# OS Awareness Manual Windows CE6/EC7/EC20

# OS Awareness Manual Windows CE6/EC7/EC20

**Version 06-Jun-2024**

## History

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

## Overview

The OS Awareness for Windows CE6/EC7/EC2013 contains special extensions to the TRACE32 Debugger. This manual describes the additional features, such as additional commands and statistic evaluations.

## Terminology

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

## 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.

## Supported Versions

Currently Windows CE is supported for the following versions:

- Windows CE 6.0 on ARM architecture, MIPS, SH4 and x86
- Windows EC 7 on ARM architecture and x86
- Windows EC 2013 on ARM architecture and x86

# Configuration

The **TASK.CONFIG** command loads an extension definition file called “wince6.t32” or “wince7.t32” (directory “~~/demo/*<processor>*/kernel/wince/ce*<ver>*”). It contains all necessary extensions.

Automatic configuration tries to locate the Windows CE internals automatically. For this purpose all Windows CE tables must be accessible at any time the OS Awareness is used.

If a system address is not available or if another address should be used for a specific system variable then the corresponding argument must be set manually with the appropriate address. In this case, use the manual configuration, which can require some additional arguments.

If you want to display the OS objects “On The Fly” while the target is running, you need to have access to memory while the target is running. In case of ICD, you have to enable **SYStem.MemAccess** or **SYStem.CpuAccess** (CPU dependent).

# Manual Configuration

It is highly recommended to use the **Automatic Configuration**.

Only, if your Windows CE image is built with unusual default values, you may try to use the manual configuration:

| | |
|---|---|
| Format: | **TASK.CONFIG** *<file> <magic_address> <kdata>* |
| *<file>*: | **wince6** \| **wince7** \| **wince8** |

*<magic_address>* Specifies the address, where the OS Awareness finds the current running thread. This is found in kernel internal structures and calculated automatically. Set it to “0”.

*<args>* This argument defines the start address of the Windows CE Kernel Data Table.

See **Hooks & Internals** for details.

## Automatic Configuration

For system resource display and trace functionality, you can do an automatic configuration of the OS Awareness. For this purpose it is necessary that all Windows CE internal tables are accessible at any time, the OS Awareness is used. Each of the **TASK.CONFIG** arguments can be substituted by '0', which means that this argument will be searched and configured automatically. For a fully automatic configuration omit all arguments:

| | |
|---|---|
| Format: | **TASK.CONFIG** *<file>* |
| *<file>*: | **wince6** \| **wince7** \| **wince8** |

If a system address is not available, or if another address should be used for a specific system variable, then the corresponding argument must be set manually with the appropriate address (see **Manual Configuration**).

See **Hooks & Internals** for details.

See also the example “`~~/demo/`*`<processor>`*`/kernel/wince/`*`<version>`*`/wince.cmm`”.

## 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 script (`~~/demo/`*`<processor>`*`/kernel/wince/`*`<version>`*`/wince.cmm`).
2. Make a copy of the PRACTICE script file “wince.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).

Now you can access the Windows CE extensions through the menu.

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

## Hooks & Internals in Windows CE

No hooks are used in the kernel.

For retrieving the kernel data structures, the OS Awareness uses the global kernel data table. Ensure that access to this table is possible every time when features of the OS Awareness are used.

Use the debug build of your platform.

# Features

The OS Awareness for Windows CE supports the following features.

## Display of Kernel Resources

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

| | |
|---|---|
| **TASK.Process** | Processes |
| **TASK.Thread** | Threads |
| **TASK.HaNDle** | Global handles |
| **TASK.ROM.MODule** | Built-in modules |
| **TASK.ROM.FILE** | Built-in files |

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

## Thread Stack Coverage

Use this feature with caution: In general, the paging mechanism of Windows CE prevents access of the stack areas of non-running threads. Thus, the display may show only the addresses without the actual maximum stack usage.

For stack usage coverage of Windows CE threads, you can use the **TASK.STacK** command. Without any parameter, this command will set up a window with all active Windows CE threads. If you specify only a magic number as parameter, the stack area of this threads will be automatically calculated.

To use the calculation of the maximum stack usage, flag memory must be mapped to the task stack areas, when working with the emulation memory. When working with the target memory, a stack pattern must be defined with the command **TASK.STacK.PATtern** (default value is zero).

To add/remove one thread to/from the thread stack coverage, you can either call the **TASK.STacK.ADD** rsp. **TASK.STacK.ReMove** commands with the thread magic number as parameter, or omit the parameter and select from the thread list window.

It is recommended to display only the thread you are interested in, because the evaluation of the used stack space is very time consuming and slows down the debugger display.

# 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**, **Data.List**, **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.

The **TASK.Thread <thread>** window contains a button ("context") to execute this command with the displayed thread, and to switch back to the current context ("current").

# 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** commands refer to this necessity.

## Windows CE MMU Basics

Windows CE divides the 32bit virtual address range into several areas.

The kernel address space covers the address range 0x80000000--0xFFFFFFFF.

All processes run in the same virtual address range that is 0x0--0x7FFFFFFF. If a process switch occurs, the MMU of the CPU is reprogrammed, so that this address range points to the current running process.

DLLs are mapped commonly for all processes into the address range 0x40000000--0x5fffffff.

The address range 0x70000000--0x7fffffff contains a shared heap, common to all processes.

## Space IDs

Processes of Windows CE may reside virtually on the same address. To distinguish those addresses, the Debugger uses an additional space ID that specifies to which virtual memory space the address refers. The command **SYStem.Option.MMUSPACES ON** enables the additional space ID. Space ID zero is reserved for the kernel. The space ID of processes equaled their process ID.

You may scan the whole system for space IDs using the command **TRANSlation.ScanID**. Use **TRANSlation.ListID** to get a list of all recognized space IDs.

The function **task.proc.spaceid("<process>")** returns the space ID for a given process. If the space ID is not equal to zero, load the symbols of a process to this space ID:

```
LOCAL &spaceid
&spaceid=task.proc.spaceid("myProcess")
Data.LOAD.eXe myProcess.exe &spaceid:0 /NoCODE /NoClear
```

See also chapter **"Debugging User Processes and DLLs"**.

## MMU Declaration

To access the virtual and physical addresses correctly, the debugger needs to know the format of the MMU tables in the target.

The following command is used to declare the basic format of MMU tables:

| | |
|---|---|
| **MMU.FORMAT** *<format>* [*<base_address>* [*<logical_kernel_address_range>*<br>*<physical_kernel_address>*]] | Define MMU table structure |

### <format> Options for ARM:

| <format> | Description |
|---|---|
| **STD** | Standard format defined by the CPU |
| **TINY** | MMU format using a tiny page size of only 1024 bytes |
| **WINCE6** | Format used by Windows CE6 / EC7 / EC2013 |

### <format> Options for MIPS:

| <format> | Description |
|---|---|
| **WINCE6** | Format used by Windows CE6 |

### <format> Options for PowerPC:

| <format> | Description |
|---|---|
| **STD** | Standard format defined by the CPU |

### <format> Options for RISC-V:

| <format> | Description |
|---|---|
| **STD** | Automatic detection of the page table format from the SATP register. |
| **SV32** | 32-bit page table format (for SV32 targets only) |
| **SV32X4** | Stage 2 (G-stage) 32-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |
| **SV39** | 39-bit page table format (for SV64 targets only) |
| **SV39X4** | Stage 2 (G-stage) 39-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |
| **SV48** | 48-bit page table format (for SV64 targets only) |
| **SV48X4** | Stage 2 (G-stage) 48-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |
| **SV57** | 57-bit page table format (for SV64 targets only) |
| **SV57X4** | Stage 2 (G-stage) 57-bit page table format for page tables translating intermediate physical addresses. Not applicable to other page tables. |

### <format> Options for SH4:

| <format> | Description |
|---|---|
| **WINCE6** | Format used by Windows CE6 |

### <format> Options for x86:

| <format> | Description |
|---|---|
| **EPT** | Extended page table format (type autodetected) |
| **EPT4L** | Extended page table format (4-level page table) |
| **EPT5L** | Extended page table format (5-level page table) |
| **P32** | 32-bit format with 2 page table levels |
| **PAE** | Format with 3 page table levels |
| **PAE64** | 64-bit format with 4 page table levels |
| **PAE64L5** | 64-bit format with 5 page table levels |
| **STD** | Automatic detection of the page table format used by the CPU |

### <base_address>

*<base_address>* specifies the base address of the kernel translation table.
This is currently unused. Specify “0”.

### <logical_kernel_address_range>

*<logical_kernel_address_range>* specifies the virtual to physical address translation of the kernel address range. See your config.bib file and the example scripts for the kernel address range.

### <physical_kernel_address>

*<physical_kernel_address>* specifies the physical start address of the kernel.

When declaring the MMU layout, you should also create the kernel translation manually with **TRANSlation.Create**.

The kernel code, which resides in the kernel space, can be accessed by any process, regardless of the current space ID. Additionally the common DLLs and the shared heap can be accessed by any process with the same address translation. Use the command **TRANSlation.COMMON** to define the complete address ranges that are addressed by the kernel, common DLLs and shared heap.

And don’t forget to switch on the debugger’s MMU translation with **TRANSlation.ON**

Example:

```
; kernel virtual address range: 0x88000000--0x8fffffff,
; physical RAM starts at 0x20000000
MMU.FORMAT WINCE6 0 0x88000000--0x8fffffff 0x20000000
TRANSlation.Create 0x88000000--0x8fffffff 0x20000000
TRANSlation.COMMON 0x40000000--0x5fffffff 0x70000000--0xffffffff
TRANSlation.ON
```

Please see also the sample scripts in the ~~/demo directory.

To access the different process spaces correctly, the debugger needs to know the address translation of every virtual address it uses. You can either scan the MMU tables and place a copy of them into the debugger's MMU table, or you can use a table walk, where the debugger walks through the tables each time it accesses a virtual address.

The command **MMU.SCAN** *only* scans the contents of the current processor MMU settings. Use the command **MMU.SCAN ALL** to go through all space IDs and scan their MMU settings. Note that on some systems this may take a long time. In this case you may scan single processes

To scan the address translation of a specific process, use the command **MMU.SCAN TaskPageTable** *<process>*. This command scans the space ID of the specified process.

**TRANSlation.List** shows the debugger's address translation table for all scanned space IDs.

If you set **TRANSlation.TableWalk ON**, the debugger tries first to look up the address translation in its own table (**TRANSlation.List**). If this fails, it walks through the target MMU tables to find the translation for a specific address. This feature eliminates the need of scanning the MMU each time it changed, but walking through the tables for each address may result in a very slow reading of the target. The address translations found with the table walk are only temporarily valid (i.e. not stored in **TRANSlation.List**), and are invalidated at each **Go** or **Step**.

See also chapter **“Debugging Windows CE Kernel”**.

# Symbol Autoloader

The OS Awareness for Windows CE 6/7 contains an autoloader, which automatically loads symbol files. The autoloader maintains a list of address ranges, corresponding Windows CE components and the appropriate load command. Whenever the user accesses an address within an address range specified in the autoloader, the debugger invokes the appropriate command. The command is usually a call to a PRACTICE script that loads the symbol file to the appropriate addresses.

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

The autoloader can be configured to react only on processes, ROM modules, (all) libraries, or libraries of the current process (see also **TASK.sYmbol.Option AutoLoad**). It is recommended to set only those components you are interested in, because this significantly reduces the time of the autoloader checks.

The autoloader reads the target's tables for the chosen components and fills the autoloader list with the components found on the target. All necessary information, such as load addresses and space IDs, are retrieved from kernel-internal information.

**sYmbol.AutoLOAD.CHECKWINCE "***<action>***"**

*<action>* Action to take for symbol load, e.g. `"DO autoload"`

If an address is accessed that is covered by the autoloader list, the autoloader calls *<action>* and appends the load addresses and the space ID of the component to the action. Usually, *<action>* is a call to a PRACTICE script that handles the parameters and loads the symbols. Please see the example script "autoload.cmm" in the ~~/demo directory.

The point in time when the component information is retrieved from the target can be set:

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

(no argument) A single **sYmbol.AutoLOAD.CHECK** command refreshes the information about the target.

**ON** The debugger automatically reads the information on every go/halt or step cycle. This significantly slows down the debugger's speed.

**OFF** no automatic update of the autoloader table will be done, you have to manually trigger the information read when necessary. To accomplish that, execute the **sYmbol.AutoLOAD.CHECK** command without arguments.

**NOTE:** The autoloader covers only components that are already started. Components that are not in the current process, module or library table are not covered.

# 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 state line 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.

*This chapter only applies to Windows CE 7.*

# 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.MMUSPACES**, too.

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

| 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”** (glossary.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)”.

On ARM architectures, Windows CE serves the ContextID register with the address space ID (ASID) of the process. This allows tracking the program flow of the processes and evaluation of the process switches. But it does not provide performance information of threads.

To allow a detailed performance analysis on Windows CE threads, the context ID must contain the thread ID. Set the lower 8 bit of the context ID register with the process’ ASID, and set the upper 24 bit with the ID of the thread, i.e. “`(thread.dwId << 8) | process.bASID`”.

The Windows CE awareness needs to be informed about the changed format of the context ID:

**TASK.Option THRCTX ON**

To implement the above context ID setting, either patch the kernel or implement OEMReschedule:

**1) Patching the kernel**

***For Windows CE 6:***

Patch the file WINCE600\private\winceos\coreos\nk\kernel\arm\vmarm.c:

Edit the two(!) ARMSetASID() calls in MDSwitchVM() and MDSetCPUASID() and change it into:

```
ARMSetASID ((RunList.pth->dwId<<8) | pprc->bASID);
```

***For Windows Embedded Compact 7:***

Follow the instructions shown in `~~/demo/arm/kernel/wince/ce7/context-id-patch.txt`

**2) implement OEMReschedule():**

In the OAL layer, implement the OAL function OEMReschedule() that writes

```
    (dwThrdId<<8 | process)
```

to the ContextID register.):

All kernel activities up to the thread switch are added to the calling thread.

## 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”** (glossary.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 CE specific Menu

The menu file “wince6.men”/”wince7.men” contains a menu with Windows CE specific menu items. Load this menu with the **MENU.ReProgram** command.

You will find a new menu called **Windows CE**.

- 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 rsp. 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 Process on main...** 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 in Windows CE. The debugger will load the symbols and halt at main(). See also the demo script “app_debug.cmm”.

  - **Watch Processes** opens a process watch window or adds or removes processes from the process watch window. Specify a process name. See **TASK.Watch** for details.

  - **Scan Process MMU Pages...** scans the MMU pages of the specified process.
    **Scan All MMU Tables** performs a scan over all target side kernel and process MMU pages.
    See also chapter “**Scanning System and Processes**”.

- **DLL Debugging** refers to actions related to library based debugging.
  See also chapter “**Debugging DLLs**”.

  - Use **Load Symbols...** and **Delete Symbols...** to load rsp. 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**.

  - **Watch DLLs** opens a DLL watch window or adds or removes DLLs from the DLL watch window. Specify a DLL name. See **TASK.WatchDLL** for details.

  - **Scan Process MMU Pages...** scans the MMU pages of the specified process. Specify the name of the process that uses the library you want to debug.
    **Scan All MMU Tables** performs a scan over all target side kernel and process MMU pages.
    See also chapter “**Scanning System and Processes**”.

- 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 CE components should be managed by the autoloader. See also **TASK.sYmbol.Option AutoLOAD**.

- The **Stack Coverage** submenu starts and resets the Windows CE specific stack coverage and provides an easy way to add or remove threads from the stack coverage window. See also chapter “**Thread Stack Coverage**”.

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 Eboot

Eboot is the bootloader of Windows CE. It runs on virtual addresses, but with a single address space. Keep **SYStem.Option.MMUSPACES OFF** for debugging Eboot.

If you want to download Eboot with the debugger, load the plain binary "eboot.nb0" to the physical start address of the Eboot image. The virtual start address is defined in eboot.bib as "RAMIMAGE". Usually the physical start address is the start address of RAM plus the offset within the RAMIMAGE address.

Example:

```
; eboob.bib says e.g.:
;     NK    80040000    00080000    RAMIMAGE
; RAM is physically at e.g. 0x20000000
; then load image to physical address 0x20040000

Data.LOAD.Binary EBOOT.nb0 0x20040000
Register.Set PC 0x20040000
```

The Windows CE awareness contains a special extension that reads the header of the Eboot image. Load this extension and specify the physical address of the Eboot image.

Example:

```
; load eboot extension for section detection
; specify physical address of eboot image
EXTension.CONFIG eboot.t32 0x20040000
```

Now you have access to the header information and you can load the symbols of Eboot. The symbol file is called "eboot.exe/pdb". Use the following sequence to load the symbols:

```
; load eboot symbols
&text=eboot.addr(0)      ; code segment
&data=eboot.addr(1)      ; data segment
Data.LOAD.EXE eboot.exe /NoCODE /NoClear \
    /RELOC .text AT &text /RELOC .data AT &data
```

If you start from reset (MMU switched off), use an on-chip breakpoint and let the MMU initialize:

```
Go main /Onchip       ; main() or BootloaderMain()
```

After this, you're ready to debug Eboot.

# Debugging Windows CE Kernel

Windows CE runs on virtual address spaces. The kernel uses a static address translation, 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 CE kernel and the 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.

See also chapter **“MMU Support”**.

## Downloading the Kernel

The Windows CE make process can generate different outputs (usually binary file called “nk.nb0”). For downloading the Windows CE kernel, you may choose whatever format you prefer.

If you start the Windows CE kernel from Flash, or if you download the kernel via Ethernet, do this as you are doing it without debugging.

If you want to download the Windows CE image using the debugger, you may use “nk.bin” (linked executable) or “nk.nb0” (absolute binary).

The nk.bin file is linked onto virtual addresses. If the virtual addresses of the kernel equal to the physical download address (e.g. on SH architectures), you can download it without further parameters. If the virtual addresses differ from the physical addresses, you have to specify the offset (calculate it by physical-virtual).

Examples:

```
; load image to the linked addresses:
Data.LOAD.EXE nk.bin

; load image (linked to virtual 0x80000000) to physical 0x20000000
Data.LOAD.EXE nk.bin 0x20000000-0x80000000
; set PC on physical start address
```

If you use the plain binary image (nk.bin), you have to specify, to which address to download it. The Windows CE kernel image is usually located at the physical start address of the RAM (note that an eventual boot loader in RAM may be overwritten).

Examples:

```
; load the image to physical start address of RAM
Data.LOAD.Binary nk.nb0 0x20000000
```

When downloading the kernel via the debugger, remember to set startup options that the kernel may require, before booting the kernel. Check the config.bib file for the start address of “NK” and set the PC on the *physical* representation of this address:

```
; set PC to physical start address
Register.Set pc 0x20000000
```

Please see the example scripts in the ~~/demo directory.

## Debugging the Kernel Startup

Use this chapter only, if you really want to debug the very early stages of the Windows CE startup.

The kernel image starts with MMU switched off, i.e. the processor operates on physical addresses. However, all symbols of the `nk.exe` file are bound to virtual addresses. If you want to debug this (tiny) startup sequence, you have to load and relocate the symbols.

- Downloading the kernel via debugger:

  Download the kernel image as described above. The PC should be at the image entry point.

- Downloading the kernel via Ethernet:

  *Before* booting Windows CE, set an on-chip breakpoint onto the physical image start address. E.g.:

  ```
  Break.Set 0x20000000 /Onchip
  ```

  Now let the boot monitor download and start the Windows CE image. It will halt on the start address, ready to debug. Delete the breakpoint when hit.

Load the Windows CE Awareness. The MMU is not yet initialized, so we have to inform the awareness of the physical address of the image header information:

```
; load wince awareness for section detection
TASK.CONFIG wince6          ; or wince7 for CE7
TASK.ROM.PA 0x20000000      ; specify physical address of image
```

Get the section addresses from the header information and calculate the physical addresses out of these:

```
; load nk.exe symbols to physical addresses
&text=task.rom.addr("nk.exe",0)-0x80000000+0x20000000
&data=task.rom.addr("nk.exe",1)-0x80000000+0x20000000
Data.LOAD.EXE nk.exe /NoCODE \
    /RELOC .text AT &text /RELOC .data AT &data
```

You are now ready to debug the startup of the kernel up to the point, where the MMU is switched on.

As soon as the processor MMU is switched on, you have to reload the symbol to its virtual addresses. The global kernel data pointer is not yet set up, so inform the awareness of the now virtual address of the header information:

```
; explicitely specify virtual address of image
TASK.ROM.VA 0x80000000
```

And reload the symbols of nk.exe:

```
; load nk.exe symbols to physical addresses
&text=task.rom.addr("nk.exe",0)
&data=task.rom.addr("nk.exe",1)
Data.LOAD.EXE nk.exe /NoCODE \
    /RELOC .text AT &text /RELOC .data AT &data
```

Now you're ready to debug the rest of the kernel startup, with MMU switched on.

## Debugging the Kernel

For debugging the kernel itself, and for using the Windows CE awareness, you have to load the virtual addressed symbols of the kernel into the debugger. The files “nk.exe” and “nk.pdb”, which reside in your build directory, contain all kernel symbols with virtual addresses.

Load the kernel symbols with the command:

```
TASK.sYmbol.LOAD "nk.exe"
```

# Debugging User Processes and DLLs

Windows CE runs on virtual address spaces. 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 CE kernel and the 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. Using this space ID, it is possible to address a unique memory location, even if several processes use the same virtual address.

See also chapter **“MMU Support”**.

Load all process symbols with the option `/NoClear`, to preserve previously loaded symbols for other components.

## 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 and to scan the process’ MMU settings.

**Manually Load Process Symbols:**

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

```
Data.LOAD.EXE hello.exe 12.:0 /NoCODE /NoClear
```

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

If **TRANSlation.TableWalk** is OFF, you need to scan the MMU translation of this process:

```
MMU.TaskPageTable.SCAN 12.:0          ; scan MMU of process ID 12.
```

See also chapter “**Scanning System and Processes**”.

### 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 and scan MMU
```

This command loads the symbols of “hello.exe” and scans the MMU of the process “hello”. See **TASK.sYmbol.LOAD** for more information.

### Using the Symbol Autoloader:

If the symbol autoloader is configured (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"
```

### Debugging a Process From Scratch, Using a Script:

If you want to debug your process right from the beginning (at “main()” or “WinMain()”), 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. Set a breakpoint into the process start handler of Windows CE, when the process is already loaded but not yet started. The function `CELOG_ProcessCreateEx()` may serve as a good point. When the breakpoint is hit, check if the process is already loaded. If so, extract the process ID, scan the process’ MMU and load the symbols. Windows CE loads the code first, if it is accessed by the CPU. So you’re not able to set a software breakpoint yet into the process, because it will be overwritten by the swapper, when it loads actually the code. Instead, set an on-chip breakpoint to the main() routine of the process. As soon as the process is started, the code will be loaded and the breakpoint will be hit. Now you’re able to set software breakpoints. See the script “app_debug.cmm” in the ~~/demo directory, how to do this.

The “Windows CE” menu contains this procedure in a menu item: **Windows CE** -> **Process Debugging** -> **Debug Process on main**... . See also chapter “**Windows CE Specific Menu**”.

When finished debugging with a process, or if restarting the process, you have to delete the symbols and restart the application debugging. Delete the symbols with:

```
sYmbol.Delete \\hello
```

If the autoloader is configured:

```
sYmbol.AutoLOAD.CLEAR "hello"
```

**Debugging a Process From Scratch, with Automatic Detection:**

The **TASK.Watch** command group implements the above script as an automatic handler and keeps track of a process launch and the availability of the process symbols. See **TASK.Watch.View** for details.

## Debugging DLLs

If the process uses DLLs, Windows CE loads them dynamically to 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.

**Manually Load Library Symbols:**

Start your process and open a **TASK.Process** window. Double click on the “magic” of the process using the library, and expand the “modules” tree (if available). A list will appear that shows the loaded libraries and the corresponding load addresses. 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 0xff8000, use the command:

```
Data.LOAD.EXE mylib.dll 12.:0xff8000 /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.LOADDLL "coredll"
```

tries to automatically load and relocate the appropriate DLL. It additionally sets the debugger MMU to allow access to the DLL, even if it is not paged in. See **TASK.sYmbol.LOADDLL** for details.

**Using the Symbol Autoloader:**

If the symbol autoloader is configured (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"
```

**Debugging a DLL From Scratch, with Automatic Detection:**

The **TASK.WatchDLL** command group implements an automatic handler and keeps track of a DLL launch and the availability of the DLL symbols. See **TASK.WatchDLL.View** for details.

## Trapping Unhandled Exceptions

An “unhandled exception” happens, if the code tries to access a memory location that cannot be mapped in an appropriate way. E.g. if a process tries to write to a read-only area, or if the kernel tries to read from a non-existent address. An unhandled exception is detected inside the kernel, if the mapping of page fails. If so, the kernel (usually) prints out an error message with “PrintException()”.

To trap unhandled exceptions, set a breakpoint onto the label “PrintException”. When halted there, execute one single HLL step to set up local variables, then use **Var.Local** to display the local variables of “PrintException()”. This function is called with three parameters:

- “dwExcpId” contains the exception ID that happened;
- “pExr” points to some more information about the exception;
- “pCtx” points to a structure containing the complete register set at the location, where the fault occurred.

When halted at “PrintException”, you may load the temporary register set of TRACE32 with these values:

```
; adapt this script to your processor registers

 ; read all register values
 &r0=v.value(pCtx.R0)
 &r1=v.value(pCtx.R1)
 ; continue for all registers
 &sr=v.value(pCtx.Cpsr)
 &pc=v.value(pCtx.Pc)

 ; write all register values into temporary register set
 Register.Set R0 &r0 /Task Temporary
 Register.Set R1 &r1 /Task Temporary
 ; continue for all registers
 Register.Set SR &sr /Task Temporary
 Register.Set PC &pc /Task Temporary
```

Use **Data.List**, **Var.Local** etc. then to analyze the fault.

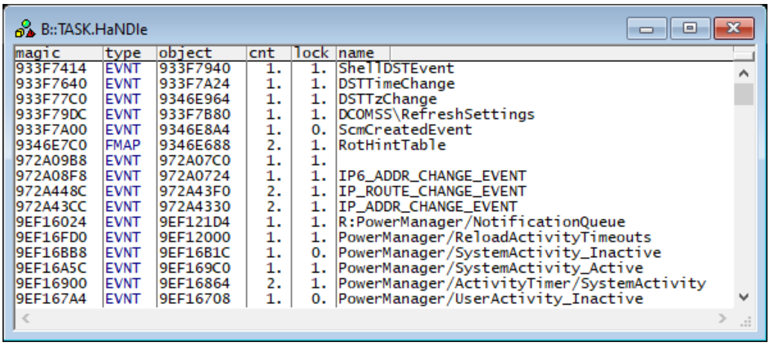
For some architectures, an example script called “exception.cmm” is prepared for this. Check the appropriate demo directory.

As soon as debugging is continued (e.g. “Step”, “Go”, ...), the original register settings at “PrintException” are restored.

## TASK.HaNDle Display global handles

Format: **TASK.HaNDle**

Displays the global handle table of Windows CE.



B::TASK.HaNDle

```
magic     type  object    cnt  lock name
933F7414  EVNT  933F7940   1.   1.  ShellDSTEvent
933F7640  EVNT  933F7A24   1.   1.  DSTTimeChange
933F77C0  EVNT  9346E964   1.   1.  DSTTzChange
933F79DC  EVNT  933F7B80   1.   1.  DCOMSS\RefreshSettings
933F7A00  EVNT  9346E8A4   1.   0.  ScmCreatedEvent
9346E7C0  FMAP  9346E688   2.   1.  RotHintTable
972A09B8  EVNT  972A07C0   1.   1.
972A08F8  EVNT  972A0724   1.   1.  IP6_ADDR_CHANGE_EVENT
972A448C  EVNT  972A43F0   2.   1.  IP_ROUTE_CHANGE_EVENT
972A43CC  EVNT  972A4330   2.   1.  IP_ADDR_CHANGE_EVENT
9EF16024  EVNT  9EF121D4   1.   1.  R:PowerManager/NotificationQueue
9EF16FD0  EVNT  9EF12000   1.   1.  PowerManager/ReloadActivityTimeouts
9EF16BB8  EVNT  9EF16B1C   1.   0.  PowerManager/SystemActivity_Inactive
9EF16A5C  EVNT  9EF169C0   1.   1.  PowerManager/SystemActivity_Active
9EF16900  EVNT  9EF16864   2.   1.  PowerManager/ActivityTimer/SystemActivity
9EF167A4  EVNT  9EF16708   1.   0.  PowerManager/UserActivity_Inactive
```

“magic” is a unique ID, used by the OS Awareness to identify a specific thread (address of the thread structure).

The fields “magic” and the address fields are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

## TASK.MMU.SCAN Scan process MMU space

Format: **TASK.MMU.SCAN** [*<process>*]

**NOTE:** If not explicitely advised, please use **MMU.TaskPageTable.SCAN** instead. See **MMU Support**.

Scans the target MMU of the space ID, specified by the given process, and sets the Debugger MMU appropriately, to cover the physical to logical address translation of this specific process.

The command walks through all page tables which are defined for the memory spaces of the process and prepares the Debugger MMU to hold the physical to logical address translation of this process. This is needed to provide full HLL support. If a process was loaded dynamically, you must set the Debugger MMU to this process, otherwise the Debugger won't know where the physical image of the process is placed.

To successfully execute this command, space IDs must be enabled (**SYStem.Option.MMUSPACES** **ON**).

| | |
|---|---|
| *<process>* | Specify a process magic, ID or name.<br>If no argument is specified, the command scans all current processes. |

**Example**:

```
; scan the memory space of the process "hello"
TASK.MMU.SCAN "hello"
```

See also **MMU Support**.

## TASK.Option    Set awareness options

| | |
|---|---|
| Format: | **TASK.Option** *<option>* |
| *<option>*: | **THRCTX** [**ON** \| **OFF**]<br>**ThrSort** [**NONE** \| **Handle**] |

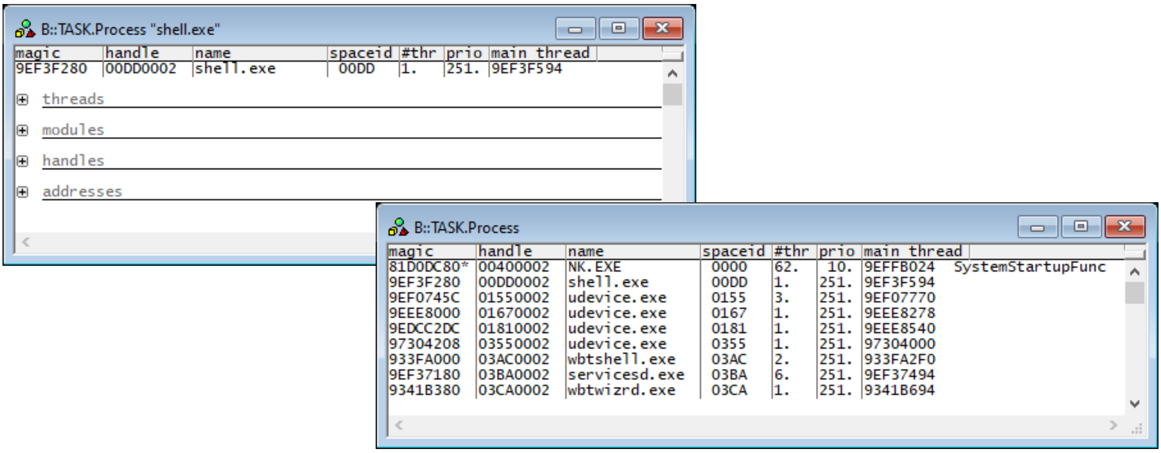
Set various options to the awareness.

| | |
|---|---|
| **THRCTX** | Set the context ID type that is recorded with the real-time trace (e.g. ETM).<br>If set to on, the context ID in the trace contains thread switch detection.<br>See **Task Runtime Statistics**. |
| **ThrSort** | Sort the TASK.Thread window according to the given item.<br>Note: sorting slows down the display of the window. |

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

Displays the process table of Windows CE or detailed information about one specific process.

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



```
magic     handle    name           spaceid #thr prio main thread
9EF3F280  00DD0002  shell.exe       00DD   1.   251. 9EF3F594

⊞ threads
⊞ modules
⊞ handles
⊞ addresses
```

```
magic     handle    name           spaceid #thr prio main thread
81D0DC80* 00400002  NK.EXE          0000   62.   10. 9EFFB024  SystemStartupFunc
9EF3F280  00DD0002  shell.exe       00DD   1.   251. 9EF3F594
9EF0745C  01550002  udevice.exe     0155   3.   251. 9EF07770
9EEE8000  01670002  udevice.exe     0167   1.   251. 9EEE8278
9EDCC2DC  01810002  udevice.exe     0181   1.   251. 9EEE8540
97304208  03550002  udevice.exe     0355   1.   251. 97304000
933FA000  03AC0002  wbtshell.exe    03AC   2.   251. 933FA2F0
9EF37180  03BA0002  servicesd.exe   03BA   6.   251. 9EF37494
9341B380  03CA0002  wbtwizrd.exe    03CA   1.   251. 9341B694
```

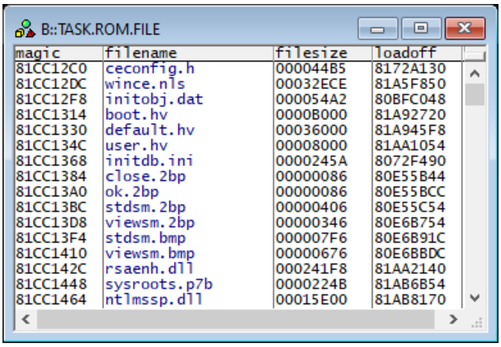
“magic” is a unique ID, used by the OS Awareness to identify a specific process (address of the process structure).

The fields “magic”, “main thread”, “thread magic” and “entry function” are mouse sensitive, double clicking on them opens appropriate windows. Right clicking on them will show a local menu.

## TASK.ROM.FILE Display built-in files

Format: **TASK.ROM.FILE**

Displays a table with all files that are built-in into the Windows CE image.



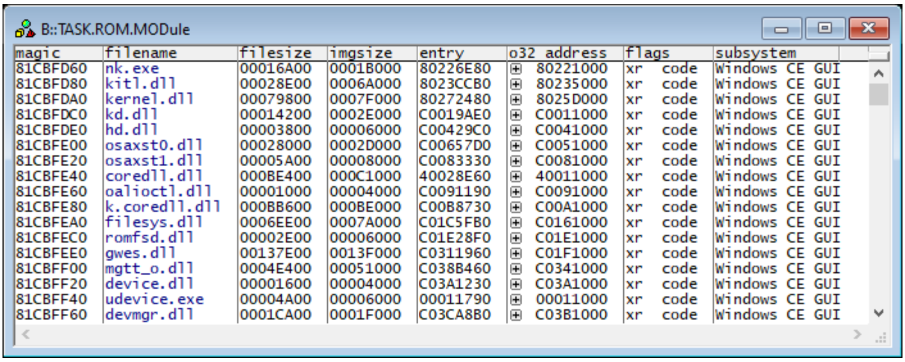
B::TASK.ROM.FILE

| magic | filename | filesize | loadoff |
|---|---|---|---|
| 81CC12C0 | ceconfig.h | 000044B5 | 8172A130 |
| 81CC12DC | wince.nls | 00032ECE | 81A5F850 |
| 81CC12F8 | initobj.dat | 000054A2 | 80BFC048 |
| 81CC1314 | boot.hv | 0000B000 | 81A92720 |
| 81CC1330 | default.hv | 00036000 | 81A945F8 |
| 81CC134C | user.hv | 00008000 | 81AA1054 |
| 81CC1368 | initdb.ini | 0000245A | 8072F490 |
| 81CC1384 | close.2bp | 00000086 | 80E55B44 |
| 81CC13A0 | ok.2bp | 00000086 | 80E55BCC |
| 81CC13BC | stdsm.2bp | 00000406 | 80E55C54 |
| 81CC13D8 | viewsm.2bp | 00000346 | 80E6B754 |
| 81CC13F4 | stdsm.bmp | 000007F6 | 80E6B91C |
| 81CC1410 | viewsm.bmp | 00000676 | 80E6BBDC |
| 81CC142C | rsaenh.dll | 000241F8 | 81AA2140 |
| 81CC1448 | sysroots.p7b | 0000224B | 81AB6B54 |
| 81CC1464 | ntlmssp.dll | 00015E00 | 81AB8170 |

## TASK.ROM.MODule Display built-in modules

Format: **TASK.ROM.MODule**

Displays a table with all modules that are built-in into the Windows CE image.



B::TASK.ROM.MODule

| magic | filename | filesize | imgsize | entry | o32 address | flags | subsystem |
|---|---|---|---|---|---|---|---|
| 81CBFD60 | nk.exe | 00016A00 | 0001B000 | 80226E80 | ⊞ 80221000 | xr code | Windows CE GUI |
| 81CBFD80 | kitl.dll | 00028E00 | 0006A000 | 8023CCB0 | ⊞ 80235000 | xr code | Windows CE GUI |
| 81CBFDA0 | kernel.dll | 00079800 | 0007F000 | 80272480 | ⊞ 8025D000 | xr code | Windows CE GUI |
| 81CBFDC0 | kd.dll | 00014200 | 0002E000 | C0019AE0 | ⊞ C0011000 | xr code | Windows CE GUI |
| 81CBFDE0 | hd.dll | 00003800 | 00006000 | C00429C0 | ⊞ C0041000 | xr code | Windows CE GUI |
| 81CBFE00 | osaxst0.dll | 00028000 | 0002D000 | C00657D0 | ⊞ C0051000 | xr code | Windows CE GUI |
| 81CBFE20 | osaxst1.dll | 00005A00 | 00008000 | C0083330 | ⊞ C0081000 | xr code | Windows CE GUI |
| 81CBFE40 | coredll.dll | 000BE400 | 000C1000 | 40028E60 | ⊞ 40011000 | xr code | Windows CE GUI |
| 81CBFE60 | oalioctl.dll | 00001000 | 00004000 | C0091190 | ⊞ C0091000 | xr code | Windows CE GUI |
| 81CBFE80 | k.coredll.dll | 000BB600 | 000BE000 | C00B8730 | ⊞ C00A1000 | xr code | Windows CE GUI |
| 81CBFEA0 | filesys.dll | 0006EE00 | 0007A000 | C01C5FB0 | ⊞ C0161000 | xr code | Windows CE GUI |
| 81CBFEC0 | romfsd.dll | 00002E00 | 00006000 | C01E28F0 | ⊞ C01E1000 | xr code | Windows CE GUI |
| 81CBFEE0 | gwes.dll | 00137E00 | 0013F000 | C0311960 | ⊞ C01F1000 | xr code | Windows CE GUI |
| 81CBFF00 | mgtt_o.dll | 0004E400 | 00051000 | C038B460 | ⊞ C0341000 | xr code | Windows CE GUI |
| 81CBFF20 | device.dll | 00001600 | 00004000 | C03A1230 | ⊞ C03A1000 | xr code | Windows CE GUI |
| 81CBFF40 | udevice.exe | 00004A00 | 00006000 | 00011790 | ⊞ 00011000 | xr code | Windows CE GUI |
| 81CBFF60 | devmgr.dll | 0001CA00 | 0001F000 | C03CA8B0 | ⊞ C03B1000 | xr code | Windows CE GUI |

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

| | |
|---|---|
| **TASK.sYmbol.LOAD** | Load process symbols and MMU |
| **TASK.sYmbol.DELete** | Unload process symbols and MMU |
| **TASK.sYmbol.LOADDLL** | Load DLL symbols and MMU |
| **TASK.sYmbol.DELeteDLL** | Unload DLL symbols and MMU |
| **TASK.sYmbol.LOADRM** | Load ROM module symbols |
| **TASK.sYmbol.DELeteRM** | Unload ROM module symbols |
| **TASK.sYmbol.Option** | Set symbol management options |

## TASK.sYmbol.DELete Unload process symbols and MMU

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

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

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

## TASK.sYmbol.DELeteDLL Unload DLL symbols and MMU

Format: **TASK.sYmbol.DELeteDLL** *<dll>*

When debugging of a DLL is finished, you should remove loaded DLL symbols and MMU entries.
This command deletes the symbols of the specified DLL and deletes its MMU entries.

| | |
|---|---|
| *<dll>* | Specify the DLL name or path (in quotes) or magic to unload the symbols of this DLL. |

## TASK.sYmbol.DELeteRM Unload ROM module symbols

Format: **TASK.sYmbol.DELeteRM** *<rom_module>*

This command deletes the symbols of the specified ROM module.

| | |
|---|---|
| *<process>* | Specify the ROM module name or path (in quotes) or magic to unload the symbols of this ROM module. |

## TASK.sYmbol.LOAD Load process symbols and MMU

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

In order to debug a user process, the debugger needs the symbols of this process, and the process specific MMU settings (see chapter “Debugging User Processes”).
This command retrieves the appropriate space ID and triggers the symbol autoloader to load the symbols of this process. Note that this command works only with processes that are already loaded in Windows CE (i.e. processes that show up in the **TASK.Process** window).

| | |
|---|---|
| *<process>* | Specify the process name or path (in quotes) or magic to load the symbols of this process. |

## TASK.sYmbol.LOADDLL Load DLL symbols and MMU

Format: **TASK.sYmbol.LOADDLL** *<dll>*

In order to debug a DLL, the debugger needs the symbols of this DLL, and the DLL specific MMU settings (see chapter **“Debugging DLLs”**, page 30).
This command retrieves the appropriate load addresses and triggers the symbol autoloader to load the symbols of this process. Note that this command works only with DLLs that are already loaded in Windows CE (i.e. DLLs that show up in the detailed process window).

| | |
|---|---|
| *<dll>* | Specify the DLL name or path (in quotes) or magic to load the symbols of this DLL. |

## TASK.sYmbol.LOADRM Load ROM module symbols

| | |
|---|---|
| Format: | **TASK.sYmbol.LOADRM** *<rom_module>* |

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

For modules (EXEs and DLLs) that are not process dependent (e.g. kernel and kernel DLLs), you can use the address information of the ROM module list (see **TASK.ROM.MODule**) to load the symbols. This command retrieves the appropriate load addresses and triggers the symbol autoloader to load the symbols of this rom module. Note that this command will not work for processes or process bound DLLs.

## TASK.sYmbol.Option Set symbol management options

| | |
|---|---|
| Format: | **TASK.sYmbol.Option** *<option>* |
| *<option>*: | **AutoLoad** *<option>* |

Set various options 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 |
| **RomMod** | Check ROM modules |
| **CurrLib** | Check only libraries of current process |
| **ALL** | Check processes, libraries and ROM modules |
| **NoProcess** | Don’t check processes |
| **NoLibrary** | Don’t check libraries |
| **NoRomMod** | Don’t check modules |
| **NONE** | Check nothing. |

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

**Example**:

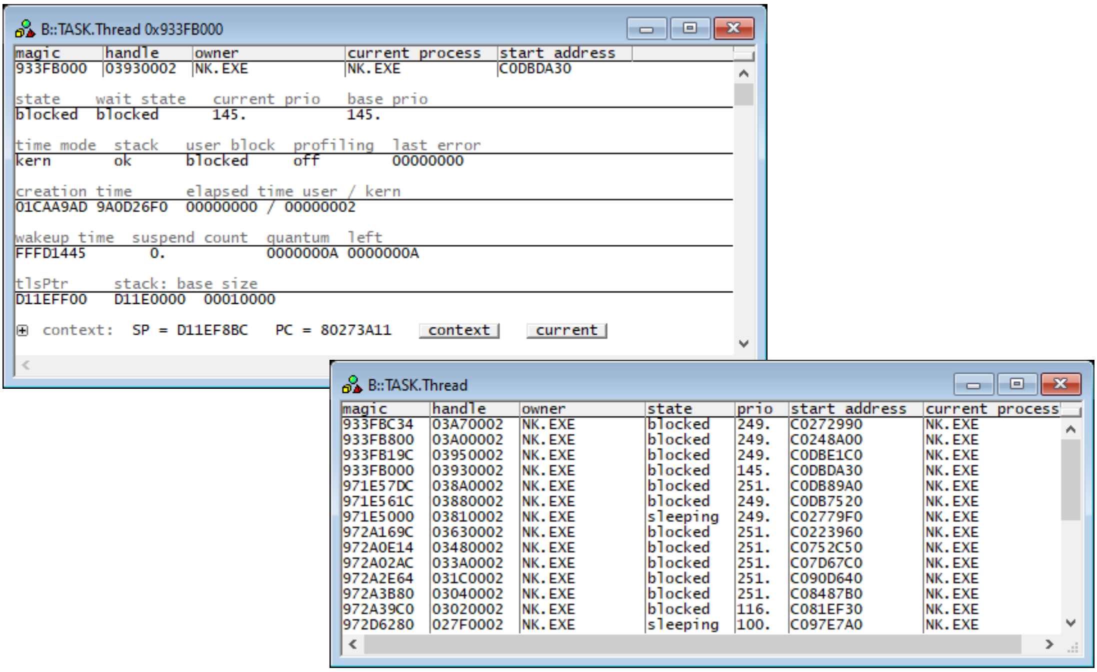
```
; check processes and ROM modules
TASK.sYmbol.Option AutoLoad Process
TASK.sYmbol.Option AutoLoad RomMod
```

## TASK.Thread                                                    Display threads

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

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

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



```
B::TASK.Thread 0x933FB000
magic     handle    owner                current process  start address
933FB000  03930002  NK.EXE               NK.EXE           C0DBDA30

state     wait state   current prio   base prio
blocked   blocked        145.         145.

time mode  stack    user block  profiling  last error
kern       ok       blocked     off        00000000

creation time      elapsed time user / kern
01CAA9AD 9A0D26F0  00000000 / 00000002

wakeup time  suspend count  quantum   left
FFFD1445         0.         0000000A  0000000A

tlsPtr     stack: base size
D11EFF00   D11E0000  00010000

⊞ context:  SP = D11EF8BC   PC = 80273A11   context   current
```

```
B::TASK.Thread
magic     handle    owner              state     prio  start address    current process
933FBC34  03A70002  NK.EXE             blocked   249.  C0272990         NK.EXE
933FB800  03A00002  NK.EXE             blocked   249.  C0248A00         NK.EXE
933FB19C  03950002  NK.EXE             blocked   249.  C0DBE1C0         NK.EXE
933FB000  03930002  NK.EXE             blocked   145.  C0DBDA30         NK.EXE
971E57DC  038A0002  NK.EXE             blocked   251.  C0DB89A0         NK.EXE
971E561C  03880002  NK.EXE             blocked   249.  C0DB7520         NK.EXE
971E5000  03810002  NK.EXE             sleeping  249.  C02779F0         NK.EXE
972A169C  03630002  NK.EXE             blocked   251.  C0223960         NK.EXE
972A0E14  03480002  NK.EXE             blocked   251.  C0752C50         NK.EXE
972A02AC  033A0002  NK.EXE             blocked   251.  C07D67C0         NK.EXE
972A2E64  031C0002  NK.EXE             blocked   251.  C090D640         NK.EXE
972A3B80  03040002  NK.EXE             blocked   251.  C08487B0         NK.EXE
972A39C0  03020002  NK.EXE             blocked   116.  C081EF30         NK.EXE
972D6280  027F0002  NK.EXE             sleeping  100.  C097E7A0         NK.EXE
```

“magic” is a unique ID, used by the OS Awareness to identify a specific thread (address of the thread structure).

The fields “magic” and the address fields are mouse sensitive. Double-clicking on them opens appropriate windows. Right clicking on them will show a local menu.

Pressing the “context” button (if available) changes the register context to this task. “current” resets it to the current context. See “**Task Context Display**”.

## TASK.Watch — Watch processes

The TASK.Watch command group build a watch system that watches your Windows CE target for specified processes. It loads and unloads process symbols automatically. Additionally it covers process creation and may stop watched processes at their entry points.

In particular the watch commands are:

| | |
|---|---|
| **TASK.Watch.View** | Activate watch system and show watched processes |
| **TASK.Watch.ADD** | Add process to watch list |
| **TASK.Watch.DELete** | Remove process from watch list |
| **TASK.Watch.DISable** | Disable watch system |
| **TASK.Watch.ENable** | Enable watch system |
| **TASK.Watch.DISableBP** | Disable process creation breakpoints |
| **TASK.Watch.ENableBP** | Enable process creation breakpoints |
| **TASK.Watch.Option** | Set watch system options |

## TASK.Watch.ADD — Add process to watch list

Format: **TASK.Watch.ADD** *<process>*

Adds a process to the watch list.

*<process>* Specify the process name (in quotes) or magic.

Please see **TASK.Watch.View** for details.

## TASK.Watch.DELete — Remove process from watch list

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

Removes a process from the watch list.

*<process>* Specify the process name (in quotes) or magic.

Please see **TASK.Watch.View** for details.

## TASK.Watch.DISable — Disable watch system

Format: **TASK.Watch.DISable**

Disables the complete watch system. The watched processes list is no longer checked against the target and is not updated. You'll see the **TASK.Watch.View** window grayed out.

This feature is useful if you want to keep process symbols in the debugger, even if the process terminated.

## TASK.Watch.DISableBP — Disable process creation breakpoints

Format: **TASK.Watch.DISableBP**

Prevents the debugger from setting breakpoints for the detection of process creation. After executing this command, the target will run in real time. However, the watch system can no longer detect process creation. Automatic loading of process symbols will still work.

This feature is useful if you'd like to use limited breakpoints for other purposes.

Please see **TASK.Watch.View** for details.

## TASK.Watch.ENable — Enable watch system

Format: **TASK.Watch.ENable**

Enables the previously disabled watch system. It enables the automatic loading of process symbols as well as the detection of process creation.

Please see **TASK.Watch.View** for details.

## TASK.Watch.ENableBP — Enable process creation breakpoints

|  |  |
|---|---|
| Format: | **TASK.Watch.ENable** |

Enables the previously disabled breakpoints for detection of process creation.
Please see **TASK.Watch.View** for details.

## TASK.Watch.Option — Set watch system options

|  |  |
|---|---|
| Format: | **TASK.Watch.Option** *<option>* |
| *<option>*: | **BreakOptC** *<option>*<br>**BreakOptM** *<option>* |

Set various options to the watch system.

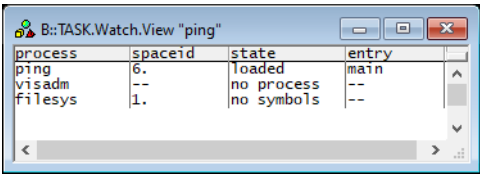
|  |  |
|---|---|
| **BreakOptC** | Set the option in double quotes, which is used to set the breakpoint on the process creation handler. The default option is “/Onchip”.<br>**Example**:<br>`TASK.Watch.Option BreakOptC "/SOFT"` |
| **BreakOptM** | Set the option in double quotes, which is used to set the breakpoint on the main entry point of the process. The default option is “/Onchip”.<br>**Example**:<br>`TASK.Watch.Option BreakOptC "/Hard"`<br>**NOTE**: The actual code of the process may not yet be loaded. Thus, setting Software breakpoints is not recommended. |

Please see **TASK.Watch.View** for details.

| | |
|---|---|
| Format: | **TASK.Watch.View** [*<process>*] |

Activates the watch system for processes and shows a table of the watched processes.

| | |
|---|---|
| **NOTE:** | **This feature may affect the real-time behavior of the target application!**<br>Please see below for details. |



```
process   spaceid  state       entry
ping      6.       loaded      main
visadm    --       no process  --
filesys   1.       no symbols  --
```

*<process>* Specify a process name for the initial process to be watched.

### Description of Columns in the TASK.Watch.View Window

| | |
|---|---|
| **process** | The name of the process to be watched. |
| **spaceid** | The current space ID (= process ID) of the watched process.<br>If grayed, the debugger is currently not able to determine the space ID of the process (e.g. the target is running). |
| **state** | The current watch state of the process.<br>If grayed, the debugger is currently not able to determine the watch state.<br>**no process**: The debugger couldn’t find the process in the current Windows CE process list.<br>**no symbols**: The debugger found the process and loaded the MMU settings of the process but couldn’t load the symbols of the process (most likely because the corresponding .exe and .pdb files were missing).<br>**loaded**: The debugger found the process and loaded the process’s MMU settings and symbols. |
| **entry** | The process entry point, which is either `main()` or `WinMain()`.<br>If grayed, the debugger is currently not able to detect the entry point or is unable to set the process entry breakpoint (e.g. because it is disabled with **TASK.Watch.DISableBP**). |

The watch system for processes is able to automatically load and unload the symbols of a process and its MMU settings, depending on their state in the target. Additionally, the watch system can detect the creation of a process and halts the process at its entry point.

| | |
|---|---|
| **TASK.Watch.ADD** | Adds processes to the watch list. |
| **TASK.Watch.DELete** | Removes processes from the watch list. |

The watch system for processes is active as long as the **TASK.Watch.View** window is open or iconized. As soon as this window is closed, the watch system will be deactivated.

### Automatic Loading and Unloading of Process Symbols

In order to detect the current processes, the debugger must have full access to the target, i.e. the target application must be stopped (with one exception, see below for creation of processes). As long as the target runs in real time, the watch system is not able to get the current process list, and the display will be grayed out (inactive).

If the target is halted (either by hitting a breakpoint, or by halting it manually), the watch system starts its work. For each of the processes in the watch list, it determines the state of this process in the target.

If a process is active on the target, which was previously not found there, the watch system scans its MMU entries and loads the appropriate symbol files. In fact, it executes **TASK.sYmbol.LOAD** for the new process.

If a watched process was previously loaded but is no longer found on the Windows CE process list, the watch system unloads the symbols and removes the MMU settings from the debugger MMU table. The watch system executes **TASK.sYmbol.DELete** for this process.

If the process was previously loaded and is now found with another space ID (e.g. if the process terminated and started again), the watch system first removes the process symbols and reloads them to the appropriate space ID.

You can disable the loading / unloading of process symbols with the command **TASK.Watch.DISable**.

### Detection of Process Creation

To halt a process at its main entry point, the watch system can detect the process creation and set the appropriate breakpoints.

To detect the process creation, the watch system sets a breakpoint on a kernel function that is called upon creation of processes. Every time the breakpoint is hit, the debugger checks if a watched process is started. If not, it simply resumes the target application. If the debugger detects the start of a newly created (and watched) process, it sets a breakpoint onto the main entry point of the process (either `main()` or `WinMain()`) and resumes the target application. A short while after this, the main breakpoint will hit and halt the target at the entry point of the process. The process is now ready to be debugged.

| | |
|---|---|
| **NOTE:** | By default, this feature uses one permanent on-chip breakpoint and one temporary on-chip breakpoint when a process is created. Please ensure that at least those two on-chip breakpoints are available when using this feature. Use **TASK.Watch.Option** to change the nature of the breakpoints.<br>Upon every process creation, the target application is halted for a short time and resumed after searching for the watched processes. **This impacts the real-time behavior of your target.** |

If you don't want the watch system to set breakpoints, you can disable them with the command **TASK.Watch.DISableBP**. Of course, detection of process creation won't work then.

The TASK.WatchDLL command group build a watch system that watches your Windows CE target for specified DLLs. It loads and unloads DLL symbols automatically. Additionally it covers DLL creation and may stop watched DLLs at their entry points.

In particular the watch commands are:

| | |
|---|---|
| **TASK.WatchDLL.View** | Activate watch system and show watched DLLs |
| **TASK.WatchDLL.ADD** | Add DLL to watch list |
| **TASK.WatchDLL.DELete** | Remove DLL from watch list |
| **TASK.WatchDLL.DISable** | Disable watch system for DLLs |
| **TASK.WatchDLL.ENable** | Enable watch system for DLLs |
| **TASK.WatchDLL.DISableBP** | Disable DLL creation breakpoints |
| **TASK.WatchDLL.ENableBP** | Enable DLL creation breakpoints |
| **TASK.WatchDLL.Option** | Set DLL watch system options |

## TASK.WatchDLL.ADD Add DLL to watch list

Format: **TASK.WatchDLL.ADD** *<dll>*

Specify the DLL name (in quotes) or magic to add this DLL to the watched DLLs list.

Please see **TASK.WatchDLL.View** for details.

## TASK.WatchDLL.DELete Remove DLL from watch list

Format: **TASK.WatchDLL.DELete** *<dll>*

Specify the DLL name (in quotes) or magic to remove this DLL from the watched DLLs list.

Please see **TASK.WatchDLL.View** for details.

## TASK.WatchDLL.DISable — Disable DLL watch system

Format: **TASK.WatchDLL.DISable**

Disables the complete watch system. The watched DLLs list is no longer checked against the target and is not updated. You'll see the **TASK.WatchDLL.View** window grayed out.

This feature is useful if you want to keep DLL symbols in the debugger, even if the DLL terminated.

## TASK.WatchDLL.DISableBP — Disable DLL creation breakpoints

Format: **TASK.WatchDLL.DISableBP**

Prevents the debugger from setting breakpoints for the detection of DLL creation. After executing this command, the target will run in real time. However, the watch system can no longer detect DLL creation. Automatic loading of DLL symbols will still work.

This feature is useful if you'd like to use limited breakpoints for other purposes.

Please see **TASK.WatchDLL.View** for details.

## TASK.WatchDLL.ENable — Enable DLL watch system

Format: **TASK.WatchDLL.ENable**

Enables the previously disabled watch system. It enables the automatic loading of DLL symbols as well as the detection of DLL creation.

Please see **TASK.WatchDLL.View** for details.

## TASK.WatchDLL.ENableBP Enable DLL creation breakpoints

Format: **TASK.WatchDLL.ENable**

Enables the previously disabled breakpoints for detection of DLL creation.

Please see **TASK.WatchDLL.View** for details.

## TASK.WatchDLL.Option Set DLL watch system options

Format: **TASK.WatchDLL.Option** *<option>*

*<option>*: **BreakOptC** *<option>*
**BreakOptM** *<option>*

Set various options to the watch system.

**BreakOptC** Set the option in double quotes, which is used to set the breakpoint on the DLL creation handler. The default option is “/Onchip”.
**Example**:

```
TASK.WatchDLL.Option BreakOptC "/SOFT"
```

**BreakOptM** Set the option in double quotes, which is used to set the breakpoint on the main entry point of the DLL. The default option is “/Onchip”.
**Example**:

```
TASK.WatchDLL.Option BreakOptC "/Hard"
```

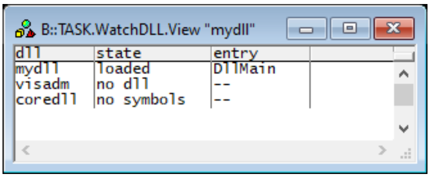
**NOTE**: The actual code of the DLL may not yet be loaded. Thus, setting Software breakpoints is not recommended.

Please see **TASK.WatchDLL.View** for details.

Format: **TASK.WatchDLL.View** [*<dll>*]

Activates the watch system for DLLs and shows a table of the watched DLLs.

**NOTE:** **This feature may affect the real-time behavior of the target application!** Please see below for details.



```
dll      state       entry
mydll    loaded      DllMain
visadm   no dll      --
coredll  no symbols  --
```

*<process>* Specify a DLL name for the initial DLL to be watched.

### Description of Columns in the TASK.WatchDLL.View Window

| | |
|---|---|
| **dll** | The name of the DLL to be watched. |
| **state** | The current watch state of the DLL.<br>If grayed, the debugger is currently not able to determine the watch state.<br>**no dll**: The debugger couldn't find the DLL in the current Windows CE DLL list.<br>**no symbols**: The debugger found the DLL and loaded the MMU settings of the DLL but couldn't load the symbols of the DLL (most likely because the corresponding .dll and .pdb files were missing).<br>**loaded**: The debugger found the DLL and loaded the DLL's MMU settings and symbols. |
| **entry** | The DLL entry point, which is usually DllMain`()`.<br>If grayed, the debugger is currently not able to detect the entry point or is unable to set the DLL entry breakpoint (e.g. because it is disabled with **TASK.Watch.DISableBP**). |

The watch system for DLLs is able to automatically load and unload the symbols of a DLL and its MMU settings, depending on their state in the target. Additionally, the watch system can detect the creation of a DLL and halts the DLL at its entry point.

| | |
|---|---|
| **TASK.WatchDLL.ADD** | Add DLLs to the watch list |
| **TASK.WatchDLL.DELete** | Remove DLLs from the watch list. |

The watch system for DLLs is active, as long as the **TASK.WatchDLL.View** window is open or iconized. As soon as this window is closed, the watch system will be deactivated.

### Automatic Loading and Unloading of DLL Symbols

In order to detect the current DLL, the debugger must have full access to the target, i.e. the target application must be stopped (with one exception, see below for creation of DLLs). As long as the target runs in real time, the watch system is not able to get the current DLL list, and the display will be grayed out (inactive).

If the target is halted (either by hitting a breakpoint, or by halting it manually), the watch system starts its work. For each of the DLL in the watch list, it determines the state of this DLL in the target.

If a DLL is active on the target, which was previously not found there, the watch system scans its MMU entries and loads the appropriate symbol files. In fact, it executes **TASK.sYmbol.LOADDLL** for the new DLL.

If a watched DLL was previously loaded, but is no longer found on the Windows CE DLL list, the watch system unloads the symbols and removes the MMU settings from the debugger MMU table. The watch system executes **TASK.sYmbol.DELeteDLL** for this DLL.

If the DLL was previously loaded, and is now found on another address (e.g. if the DLL terminated and started again), the watch system first removes the DLL symbols and reloads them to the appropriate address.

You can disable the loading / unloading of DLL symbols with the command **TASK.WatchDLL.DISable**.

### Detection of DLL Creation

To halt a DLL at its main entry point, the watch system can detect the DLL creation and set the appropriate breakpoints.

To detect the DLL creation, the watch system sets a breakpoint on a kernel function that is called upon creation of DLLs. Every time the breakpoint is hit, the debugger checks if a watched DLL is started. If not, it simply resumes the target application. If the debugger detects the start of a newly created (and watched) DLL, it sets a breakpoint onto the main entry point of the DLL (`DllMain()`) and resumes the target application. A short while after this, the main breakpoint will hit and halt the target at the entry point of the DLL. The DLL is now ready to be debugged.

| | |
|---|---|
| **NOTE:** | By default, this feature uses one permanent on-chip breakpoint and one temporary on-chip breakpoint when a DLL is created. Please ensure that at least those two on-chip breakpoints are available when using this feature. Use **TASK.WatchDLL.Option** to change the nature of the breakpoints.<br>Upon every DLL creation, the target application is halted for a short time and resumed after searching for the watched DLLs. **This impacts the real-time behavior of your target.** |

If you don't want the watch system to set breakpoints, you can disable them with the command **TASK.WatchDLL.DISableBP**. Of course, detection of DLL creation won't work then.

# Windows CE PRACTICE Functions

There are special definitions for Windows CE specific PRACTICE functions.

## TASK.CONFIG() OS Awareness configuration information

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

**Parameter and Description**:

| | |
|---|---|
| **magic** | **Parameter Type**: String (***without*** quotation marks).<br>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).<br>Returns the size of the task magic number (1, 2 or 4). |

**Return Value Type**: Hex value.

## TASK.DLL.CODEADDR() Address of code segment

Syntax: **TASK.DLL.CODEADDR("***<dll_name>***")**

Returns the address of the code segment of the specified DLL.

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

**Return Value Type**: Hex value.

## TASK.DLL.CURRENT() ‘magic’ of DLL

Syntax: **TASK.DLL.CURRENT("***<dll_name>***")**

Returns the magic number of the given DLL of the current process.

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

**Return Value Type**: Hex value.

## TASK.DLL.DATAADDR() Address of data segment

Syntax: **TASK.DLL.DATAADDR("***<dll_name>***")**

Returns the address of the data segment of the specified DLL.

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

**Return Value Type**: Hex value.

## TASK.DLL.MAGIC() ‘magic’ of DLL

Syntax: **TASK.DLL.MAGIC("***<dll_name>***",***<process_magic>***)**

Returns the magic of the given DLL belonging to the given process.

**Parameter and Description**:

| | |
|---|---|
| *<dll_name>* | **Parameter Type**: String (*with* quotation marks). |
| *<process_magic>* | **Parameter Type**: Decimal or hex or binary value. |

**Return Value Type**: Hex value.

## TASK.DLL.SECADDR() Address of section

Syntax: **TASK.DLL.SECADDR(***<dll_magic>,<section_id>***)**

Returns the address of the given section.

**Parameter and Description**:

| | |
|---|---|
| *<dll_magic>* | **Parameter Type**: Decimal or hex or binary value. |
| *<section_id>* | **Parameter Type**: Decimal or hex or binary value. |

**Return Value Type**: Hex value.

## TASK.DLL.SECNUM() Number of sections

Syntax: **TASK.DLL.SECNUM(***<dllmagic>***)**

Returns the number of sections.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.LOG2PHYS() Convert virtual address to physical address

Syntax: **TASK.LOG2PHYS(***<logical_address>***,***<process_magic>***)**

Convert virtual address of given process to physical address.

**Parameter and Description**:

| | |
|---|---|
| *<logical_address>* | **Parameter Type**: Decimal or hex or binary value. |
| *<process_magic>* | **Parameter Type**: Decimal or hex or binary value. |

**Return Value Type**: Hex value.

## TASK.PROC.CODEADDR() Address of code segment

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

Returns the address of the code segment of the specified process.

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

**Return Value Type**: Hex value.

## TASK.PROC.DATAADDR() Address of data segment

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

Returns the address of the data segment of the specified process.

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

**Return Value Type**: Hex value.

## TASK.PROC.M2S() Convert process magic number to space ID

Syntax: **TASK.PROC.M2S(*<process_magic>*)**

Converts the process magic number to the space ID.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.PROC.MAGIC() Process magic number of process

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

Returns the process magic number of the given process name.

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

**Return Value Type**: Hex value.

## TASK.PROC.S2M() Convert space ID to process magic number

Syntax: **TASK.PROC.S2M(*<space_id>*)**

Converts the space ID to the process magic number.

**Parameter Type**: Decimal or hex or binary 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.ROM.ADDR() Section address of ROM module

Syntax: **TASK.ROM.ADDR("***<module_name>***",***<section>***)**

Finds the section address of the given ROM module.

**Parameter and Description**:

| | |
|---|---|
| *<module_name>* | **Parameter Type**: String (*with* quotation marks). |
| *<section>* | **Parameter Type**: Decimal or hex or binary value. |

**Return Value Type**: Hex value.

## TASK.ROM.MAGIC() ‘Magic’ of ROM module

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

Returns the “magic” of the given ROM module.

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

**Return Value Type**: Hex value.

## TASK.ROM.SECADDR() Address of section

| Syntax: | **TASK.ROM.SECADDR(***<module>***,***<section_id>***)** |
|---|---|

Returns the address of the given section.

**Parameter and Description**:

| | |
|---|---|
| *<module>* | **Parameter Type**: Decimal or hex or binary value. |
| *<section_id>* | **Parameter Type**: Decimal or hex or binary value. |

**Return Value Type**: Hex value.

## TASK.ROM.SECNUM() Number of sections

| Syntax: | **TASK.ROM.SECNUM(***<module>***)** |
|---|---|

Returns the number of sections.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.THREAD.LIST() Thread list

| Syntax: | **TASK.THREAD.LIST(***<thread_magic>***)** |
|---|---|

Returns the next thread magic number in the thread list.

**Parameter Type**: Decimal or hex or binary value. Specify zero for the first thread.

**Return Value Type**: Hex value. Returns zero if no further thread is available.

## TASK.THREAD.PROC() Process of thread

Syntax: **TASK.THREAD.PROC(***<task_magic>***)**

Returns the magic number of the process owning the thread given by the parameter.

**Parameter Type**: Decimal or hex or binary value.

**Return Value Type**: Hex value.

## TASK.Y.O() Symbol option parameters

Syntax: **TASK.Y.O(***<item>* | **autoload)**

**Parameter and Description**:

| | |
|---|---|
| *<item>* | **Parameter Type**: String (***without*** quotation marks).<br>Reports symbol option parameters. |
| **autoload** | **Parameter Type**: String (***without*** quotation marks).<br>Returns the flags which components are checked by the symbol autoloader. |

**Return Value Type**: Hex value.