OS Awareness Manual Chorus Classic

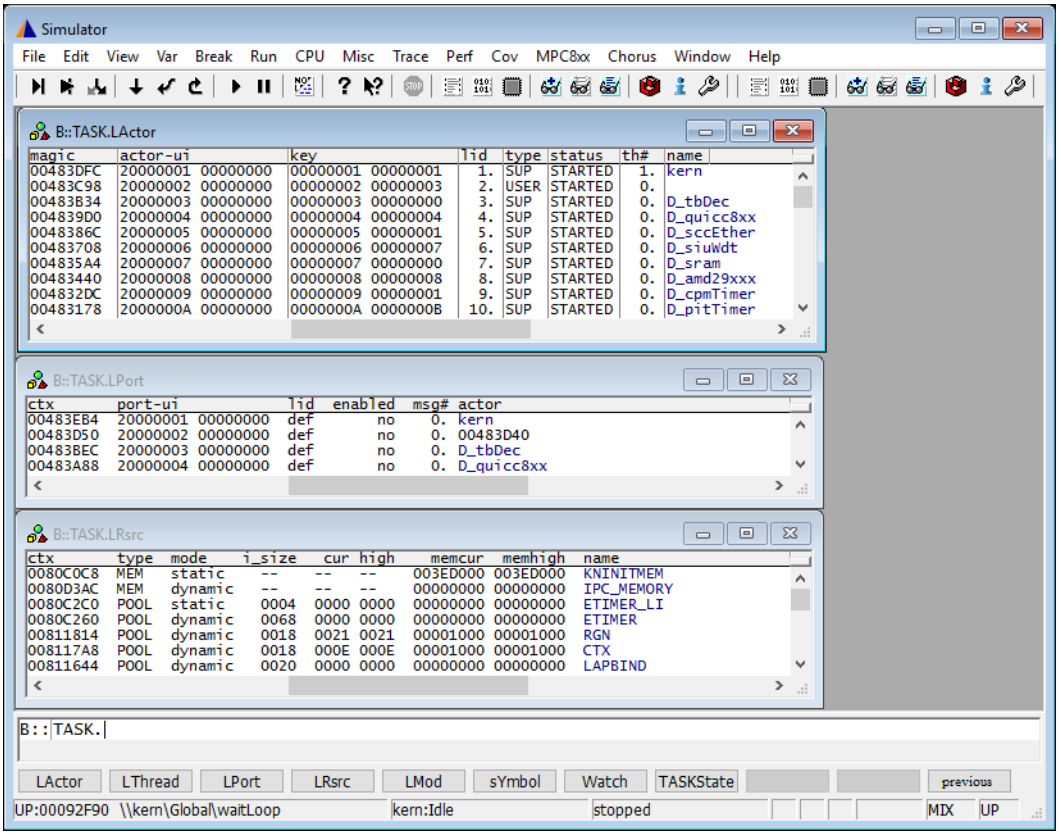
OS Awareness Manual Chorus Classic

Version 18-Nov-2020

History

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

Overview



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

Terminology

ChorusOS uses the terms “Actors” and “Threads”. If not otherwise specified the TRACE32 term “Task” corresponds to ChorusOS threads.

Brief Overview of Documents for New Users

Architecture-independent information:

- **“Debugger Basics - Training”** (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 ChorusOS is supported for the following versions:

- R3.0.2 on 68360
- R3.1b basic and micro on ARM7 (thumb & arm mode)
- R3.1 on 386/486 with PRM memory model
- R3.2 on PowerPC
- R4.0 and 4.0.1 on PowerPC 860 (PRM & FLM)

For the description of micro kernel support see **“OS Awareness Manual Chorus Micro”** (rtos_chorus_micro.pdf).

Configuration

The **TASK.CONFIG** command loads an extension definition file called “chorus.t32” (directory “~~/demo/*<processor>*/kernel/chorus”). It contains all necessary extensions.
Automatic configuration tries to locate the ChorusOS internals 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. This can be done by manual configuration which can require some additional arguments, too.
If you want to have dual port access for the display functions (display “On The Fly”), you have to map emulation memory to the address space of all used system tables.

Manual Configuration

Manual configuration of the OS Awareness for ChorusOS can be used to explicitly define some memory locations. It is recommended to use automatic configuration (except x86).

Format:	**TASK.CONFIG chorus** *<magic_address>* **0** *<args>*

<magic_address>	Specifies a memory location that contains the current running thread. This address can be found at *<cpu_context>*.
<args> (68k/arm/ppc)	0 *<chorus_context>*
<args> (x86)	*<chorus_context>* *<page_directory>*

See **Hooks & Internals** for details on *<cpu_context>* and *<chorus_context>*.

For ChorusOS on x86 add the address of the page directory (can be found in the processor register CR3, after startup).

For examples see the “chorus.cmm” file in the “~~/demo/*<processor>*/kernel/chorus” directory.

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. 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 other arguments:

Format:	**TASK.CONFIG chorus**

It is not recommended to use automatic configuration on x86 systems.

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).

If you want to have dual port access for the display functions (display “On The Fly”), you have to map emulation memory to the address space of all used system tables.

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

Quick Configuration Guide

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

1. Run the demo script (~~/demo/<*processor*>/kernel/chorus/chorus.cmm). Start the demo with “`DO chorus`” and “`Go`”. The result should be a list of threads, which continuously change their state.
2. Make a copy of the PRACTICE script “chorus.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 analyzer functions.

The current running thread is calculated using the cpu context pointer.This pointer can be found on one of the following symbols:
`SUP_SOFT_CTX, CPU_CONTEXT, cpuContext_p, _cpuContext_f, f_cpuContext.`

All system tables are calculated by using the global chorus context pointer. This pointer can be found on one of the following symbols:
`CHORUS_CONTEXT, chorusContext_p, _chorusContext_f, f_chContext.`

The values for these symbols are specified in the file 'src/kern/conf/memMap.h' rsp. 'src/kern/conf/f_map_p.h' of your ChorusOS configuration source.

In x86 protected systems, the OS Awareness must know the physical addresses of the actors. For this calculation, additionally the address of the page directory is used.

Features

The OS Awareness for ChorusOS supports the following features.

KDB Terminal Emulation

The terminal emulation window can be used to communicate with the target resident Chorus kernel debugger, called KDB. The communication via two memory cells requires no external interface. See the **TERM** command group for a description of the terminal emulation. On request we can provide you with the source code for the target interface routine.

NOTE: Only available for 68k and ARM7 systems.

Display of Kernel Resources

The extension defines new commands to display various kernel resources. The following information can be displayed:

TASK.LActor	Actors
TASK.LThread	Threads
TASK.LPort	Ports
TASK.LRsrc	Resources

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

When working with emulation memory or shadow memory, these resources can be displayed “On The Fly”, i.e. while the target application is running, without any intrusion to the application. If using this dual port memory feature, be sure that emulation memory is mapped to all places, where ChorusOS holds its tables.

When working only with target memory, the information will only be displayed if the target application is stopped.

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)".

Task State Analysis

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)".

Function Runtime Statistics

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

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

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

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

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

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

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

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

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

Task Stack Coverage

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

To add/remove one 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.

(This chapter currently only applies to ChorusOS 4.x on PowerPC)

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.

ChorusOS uses different techniques to program the MMU:

- Protected Memory Model (PRM)
- Virtual Memory Model (VM).

Which one is used mainly depends on the processor used.

- PowerPC 860 and derivatives use PRM.
- PowerPC 603e and PowerPCs with the same MMU (like 8260) use VM.

Space IDs

Actors of ChorusOS 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 actors using the kernel address space, the space ID is zero. For actors using their own address space, the space ID is equal to the actor 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.ACTOR.SPACE("<actor>")** returns the space ID for a given actor. If the space ID is not equal to zero, load the symbols of an actor to this space ID:

```
LOCAL &spaceid
&spaceid=task.actor.space("myActor_u")
Data.LOAD.auto myActor_u &spaceid:0 /NoCODE /NoClear
```

Scanning a PRM System and Actors

The command **MMU.SCAN** scans the contents of the current MMU contents. Use the command **TRANSlation.SCANall** to go through all space IDs and scan their MMU settings (ChorusOS calls them “contexts”). **TRANSlation.List** shows the address translation table for all space IDs.

The kernel code, which resides below the variable “space_barrier” (defined in the Embedded WorkShop) can be accessed by any actor, regardless the current space ID. The command **TRANSlation.COMMON** defines those commonly used areas.

Scanning a VM System and Actors

The 603e-type MMU has an address translation that cannot be scanned fully automatically. However, the current used memory areas can be scanned with **MMU.SCAN BAT** and **MMU.SCAN PTE** (ChorusOS uses both, BATs and PTEs).

To scan the address translation of a specific space ID, use the command **TASK.MMU.SCAN "<actor>"**. This command scans the space ID of the specified actor. To scan the kernel space, use:

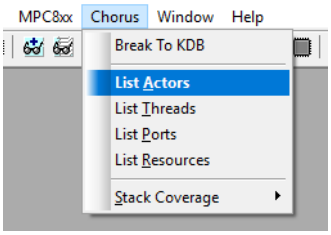
```
TASK.MMU.SCAN "kern"
```

Use **TRANSlation.List** to check the scanned memory areas.

ChorusOS specific Menu

The menu file “chorus.men” contains a menu with ChorusOS specific menu items. Load this menu with the **MENU.ReProgram** command.

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



- The **KDB Terminal** menu item (if available) brings up a terminal emulation window, which communicates with the preconfigured KDB debugger.
- The **List** menu items launch the kernel resource display windows.
- The **Stack Coverage** submenu starts and resets the ChorusOS specific stack coverage and provides an easy way to add or remove threads from the stack coverage window.

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

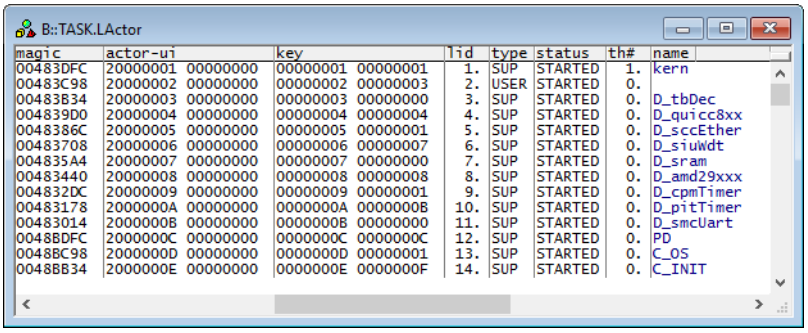
- The **Trace** menu is extended. In the **List** submenu, you can choose if you want a trace list window to show only the task switches (if any) or task switches together with default display.
- The **Perf** menu is prepared to execute thread runtime statistics, thread related function runtime statistics or statistics on the thread states.

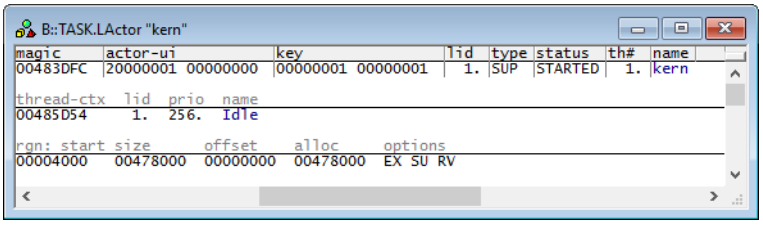
TASK.LActor — List actor table

Format:	**TASK.LActor** *<actor>*

Displays the actor table of ChorusOS or detailed information about one specific actor. The display is similar to the KDB 'l' dump.

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



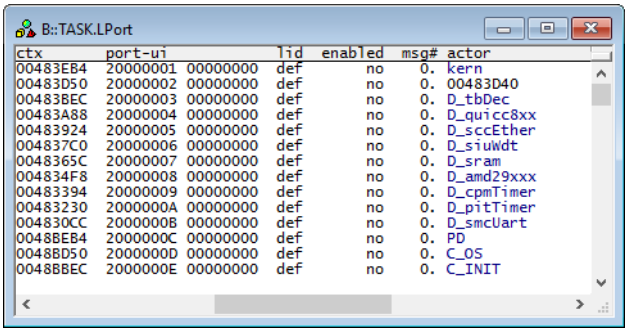


"magic" is a unique ID, used by the OS Awareness to identify a specific actor (address of the actor object).

The fields "magic", "name", "lid" and "thread-ctx" are mouse sensitive, double clicking on them opens appropriate windows.

Format: **TASK.LPort** [*<port>*]

Displays the port table of ChorusOS or detailed information about one specific port.
The display is similar to the KDB '"lp" dump.

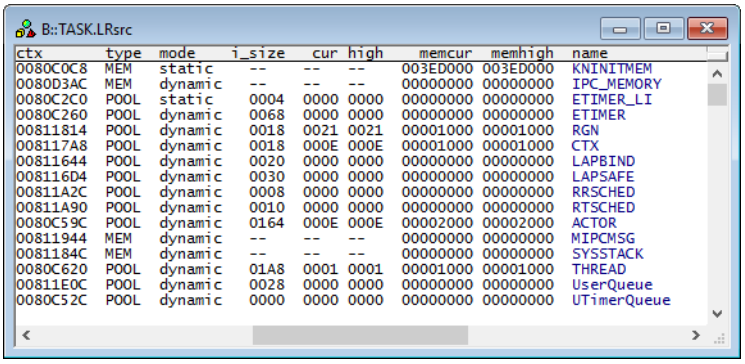


<port> Without any arguments, a table with all created ports will be shown. Specify a port ID or magic number to display detailed information on that port.

The fields “ctx”, “lid” and “actor” are mouse sensitive. Double-clicking on them will open appropriate windows.

Format: **TASK.LRsrc**

Displays a table with all created resources of ChorusOS.
The display is similar to the KDB “lr” dump.

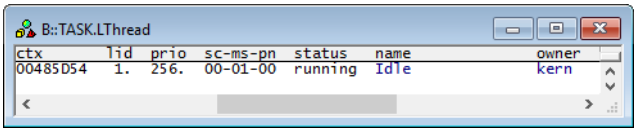


TASK.LThread List thread table

Format: **TASK.LThread** *<thread>*

Displays the thread table of ChorusOS or detailed information about one specific thread.The display is similar to the KDB ‘”lt” dump.

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



“ctx” specifies the value, with which the OS Awareness identifies a specific thread.

The fields “ctx”, “lid”, “name” and “owner” are mouse sensitive. Double-clicking on them opens appropriate windows.

Only available for PowerPC systems using a 603e type MMU.

Format: **TASK.MMU.SCAN** *<actor>*

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

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

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

<actor> Specify an actor name, ID or magic as parameter.

Example:

```
; scan the memory space of the actor helloSync
TASK.MMU.SCAN "helloSync"
```

See also **MMU Support**.

Format: **TASK.MmuSet** *<actor>*

NOTE: Only available for 386/486 systems.

Sets the emulator MMU to cover the physical to logical address conversion of one specific actor.

Specify an actor name, ID or magic as parameter.

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

Example:

```
; set the emulator MMU for actor helloSync
task.mmuset "helloSync"
```

TASK.TASKState Mark task state words

Format: **TASK.TASKState**

This command sets Alpha breakpoints on all thread status words.

The statistic evaluation of thread states (see **Task State Analysis**) requires recording of the accesses to the thread state words. By setting Alpha breakpoints to these words and selectively recording Alpha's, you can do a selective recording of thread state transitions.

Because setting the Alpha breakpoints by hand is very hard to do, this utility command automatically sets the Alpha's to the status words of all threads currently created. It does NOT set breakpoints to threads that terminated or haven't yet been created.

ChorusOS PRACTICE Functions

There are special definitions for ChorusOS specific PRACTICE functions.

TASK.ACTOR.SPACE() Space ID of actor

Syntax: **TASK.ACTOR.SPACE("***<actor_name>***")**

Returns the debugger MMU space ID of the specified actor.

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

Return Value Type: Hex value.

TASK.ACTOR.START() Start address of specified region

Syntax: **TASK.ACTOR.START("***<actor_name>***", ***<region>***)**

Returns the start address of the specified region owned by the specified actor.

Parameter and Description:

<actor_name>	**Parameter Type**: String (***without*** quotation marks).
<region>	**Parameter Type**: Decimal or hex or binary value. • **1** usually is the code region. • **2** usually is the data region.

Return Value Type: Hex value.

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

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).
kernel	**Parameter Type**: String (***without*** quotation marks). Returns the address of the kernel state variable.

Return Value Type: Hex value.

Frequently-Asked Questions

No information available