are started and stopped and kernel modules or libraries are loaded or unloaded. The command **sYmbol.AutoLOAD.CHECK** configures the strategy, when to “check” the kernel data structures for changes in order to keep the debugger’s information regarding the components up-to-date.

Without parameters, the **sYmbol.AutoLOAD.CHECK** command *immediately* updates the component information by reading the kernel data structures. This information includes the component name, the load address and the space ID and is used to fill the autoloader list (shown via **sYmbol.AutoLOAD.List**).

With **sYmbol.AutoLOAD.CHECK ON**, the debugger *automatically* reads the component information *each time the target stops executing* (even after assembly steps), having to assume that the component information might have changed. This significantly slows down the debugger which is inconvenient and often superfluous, e.g. when stepping through code that does not load or unload components.

With the parameter **ONGO**, the debugger checks for changed component info like with **ON**, but *not when performing single steps*.

With **sYmbol.AutoLOAD.CHECK OFF**, no automatic read is performed. In this case, the update has to be triggered manually when considered necessary by the user.

The command **TASK.sYmbol.Option AutoLoad** configures which types of components the autoloader shall consider:

- Processes
- Kernel modules
- All libraries, or
- Libraries of the current process.

It is recommended to restrict the components to the minimal set of interest (rather than all components), because it makes the autoloader checks much faster. By default, only processes are checked by the autoloader.

When configuring the OS Awareness for Windows Standard with the **TASK.CONFIG** command, it automatically sets the autoloader:

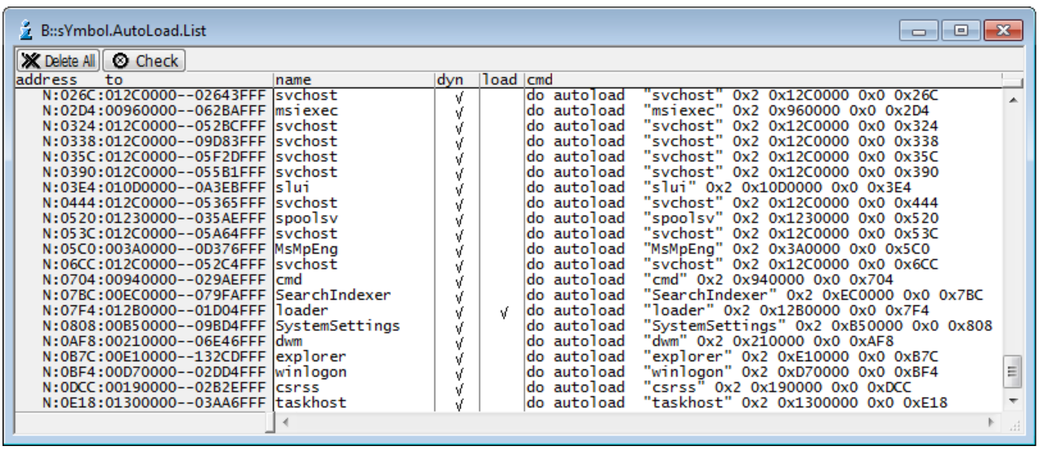
Format:	**sYmbol.AutoLOAD.CHECKWINDOWS** "*<action>*"
<action>:	action to take for symbol load, e.g. "`do autoload`"

The command **sYmbol.AutoLOAD.CHECKWINDOWS** is used to define which action is to be taken, for loading the symbols corresponding to a specific address. The action defined is invoked with specific parameters (see below). With Windows Standard, the pre-defined action is to call the script ~~/demo/*<arch>*/kernel/windows/autoload.cmm.

NOTE:	•	The action parameter needs to be written with quotation marks (for the parser it is a string).
	•	*Defining* this action does not cause its execution.

The action is executed on demand, i.e. when the address is actually accessed by the debugger e.g. in the **List.auto** or **Trace.List** window. In this case the autoloader executes the *<action>* appending parameters indicating the name of the component, its type (process, library, kernel module), the load address and space ID.

For checking the currently active components use the command **sYmbol.AutoLOAD.List**. Together with the component name, it shows details like the load address, the space ID, and the command that will be executed to load the corresponding object files with symbol information. Only components shown in this list are handled by the autoloader.



The symbol autoloader - moreover the script that is invoked by the symbol autoloader (autoload.cmm), takes care of symbol storages. **TASK.sYmbol.Option SymCache** configures a path, where the debugger stores symbol files that it once retrieved, to prevent it from re-loading it from external sources. Additionally an external program (getsymfile.exe) is used to load Microsoft specific symbol files from the public Microsoft symbol server, if desired.

SMP Support

The OS Awareness supports symmetric multiprocessing (SMP).

An SMP system consists of multiple similar CPU cores. The operating system schedules the threads that are ready to execute on any of the available cores, so that several threads may execute in parallel. Consequently an application may run on any available core. Moreover, the core at which the application runs may change over time.

To support such SMP systems, the debugger allows a “system view”, where one TRACE32 PowerView GUI is used for the whole system, i.e. for all cores that are used by the SMP OS. For information about how to set up the debugger with SMP support, please refer to the **Processor Architecture Manuals**.

All core relevant windows (e.g. **Register.view**) show the information of the current core. The status bar of the debugger indicates the current core. You can switch the core view with the **CORE.select** command.

Target breaks, be they manual breaks or halting at a breakpoint, halt all cores synchronously. Similarly, a **Go** command starts all cores synchronously. When halting at a breakpoint, the debugger automatically switches the view to the core that hit the breakpoint.

Because it is undetermined, at which core an application runs, breakpoints are set on all cores simultaneously. This means, the breakpoint will always hit independently on which core the application actually runs.

Crash Dump Analysis

The OS Awareness for Windows Standard implements some facilities to help analyzing a Windows Crash Dump file. The memory image is loaded into e.g the TRACE32 Instruction Set Simulator using the command **Data.LOAD.CrashDump**. Then, the command **TASK.CrashDump.LOADNT** could be used to retrieve and autoload the Windows kernel debug symbols from the Microsoft Symbol Store or from the specified symbol cache directory. After correctly loading the Windows kernel debug symbols, the command **TASK.CrashDump.LOADREG** could be used. This will set the context of all the cores available in the Crash Dump file to the state of the system when the crash happened. The context includes the core registers and some special registers that are relative to the memory management unit configuration.

An example script is available in the folder of the OS Awareness for Windows. It shows how to load and analyze a Windows Crash Dump using the TRACE32 Instruction Set Simulator. Currently, only the architectures Intel® x86 and Intel® x64 are supported.

```
DO ~~/demo/<arch>/kernel/windows/crashdump.cmm MEMORY.DMP
```

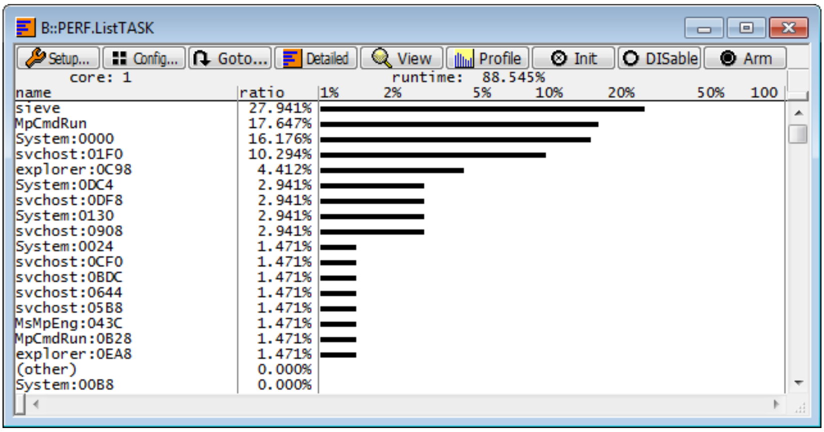
After the Crash Dump is correctly loaded, the developer can inspect the crash reason using the **Frame.view** window. It is also possible to inspect the running processes, threads and libraries, as well as the kernel modules and drivers at the moment of the crash.

Dynamic Task Performance Measurement

The debugger can execute a dynamic performance measurement by evaluating the current running task in changing time intervals. Start the measurement with the commands **PERF.Mode TASK** and **PERF.Arm**, and view the contents with **PERF.ListTASK**. The evaluation is done by reading the ‘magic’ location (= current running task) in memory. This memory read may be non-intrusive or intrusive, depending on the **PERF.METHOD** used.

If **PERF** collects the PC for function profiling of processes in MMU-based operating systems (**SYStem.Option.MMUSPACES ON**), then you need to set **PERF.CONFIG.MMUSPACES**, too.

For a general description of the **PERF** command group, refer to **“General Commands Reference Guide P”** (general_ref_p.pdf).



Task Runtime Statistics

NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **LOGGER**). For details, refer to **“OS-aware Tracing”** in TRACE32 Concepts, page 36 (trace32_concepts.pdf).

Based on the recordings made by the **Trace** (if available), the debugger is able to evaluate the time spent in a task and display it statistically and graphically.

To evaluate the contents of the trace buffer, use these commands:

Trace.List List.TASK DEFault	Display trace buffer and task switches
Trace.STATistic.TASK	Display task runtime statistic evaluation
Trace.Chart.TASK	Display task runtime timechart
Trace.PROfileSTATistic.TASK	Display task runtime within fixed time intervals statistically
Trace.PROfileChart.TASK	Display task runtime within fixed time intervals as colored graph
Trace.FindAll Address TASK.CONFIG(magic)	Display all data access records to the “magic” location
Trace.FindAll CYcle owner OR CYcle context	Display all context ID records

The start of the recording time, when the calculation doesn’t know which task is running, is calculated as “(unknown)”.

Function Runtime Statistics

NOTE:	This feature is *only* available, if your debug environment is able to trace task switches (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate task information (eg. data trace), or a software instrumentation feeding one of TRACE32 software based traces (e.g. **FDX** or **LOGGER**). For details, refer to **“OS-aware Tracing”** in TRACE32 Concepts, page 36 (trace32_concepts.pdf).

All function-related statistic and time chart evaluations can be used with task-specific information. The function timings will be calculated dependent on the task that called this function. To do this, in addition to the function entries and exits, the task switches must be recorded.

To do a selective recording on task-related function runtimes based on the data accesses, use the following command:

```
; Enable flow trace and accesses to the magic location
Break.Set TASK.CONFIG(magic) /TraceData
```

To do a selective recording on task-related function runtimes, based on the Arm Context ID, use the following command:

```
; Enable flow trace with Arm Context ID (e.g. 32bit)
ETM.ContextID 32
```

To evaluate the contents of the trace buffer, use these commands:

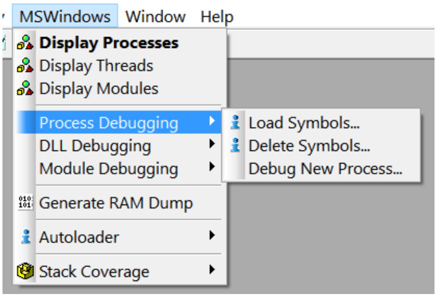
Trace.ListNesting	Display function nesting
Trace.STATistic.Func	Display function runtime statistic
Trace.STATistic.TREE	Display functions as call tree
Trace.STATistic.sYmbol /SplitTASK	Display flat runtime analysis
Trace.Chart.Func	Display function timechart
Trace.Chart.sYmbol /SplitTASK	Display flat runtime timechart

The start of the recording time, when the calculation doesn't know which task is running, is calculated as "(unknown)".

Windows Specific Menu

The OS Awareness for Windows Standard installs a Windows specific menu (win<*x*>.men). You can reload this menu with the **MENU.ReProgram** command.

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



- The **Display** menu items launch the kernel resource display windows. See chapter "**Display of Kernel Resources**".
- **Process Debugging** refers to actions related to process based debugging.
  See also chapter "**Debugging the Process**".
  - Use **Load Symbols** and **Delete Symbols** to load or delete the symbols of a specific process. You may select a symbol file on the host with the **Browse** button. See also **TASK.sYmbol**.
  - **Debug New Process** allows you to start debugging a process on its main() function. Select this prior to starting the process. Specify the name of the process you want to debug. Then start the process on your target machine. The debugger will load the symbols and halt at main(). See also the script "app_debug.cmm".
- **Module Debugging** refers to actions related to kernel module based debugging.
  See also chapter "**Kernel Modules**".
  - Use **Load Symbols** and **Delete Symbols** to load or delete the symbols of a specific kernel module. You may select a symbol file on the host with the **Browse** button. See also **TASK.sYmbol**.
  - **Debug Module on init** allows you to start debugging a kernel module on its init function. Select this prior to loading the module. Specify the name of the module you want to debug. Then load the module on your target machine. The debugger will load the symbols and halt at the init function (if available). See also the demo script "mod_debug.cmm".
- **DLL Debugging** refers to actions related to library based debugging.
  See also chapter "**Debugging into Libraries**".
  - Use **Load Symbols** and **Delete Symbols** to load or delete the symbols of a specific library. Please specify the library name and the process name that uses this library. You may select a symbol file on the host with the **Browse** button. See also **TASK.sYmbol**.
- Use the **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.
- Use **Set Components Checked** to specify, which Windows components should be managed by the autoloader. See also **TASK.sYmbol.Option AutoLOAD**.

In addition, the menu file (*.men) modifies these menus on the TRACE32 main menu bar:

- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only task switches (if any) or task switches together with the default display.
- The **Perf** menu contains additional submenus for task runtime statistics, task-related function runtime statistics or statistics on task states, if a trace is available. See also chapter “**Task Runtime Statistics**”.

Debugging Windows Components

Windows runs on virtual address spaces. The kernel uses a static address translation. For example for Windows 32bit versions the kernel is usually starting from virtual address 0x80000000 mapped to the physical start address of the RAM. Each user process gets its own user address space when loaded, usually starting from virtual 0x0, mapped to any physical RAM area that is currently free. Due to this address translations, debugging the Windows kernel components and user processes requires some settings to the Debugger.

To distinguish those different memory mappings, TRACE32 uses “space IDs”, defining individual address translations for each ID. The kernel itself is attached to the space ID zero. Each process that has its own memory space gets a space ID that is equal to its process ID. Windows threads get the space ID of the parent process.

See also chapter “**MMU Support**”.

Windows Kernel

The Windows awareness needs several kernel symbols, it is necessary to load at least the symbols for the Windows kernel. The command **TASK.sYmbol.LOADNT** could be used to automatically retrieve the symbol file from the Microsoft Symbol Store, and to load the symbols into the debugger. It is also possible to load the symbols from the specified symbol cache directory.

```
TASK.sYmbol.Option SymCache "C:\Symbols"    ; set symbol cache directory
TASK.sYmbol.LOADNT                          ; load the kernel symbols
```

User Processes

Each user process in Windows gets its own virtual memory space, each usually starting at address zero. To distinguish the different memory spaces, the debugger assigns a “space ID”, which is equal to the process ID. Using this space ID, it is possible to address a unique memory location, even if several processes use the same virtual address.

Windows uses the “on demand paging” mechanism to load the code and data of processes and shared libraries. Debugging those pages is not trivial, see “**On Demand Paging**” for details and workaround.

Note that at every time the Windows awareness is used, it needs the kernel symbols. Please see the chapters above on how to load them. Hence, load all process symbols with the option `/NoClear`, to preserve the kernel symbols.

Debugging the Process

To correlate the symbols of a user process with the virtual addresses of this process, it is necessary to load the symbols into this space ID.

Please watch out for demand paging (see chapter “**On Demand Paging**”).

Manually Load Process Symbols:

For example, if you’ve got a a process called “hello.exe” with the process ID **`12.`** (the dot specifies a decimal number!):

```
Data.LOAD.eXe hello.pdb 12.:0 /NoCODE /NoClear
```

The space ID of a process may also be calculated by using the PRACTICE function `task.proc.spaceid()` (see chapter “**PRACTICE Functions**”).

Automatically Load Process Symbols:

If a process name is unique, and if the symbol files are accessible at the standard search paths, you can use an automatic load command

```
TASK.sYmbol.LOAD "hello"              ; load symbols of "hello.exe"
```

This command loads the symbols of “hello”. See **TASK.sYmbol.LOAD** for more information.

Using the Symbol Autoloader:

With the symbol autoloader (see chapter “**Symbol Autoloader**”), the symbols will be automatically loaded when accessing an address inside the process. You can also force the loading of the symbols of a process with

```
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.TOUCH "hello"
```

Using the Menus:

Select the menu item **MSWindows** > **Process Debugging** > **Load Symbols** to load the symbols of a specific process. Alternatively, select **Display Processes**, right click on the “magic” of a process, and select **Load Symbols**.

Debugging a Process From Scratch, Using a Script:

If you want to debug your process right from the beginning (at “main()”), you have to load the symbols *before* starting the process. This is a tricky thing because you have to know the process ID, which is assigned first at the process start-up. The demo directory contains a script “app_debug.cmm” that assists you for this purpose. Call the script with the process name as argument before the process is started:

```
DO ~~/demo/<arch>/kernel/windows/app_debug.cmm hello
```

Then, start the process in Windows. The debugger should automatically halt at the entry point of the process. You can also use the menu item **MSWindows** > **Process Debugging** > **Debug New Process**, which does essentially the same within a dialog.

Debugging into Libraries

If the process uses libraries (DLLs), Windows loads them into the address space of the process. The process itself contains no symbols of the libraries. If you want to debug those libraries, you have to load the corresponding symbols into the debugger.

Please watch out for demand paging (see chapter "**On Demand Paging**").

Manually Load Library Symbols:

1. Start your process and open a **TASK.Process** window.
2. Double-click the magic value of the process that uses the library.
3. Expand the "modules" tree (if available).

   A list will appear that shows the loaded libraries and the corresponding base addresses.
4. Load the symbols to this address and into the space ID of the process.

   E.g. if the process has the space ID `12.`, the library is called "mylib.dll" and it is loaded on address 0x76550000, then use the command:

   ```
   Data.LOAD.EXE mylib.pdb 12.:0x76550000 /NoCODE /NoClear
   ```

   Of course, this library must be compiled with debugging information.

Automatically Load Library Symbols:

If a library name is unique, and if the symbol files are accessible at the standard search paths, you can use an automatic load command:

```
TASK.sYmbol.LOADLib "hello" "mylib.dll"    ; load library symbols
```

This command loads the symbols of the library "mylib.dll", used by the process "hello". See **TASK.sYmbol.LOADDLL** for more information.

Using the Symbol Autoloader:

With the symbol autoloader (see chapter "**Symbol Autoloader**"), the symbols will be automatically loaded when accessing an address inside the library. You can also force the loading of the symbols of a library with

```
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.TOUCH "mylib.dll"
```

Using the Menus:

Select the menu item **MSWindows** > **DLL Debugging** > **Load Symbols** to load the symbols of a specific library. Alternatively, select **Display Processes**, double click on the “magic” of the process, expand the “modules” section, right click on the “magic” of a library and select **Load Symbols**.

Debugging Windows Threads

Windows threads share the same virtual memory of the parent process. The OS Awareness for Windows Standard assigns one space ID for all threads that belong to a specific process. It is sufficient, to load the debug information of this process only once (onto its space ID) to debug all threads of this process. See chapter “**Debugging the Process**” for loading the process’ symbols.

The **TASK.Thread** window shows which thread is currently running (“current”).

On Demand Paging

When a process is started, Windows doesn’t load any code or data of this process. Instead, it uses the “on demand paging” mechanism. This means, Windows loads memory pages first, when they are accessed. As long as they aren't accessed by the CPU, they're not present in the system.

A “memory page” is a 4 KByte continuous memory region, with a dedicated virtual and physical address range. The MMU handles the whole (user space) memory in such pages.

When starting a process, Windows just sets up its kernel structures and loads the characteristics of the process’ code and data sections from the process’ file (size and addresses of the sections), but not the sections themselves. Then the kernel jumps to the “main” routine of the process. The first instruction fetch will then cause a code page fault, because the code is not yet present. The page fault handler then loads the actual code page (4 KByte) that contains the code of the “main” entry point, from the file. Note that only one page is loaded. If the program jumps to a location outside this page, or steps over a page boundary, another code page fault happens. While running, more and more pages will be loaded. Note that, if RAM becomes low, pages may also be discarded. If a process terminates, all pages of this process are removed.

The same page loading mechanism applies to data and stack addresses. Variables are first visible to the system, after the CPU accessed them (by reading or writing the address and thus urging a page load). The stack grows page wise, as it is used.

When debugging those paged processes, you have to take care about this paging.

- The process’ code and data is first visible to the debugger, after the pages were loaded.
- You cannot set a software breakpoint onto a function that is located in a page which is not yet loaded. The code for this function simply not yet exists, and thus cannot be patched with the breakpoint instruction. In such cases, use on-chip breakpoints instead.
- The CPU handles on-chip breakpoints *before* code page faults. If the CPU jumps onto an on-chip breakpoint, and the appropriate page is not yet loaded, the debugger will halt before the page is loaded. You’ll see the program counter on a location with no actual code (usually the debugger shows “???” then). The same may happen, if you single step over a page boundary. In such cases, set an on-chip(!) breakpoint onto the next instruction and let the system “Go”. The page fault handler will then load the page, the processor will execute the first instruction and halt on the next breakpoint. A simple workaround for functions is to set the breakpoint at the function entry plus 4 (e.g. "main+4"). Then the application will halt *after* the page was loaded.

The on demand paging is a basic design feature of Windows that cannot be switched off.

Kernel Modules

Kernel modules, aka device drivers, are dynamically loaded and linked by the kernel into the kernel space. If you want to debug kernel modules, you have to load the symbols of the kernel module into the debugger, and to relocate the code and data address information.

Manually Load Module Symbols:

Load your module and open a **TASK.MODule** window. A list will appear that shows all loaded modules and the corresponding base addresses. Load the symbols to this address and into space ID zero (kernel). E.g. if the module is called “mydriver.sys” and it is loaded on address 0x95DD3000, use the command:

```
Data.LOAD.EXE mydriver.pdb 0:0x39DD30000 /NoCODE /NoClear
```

Of course, this module must be compiled with debugging information.

Automatically Load Module Symbols:

If a module name is unique, and if the symbol files are accessible at the standard search paths, you can use an automatic load command

```
TASK.sYmbol.LOADKM "mydriver"              ; load module symbols
```

This command loads the symbols of the module “mydriver.sys”. See **TASK.sYmbol.LOADKM** for more information.

Using the Symbol Autoloader:

With the symbol autoloader (see chapter “**Symbol Autoloader**”), the symbols will be automatically loaded when accessing an address inside the kernel module. You can also force the loading of the symbols of a kernel module with

```
sYmbol.AutoLOAD.CHECK
sYmbol.AutoLOAD.TOUCH "mydriver"
```

Using the Menus:

Select the menu item **MSWindows** > **Module Debugging** > **Load Symbols** to load the symbols of a specific process. Alternatively, select **Display Modules**, right click on the “magic” of a module, and select **Load Symbols**.

Debugging the kernel module’s init routine:

If you want to debug your module's init routine, you have to load the symbols *before* initializing the module. The demo directory contains a script "mod_debug.cmm" that assists you for this purpose. Call the script with the module name as argument before the module is loaded:

```
DO ~~/demo/<arch>/kernel/windows/mod_debug.cmm mydriver
```

Then, load the module in Windows. The debugger should automatically halt at the entry point of the module. You can also use the menu item **MSWindows** > **Module Debugging** > **Debug Module on init**, which does essentially the same within a dialog.

TASK.CrashDump

Windows crash dump analysis

Format:	**TASK.CrashDump**

The **TASK.CrashDump** command group helps to analyze the Windows crash dump:

TASK.CrashDump.LOADNT	Load the kernel debug symbols of the loaded memory dump.
TASK.CrashDump.LOADREG	Load the context of all the available cores from the loaded memory dump.

TASK.CrashDump.LOADNT

Load the kernel debug symbols

Format:	**TASK.CrashDump.LOADNT** *<address>*

The **TASK.CrashDump.LOADNT** command helps to retrieve and auto-load the Windows kernel debug symbols that are relative to the loaded memory dump. The address parameter specifies the virtual address of the kernel debugger data block (KdDebuggerDataBlock).

This address could easily be found in the crash dump file header at a fixed offset depending on the windows bit-ness. For a 32-bit Windows the KdDebuggerDataBlock is a 4 bytes address at the file offset 0x60 and for 64-bit Windows KdDebuggerDataBlock is an 8 bytes address at the file offset 0x80.

TASK.CrashDump.LOADREG

Load the registers from the crash dump

Format:	**TASK.CrashDump.LOADREG**

The **TASK.CrashDump.LOADREG** command helps to load the context of all the cores available in the Crash Dump file to the state of the system when the crash happened. The context include the core registers and some special registers that are relative to the memory management unit configuration.

TASK.KDBG.SET

Set kernel debugger data block address

Format:	**TASK.KDBG.SET** *<address>*

This command sets the virtual address of the kernel debugger data block (KdDebuggerDataBlock). This address is changing each time the target is started.

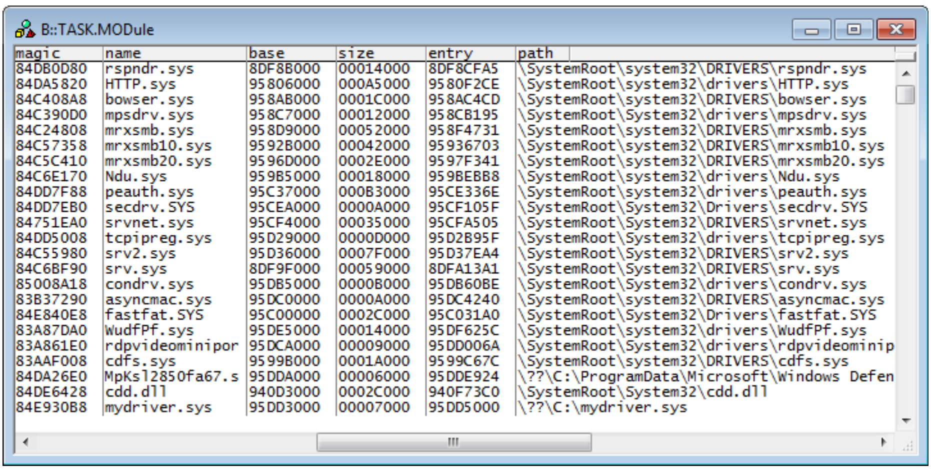
For the Windows awareness version November 2016 and newer, setting the kernel debugger data block address is no longer needed when working on a live debug session. This may help in postmortem debug session when loading a raw memory dump into TRACE32 simulator.

TASK.MODule

Display kernel modules

Format:	**TASK.MODule**

Displays a table with all loaded kernel modules / device drivers of Windows.


B::TASK.MODule

magic	name	base	size	entry	path
84DB0D80	rspndr.sys	8DF8B000	00014000	8DF8CFA5	\SystemRoot\system32\DRIVERS\rspndr.sys
84DA5820	HTTP.sys	95806000	000A5000	9580F2CE	\SystemRoot\system32\drivers\HTTP.sys
84C408A8	bowser.sys	958AB000	0001C000	958AC4CD	\SystemRoot\system32\DRIVERS\bowser.sys
84C390D0	mpsdrv.sys	958C7000	00012000	958CB195	\SystemRoot\System32\drivers\mpsdrv.sys
84C24808	mrxsmb.sys	958D9000	00052000	958F4731	\SystemRoot\system32\DRIVERS\mrxsmb.sys
84C57358	mrxsmb10.sys	9592B000	00042000	95936703	\SystemRoot\system32\DRIVERS\mrxsmb10.sys
84C5C410	mrxsmb20.sys	9596D000	0002E000	9597F341	\SystemRoot\system32\DRIVERS\mrxsmb20.sys
84C6E170	Ndu.sys	959B5000	00018000	959BEBB8	\SystemRoot\system32\drivers\Ndu.sys
84DD7F88	peauth.sys	95C37000	000B3000	95CE336E	\SystemRoot\system32\drivers\peauth.sys
84DD7EB0	secdrv.SYS	95CEA000	0000A000	95CF105F	\SystemRoot\System32\Drivers\secdrv.SYS
84751EA0	srvnet.sys	95CF4000	00035000	95CFA505	\SystemRoot\System32\DRIVERS\srvnet.sys
84DD5008	tcpipreg.sys	95D29000	0000D000	95D2B95F	\SystemRoot\System32\drivers\tcpipreg.sys
84C55980	srv2.sys	95D36000	0007F000	95D37EA4	\SystemRoot\System32\DRIVERS\srv2.sys
84C6BF90	srv.sys	8DF9F000	00059000	8DFA13A1	\SystemRoot\System32\DRIVERS\srv.sys
85008A18	condrv.sys	95DB5000	0000B000	95DB60BE	\SystemRoot\System32\drivers\condrv.sys
83B37290	asyncmac.sys	95DC0000	0000A000	95DC4240	\SystemRoot\system32\DRIVERS\asyncmac.sys
84E840E8	fastfat.SYS	95C00000	0002C000	95C031A0	\SystemRoot\System32\Drivers\fastfat.SYS
83A87DA0	WudfPf.sys	95DE5000	00014000	95DF625C	\SystemRoot\system32\drivers\WudfPf.sys
83A861E0	rdpvideominipor	95DCA000	00009000	95DD006A	\SystemRoot\System32\drivers\rdpvideominip
83AAF008	cdfs.sys	9599B000	0001A000	9599C67C	\SystemRoot\system32\DRIVERS\cdfs.sys
84DA26E0	MpKsl2850fa67.s	95DDA000	00006000	95DDE924	\??\C:\ProgramData\Microsoft\Windows Defen
84DE6428	cdd.dll	940D3000	0002C000	940F73C0	\SystemRoot\System32\cdd.dll
84E930B8	mydriver.sys	95DD3000	00007000	95DD5000	\??\C:\mydriver.sys

“magic” is a unique ID, used by the OS Awareness to identify a module.
The field “magic” is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

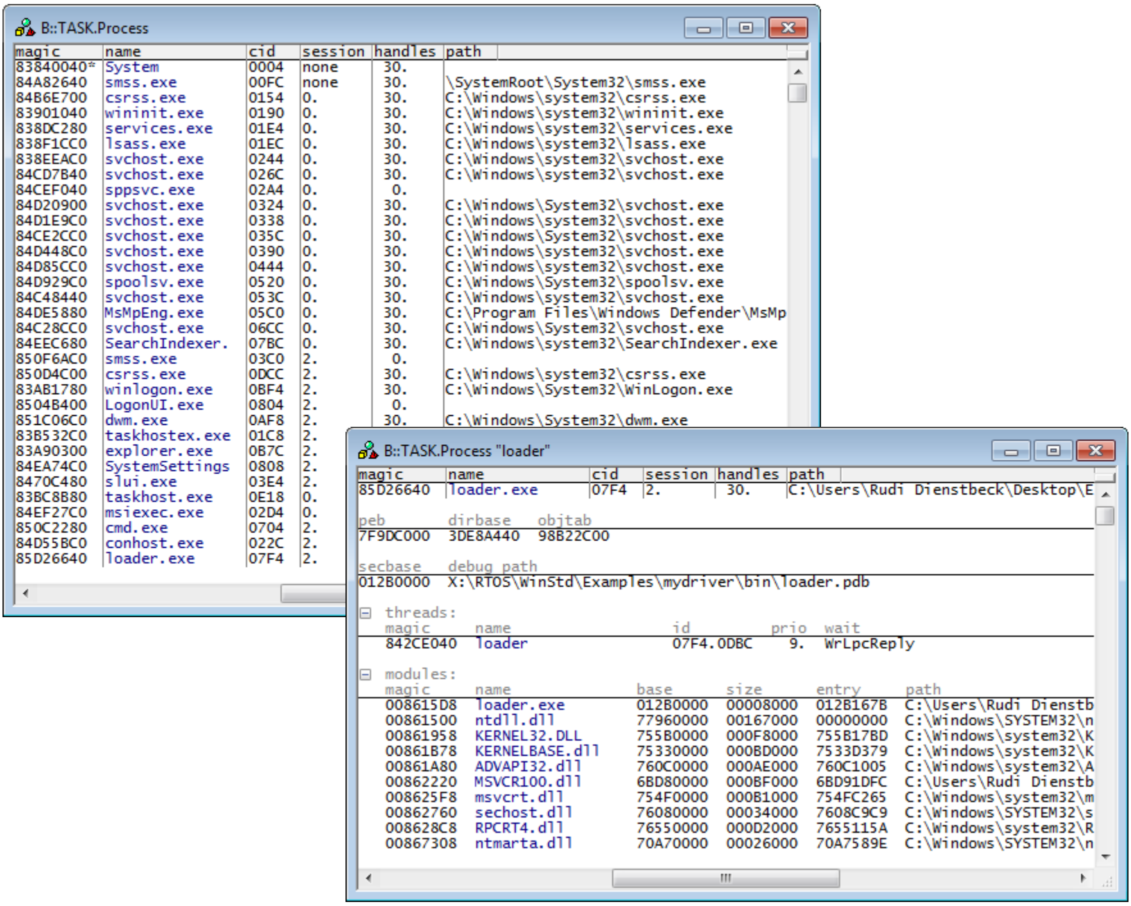
Format:	**TASK.NTBASE** *<address>*

The Windows extension tries to detect the kernel base address when loading the kernel symbols using the command **TASK.sYmbol.LOADNT.** If any, the kernel base address detection fails, this command could be used to set it manually.

Format:	**TASK.Process** [*<process>*]

Display all active processes or detailed information about one specific process.

Without any arguments, this command displays a table with all created processes.
Specify a process magic, ID or name to see the detailed information on this process.



“magic” is a unique ID, used by the OS Awareness to identify a specific process.
The fields “magic”, “name” and other detailed fields are mouse sensitive, double clicking on them open appropriate windows. Right clicking on them will show a local menu.

Format:	**TASK.sYmbol**

The **TASK.sYmbol** command group helps to load and unload symbols of a given process or module. In particular the commands are:

TASK.sYmbol.LOAD	Load process symbols
TASK.sYmbol.DELete	Unload process symbols
TASK.sYmbol.LOADNT	Load the kernel symbols
TASK.sYmbol.LOADKM	Load module symbols
TASK.sYmbol.DELeteKM	Unload module symbols
TASK.sYmbol.LOADUM	Load uefi module symbols
TASK.sYmbol.DELeteUM	Unload uefi module symbols
TASK.sYmbol.LOADDLL	Load library symbols
TASK.sYmbol.DELeteDLL	Unload library symbols
TASK.sYmbol.Option	Set symbol management options

TASK.sYmbol.DELete Unload process symbols

Format:	**TASK.sYmbol.DELete** *<process>*

When debugging of a process is finished, or if the process exited, you should remove loaded process symbols. Otherwise the remaining entries may interfere with further debugging.
This command deletes the symbols of the specified process.

<process>	Specify the process name or path (in quotes) or magic to unload the symbols of this process.

TASK.sYmbol.DELeteDLL

Unload library symbols

Format:	**TASK.sYmbol.DELeteDLL** *<process> <library>*

When debugging of a library is finished, or if the library is removed from the kernel, you should remove loaded library symbols. Otherwise the remaining entries may interfere with further debugging. This command deletes the symbols of the specified library.

<process>	As first parameter, specify the process to which the desired library belongs (name in quotes or magic).
<library>	Specify the library name in quotes as second parameter. The library name **must** match the name as shown in **TASK.Process** *<process>*, ”modules”.

See also chapter “**Debugging Into Shared Libraries**”.

TASK.sYmbol.DELeteKM

Unload module symbols

Format:	**TASK.sYmbol.DELeteKM** *<module>*

Specify the module name (in quotes) or magic to unload the symbols of this kernel module.

When debugging of a module is finished, or if the module is removed from the kernel, you should remove loaded module symbols. Otherwise the remaining entries may interfere with further debugging.
This command deletes the symbols of the specified module.

See also chapter “**Debugging Kernel Modules**”.

TASK.sYmbol.DELeteUM

Unload UEFI module symbols

Format:	**TASK.sYmbol.DELeteUM** *<umodule>*

Specify the uefi runtime service module name (in quotes) or the magic to unload the symbols of this module.

This command deletes the symbols of the specified UEFI module.

TASK.sYmbol.LOAD

Load process symbols

Format:	**TASK.sYmbol.LOAD** *<process>*

Specify the process name or path (in quotes) or magic to load the symbols of this process.

In order to debug a user process, the debugger needs the symbols of this process (see chapter “Debugging User Processes”).

This command retrieves the appropriate space ID and loads the symbol file of an existing process. Note that this command works only with processes that are already loaded in Windows (i.e. processes that show up in the **TASK.Process** window).

TASK.sYmbol.LOADDLL

Load library symbols

Format:	**TASK.sYmbol.LOADDLL** *<process> <library>*

In order to debug a library, the debugger needs the symbols of this library, relocated to the correct addresses where Windows linked this library. This command retrieves the appropriate load addresses and loads the symbol file of an existing library. Note that this command works only with libraries that are already loaded in Windows (i.e. processes that show up in the **TASK.Process** *<process>* window).

<process>	Specify the process to which the desired library belongs (name in quotes or magic).
<library>	Specify the library name in quotes. The library name **must** match the name as shown in **TASK.Process** *<process>*, ”modules”.

TASK.sYmbol.LOADKM — Load module symbols

Format:	**TASK.sYmbol.LOADKM** *<module>*

In order to debug a kernel module, the debugger needs the symbols of this module. This command retrieves the appropriate load addresses and loads the symbol file of an existing module. Note that this command works only with modules that are already loaded in Windows (i.e. modules that show up in the **TASK.MODule** window).

<module>	Specify the module name (in quotes) or magic to load the symbols of this module.

See also chapter “**Debugging Kernel Modules**”.

TASK.sYmbol.LOADNT — Load the kernel symbols

Format:	**TASK.sYmbol.LOADNT**

This command tries to locate the kernel base address and load the Windows kernel symbols.

TASK.sYmbol.LOADUM — Load UEFI runtime service module symbols

Format:	**TASK.sYmbol.LOADUM***<umodule>*

In order to debug a UEFI runtime service module, the debugger needs the symbols of this module. This command retrieves the appropriate load addresses and loads the symbol file of an existing uefi module. Note that this command works only with modules that are already loaded in Windows (i.e. modules that show up in the **TASK.UefiMODule** window).

<umodule>	Specify the uefi module name (in quotes) or magic to load the symbols of this module.

Format:	**TASK.sYmbol.Option** *<option>*
<option>:	**AutoLoad** *<option>* **sYmCache** *<path>*

Set a specific option to the symbol management.

AutoLoad:
This option controls, which components are checked and managed by the **symbol autoloader**:

Process	Check processes
Library	Check all libraries of all processes
KModule	Check kernel modules
UModule	Check UEFI modules
CurrLib	Check only libraries of current process
ProcLib *<process>*	Check libraries of specified process
ALL	Check processes, libraries and kernel modules
NoProcess	Don't check processes
NoLibrary	Don't check libraries
NoKModule	Don't check kernel modules
NoUModule	Don't check UEFI modules
NONE	Check nothing.

The options are set *additionally*, not removing previous settings.

The default is "Process", i.e. only the processes are checked by the symbol autoloader.

Example:

```
; check processes and kernel modules
TASK.sYmbol.Option AutoLoad Process
TASK.sYmbol.Option AutoLoad KModule
```

sYmCache:

If this option is set, the symbol autoloader tries to find the symbol files in the specified symbol cache directory. The directory is populated and searched referencing the file's name and GUID.

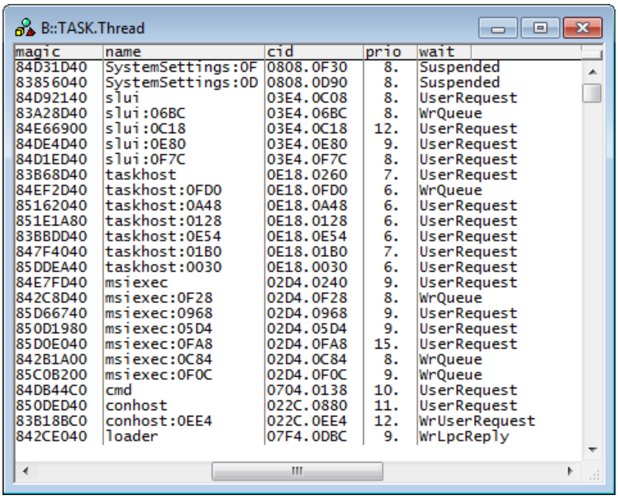
TASK.Thread

Display threads

Format:	**TASK.Thread** [*<thread>*]

Displays the thread table of Windows or detailed information about one specific task.

Without any arguments, a table with all created threads will be shown.
Specify a thread name, ID or magic number to display detailed information on that thread.


B::TASK.Thread

magic	name	cid	prio	wait
84D31D40	SystemSettings:0F	0808.0F30	8.	Suspended
83856040	SystemSettings:0D	0808.0D90	8.	Suspended
84D92140	slui	03E4.0C08	8.	UserRequest
83A28D40	slui:06BC	03E4.06BC	8.	WrQueue
84E66900	slui:0C18	03E4.0C18	12.	UserRequest
84DE4D40	slui:0E80	03E4.0E80	9.	UserRequest
84D1ED40	slui:0F7C	03E4.0F7C	8.	UserRequest
83B68D40	taskhost	0E18.0260	7.	UserRequest
84EF2D40	taskhost:0FD0	0E18.0FD0	6.	WrQueue
85162040	taskhost:0A48	0E18.0A48	6.	UserRequest
851E1A80	taskhost:0128	0E18.0128	6.	UserRequest
83BBDD40	taskhost:0E54	0E18.0E54	6.	UserRequest
847F4040	taskhost:01B0	0E18.01B0	7.	UserRequest
85DDEA40	taskhost:0030	0E18.0030	6.	UserRequest
84E7FD40	msiexec	02D4.0240	9.	UserRequest
842C8D40	msiexec:0F28	02D4.0F28	8.	WrQueue
85D66740	msiexec:0968	02D4.0968	9.	UserRequest
850D1980	msiexec:05D4	02D4.05D4	9.	UserRequest
85D0E040	msiexec:0FA8	02D4.0FA8	15.	UserRequest
842B1A00	msiexec:0C84	02D4.0C84	8.	WrQueue
85C0B200	msiexec:0F0C	02D4.0F0C	9.	WrQueue
84DB44C0	cmd	0704.0138	10.	UserRequest
850DED40	conhost	022C.0880	11.	UserRequest
83B18BC0	conhost:0EE4	022C.0EE4	12.	WrUserRequest
842CE040	loader	07F4.0DBC	9.	WrLpcReply

“magic” is a unique ID, used by the OS Awareness to identify a specific thread.
The fields “magic”, “name” and other detailed fields are mouse sensitive, double clicking on them open appropriate windows. Right clicking on them will show a local menu.

TASK.UefiMODule

Display UEFI runtime service modules

Format:	**TASK.UefiMODule** [*<umodule>*]

Displays a table with all UEFI runtime service modules used by windows. This requires that the UEFI bios is compiled with debug information.

PRACTICE Functions

There are special definitions for Windows specific PRACTICE functions.

TASK.CONFIG() OS Awareness configuration information

Syntax:	**TASK.CONFIG(**magic | magicsize**)**

Parameter and Description:

magic	**Parameter Type**: String (***without*** quotation marks). Returns the magic address, which is the location that contains the currently running task (i.e. its task magic number).
magicsize	**Parameter Type**: String (***without*** quotation marks). Returns the size of the task magic number (1, 2 or 4).

Return Value Type: Hex value.

TASK.KDBG() Kernel debugger data block

Syntax:	**TASK.KDBG()**

Returns the kernel debugger data block as configured by the extension.

Return Value Type: Hex value.

TASK.KERNELPT() Kernel page table

Syntax:	**TASK.KERNELPT()**

Returns the kernel page table.

Return Value Type: Hex value.

TASK.LIB.DEBUG() Library with debug information

Syntax: **TASK.LIB.DEBUG(**<*librarymagic*>**,** <*process_magic*>**)**

Returns if debug information could be detected from library, loaded by the specified process.

Parameter and Description: **Parameter Type**: Hex value.

<*librarymagic*>	**Parameter Type**: Hex value.
<*process_magic*>	**Parameter Type**: Hex value.

Return Value and Description:

0	debug information couldn't be detected.
1	debug information could be detected.

TASK.LIB.GUID() GUID of library

Syntax: **TASK.LIB.GUID(**<*librarymagic*>**,** <*process_magic*>**)**

Returns the GUID of the library, loaded by the specified process.

Parameter and Description:

<*librarymagic*>	**Parameter Type**: Hex value.
<*process_magic*>	**Parameter Type**: Hex value.

Return Value Type: String.

TASK.LIB.MACHINE()

32bit or 64bit setting of library

Syntax: **TASK.LIB.MACHINE(***<library_magic>***,** *<process_magic>***)**

Returns the detected 32bit/64bit setting of the library loaded by the specified process.

Parameter and Description:

<library_magic>	**Parameter Type**: Hex value.
<process_magic>	**Parameter Type**: Hex value.

Return Value Type: Hex value.

Return Value and Description:

0	0 for 32bit.
1	1 for 64bit.

TASK.LIB.MAGIC()

Magic number of library

Syntax: **TASK.LIB.MAGIC("***<library_name>***",** *<process_magic>***)**

Returns the “magic number” of the library, loaded by the specified process.

Parameter and Description:

<library_name>	**Parameter Type**: String (*with* quotation marks).
<process_magic>	**Parameter Type**: Hex value.

Return Value Type: Hex value.

TASK.LIB.PDBPATH() Path to PDB file of library

Syntax:	**TASK.MOD.PDBPATH(***<library_magic>*,*<process_magic>***)**

Returns the path to the PDB file of the library, loaded by the specified process.

Parameter and Description:

<library_magic>	**Parameter Type**: Hex value.
<process_magic>	**Parameter Type**: Hex value.

Return Value Type: String.

TASK.MOD.BASE() Base address of module

Syntax:	**TASK.MOD.BASE(***<module_magic>***)**

Returns the base address of the module.

Parameter Type: Hex value.

Return Value Type: Hex value.

TASK.MOD.DEBUG() Module with debug information

Syntax:	**TASK.MOD.DEBUG(***<module_magic>***)**

Returns if debug information could be detected from the loaded module.

Parameter Type: Hex value.

Return Value Type: Hex value.

Return Value and Description:

0	debug information couldn't be detected.
1	debug information could be detected.

TASK.MOD.ENTRY() Entry address of module

Syntax: **TASK.MOD.ENTRY(***<module_magic>***)**

Returns the entry address of the module.

Parameter Type: Hex value.

Return Value Type: Hex value.

TASK.MOD.GUID() GUID of module

Syntax: **TASK.MOD.GUID(***<module_magic>***)**

Returns the GUID of the module magic.

Parameter Type: Hex value.

Return Value Type: String.

TASK.MOD.MACHINE() 32bit or 64bit setting of the module

Syntax: **TASK.MOD.MACHINE(***<module_magic>***)**

Returns the detected 32bit/64bit setting of the module.

Parameter Type: Hex value.

Return Value Type: Hex value.

Return Value and Description:

0	0 for 32bit.
1	1 for 64bit.

TASK.MOD.MAGIC() Magic number of module name

Syntax: **TASK.MOD.MAGIC("***<module_name>***")**

Returns the “magic” value of the module.

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

Return Value Type: Hex value.

TASK.MOD.PDBPATH() Path to PDB file of module

Syntax: **TASK.MOD.PDBPATH(***<module_magic>***)**

Returns the path to the PDB file of the module.

Parameter Type: Hex value.

Return Value Type: String.

TASK.MOD.YF2M() Magic number of module symbol file

Syntax: **TASK.MOD.YF2M("***<modulesymfile>***")**

Returns the “magic number” of the module that fits to the given symbol file.

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

Return Value Type: Hex value.

TASK.NTBASE() Kernel base address

Syntax: **TASK.NTBASE()**

Returns the kernel base address as located by the extension.

Return Value Type: Hex value.

TASK.PHYMEMBLOCK() Kernel physical memory descriptor

Syntax: **TASK.PHYMEMBLOCK()**

Returns the address of physical memory blocks descriptor as configured by Windows.

Return Value Type: Hex value.

TASK.PROC.DEBUG() Process with debug information

Syntax: **TASK.PROC.DEBUG(**<*process_magic*>**)**

Returns if debug information could be detected from the loaded process.

Parameter Type: Hex value.

Return Value Type: Hex value.

Return Value and Description:

0	debug information couldn't be detected.
1	debug information could be detected.

TASK.PROC.GUID() GUID of the process magic

Syntax: **TASK.PROC.GUID(**<*process_magic*>**)**

Returns the GUID of the process magic.

Parameter Type: Hex value.

Return Value Type: String.

TASK.PROC.MACHINE() 32-bit or 64-bit setting of process

Syntax:	**TASK.PROC.MACHINE(**<*process_magic*>**)**

Returns the detected 32-bit/64-bit setting of the process.

Parameter Type: Hex value.

Return Value Type: Hex value.

Return Value and Description:

0	0 for 32-bit.
1	1 for 64-bit.

TASK.PROC.MAGIC() Magic value of process

Syntax:	**TASK.PROC.MAGIC("**<*process_name*>**")**

Returns the “magic” value of the process.

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

Return Value Type: Hex value.

TASK.PROC.PDBPATH() Path to PDB file of process

Syntax:	**TASK.PROC.PDBPATH(**<*process_magic*>**)**

Returns the path to the PDB file of the process.

Parameter Type: Hex value.

Return Value Type: String.

TASK.PROC.SID2MAGIC() Magic number of process

Syntax: **TASK.PROC.SID2MAGIC(***<space_id>***)**

Returns the “magic number” of the process that fits to the given space ID.

Parameter Type: Hex value.

Return Value Type: Hex value.

TASK.PROC.SPACEID() Space ID of process

Syntax: **TASK.PROC.SPACEID("***<process_name>***")**

Returns the space ID of the specified process.

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

Return Value Type: Hex value.

TASK.PROC.TRACEID() Trace ID of process

Syntax: **TASK.PROC.TRACEID("***<process_name>***")**

Returns the trace ID of the specified process.

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

Return Value Type: Hex value.

TASK.UMOD.MACHINE()

32-bit or 64-bit setting of UEFI module

Syntax:	**TASK.UMOD.MACHINE(***<umod_magic>***)**

Returns the detected 32-bit/64-bit setting of the UEFI module.

Parameter Type: Hex value.

Return Value Type: Hex value.

Return Value and Description:

0	0 for 32-bit.
1	1 for 64-bit.

TASK.UMOD.MAGIC()

Magic value of UEFI module

Syntax:	**TASK.UMOD.MAGIC("***<umod_name>***")**

Returns the “magic” value of the UEFI module.

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

Return Value Type: Hex value.

TASK.UMOD.PDBPATH()

Path to PDB file of UEFI module

Syntax:	**TASK.UMOD.PDBPATH(***<umod_magic>***)**

Returns the path to the PDB file of the uefi module.

Parameter Type: Hex value.

Return Value Type: String.