LAUTERBACH
DEVELOPMENT TOOLS

OS Awareness Manual uClinux

MANUAL

OS Awareness Manual uClinux

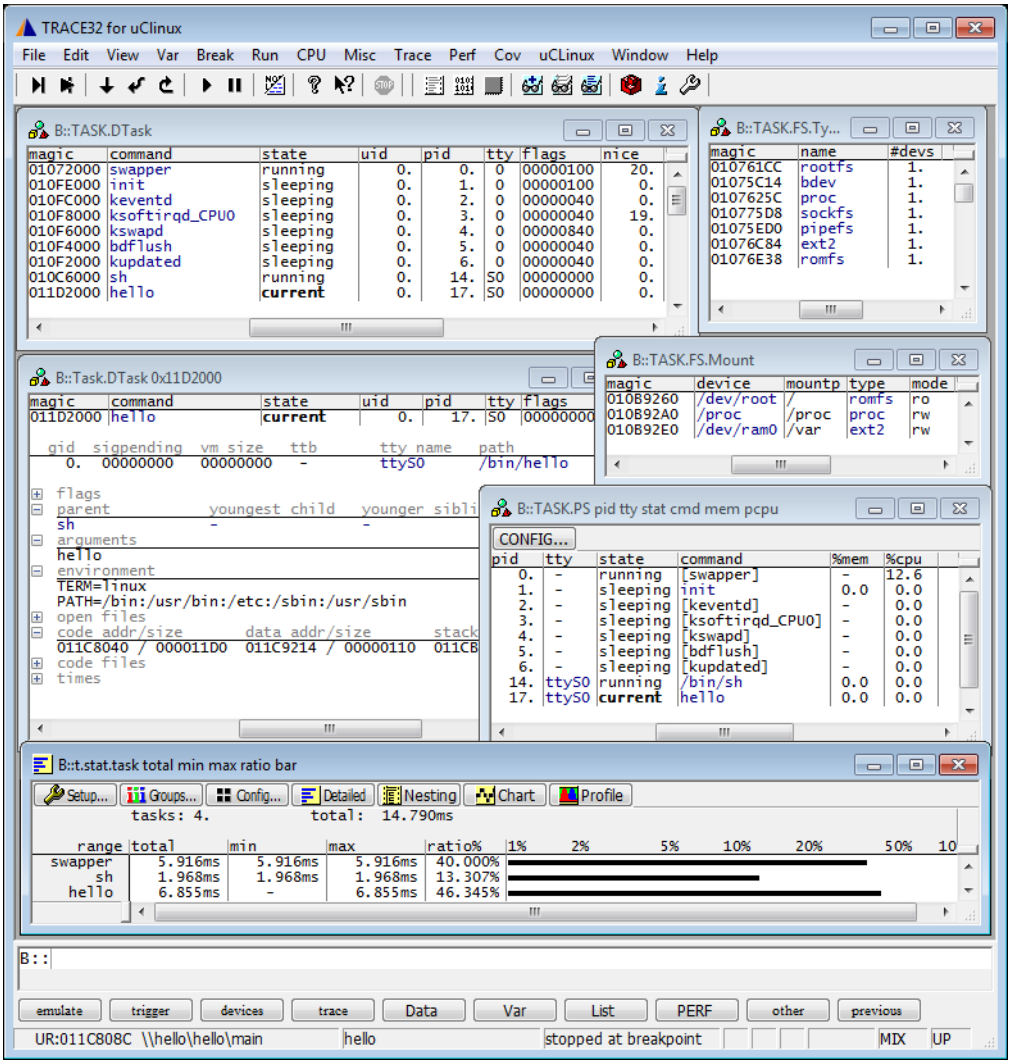
OS Awareness Manual uClinux

Version 06-Jun-2024

History

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

Overview



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

Terminology

uCLinux uses the terms “processes” and “tasks”. If not otherwise specified, the TRACE32 term “task” corresponds to uCLinux tasks (executing processes).

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 uCLinux is supported for the following versions:

- uCLinux (Linux kernel version 2.4 and 2.6) on 68k/ColdFire, ARM architecture, Blackfin, MicroBlaze, MIPS32 and Nios-II

Configuration

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

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

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).

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 uclinux**

Note that the default uCLinux configuration does not include any kernel symbols to the debug information. Please change your configuration to generate kernel debug information.

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

See also the example “~~/demo/<*processor*>/kernel/uclinux/uclinux.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/<*arch*>/kernel/uclinux/uclinux.cmm).
2. Make a copy of the PRACTICE script file “uclinux.cmm”. Modify the file according to your application.
3. Run the modified version in your environment. This should allow you to display the kernel resources, use the trace functions (if available) and debug processes.

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

Hooks & Internals in uCLinux

No hooks are used in the kernel.

For retrieving the kernel data structures, the OS Awareness uses the global kernel symbols and structure definitions. Ensure that access to those structures is possible every time when features of the OS Awareness are used. This requires that the whole Linux kernel is compiled with debug symbols switched on, and that the symbols of the “vmlinux” file are loaded.

If you control the compile stage by hand, just switch on debug symbols by adding the option “-g” to gcc. In most kernel configuration scripts, you have an option “Kernel Hacking” -> “Compile kernel with debug info” that enables debug symbols to the kernel.

Features

The OS Awareness for uCLinux supports the following features.

Terminal Emulation

TRACE32 Terminal Emulation is available for two interfaces: printk debug outputs and console driver. The communication via two memory buffers requires no external interface.

The demo directory contains a PRACTICE script file (patch_printk.cmm) to reroute printk outputs to a TRACE32 terminal. Please adjust the output addresses before using the patch in your application.

To use a terminal emulation as console device, you have to include the TRACE32 serial driver (t32serial.c) as console driver into your uCLinux image. See the “uclinux.cmm” file for details on starting the terminal.

See also the **TERM** command for a description of the terminal emulation.

Display of Kernel Resources

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

TASK.DTask	Tasks
TASK.PS	“ps” outputs
TASK.MODule	Kernel modules
TASK.FS	File system internals

For a detailed description of each command, refer to chapter “**uCLinux 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-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.

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 OS Awareness for uClinux contains an autoloader, which automatically loads symbol files. The autoloader maintains a list of address ranges, corresponding uClinux 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, kernel 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, are retrieved from kernel-internal information.

sYmbol.AutoLOAD.CHECKLINUX "*<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)".

NOTE:	This feature is *only* available, if your debug environment is able to trace task switches and data accesses (program flow trace is not sufficient). It requires either an on-chip trace logic that is able to generate a 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).

The time different tasks are in a certain state (running, ready, suspended or waiting) can be evaluated statistically or displayed graphically.

This feature requires that the following data accesses are recorded:

- All accesses to the status words of all tasks
- Accesses to the current task variable (= magic address)

Adjust your trace logic to record all data write accesses, or limit the recorded data to the area where all TCBs are located (plus the current task pointer).

Example: This script assumes that the TCBs are located in an array named TCB_array and consequently limits the tracing to data write accesses on the TCBs and the task switch.

```
Break.Set Var.RANGE(TCB_array) /Write /TraceData
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

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

Trace.STATistic.TASKState	Display task state statistic
Trace.Chart.TASKState	Display task state timechart

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

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)".

uCLinux specific Menu

The menu file "uclinux.men" contains a menu with uCLinux specific menu items. Load this menu with the **MENU.ReProgram** command.

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

- 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 it's main() function. Select this prior to starting the process. Specify the name of the process you want to debug. Then start the process in your Linux terminal. 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.
- **Module Debugging** refers to actions related to kernel module based debugging.
  See also chapter "**Kernel Modules**".
  - Use **Load Symbols** and **Delete Symbols** to load rsp. delete the symbols of a specific kernel module. You may select a symbol file on the host with the **Browse** button. See also **TASK.sYmbol**.
- 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 Linux components should be managed by the autoloader. See also **TASK.sYmbol.Option AutoLOAD**.
- **Linux Terminal** opens a terminal window that can be configured prior to opening with **Configure Terminal**.
- The **Stack Coverage** submenu starts and resets the uCLinux specific stack coverage and provides an easy way to add or remove tasks from the stack coverage window.

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 uCLinux Kernel and User Processes

uClinux runs on physical address spaces. Each user process gets its own unique address space and is linked to this address when loaded. This linking stage needs some attention when debugging uClinux processes.

uCLinux Kernel

The uCLinux make process can generate different outputs (e.g. zipped, non-zipped, with or without debug info). For downloading the uClinux kernel, you may choose whatever format you prefer. However, the uCLinux awareness needs several kernel symbols, i.e. you have to compile your kernel with debug information and preserve the resulting kernel file (usually “image.elf”). This file is in ELF format, and all other kernel images are derived from this file.

Downloading the Kernel

If you start the uClinux 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 kernel image using the debugger, you have to specify, to which address to download it. The uCLinux kernel image is usually located at the physical start address of the RAM (sometimes the vector table is skipped, check label `_stext` in the system map).

When downloading a binary image, specify the start address, where to load. E.g., if the physical address starts at `0x01000000`:

```
Data.LOAD.Binary linux.bin 0x01000000 /nosymbol
```

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

Debugging the Kernel

For debugging the kernel itself, and for using the uCLinux awareness, you have to load the symbols of the kernel into the debugger. The uClinux ELF image contains all addresses linked correctly, so it’s enough to simply load the file:

```
Data.LOAD.Elf vmlinux /GNU /NoCODE
```

User Processes

Each user process in uCLinux gets its own memory area, to which the process is linked when loaded.

Note that at every time the uCLinux 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 linked addresses of this process, it is necessary to load the symbols and relocate them to the appropriate addresses.

Manually Load Process Symbols:

The debugger can read out the load addresses of a process and relocate the symbols accordingly. The option **/RELOCTYPE** of the **Data.LOAD.Elf** command instructs the debugger to use the relocation information on the target. Internally to the Linux Awareness, processes have the type 1, so specify this as reloctype. Please note that access to the kernel variables must be possible whenever executing this command.

For example, if you've got a process called "hello":

```
Data.LOAD.Elf hello /NoCODE /NoClear /RELOCTYPE 1
```

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 relocate
```

This command loads the symbols of "hello" and relocates the symbols to the appropriate addresses. 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:

The uCLinux awareness provides the script `app_debug.cmm` that allows to debug a process from the start. This script can be found in the path of the uCLinux awareness.

The **uCLinux** menu offers the same feature in a menu item: **uCinux** -> **Process Debugging** -> **Debug Process on main** which is based on the `app_debug.cmm` script. See also chapter "**uCinux 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 into Shared Libraries

If the process uses shared libraries, uCLinux loads them into own address areas. 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:

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

   A list will appear that shows the loaded libraries and the corresponding load addresses.
4. Load the symbols to this address.

   E.g. if the library is called "lib.so" and it is loaded on address 0xff8000, then use the command:

   ```
   Data.LOAD.Elf lib.so 0xff8000 /GNU /NoCODE /NoClear
   ```

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

Automatically Load Library Symbols:

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

```
TASK.sYmbol.LOADLib "hello" "libc.so"      ; load symbols
```

This command loads the symbols of the library “libc.so”, used by the process “hello”. See **TASK.sYmbol.LOADLib** 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 library. You can also force the loading of the symbols of a library with

```
sYmbol.AutoLoad.CHECK
sYmbol.AutoLoad.TOUCH "libc.so"
```

Debugging uCLinux Threads

uCLinux Threads are implemented as tasks that share the same memory. It is sufficient, to load the debug information of this process only once to debug all threads of this process. See chapter “**Debugging the Process**” for loading the process’ symbols.

There are several different mechanisms how threads are managed inside the Linux kernel. The Linux Awareness tries to detect them automatically, but this may fail on some systems. If the **TASK.DTask** window doesn’t show all threads of a process, declare the threading method manually with the **TASK.Option Threading** command.

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

Kernel Modules

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

Please see the document “**OS Awareness Manual Linux**”, chapter “**Kernel Modules**” for a detailed description how to load kernel module symbols. Ignore all MMU statements done there.

TASK.DMESG — Display the kernel ring buffer

Format: **TASK.DMESG** [/*<option>*]

<option>: **Level** *<log_level>*
Facility *<log_facility>*
DETAILED
COLOR

Display the kernel messages in a window similar to the dmesg Linux command.

Level — Restrict the displayed log messages to the specified log levels. This option can be used multiple times. Log levels names or numbers can be used with this options. Level names: EMERG, ALERT, CRIT, ERR, WARN, NOTICE, INFO, DEBUG.

Facility — Restrict the displayed log messages to the specified log facilities. This option can be used multiple times. Log facility names or numbers can be used with this options. Facility names: KERN, USER, MAIL, DAEMON, AUTH, SYLOG, LPR, NEWS.

DETAILED — Display the log level and facility as readable strings.

COLOR — Enable the coloring of the log messages according to the log level and facility.

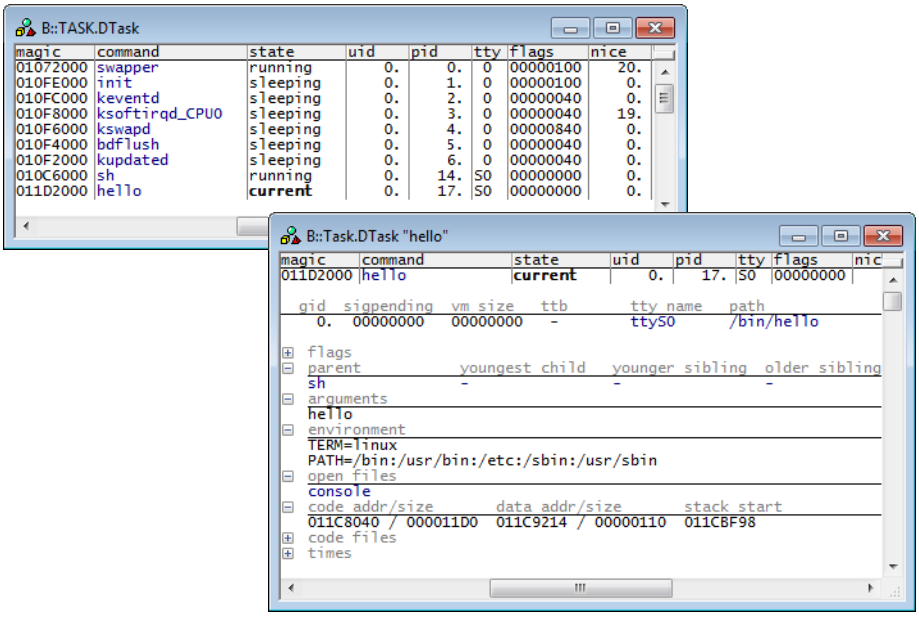
TASK.DTask — Display tasks

Format: **TASK.DTask** [*<task>*]

Displays the task table of uCLinux or detailed information about one specific task.
“Tasks” are activated processes.

Without any arguments, a table with all created tasks will be shown.

Specify a task name, ID or magic number to display detailed information on that task.



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

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

TASK.DTB — Display the device tree blob

Format:	**TASK.DTB**

Display the device tree blob as a tree view.

TASK.DTS — Display the device tree source

Format:	**TASK.DTS**

Display the source code of the device tree.

Format: **TASK.FS.**<*sub_cmd*>

<*sub_cmd*>: **Types** | **MountDevs** | **PROC**

This command displays internal data structures of the used file systems.See the appropriate command description for details.

TASK.FS.MountDevs Display mounted devices

Format: **TASK.FS.MountDevs**

This command displays all currently mounted devices (i.e. super blocks).

TASK.FS.PROC Display /proc file system

Format: **TASK.FS.PROC**

This command displays the contents of the “/proc” file system (procfs), even if it is not mounted.

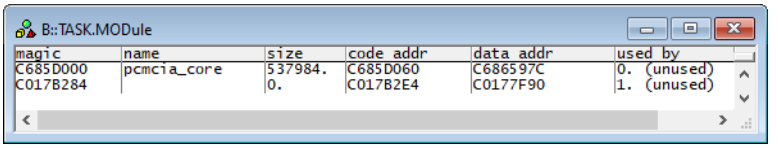
TASK.FS.Types Display file system types

Format: **TASK.FS.Types**

This command displays the all file system types that are currently registered in the uCLinux kernel.

Format: **TASK.MODule**

Displays a table with all loaded kernel modules of uCLinux. The display is similar to the output of “lsmod”.



“magic” is a unique ID, used by the OS Awareness to identify a module (address of the module struct).
“code addr” and “data addr” specify the address of the .text segment rsp. the .data segment.
The field “magic” is mouse sensitive, double clicking on it opens an appropriate window. Right clicking on it will show a local menu.

TASK.MAPS Display process maps

Format: **TASK.MAPS** <*process*>

Display the mapped memory regions and their access permissions similar to the /proc/[pid]/maps file.

TASK.NET Display network devices

Format: **TASK.NET**

Display network devices.

Format:	**TASK.Option** <*option*>
<*option*>:	**Threading** <*threading*> [**ON** | **OFF**] **NameMode** [**comm** | **TaskName** | **ARG0** | **ARG0COMM**] **THRCTX** [**ON** | **OFF**]

Sets various options to the awareness.

Threading Set the Threading type used by Linux.
TGROUP: threads are organized by the thread_group list.
See also chapter "**Debugging Threads**".

NameMode Set the mode how the task names are evaluated.
comm: use the "comm" field in the task structure (default).
TaskName: use the name evaluated by TASK.NAME and "comm" (allows renaming of the tasks with TASK.NAME.Set).
ARG0: use the arg[0] statement of the process call.
ARG0COMM: use arg[0] as process name and "comm" as thread name (suitable for Android)

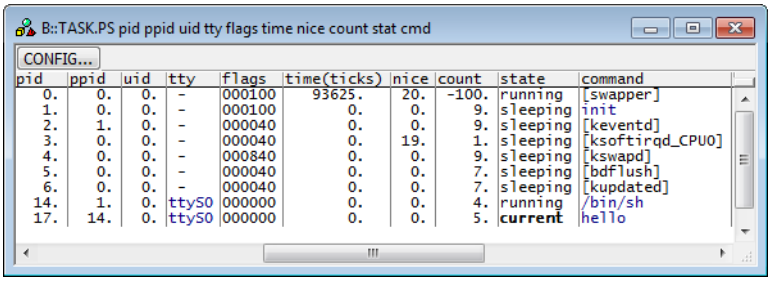
THRCTX Set the context ID type that is recorded with the real-time trace (e.g. ETM).
If set to on, the context ID in the trace contains thread switch detection.
See **Task Runtime Statistics**.

TASK.PS Display "ps" output

Format:	**TASK.PS** <*items*>
<*items*>:	**pid** | **ppid** | **uid** | **sid** | **pgid** | **cmd** | **pri** | **flags** | **tty** | **time** | **stat** | **nice** | **stackp** | **tmout** | **alarm** | **pending** | **blocked** | **vsz** | **rss** | **start** | **majflt** | **minflt** | **trs** | **drs** | **rss** | **count** | **nswap** | **ttb**

Displays the process table of uCLinux.

The display is similar to the output of the “ps” shell command.



pid	ppid	uid	tty	flags	time(ticks)	nice	count	state	command
0.	0.	0.	-	000100	93625.	20.	-100.	running	[swapper]
1.	0.	0.	-	000100	0.	0.	9.	sleeping	init
2.	1.	0.	-	000040	0.	0.	9.	sleeping	[keventd]
3.	0.	0.	-	000040	0.	19.	1.	sleeping	[ksoftirqd_CPU0]
4.	0.	0.	-	000840	0.	0.	9.	sleeping	[kswapd]
5.	0.	0.	-	000040	0.	0.	7.	sleeping	[bdflush]
6.	0.	0.	-	000040	0.	0.	7.	sleeping	[kupdated]
14.	1.	0.	ttyS0	000000	0.	0.	4.	running	/bin/sh
17.	14.	0.	ttyS0	000000	0.	0.	5.	**current**	hello

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

TASK.sYmbol.LOAD	Load process symbols
TASK.sYmbol.DELete	Unload process symbols
TASK.sYmbol.LOADMod	Load module symbols
TASK.sYmbol.DELeteMod	Unload module symbols
TASK.sYmbol.LOADLib	Load library symbols
TASK.sYmbol.DELeteLib	Unload library symbols
TASK.sYmbol.Option	Set symbol management options

TASK.sYmbol.DELete Unload process symbols

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

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

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

Example: When deleting the above loaded symbols with the command:

```
TASK.sYmbol.DELete "hello"
```

the debugger will internally execute the commands:

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

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

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

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

Example:

```
TASK.sYmbol.DELeteLib "hello" "libc-2.2.1.so"
```

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

TASK.sYmbol.DELeteMod Unload module symbols

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

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

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

Example:

```
TASK.sYmbol.DELeteMod "pcmcia_core"
```

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

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

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

In order to debug a user process, the debugger needs the symbols of this process (see chapter “Debugging User Processes”).
This command retrieves the appropriate addresses, loads the symbol file of an existing process and relocates the symbol information. Note that this command works only with processes that are already loaded in uCLinux (i.e. processes that show up in the **TASK.DTask** window).

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

```
magic___|command_|state___|uid___|pid___|tty|flags___|nice__|
80044000|hello   |current |    0.|   24.| 0 |00000000|    0.|
```

the command:

```
TASK.sYmbol.LOAD "hello"
```

will internally execute the commands:

```
Data.LOAD.Elf hello /GNU /NoCODE /NoClear /RELOC .text at 0x…
```

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

```
TASK.sYmbol.LOAD "C:\mypath\hello"
```

Loads the ELF file “C:\mypath\hello” of the process “hello”. Note that the process name must equal to the filename of the ELF file.

TASK.sYmbol.LOADLib Load library symbols

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

As first parameter, specify the process to which the desired library belongs (name in quotes or magic).

Specify the library name in quotes as second parameter. The library name **must** match the name as shown in **TASK.DTASK** *<process>*, “code files”.

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

Example:

```
TASK.sYmbol.LOADLib "hello" "libc-2.2.1.so"
```

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

TASK.sYmbol.LOADMod Load module symbols

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

Specify the module name (in quotes) or magic to load the symbols of this module.
In order to debug a kernel module, the debugger needs the symbols of this module (see chapter “Debugging Kernel Modules”).

This command retrieves the appropriate load addresses and loads the .o/.ko symbol file of an existing module. Note that this command works only with modules that are already loaded in uCLinux (i.e. modules that show up in the **TASK.MODule** window).

Example:

```
TASK.sYmbol.LOADMod "pcmcia_core"
```

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

Format:	**TASK.sYmbol.Option** <*option*>
<*option*>:	**LOADCMD** <*command*> **LOADMCMD** <*command*> **LOADLCMD** <*command*> **AutoLoad** <*option*>

Sets a specific option 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
%v	space ID of the process

Examples:

```
TASK.sYmbol.Option LOADCMD "data.load.elf %s 0x%x:0 /NoCODE /NoClear"

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

LOADMCMD:
This setting is only active, if the symbol autoloader for kernel modules is off.
TASK.sYmbol.LOADMod uses a default load command to load the symbol file of the module. 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:

Examples:

```
TASK.sYmbol.Option LOADMCMD "data.load.elf %s /NoCODE /NoClear /gcc3
     /reloc .text at 0x%x /reloc .data at 0x%x /reloc .bss after .data"

TASK.sYmbol.Option LOADMCMD "do myloadmscript %s 0x%x 0x%x"
```

%s	name of the module
%x	start (=code) address of the module
%x	data address of the module (if applicable)

LOADLCMD:
This setting is only active, if the symbol autoloader for libraries is off.
TASK.sYmbol.LOADLib uses a default load command to load the symbol file of the library. 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 library
%x	space ID of the library
%x	oad address of the library

Examples:

```
TASK.sYmbol.Option LOADLCMD "D.LOAD.Elf %s 0x%x:0x%x /NoCODE /NoClear"

TASK.sYmbol.Option LOADMCMD "do myloadlscript %s 0x%x 0x%x"
```

AutoLoad:
This option controls, which components are checked and managed by the AutoLoader:

Process	check processes
Library	check all libraries of all processes
Module	check kernel modules
CurrLib	check only libraries of current process
ALL	check processes, libraries and kernel modules
NoProcess	don't check processes
NoLibrary	don't check libraries
NoModule	don't check modules
NONE	check nothing.

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

Example:

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

Format: **TASK.VMAINFO**

(Not available for all processors!)

The TASK.Watch command group build a watch system that watches your uClinux 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 on-chip 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 the on-chip 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 on-chip breakpoints for detection of process creation.

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

Format:	**TASK.Watch.Option** *<option>*
<option>:	**BreakFunC** *<function>*

Set various options to the watch system.

BreakFunC Set the breakpoint location for process creation detection.
Depending on the target CPU, this may be “start_thread” or “set_binfmt”.
Example:

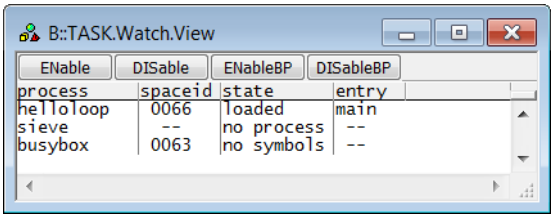
```
TASK.Watch.Option BreakFunC set_binfmt
```

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.
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 uClinux process list. **no symbols**: The debugger found the process but couldn't load the symbols of the process (most likely because the corresponding symbol files were missing). **loaded**: The debugger found the process and loaded the symbols of the process.
entry	The process entry point, which is `main()`. 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, 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 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 Linux process list, the watch system unloads the symbols. The watch system executes **TASK.sYmbol.DELete** for this process.

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

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 an on-chip 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 an on-chip breakpoint onto the main entry point of the process (`main()`) 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:	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. 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.

uCLinux PRACTICE Functions

There are special definitions for uCLinux 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.ERROR.CODE() Error code

Syntax: **TASK.ERROR.CODE()**

Checks for Awareness errors and returns the error code.

Return Value Type: Hex value.

Return Value and Description:

0	No error.
1	Failed to detect kernel symbols.
2	Failed to detect kernel structures.
4	Failed to detect kernel structure members.
8	Pointer size does not fit.

TASK.ERROR.HELP() Error help ID

Syntax: **TASK.ERROR.HELP()**

Checks for Awareness errors and returns the error help ID.

Return Value Type: String.

TASK.LIB.ADDRESS() Load address of library

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

Returns the load address of the library, loaded by the specified process.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.LIB.CODESIZE() Code size of library

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

Returns the code size of the library, loaded by the specified process.

Parameter and Description:

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

Return Value Type: Hex value.

TASK.MOD.CODEADDR() Code start address of module

Syntax: **TASK.MOD.CODEADDR(***<module_name>***)**

Returns the code start address of the module.

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

Return Value Type: Hex value.

TASK.MOD.DATAADDR() Data start address of module

Syntax: **TASK.MOD.DATAADDR(**<*module_name*>**)**

Returns the data start address of the module.

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

Return Value Type: Hex value.

TASK.MOD.MAGIC() Magic value of module

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

Returns the “magic” value of the module.

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

Return Value Type: Hex value.

TASK.MOD.NAME() Name of module magic

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

Returns the name of the given module magic.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

TASK.MOD.SECTION() Address of module

Syntax: **TASK.MOD.SECTION("***<section_name>***","***<module_magic>***")**

Returns the address of the section of the specified module.

Parameter and Description:

<section_name>	**Parameter Type**: String (*with* quotation marks).
<module_magic>	**Parameter Type**: String (*with* quotation marks).

Return Value Type: Hex value.

TASK.PROC.CODEADDR() Code start address of process

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

Returns the code start address of the process.

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

Return Value Type: Hex value.

TASK.PROC.CODESIZE() Code size of process

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

Returns the code size of the process.

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

Return Value Type: Hex value.

TASK.PROC.DATAADDR() Data start address of process

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

Returns the data start address of the process.

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

Return Value Type: Hex value.

TASK.PROC.DATASIZE() Data size of process

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

Returns the data size of the process.

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

Return Value Type: Hex value.

TASK.PROC.MAGIC() Magic value of process

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

Returns the “magic” value of the process.

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

Return Value Type: Hex value.

TASK.PROC.NAME() Name of process

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

Returns the name of the specified process.

Parameter Type: Decimal or hex or binary value.

Return Value Type: String.

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

Returns the process ID of the specified process.

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

Return Value Type: Hex value.