LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual
Windows CE4/CE5

MANUAL

OS Awareness Manual Windows CE4/CE5

OS Awareness Manual Windows CE4/CE5

Version 06-Jun-2024

History

04-Feb-21 Removing legacy command TASK.TASKState.

Overview

The OS Awareness for Windows CE 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 4.0, 4.1, 4.2 and 5.0 on ARM, MIPS, SH4 and XScale.

Configuration

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

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

If a system symbol 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 build doesn’t use standard symbols, you may try to use the manual configuration:

Format:	**TASK.CONFIG wince** *<magic_address> <oem_address_tab>*

<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”.

<oem_address_tab> This argument defines the start address of the OEM address translation table. The symbols “OEMAddressTable” (used by Windows CE 4.x) and “g_oalAddressTable” (used by Windows CE 5.x) are searched automatically. Set this parameter only, if you use different symbols for this table; set it to “0” otherwise.

See **Hooks & Internals** for details on the used symbols.

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 system internal symbols are loaded and 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 wince**

If a system symbol 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**).

Note that you have to use the Debug Build of your platform to get all necessary symbols.

See **Hooks & Internals** for details on the used symbols.

See also the example "`~~/demo/<processor>/kernel/wince/<version>/<board>/wince.cmm`".

Quick Configuration Guide

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

1. Carefully read the PRACTICE demo start-up script (`~~/demo/<processor>/kernel/wince/<version>/<board>/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.

No hooks are used in the kernel.

For retrieving the kernel data structures, the OS Awareness uses global kernel symbols and structure definitions. Ensure that access to those structures is possible every time when features of the OS Awareness are used.

Use the debug build of your platform. The Windows CE files “nk.exe” and “nk.pdb” contain the necessary symbols. Ensure that the symbols of “nk.exe” are loaded (see example script).

To get the kernel address translation, the OS Awareness searches for the symbols “`OEMAddressTranslation`” (CE 4.x) or “`g_oalAddressTranslation`” (CE 5.x). If none of these symbols is available, use the **Manual Configuration**.

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.MODule, TASK.DLL	Libraries (DLLs)
TASK.Event	Events
TASK.Mutex	Mutexes
TASK.Semaphore	Semaphores
TASK.ROM.MODule	Built-in modules
TASK.ROM.FILE	Built-in files

For a detailed description of each command, 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 Stack Coverage

For stack usage coverage of tasks, you can use the **TASK.STacK** command. Without any parameter, this command will open a window displaying with all active tasks. If you specify only a task magic number as parameter, the stack area of this task will be automatically calculated.

To use the calculation of the maximum stack usage, a stack pattern must be defined with the command **TASK.STacK.PATtern** (default value is zero).

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

It is recommended to display only the tasks 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.

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--0x01FFFFFF. If a process switch occurs, the MMU of the CPU is reprogrammed, so that this address range points to the current running process.

Additionally, the processes are mapped to higher addresses somewhere in between 0x04000000--0x7FFFFFFF, but this is not relevant for debugging.

DLLs are mapped, depending on their type. If they are private, they are mapped into the virtual address range of the owning process. If they are common to all processes, they are mapped into the address range 0x02000000--0x03FFFFFF.

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. For all processes using the kernel address space and for the kernel itself, the space ID is zero. For processes using their own address space, the space ID equaled the 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 "**User Processes**".

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>* *<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
WINCE5	Format used by Windows CE5

<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 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.
Specify **OEMAddressTable**

<logical_kernel_address_range>

<logical_kernel_address_range> specifies the virtual to physical address translation of the kernel address range. Currently not necessary.

<physical_kernel_address>

<physical_kernel_address> specifies the physical start address of the kernel. Currently not necessary.

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 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 and common DLLs.

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

Example:

```
MMU.FORMAT WINCE5 OEMAdressTable
TRANSlation.COMMON 0x02000000--0x03ffffff 0x80000000--0xffffffff
TRANSlation.ON
```

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

Scanning System and Processes

The command **MMU.SCAN** *only* scans the contents of the current processor MMU settings.

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

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

See also chapter **"Debugging Windows CE Kernel and User Processes"**.

Symbol Autoloader

The OS Awareness for Windows CE 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, or libraries (see also **TASK.sYmbol.Option AutoLoad**). It is recommended to set only those components you are interested in, because this decreases the time of the autoloader checks highly.

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.

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).

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"** (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, the ContextID register can be used to allow a detailed performance analysis on Windows CE threads. Set the upper 24 bit of the ContextID register with the ID of the thread, i.e. "`(thread.dwThrdId << 8)`".

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 armtrap.S or implement OEMReschedule:

1) Patching armtrap.S:

in _SetCPUASID, after "LEAF_ENTRY _SetCPUASID", change the lines:

```
ldr  r12, =KData                ; (r12) = ptr to KDataStruct
ldr  r1, [r0, #ThProc]          ; (r1) = ptr to thread's current process
ldr  r2, [r12, #CeLogStatus]    ; (r2) = KInfoTable[KINX_CELOGSTATUS]
```

to:

```
ldr  r12, =KData                ; (r12) = ptr to KDataStruct
ldr  r1, [r0, #ThProc]          ; (r1) = ptr to thread's current process
ldr  r2, [r0, #ThHandle]        ; (r2) = thread handle
mov  r2, r2, lsl #8             ; (r2) = thread context ID
ldrb r0, [r1, #PrcID]           ; (r0) = pCurProc->procnum
orr  r2, r2, r0                 ; (r2) = thread context ID + PID
mcr  p15, 0, r2, c13, c0, 1     ; write ContextID register
ldr  r2, [r12, #CeLogStatus]    ; (r2) = KInfoTable[KINX_CELOGSTATUS]
```

2) implement OEMReschedule():

in the OAL layer, implement the OAL function OEMReschedule() that writes
`(dwThrdId<<8)`
to the ContextID register.

NOTE: This feature is *only* available, if your debugger equipment is able to trace memory data accesses (flow trace is not sufficient).

The time different threads are in a certain state (running, ready, suspended or waiting) can be evaluated statistically or displayed graphically. This feature is implemented by recording all accesses to the status words of all threads. Additionally the accesses to the current thread pointer (=magic) are traced.

To do a selective recording on thread states, the following PRACTICE commands can be used:

```
; Mark the magic location with an Alpha breakpoint
Break.Set task.config(magic)++(task.config(magicsize)-1) /Alpha

; Program the Analyzer to record task state transitions
Analyzer.ReProgram
(
     Sample.Enable if AlphaBreak&&Write
)
```

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

Trace.STATistic.TASKState	Display task state statistic
Trace.Chart.TASKState	Display task state time chart

All kernel activities up to the thread switch are added to the calling thread. The start of the recording time, when the calculation doesn't know, which thread is running, is calculated as "(root)".

NOTE: This feature is *only* available, if your debugger equipment is able to trace memory data accesses (flow trace is not sufficient).

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

To do a selective recording on thread related function runtimes, the following PRACTICE commands can be used:

```
; Mark the magic location with an Alpha breakpoint
Break.Set task.config(magic)++(task.config(magicsize)-1) /Alpha

; Mark the function entries/exits with Alpha/Beta breakpoints
Break.SetFunc

; Program the Analyzer to record function entries/exits
; and task switches
Analyzer.ReProgram
(
     Sample.Enable if AlphaBreak||BetaBreak
     Mark.A        if AlphaBreak
     Mark.B        if BetaBreak
)
```

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

Trace.List List.TASK FUNC	Display function nesting
Trace.STATistic.TASKFunc	Display function runtime statistic
Trace.STATistic.TASKTREE	Display functions as call tree
Trace.Chart.TASKFunc	Display function time chart

All kernel activities up to the thread switch are added to the calling thread. The start of the recording time, when the calculation doesn't know, which thread is running, is calculated as "(root)".

Windows CE specific Menu

The menu file “wince.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**.
  - **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.
    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.
    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 “**Task 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 Windows CE Kernel and User Processes

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”**.

Windows CE 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. However, the Windows CE awareness needs several kernel symbols, that are stored in the files “nk.exe” and “nk.pdb”. Preserve those files.

Downloading the Kernel

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). If you use the binary, 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 may be overwritten).

Examples:

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

or:

```
Data.LOAD.Binary nk.nb0 0x0    ; loads the binary to address 0x0
```

When downloading the kernel via the debugger, remember to set startup options, that the kernel may require, before booting the kernel.

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 virtual addresses. If you want to debug this (tiny) startup sequence, you have to load and relocate the symbols. Relocate the ".text" segment to the physical start address of your kernel.

- Downloading the kernel via debugger:

  Download the kernel image and separately download the kernel symbols with relocation:

  ```
  Data.LOAD.EXE nk.bin
  Data.LOAD.EXE nk.exe /NoCODE /RELOC .text AT 0x81000
  ```

  After downloading, set your PC to the physical start address, and you're ready to debug.

- Downloading the kernel via Ethernet:

  Just load the symbols into the debugger *before* the image is downloaded by the boot monitor:

  ```
  Data.LOAD.EXE nk.exe /NoCODE /RELOC .text AT 0x81000
  ```

  Then, set an on-chip(!) breakpoint to the physical start address of the kernel (software breakpoints won't work, as they would be overwritten by the kernel download):

  ```
  Break.Set 0x81000 /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.

As soon as the processor MMU is switched on, you have to reload the symbol to its virtual addresses. See the next chapter on how to debug the kernel in the virtual address space.

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.

The symbols have to be relocated to the target’s address configuration. The build process generates a log file called *<project>*.plg in your project directory. This log file contains all necessary information.

Retrieving the relocation information manually:

To extract the addresses of nk.exe, use the DOS “find” command:

```
find /i "nk.exe" myproject.plg
```

Relocate the `.text` segment and the `.pdata` segment as given. Relocate the `.KDATA` segment and the `.data` segment NOT to the first given address, but to the address given after "`FILLER->`". E.g., if the output of the find command gives:

```
---------- PCA1.PLG
No imports for nk.exe
nk.exe          .text    98381000  282624  282624  282492 o32_rva=00001000
nk.exe          .pdata   983c6000    8192    8192    8104 o32_rva=00066000
nk.exe          .KDATA   98939000       0       0   40960 FILLER->82090000
nk.exe          .data    9884792c    1556    1556   86060 FILLER->8209a000
nk.exe          E32      98fdaf8c     108                 FILLER
nk.exe          O32      98671fa0      96                 FILLER
nk.exe          FileName 9863bff8       7                 FILLER
```

Then load the kernel symbols with the command (all in one line):

```
Data.LOAD.EXE nk.exe /NoCODE /RELOC .text AT 0x98381000 /RELOC .pdata AT
0x983c6000 /RELOC .KDATA AT 0x82090000 /RELOC .data AT 0x8209a000
```

Using a script for generating the load command automatically:

The demo directory contains a simple script file called “`nk.cmm`” to extract the relocation information and automatically generate the load command, stored in a file called “loadnk.cmm”. You may use this script then to load the kernel symbols:

```
do nk myproject         ; scans "myproject.plg" and generates
                        ; "loadnk.cmm"
do loadnk               ; actually loads and relocates "nk.exe"
```

Kernel MMU Settings:

The kernel address space (0x80000000 to 0xffffffff) and the XIP DLL address space (0x20000000 to 0x3fffffff) are visible to all processes, so specify the address ranges to be common to all space IDs:

```
TRANSlation.COMMON 0x20000000--0x3fffffff 0x80000000--0xffffffff
```

And switch on the debugger MMU translation:

```
TRANSlation.ON
```

User Processes

Each user process in Windows CE 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.

Note that at every time the Windows CE 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 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**").

Additionally, you have to scan the MMU translation table of this process:

```
TASK.MMU.SCAN 12.            ; scan MMU of process ID 12
```

It is possible to scan the translation tables of all processes at once. On some processors, and depending on your number of active processes, this may take a very long time. In this case use the scanning of single processes, mentioned above. Scanning all processes:

```
TASK.MMU.SCAN                ; scan MMU entries of all 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 `MDCreateMainThread1()` or `LoadSwitch()` (if available) 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.

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.

Manual Load:

Open a **TASK.DLL** window. Load the symbols and relocate the ".text" segment to the base address and the ".data" segment to the "rw base" address of the DLL. E.g., if the window contains the line:

```
magic    name         in_use base     entry    rw_base  referenced_by
83C8B500 coredll.dll  127.  03F71000 03F74AAC 01FFF000 0. 1. 2. 3.
```

Then load the appropriate DLL with the command (all in one line):

```
D.LOAD.EXE coredll.dll /NoCODE /NoClear /RELOC .text AT 0x03F71000 /RELOC
.data AT 0x01FFF000
```

Automatic Load:

If the files are in the standard search path, the 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.

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 “DumpFrame()”.

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

- “pth” points to the thread that caused the exception;
- “pCtx” points to a structure containing the complete register set at the location, where the fault occurred.
- “id” contains the exception ID that happened;
- “addr” contains the faulty address;
- “level” indicates the exception level.

When halted at “DumpFrame”, 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 “DumpFrame” are restored.

TASK.DLL

Display libraries

Format: **TASK.DLL**

Alias for TASK.MODule. See there.

TASK.Event

Display events

Format: **TASK.Event**

Display a table with all created synchronization objects of type “event”.

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

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.

Space IDs must be enabled (**SYStem.Option.MMU ON**) to successfully execute this command.

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

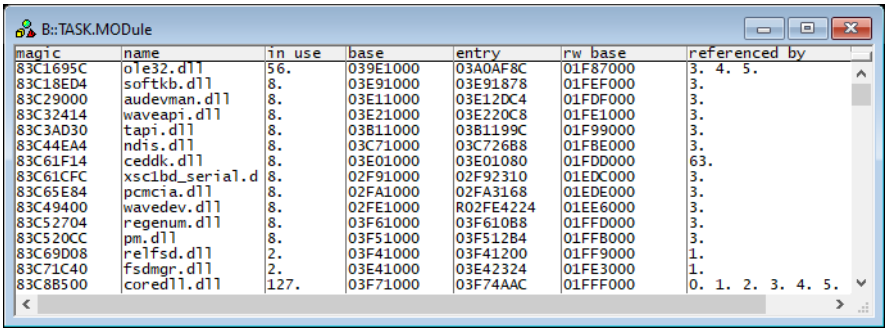
Example:

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

See also **MMU Support**.

Format: **TASK.MODULE**

Displays a table with all loaded modules (DLLs) of Windows CE.



B::TASK.MODule

magic	name	in use	base	entry	rw base	referenced by
83C1695C	ole32.dll	56.	039E1000	03A0AF8C	01F87000	3. 4. 5.
83C18ED4	softkb.dll	8.	03E91000	03E91878	01FEF000	3.
83C29000	audevman.dll	8.	03E11000	03E12DC4	01FDF000	3.
83C32414	waveapi.dll	8.	03E21000	03E220C8	01FE1000	3.
83C3AD30	tapi.dll	8.	03B11000	03B1199C	01F99000	3.
83C44EA4	ndis.dll	8.	03C71000	03C726B8	01FBE000	3.
83C61F14	ceddk.dll	8.	03E01000	03E01080	01FDD000	63.
83C61CFC	xsc1bd_serial.d	8.	02F91000	02F92310	01EDC000	3.
83C65E84	pcmcia.dll	8.	02FA1000	02FA3168	01EDE000	3.
83C49400	wavedev.dll	8.	02FE1000	R02FE4224	01EE6000	3.
83C52704	regenum.dll	8.	03F61000	03F610B8	01FFD000	3.
83C520CC	pm.dll	8.	03F51000	03F512B4	01FFB000	3.
83C69D08	relfsd.dll	2.	03F41000	03F41200	01FF9000	1.
83C71C40	fsdmgr.dll	2.	03E41000	03E42324	01FE3000	1.
83C8B500	coredll.dll	127.	03F71000	03F74AAC	01FFF000	0. 1. 2. 3. 4. 5.

“magic” is a unique ID, used by the OS Awareness to identify a specific module (address of the module structure).
“base” and “rw base” specify the code and data address of the module.
“entry” is the DLL entry point.
“referenced by” lists all process IDs, that loaded this module.

The field “magic” is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

TASK.Mutex Display mutexes

Format: **TASK.Mutex**

Display a table with all created synchronization objects of type “mutex”.

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

<option>: **THRCTX** [**ON** | **OFF**]

Set various options to the awareness.

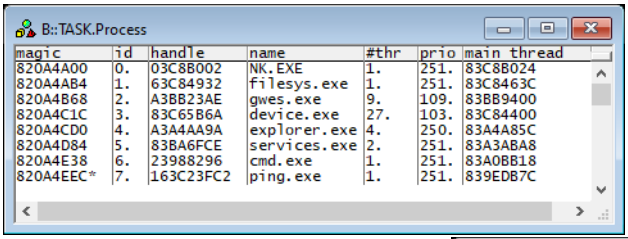
THRCTX Set the context ID type, that is recorded with the real-time trace (e.g. ETM).
If set to on, the context ID in the trace contains thread switch detection.
See **Task Runtime Statistics**.

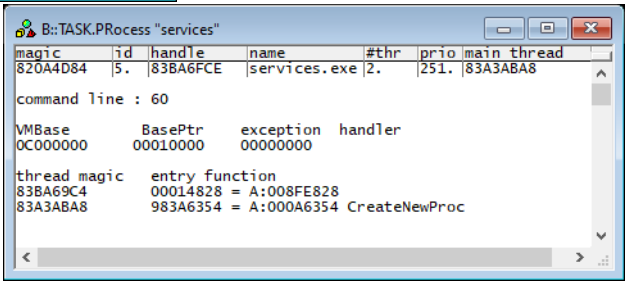
TASK.Process Display processes

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.





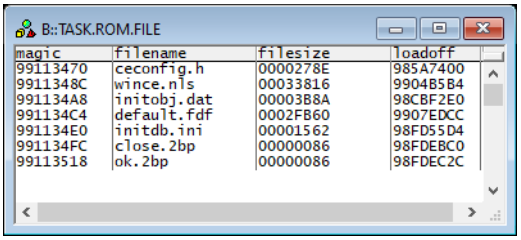
“magic” is a unique ID, used by the OS Awareness to identify a specific process (address of the process structure).

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.

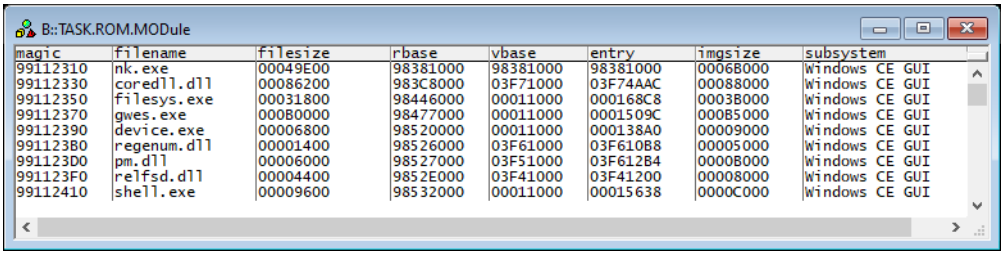


magic	filename	filesize	loadoff
99113470	ceconfig.h	0000278E	985A7400
9911348C	wince.nls	00033816	9904B5B4
991134A8	initobj.dat	00003B8A	98CBF2E0
991134C4	default.fdf	0002FB60	9907EDCC
991134E0	initdb.ini	00001562	98FD55D4
991134FC	close.2bp	00000086	98FDEBC0
99113518	ok.2bp	00000086	98FDEC2C

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.



magic	filename	filesize	rbase	vbase	entry	imgsize	subsystem
99112310	nk.exe	00049E00	98381000	98381000	98381000	0006B000	Windows CE GUI
99112330	coredll.dll	00086200	983C8000	03F71000	03F74AAC	00088000	Windows CE GUI
99112350	filesys.exe	00031800	98446000	00011000	000168C8	0003B000	Windows CE GUI
99112370	gwes.exe	000B0000	98477000	00011000	0001509C	000B5000	Windows CE GUI
99112390	device.exe	00006800	98520000	00011000	000138A0	00009000	Windows CE GUI
991123B0	regenum.dll	00001400	98526000	03F61000	03F610B8	00005000	Windows CE GUI
991123D0	pm.dll	00006000	98527000	03F51000	03F612B4	0000B000	Windows CE GUI
991123F0	relfsd.dll	00004400	9852E000	03F41000	03F41200	00008000	Windows CE GUI
99112410	shell.exe	00009600	98532000	00011000	00015638	0000C000	Windows CE GUI

TASK.Semaphore — Display semaphores

Format: **TASK.Semaphore**

Display a table with all created synchronization objects of type “semaphore”.

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.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.

Example:

When deleting the above loaded symbols with the command:

```
TASK.sYmbol.LOAD "ping"
```

the debugger will internally execute the commands:

```
TRANSlation.Delete 6.:0--0xffffffff
sYmbol.Delete \\pingTASK.sYmbol.LOAD "ping"
```

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.

Example:

When deleting the above loaded symbols with the command:

```
TASK.sYmbol.DELeteDLL "coredll"
```

the debugger will internally execute the commands:

```
TRANSlation.Delete 0:0x03F71000 …
sYmbol.Delete \\coredll
```

TASK.sYmbol.LOAD Load process symbols and MMU

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, and the process specific MMU settings (see chapter “Debugging User Processes”).
This command retrieves the appropriate space ID, loads the .exe and .pdb file of an existing process and reads its MMU entries. 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).

Example: If the **TASK.Process** window shows the entry:

```
magic____|id_|handle__|name___|#thr|prio|main_thread|
820A4E38*| 6.|839A7002|ping   |  1.|251.|83BA6964
```

the command:

```
TASK.sYmbol.LOAD "ping"
```

will internally execute the commands:

```
TASK.MMU.SCAN 6.
Data.LOAD.EXE ping.exe 6.:0 /NoCODE /NoClear
```

If the symbol file is not within the current directory, specify the path to the .exe file. E.g.:

```
TASK.sYmbol.LOAD "C:\mypath\ping.exe"
```

Loads the .exe/.pdb files “C:\mypath\ping.exe” of the process “ping”. Note that the process name must equal to the filename of the .exe file.

The command used to load the symbols can be customized with **TASK.sYmbol.Option LOADCMD**.

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 28).
This command retrieves the appropriate load addresses, loads the .dll and .pdb file of an existing DLL and reads configures the debugger MMU. Note that this command works only with DLLs, that are already loaded in Windows CE (i.e. DLLs that show up in the **TASK.DLL** window).

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

Example:

If the **TASK.DLL** window shows the entry:

```
magic____|name_______|in_use|base____|entry____|rw_base__|referenced_by|
83C8B500 |coredll.dll|127.  |03F71000|03F74AAC |01FFF000 |0. 1. 2. 3.
```

the command:

```
TASK.sYmbol.LOADDLL "coredll"
```

will internally execute the commands:

```
TRANSlation.Create 0:0x03F71000 …
Data.LOAD.EXE coredll.dll /NoCODE /NoClear /RELOC .text AT 0x03F71000 /
     RELOC .data AT 0x01FFF000
```

If the symbol file is not within the current directory, specify the path to the .dll file. E.g.:

```
TASK.sYmbol.LOAD "C:\mypath\coredll.dll"
```

Loads the .dll/.pdb files "C:\mypath\coredll.dll" of the DLL "coredll". Note that the DLL name must equal to the filename of the .dll file.

The command used to load the symbols can be customized with **TASK.sYmbol.Option LOADDCMD**.

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

<option>: **LOADCMD** *<command>*
LOADDCMD *<command>*
MMUSCAN [**ON** | **OFF**]
AutoLoad *<option>*

Set various options to the symbol management.

LOADCMD:
This setting is only active, if the symbol autoloader for processes is off.

TASK.sYmbol.LOAD uses a default load command to load the symbol file of the process. This loading command can be customized using this option with the command enclosed in quotes. Two parameters are passed to the command in a fixed order:
%s name of the process
%x space ID of the process.

Examples:

```
TASK.sYmbol.Option LOADCMD "Data.LOAD.EXE %s.exe 0x%x:0 /NoCODE /NoClear"

TASK.sYmbol.Option LOADCMD "do myloadscript %s 0x%x"
```

LOADDCMD:
This setting is only active, if the symbol autoloader for DLLs is off.

TASK.sYmbol.LOADDLL uses a default load command to load the symbol file of the DLL. This loading command can be customized using this option with the command enclosed in quotes. Three parameters are passed to the command in a fixed order:
%s name of the DLL
%x code address of the DLL
%x data address of the DLL.

Example:

```
TASK.sYmbol.Option LOADCMD "Data.LOAD.EXE %s.dll 0:0
     /NoCODE /NoClear /RELOC .text AT %x /RELOC .data AT %x"
```

MMUSCAN:
This option controls, if the symbol loading mechanisms of **TASK.sYmbol** scan the MMU page tables of the loaded components, too. When using **TRANSlation.TableWalk**, then switch this off.

AutoLoad:

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

Process	check processes
Library	check libraries
RomMod	check ROM modules
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.

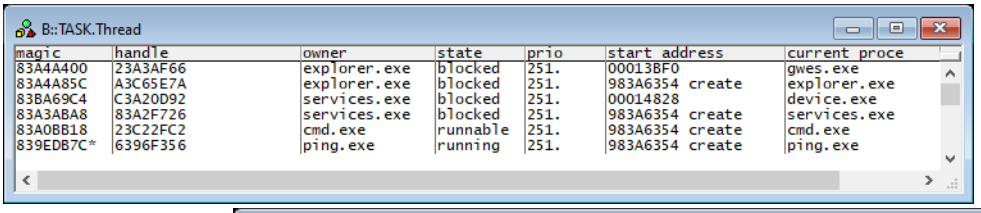
Example:

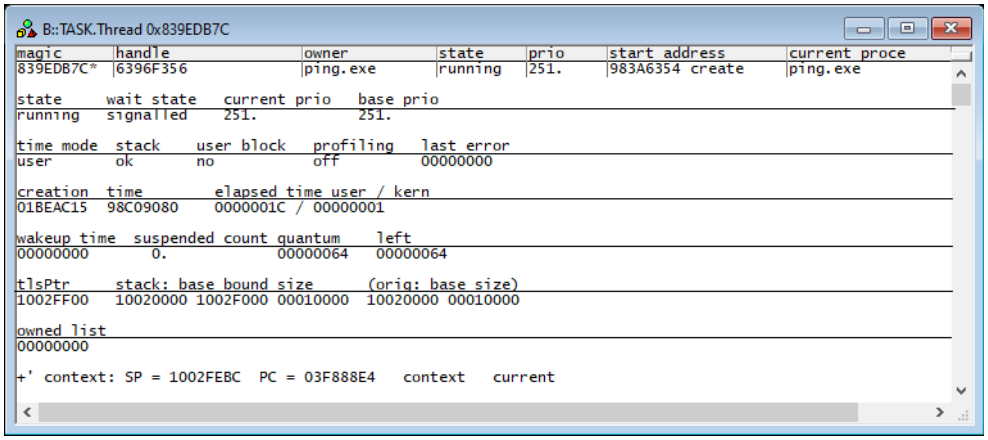
```
; check processes and ROM modules
TASK.sYmbol.Option AutoLoad Process
TASK.sYmbol.Option AutoLoad RomMod
```

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.





“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**”.

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.

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>* **BreakOptM** *<option>*

Set various options to the watch system.

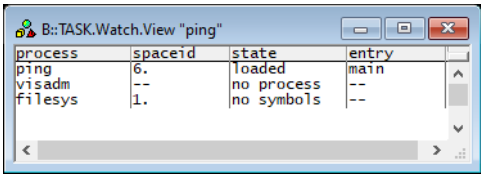
BreakOptC	Set the option in double quotes, which is used to set the breakpoint on the process creation handler. The default option is “/Onchip”. **Example**: `TASK.Watch.Option BreakOptC "/SOFT"`
BreakOptM	Set the option in double quotes, which is used to set the breakpoint on the main entry point of the process. The default option is “/Onchip”. **Example**: `TASK.Watch.Option BreakOptC "/Hard"` **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!** Please see below for details.



<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. 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. If grayed, the debugger is currently not able to determine the watch state. **no process**: The debugger couldn't find the process in the current Windows CE process list. **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). **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()`. 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. 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.

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.

Format: **TASK.WatchDLL.ENable**

Enables the previously disabled breakpoints for detection of DLL creation.

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.Option Set DLL watch system options

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

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

Set various options to the watch system.

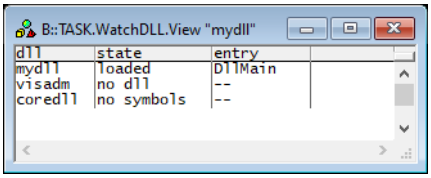
BreakOptC	Set the option in double quotes, which is used to set the breakpoint on the DLL creation handler. The default option is “/Onchip”. **Example**: `TASK.WatchDLL.Option BreakOptC "/SOFT"`
BreakOptM	Set the option in double quotes, which is used to set the breakpoint on the main entry point of 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.



<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. If grayed, the debugger is currently not able to determine the watch state. **no dll**: The debugger couldn't find the DLL in the current Windows CE DLL list. **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). **loaded**: The debugger found the DLL and loaded the DLL's MMU settings and symbols.
entry	The DLL entry point, which is usually DllMain ( ) . 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.

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). 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.CURRENT() 'vmbase' address of process

Syntax: **TASK.CURRENT(vmbase)**

Returns the vmbase address of the current process (0 for kernel).

Parameter Type: String (***without*** quotation marks).

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.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.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.SPACEID() Space ID of process

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

Returns the debugger MMU 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*>**)**

Find section address of the given ROM modules.

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.

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

Parameter and Description:

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

Return Value Type: Hex value.