User's Guide

HP B3084A
Real-Time OS Measurement Tool for VxWorks

Notice

Hewlett-Packard
P.O. Box 2197
1900 Garden of the Gods Road
Colorado Springs, CO 80901-2197, U.S.A.

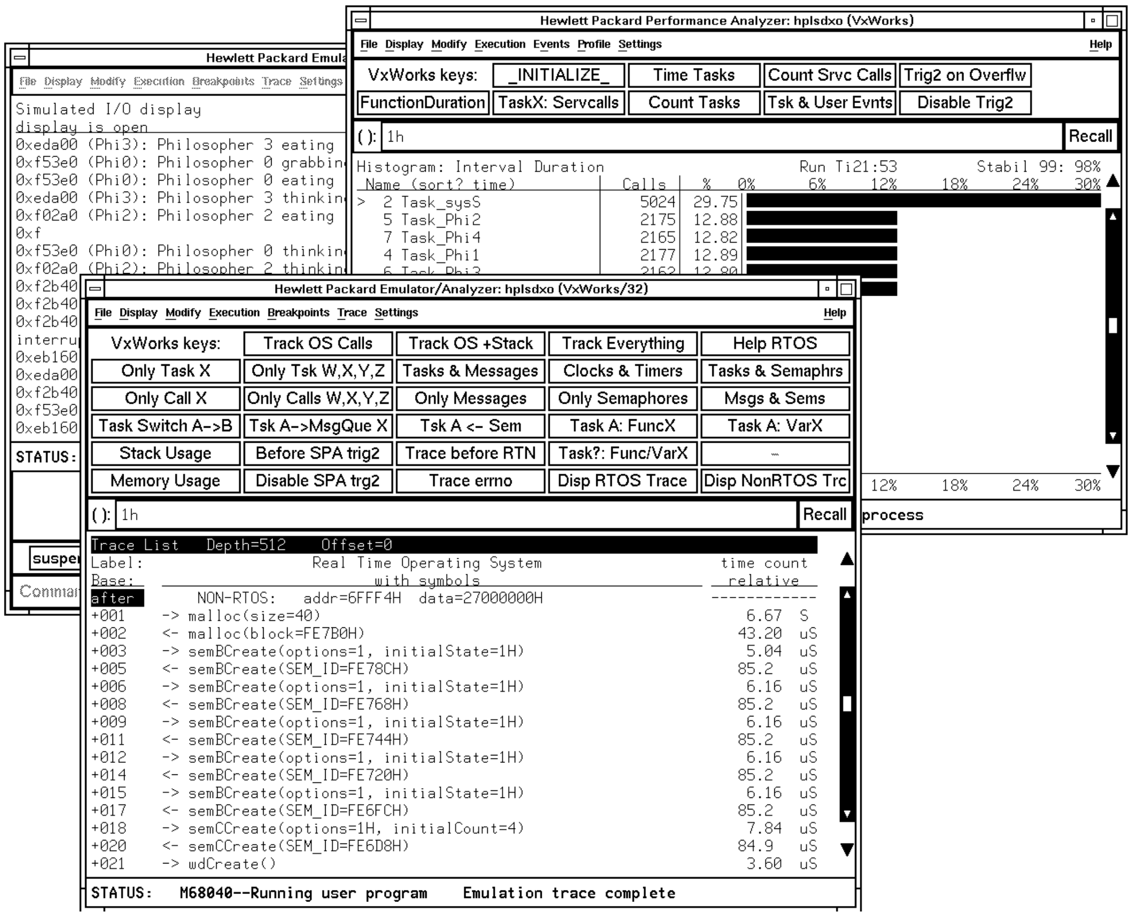
Printing History

New editions are complete revisions of the manual. The date on the title page changes only when a new edition is published.

Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition 1 B3084-97000, September 1995

Measurements for the VxWorks Real-Time Operating System



```
Simulated I/O display
display is open
0xeda00 (Phi3): Philosopher 3 eating
0xf53e0 (Phi0): Philosopher 0 grabbin
0xf53e0 (Phi0): Philosopher 0 eating
0xeda00 (Phi3): Philosopher 3 thinkin
0xf02a0 (Phi2): Philosopher 2 eating
0xf
0xf53e0 (Phi0): Philosopher 0 thinkin
0xf02a0 (Phi2): Philosopher 2 thinkin
0xf2b40
0xf2b40
0xf2b40
interru
0xeb160
0xeda00
0xf2b40
0xf53e0
0xeb160
```

```
Histogram: Interval Duration                   Run Ti21:53        Stabil 99: 98%
 Name (sort? time)          Calls    %   0%      6%     12%     18%     24%    30%
>  2 Task_sysS               5024  29.75
   5 Task_Phi2               2175  12.88
   7 Task_Phi4               2165  12.82
   4 Task_Phi1               2177  12.89
                                                        12%     18%     24%    30%
```

```
Trace List    Depth=512    Offset=0
Label:                  Real Time Operating System                    time count
Base:                          with symbols                            relative
after      NON-RTOS:   addr=6FFF4H  data=27000000H                  ------------
+001     -> malloc(size=40)                                             6.67  S
+002     <- malloc(block=FE7B0H)                                       43.20  uS
+003     -> semBCreate(options=1, initialState=1H)                      5.04  uS
+005     <- semBCreate(SEM_ID=FE78CH)                                  85.2   uS
+006     -> semBCreate(options=1, initialState=1H)                      6.16  uS
+008     <- semBCreate(SEM_ID=FE768H)                                  85.2   uS
+009     -> semBCreate(options=1, initialState=1H)                      6.16  uS
+011     <- semBCreate(SEM_ID=FE744H)                                  85.2   uS
+012     -> semBCreate(options=1, initialState=1H)                      6.16  uS
+014     <- semBCreate(SEM_ID=FE720H)                                  85.2   uS
+015     -> semBCreate(options=1, initialState=1H)                      6.16  uS
+017     <- semBCreate(SEM_ID=FE6FCH)                                  85.2   uS
+018     -> semCCreate(options=1H, initialCount=4)                      7.84  uS
+020     <- semCCreate(SEM_ID=FE6D8H)                                  84.9   uS
+021     -> wdCreate()                                                  3.60  uS
STATUS:   M68040--Running user program    Emulation trace complete
```

The HP B3084A Real-Time Operating System Measurement Tool for VxWorks supports the Wind River VxWorks operating system for Motorola processors including the 68360 and 68040.

The RTOS Measurement Tool is a collection of files that are used with your real-time OS application and the HP 64700 emulation/analysis system to view program execution in the context of the real-time OS. For example, you can view tooled calls and their parameters, task switches, clock ticks, and dynamic memory usage.

By linking your real-time OS application with an "instrumented" service call library (an interface library with instructions that write to a data table), you can capture writes to the data table with the HP 64700 emulation bus analyzer. A special inverse assembler decodes the captured information and displays it in an easy-to-read format. You can also use the software performance analyzer to measure time taken by tasks.

Command files are provided for common RTOS measurements, and you can run them by clicking on action keys. You can also create custom command files and action keys for your own RTOS measurements.

With an Emulation Bus Analyzer, You Can ...

- View problems at the task level.
- Use one button point-and-click commands (or run command files in the command line).
- Display the real-time OS trace with the native service call mnemonics of your OS.
- Track tooled OS service calls and display entry parameters and return values.
- Capture task switches caused by OS service calls or system clock ticks.
- Understand how interrupts are affecting your high level task flow.
- Stop program execution if any OS service call ever fails.
- Identify which tasks access a shared function or variable.
- Trigger when a certain message is sent to a specified mailbox.
- Capture activity after task A switches into task B in sequence.
- Detect attempts to free invalid memory segments.
- Display location of local stacks.
- Track all dynamic memory allocation and freeing.
- Trigger on stack overflow.

With the Software Performance Analyzer, You Can ...

- Perform time profiling of task durations in your application.
- Measure time spent in OS kernel versus application tasks.
- Measure the percentage of time spent in each application task.
- Stop program execution if a task exceeds a maximum time.
- Find out how often each OS service call is invoked.

In This Book

This book describes the HP B3084A Real-Time Operating System Measurement Tool for the VxWorks operating system.

This book assumes you are familiar with the Emulator/Analyzer graphical interface.

This book is organized into three parts:

Part 1. User's Guide

Part 2. Concept Guide

Part 3. Installation Guide

Contents

Part 1

User's Guide

How-to instructions and problem-solving guidelines.

Part 1

1

Preparing Your Application for RTOS Measurements

Preparing Your Application for RTOS Measurements

Requirements

Before preparing your application for RTOS measurements, you should have already:

- Installed the emulator, emulation bus analyzer, and Graphical User Interface as described in their *User's Guide* manuals. The emulator/analyzer interface software must be version C.05.20 or greater. Note that if you have installed another Graphical User Interface after you installed the HP B3084A Real-Time Operating System Measurement Tool, you must re-run the HP B3084A "customize" script.
- Installed the HP B3084A Real-Time Operating System Measurement Tool as outlined in the "Installation" chapter of this manual.

If you wish to make profile measurements on RTOS tasks and service calls, you should have already:

- Installed the HP 64708A Software Performance Analyzer and its interface software (HP B1487) as described in the *Software Performance Analyzer User's Guide*.

It's helpful if you are already familiar with your emulator, the software performance analyzer, and their interfaces before preparing your multi-tasking application for real-time operating system measurements. It's best if you have already loaded and run the application under the emulator.

With the emulator/analyzer interface already running, you should see two new entries under the **File→Emul700** pulldown menu: **VxWorks Emulator/Analyzer ...** and **VxWorks Performance Analyzer ...**. If you do not see these new entries, review the installation procedure to make sure it was done correctly, and make sure the /system/B3084A/customize script was run. If you still do not see these new entries, contact your Hewlett-Packard representative.

VxWORKS Versions

This product is compatible with VxWORKS versions 5.1 and 5.2.

Task list control file

Both the emulator/analyzer interface and the Software Performance Analyzer need to know the names of the tasks in your application. The emulator/analyzer looks for the task names in the file "tables.c". The Software Performance Analyzer looks in your "s_init" file.

A script, called **rtos_edit_vxworks**, has been provided to help you create the "tables.c" and "s_init" files. The first time you run the script, it will save the names of the tasks in a *task list control file*. As you make changes to your application, keep the task list control file up-to-date and re-run the **rtos_edit_vxworks** script so that the Real-Time Operating System Measurement Tool can track all of the application's tasks.

Preparing your application for RTOS measurements

To prepare your application for real-time operating system measurements with the emulation bus analyzer and the software performance analyzer, take the following steps:

1 Make a new source directory.
2 Retrieve the RTOS measurement source files.
3 Create the task table.
4 Create the Software Performance Analyzer initialization file.
5 Add the RTOS measurement files to your application.
6 Build the new application file.
7 Start the emulator interface.
8 Configure the emulator and load the application.
9 Test the RTOS measurement tool.
10 Test the Software Performance Analyzer.

The remainder of this chapter describes these steps in detail.

Step 1: Make a new source directory

- **Make a new directory, for example ".../hprtos_src", to hold the instrumented code which needs to be linked to your existing application.**

Create the directory somewhere convenient for linking its files to your application.

Step 2: Retrieve the RTOS source files

If you have already installed the RTOS Measurement Tool, source files will be found under the $HP64000/rtos/B3084A directory. If you haven't installed the product, refer to the "Installation" chapter.

During installation, you should have set the environment variable HP64000 to the directory in which the HP 64000 software has been installed.

1 Copy the product files into the directory that was created in Step 1.

The files are found under $HP64000/rtos/B3084A.

You must copy the following files:

track_os.c	(instrumented service call data)
track_il.c	(instrumented C service call routines)
callout.c	(task hook routines)
HPIL.h	(function override #defines)

Step 3: Create the task table

To create the task table and the Software Performance Analyzer initialization file, you will need a *task list control file*. The "rtos_edit_vxworks" script will create this file for you when you use the "-i" (initialize) option.

- **If you have not prepared a task list control file, run the $HP64000/bin/rtos_edit_vxworks script. Type:**

```
rtos_edit_vxworks -i -tables <task_name_file>
```

where <*task_name_file*> is the name of the task list control file.

The "rtos_edit_vxworks" script asks you for the names of the tasks in your application.

The script creates your application specific "tables.c" file. This file contains information that customizes the RTOS tool for your application. This file will be compiled and linked in with your application code. Tables.c allows a "bucket" to be created in memory for each task entry you define. Information is written to the buckets when task switches occur.

The "rtos_edit_vxworks" script may be run any time you wish to add or delete task name information.

If a task list control file does not exist, running "rtos_edit_vxworks -i" will create a task list control file. If the file already exists, it will not be modified.

You can edit the task list control file to add or delete tasks. You can use any text editor, such as **vi** or **emacs**, to edit the file. If you make any changes, be sure to run the "rtos_edit_vxworks" script to create a new task table and Software Performance Analyzer initialization file.

Although "rtos_edit_vxworks" may be run interactively, it is recommended that you prepare a task list control file.

The task list control file consists of three parts:

- The keyword TASK_FILE.
- The keyword BUS:, followed on the next line by the bus size of your processor.
- The keyword TASKS:, followed by a list of all of the task names.

This is a good time to look at your task names. The HP Real-Time Operating System Measurement Tool uses integer (32-bit) task names, corresponding to the first four characters of the name assigned as the task was spawned. Make sure the tasks you want to track are unique in the first four characters of their names.

Example

Here is an example of a task list control file:

```
TASK_FILE
BUS:
32
TASKS:
sysS
tExc
tP1
tP2
tP3
(EOF)
```

See Also

Page 27 for instructions on how to add the "rtos_edit_vxworks" script to your makefile.

Step 4: Create the Software Performance Analyzer initialization file

1 Create the "s_init" file. Type:

```
rtos_edit_vxworks -s_init <task_name_file>
```

where *<task_name_file>* is the name of the task list control file.

The "s_init" file will be created in your home directory as "~/.rtos/vxworks/s_init". This is a command file that customizes the Software Performance Analyzer system to your application.

Note that each user has a separate "s_init" file. This allows individual users to track different sets of functions and tasks, if they wish.

The contents of any existing s_init file will be lost. If you have several task list control files, you may want to make a copy of the s_init file before using rtos_edit_vxworks with a new task list control file. In this case, be careful that the correct s_init file is installed before you start an emulator interface.

Step 5: Add the RTOS measurement files to your application

To track task creation, switching, and deletion, the capabilities of the VxWorks taskHookLib must be initialized to point to the RTOS callout routines found in "callout.c".

1 Add the following code to your application:

```
#include "taskHookLib.h"

taskHookInit();
taskCreateHookAdd((FUNCPTR) HP_createTask);
taskSwitchHookAdd((FUNCPTR) HP_switchTask);
taskDeleteHookAdd((FUNCPTR) HP_deleteTask);
```

2 Add "track_os.c", "track_il.c", "callout.c" and "tables.c" to your "makefile" and "linker" files.

Note "Track_il.c" must be compiled without optimization.

It is recommended that "track_os.c", "callout.c" and "tables.c" also be compiled without optimization to avoid unknown side effects.

The large data table that resides in "track_os.c" and spans from the symbol HP_RTOS_TRACK_START through HP_RTOS_TRACK_END only needs to be in an address range that is writable. The data table is never read from and needs no real memory.

3 **Add the header file "HPIL.h" to every .c source file that contains VxWorks service calls. This header file will redirect the VxWorks service routine to a wrapper routine that will provide the tracking measurements.**

"Track_os.c" contains code that allows a user to call the VxWorks OS service call routines. This file also contains special code that writes out RTOS information to the analyzer any time an OS service call is invoked.

Step 6: Build the new application file

- **Rebuild your application with the new files.**

  The service routines in "track_il.c" have been defined according to the VxWorks standard so your application should require no other changes.

Step 7: Start the emulator interface

- **Start the RTOS emulation window using the "emulrtos_vxworks" command:**

```
emulrtos_vxworks [-quiet] [-xrm <resource_string>]
   [-c <command_file>] [-p <PROCESSOR>] [8|16|32]
   <emulator_name> &
```

This is a script which sets up a few things before calling **emul700** with your given emulator name. The command and the options you choose should all be entered on one line.

The "emulrtos_vxworks" script does the following before calling **emul700** with your given emulator name:

1. Sets HP64000 if it is not already set.
2. Sets HP64RTOSIAL based on the determined bus width.
3. Defines the environment variable HP64KPATH so the command files related to the action keys are found.
4. Defines the PATH variable so shell scripts needed by command files will be found. If you have used the **emul700** command to start the emulator/analyzer interface, you can choose the **File→Emul700→VxWORKS RTOS Measurement Tool** pulldown menu item to open the RTOS emulation window. This will work only if the $HP64RTOSIAL environment variable has been set. If you need to find out how to set the $HP64RTOSIAL variable, examine the "emulrtos_vxworks" script.

Step 8: Configure the emulator and load the application

- **Now, load an emulator configuration and your application program into the emulator.**

A few notes on the configuration:

1 You may set the emulator to be restricted to real-time runs. The RTOS measurements are done without breaking into the emulation monitor.

2 You may use either a foreground or background monitor.

You are now ready to test your application.

See Also

The *Emulator/Analyzer User's Guide* for information about loading configuration files and application programs.

Step 9: Test the RTOS measurement tool

1 Click the **Track OS calls** action key.

2 Start your application running from its start address (assuming the start address has initialization code and starts your root task).

You should now see a trace display of your root task setting up application tasks and performing any other initializations.

If you page down the display, you will see all of the root task's OS activity and possibly the start of your application's tasks.

3 Click the **Track OS calls** action key again to see a "running snapshot" of what your application is currently doing.

The action keys for RTOS measurements are described in the "Making RTOS Measurements with the Emulator/Analyzer" chapter.

Step 10: Test the Software Performance Analyzer

If your HP 64700 emulation system includes a Software Performance Analyzer, you can test it by performing the following steps.

1 Bring up SPA window by choosing the **File→Emul700→VxWorks Performance Analyzer** pulldown menu item.

2 If you wish to make cross-trigger measurements between SPA and the emulation system, make sure the emulation configuration for "Should Analyzer drive or receive Trig2?" is set to "Receive".

To do this, choose **Modify→Emulator Config...**. Choose **Interactive Measurement Specification**. For **Analyzer on Trig2**, select **Receive**.

3 In Step 4, when you ran the "rtos_edit_vxworks" script, a command file "s_init" should also have been created. If not, rerun "rtos_edit_vxworks".

4 Click the **Initialize** action key in SPA to define the events that correspond to each task. This uses the command file "s_init" that you just created.

5 Click the **Time Tasks** action key to see a dynamic histogram of the currently running tasks.

If your application isn't running, start it running from the emulation window either before or after the action key is pressed.

If you have multiple projects on one machine, you'll need to set up unique SPA windows for each project. For more information, refer to the "Handling Multiple Projects on One Machine" section of the "Making RTOS Measurements with the SPA" chapter.

See Also

Refer to your emulator/analyzer *User's Guide* for information on modifying the emulator configuration.

Suggestions for Easier Software Development

- Add rtos_edit_vxworks to your makefile.
- Use the sample configuration tables.

To add rtos_edit_vxworks to your makefile

The "rtos_edit_vxworks" script must be run every time you add or delete a task. To simplify this process, you can add rtos_edit_vxworks to your makefile.

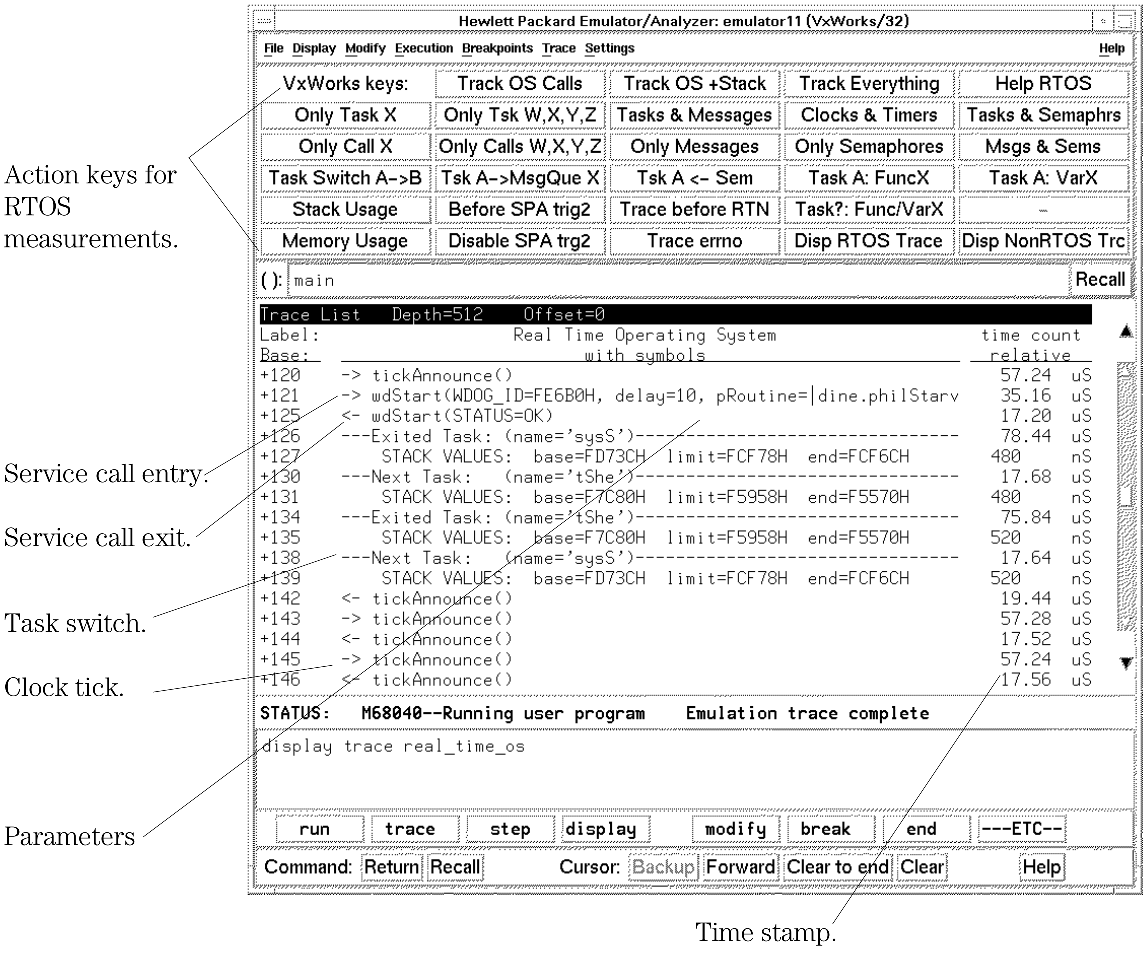
- **Add the following dependencies to your makefile:**

```
~/.rtos/vxworks/s_init: <task_file_name>
        rtos_edit_vxworks -s_init <task_file_name>
tables.c: <task_file_name>
        rtos_edit_vxworks -tables <task_file_name>
```

2

Making RTOS Measurements with the Emulator/Analyzer

Making RTOS Measurements with the Emulator/Analyzer



```
Trace List   Depth=512    Offset=0
Label:                 Real Time Operating System                  time count
Base:                          with symbols                         relative
+120    -> tickAnnounce()                                            57.24  uS
+121    -> wdStart(WDOG_ID=FE6B0H, delay=10, pRoutine=|dine.philStarv  35.16  uS
+125    <- wdStart(STATUS=OK)                                        17.20  uS
+126    ---Exited Task: (name='sysS')--------------------------------  78.44  uS
+127       STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       480    nS
+130    ---Next Task:   (name='tShe')--------------------------------  17.68  uS
+131       STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       480    nS
+134    ---Exited Task: (name='tShe')--------------------------------  75.84  uS
+135       STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       520    nS
+138    ---Next Task:   (name='sysS')--------------------------------  17.64  uS
+139       STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       520    nS
+142    <- tickAnnounce()                                            19.44  uS
+143    -> tickAnnounce()                                            57.28  uS
+144    <- tickAnnounce()                                            17.52  uS
+145    -> tickAnnounce()                                            57.24  uS
+146    <- tickAnnounce()                                            17.56  uS

STATUS:   M68040--Running user program    Emulation trace complete

display trace real_time_os
```

RTOS measurements are easy to set up and use. To set up a measurement you simply point and click on the appropriate action key (which runs a command file), and the setup is done automatically. If parameters are required, you are prompted for them. In the graphical interface, these prompts appear as dialog boxes in which you can either type or cut-and-paste the required parameters.

You can modify the provided command files and set up action keys for your own RTOS measurements (refer to the "Creating Your Own RTOS Measurements" chapter for more information).

Interpreting the measurement output is also very easy. VxWorks service calls are displayed just as they appear in the VxWorks manual. Macro definitions, however, are not decoded.

Real-time OS measurements in the emulator/analyzer interface are made using the HP 64700 series emulation bus analyzers. The analyzer traces real-time OS activity such as service calls, task switches, and dynamic memory usage.

Each state stored in the trace has a time stamp that shows relative or absolute time. This is useful for verifying the system clock tick interval, measuring non-running time of tasks, and understanding the timing needs of various communications mechanisms such as sending a message or responding to a semaphore.

Groups of action keys

The RTOS Measurement Tool comes with a default set of measurements that appear as action keys and are grouped into the following sections:

- Tracking the flow of OS activity.
- Tracking particular OS service calls.
- Tracking particular tasks.
- Tracking accesses to functions or variables.
- Tracking dynamic memory usage.
- Displaying traces.

Additional measurements exist as command files and can be put on action keys or run directly from the command line. A complete list of these measurements can be found in the files $HP64000/rtos/B3084A/CMDLIST16 or CMDLIST32 (depending on whether a 16- or 32-bit processor is being used).

Tracking the Flow of OS Activity

The HP 64700 series emulation bus analyzer can measure the real-time task flow that is occurring in your system. As your application calls into the real-time OS kernel through OS service calls, the emulation bus analyzer captures the activity including the value of input and output parameters and the return values. If the OS switches context into another task, the analyzer can also capture this information. One simple measurement monitors the service call return values while tracking OS activity and stops if a user-definable value is detected; this helps designers guard against unchecked return values.

Tooled calls

Not all VxWorks service calls can be tracked. Due to the large number of calls available in VxWorks, not all of them could be "tooled."

The calls which are tooled and can be tracked are:

taskSpawn
taskInit
taskDelete
taskDelay
taskPrioritySet
taskSuspend
taskResume
taskSafe
taskUnsafe
taskLock
taskUnlock
semBCreate
semCCreate
semMCreate
semDelete
semTake
semGive
semFlush
semCSmCreate
semBSmCreate
tickAnnounce[1]
wdCreate

wdDelete
wdStart
wdCancel
msgQCreate
msgQDelete
msgQReceive
msgQSend
msgQSmCreate
signal
sigsuspend[2]
pause[2]
kill
memalign
calloc
cfree
memPartCreate
memPartAlignedAlloc
memPartAlloc
memPartFree
malloc
free

[1] tickAnnounce is not tracked when you use the **Track OS Calls** action key because, in general, the number of occurrences would dominate the RTOS display. However, it tooled, and can be tracked with the action keys **Track Everything** and **Clocks and Timers**.

[2] These calls do not return from the kernel to the caller until the calling task has been restarted. Therefore the return may not show up on the RTOS display.

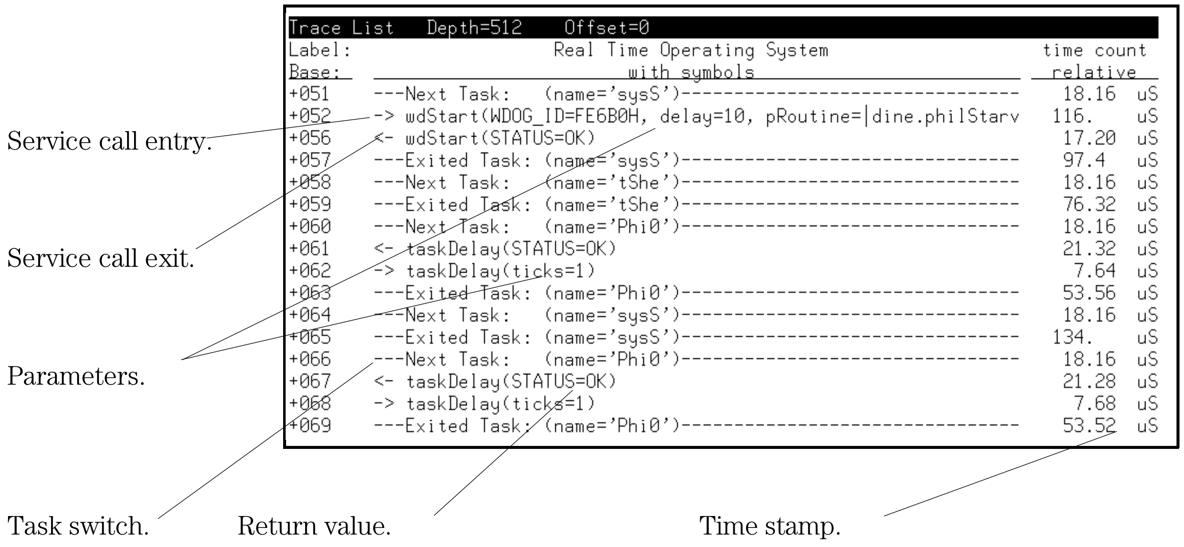
This section shows you how to:

- Track tooled calls (including device calls).
- Track tooled calls plus the stack activity.
- Track tooled calls before an error occurs.
- To track all tooled calls before a return value occurs
- Track everything.

To track all tooled calls

- Click on the **Track OS Calls** action key (or run the **e_trkcalls** command file by entering it on the command line).

This command takes a trace of all OS service calls and task switches.



```
Trace List   Depth=512    Offset=0
Label:                 Real Time Operating System                      time count
Base:                         with symbols                              relative
+051    ---Next Task:   (name='sysS')--------------------------------   18.16  uS
+052    -> wdStart(WDOG_ID=FE6B0H, delay=10, pRoutine=|dine.philStarv  116.    uS
+056    <- wdStart(STATUS=OK)                                           17.20  uS
+057    ---Exited Task: (name='sysS')--------------------------------   97.4   uS
+058    ---Next Task:   (name='tShe')--------------------------------   18.16  uS
+059    ---Exited Task: (name='tShe')--------------------------------   76.32  uS
+060    ---Next Task:   (name='Phi0')--------------------------------   18.16  uS
+061    <- taskDelay(STATUS=OK)                                         21.32  uS
+062    -> taskDelay(ticks=1)                                            7.64  uS
+063    ---Exited Task: (name='Phi0')--------------------------------   53.56  uS
+064    ---Next Task:   (name='sysS')--------------------------------   18.16  uS
+065    ---Exited Task: (name='sysS')--------------------------------  134.    uS
+066    ---Next Task:   (name='Phi0')--------------------------------   18.16  uS
+067    <- taskDelay(STATUS=OK)                                         21.28  uS
+068    -> taskDelay(ticks=1)                                            7.68  uS
+069    ---Exited Task: (name='Phi0')--------------------------------   53.52  uS
```

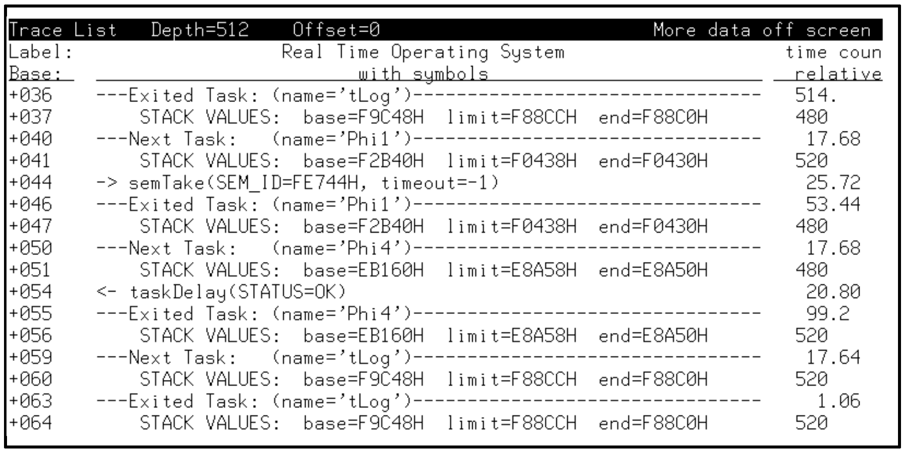
Note that there are entry and exit arrows on the left of the screen to show when a tooled call is entered and, on a separate line, to show when a tooled call is exited. This is important since an OS service call may switch to another task while in the OS and *not* return to the calling service call for a long time, if ever.

When appropriate, the trace information is decoded. The OS service calls are decoded into the same mnemonics that appear in the OS manual. The parameters and return values that are associated with service calls are displayed. The parameter variable names also appear as they do in the OS manual decoded into their English mnemonics. Some of the parameter values and all return values are also decoded whenever appropriate.

Calls to tickAnnounce() are not tracked using this action key. To track clock ticks, use the **Clocks & Timers** or **Track Everything** action key.

To track all tooled calls plus the stack activity

- Click on the **Track OS +Stack** action key (or run the **e_trk_stack** command file by entering it on the command line).



```
Trace List   Depth=512    Offset=0                           More data off screen
Label:                  Real Time Operating System                       time coun
Base:                          with symbols                               relative
+036    ---Exited Task: (name='tLog')--------------------------------    514.
+037        STACK VALUES:  base=F9C48H  limit=F88CCH  end=F88C0H         480
+040    ---Next Task:   (name='Phi1')--------------------------------     17.68
+041        STACK VALUES:  base=F2B40H  limit=F0438H  end=F0430H         520
+044    -> semTake(SEM_ID=FE744H, timeout=-1)                             25.72
+046    ---Exited Task: (name='Phi1')--------------------------------     53.44
+047        STACK VALUES:  base=F2B40H  limit=F0438H  end=F0430H         480
+050    ---Next Task:   (name='Phi4')--------------------------------     17.68
+051        STACK VALUES:  base=EB160H  limit=E8A58H  end=E8A50H         480
+054    <- taskDelay(STATUS=OK)                                           20.80
+055    ---Exited Task: (name='Phi4')--------------------------------     99.2
+056        STACK VALUES:  base=EB160H  limit=E8A58H  end=E8A50H         520
+059    ---Next Task:   (name='tLog')--------------------------------     17.64
+060        STACK VALUES:  base=F9C48H  limit=F88CCH  end=F88C0H         520
+063    ---Exited Task: (name='tLog')--------------------------------      1.06
+064        STACK VALUES:  base=F9C48H  limit=F88CCH  end=F88C0H         520
```

This measurement is useful not only if you want to see the stack usage as you enter and exit tasks but also if you want to see what service calls may have changed the stack usage. It will give you all service call activity for tooled calls, plus show you when the task switches occur.

For more information on stack activity measurements, see the "Tracking Dynamic Memory Usage" section that follows.

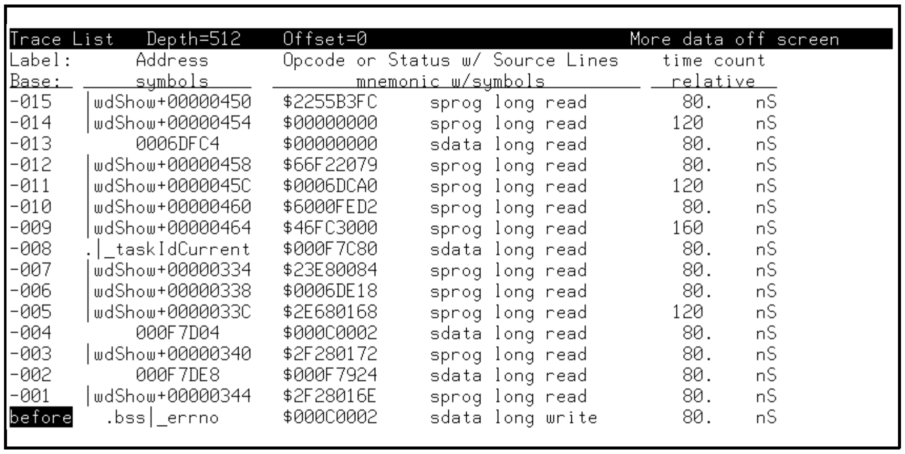
To track all tooled calls before an error occurs

- Click on the **Trace errno** action key (or run the **e_errno** command file by entering it on the command line).

This command lets you use the analyzer to continuously monitor the global error value, errno, and check for any non-zero writes, even if the target code is not checking that value.

When the trace completes, you can see the activity that occurred before the error.

Note: The trace may be modified to break emulator execution on any error occurrence by adding "break_on_trigger" to the end of the trace specification either on the command line or in the command file.



```
Trace List   Depth=512    Offset=0                             More data off screen
Label:       Address      Opcode or Status w/ Source Lines    time count
Base:        symbols            mnemonic w/symbols              relative
-015   |wdShow+00000450  $2255B3FC     sprog long read         80.    nS
-014   |wdShow+00000454  $00000000     sprog long read        120     nS
-013         0006DFC4    $00000000     sdata long read         80.    nS
-012   |wdShow+00000458  $66F22079     sprog long read         80.    nS
-011   |wdShow+0000045C  $0006DCA0     sprog long read        120     nS
-010   |wdShow+00000460  $6000FED2     sprog long read         80.    nS
-009   |wdShow+00000464  $46FC3000     sprog long read        160     nS
-008  .|_taskIdCurrent   $000F7C80     sdata long read         80.    nS
-007   |wdShow+00000334  $23E80084     sprog long read         80.    nS
-006   |wdShow+00000338  $0006DE18     sprog long read         80.    nS
-005   |wdShow+0000033C  $2E680168     sprog long read        120     nS
-004         000F7D04    $000C0002     sdata long read         80.    nS
-003   |wdShow+00000340  $2F280172     sprog long read         80.    nS
-002         000F7DE8    $000F7924     sdata long read         80.    nS
-001   |wdShow+00000344  $2F28016E     sprog long read         80.    nS
before    .bss|_errno    $000C0002     sdata long write        80.    nS
```

To track all tooled calls before a return value occurs

1 Click on the **Trace before RTN** action key (or run the **e_before_return** command file by entering it on the command line).

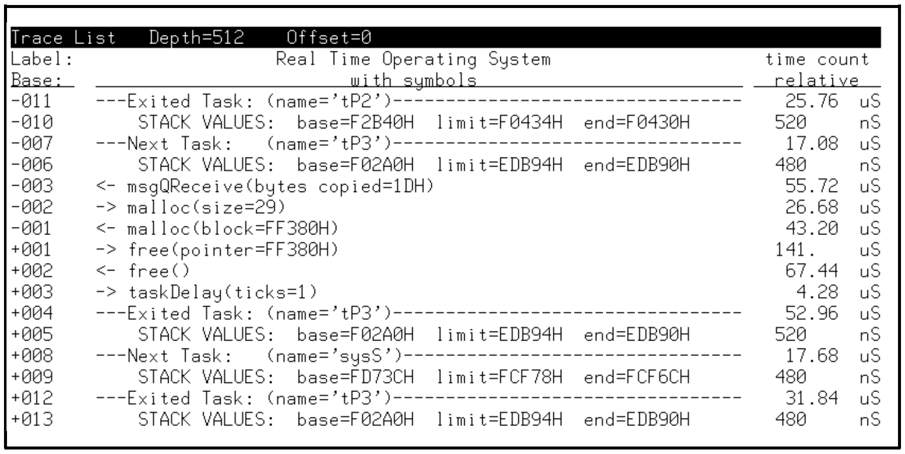
2 In the dialog box, enter the return value you are looking for, then click OK.

One common problem for software developers is the habit of not checking return values from system service calls that "should" never fail. Unfortunately, when one does fail, it can be very difficult to locate.

This command lets you use the analyzer to continuously monitor the system and check if any tooled service call returns a specific value.

When the trace completes, you can see the activity that occurred before the return, and the return value itself.

Note: The trace may be modified to break emulator execution on any error occurrence by adding "break_on_trigger" to the end of the trace specification either on the command line or in the command file.

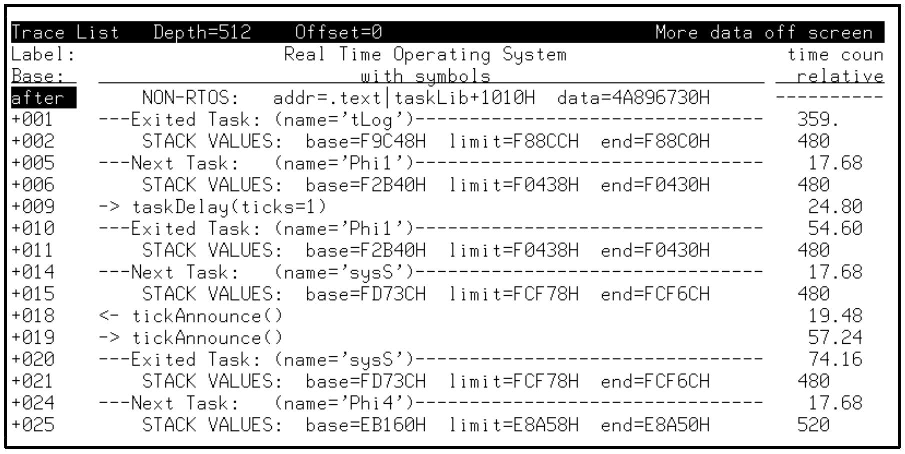


```
Trace List   Depth=512    Offset=0
Label:                   Real Time Operating System                     time count
Base:                          with symbols                              relative
-011    ---Exited Task: (name='tP2')---------------------------------    25.76  uS
-010        STACK VALUES:  base=F2B40H  limit=F0434H  end=F0430H        520     nS
-007    ---Next Task:   (name='tP3')---------------------------------    17.08  uS
-006        STACK VALUES:  base=F02A0H  limit=EDB94H  end=EDB90H        480     nS
-003    <- msgQReceive(bytes copied=1DH)                                 55.72  uS
-002    -> malloc(size=29)                                               26.68  uS
-001    <- malloc(block=FF380H)                                          43.20  uS
+001    -> free(pointer=FF380H)                                         141.    uS
+002    <- free()                                                        67.44  uS
+003    -> taskDelay(ticks=1)                                             4.28  uS
+004    ---Exited Task: (name='tP3')---------------------------------    52.96  uS
+005        STACK VALUES:  base=F02A0H  limit=EDB94H  end=EDB90H        520     nS
+008    ---Next Task:   (name='sysS')--------------------------------    17.68  uS
+009        STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH        480     nS
+012    ---Exited Task: (name='tP3')---------------------------------    31.84  uS
+013        STACK VALUES:  base=F02A0H  limit=EDB94H  end=EDB90H        480     nS
```

In this example, a return value of FF380H was specified. A call to malloc() was found which returned this value.

To track everything

- Click on the **Track Everything** action key (or run the **e_trkall** command file by entering it on the command line).



```
Trace List   Depth=512    Offset=0                         More data off screen
Label:                  Real Time Operating System                  time coun
Base:                          with symbols                          relative
after        NON-RTOS:   addr=.text|taskLib+1010H  data=4A896730H    ----------
+001     ---Exited Task: (name='tLog')--------------------------------   359.
+002         STACK VALUES:  base=F9C48H  limit=F88CCH  end=F88C0H       480
+005     ---Next Task:   (name='Phi1')--------------------------------    17.68
+006         STACK VALUES:  base=F2B40H  limit=F0438H  end=F0430H       480
+009     -> taskDelay(ticks=1)                                            24.80
+010     ---Exited Task: (name='Phi1')--------------------------------    54.60
+011         STACK VALUES:  base=F2B40H  limit=F0438H  end=F0430H       480
+014     ---Next Task:   (name='sysS')--------------------------------    17.68
+015         STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       480
+018     <- tickAnnounce()                                                19.48
+019     -> tickAnnounce()                                                57.24
+020     ---Exited Task: (name='sysS')--------------------------------    74.16
+021         STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       480
+024     ---Next Task:   (name='Phi4')--------------------------------    17.68
+025         STACK VALUES:  base=EB160H  limit=E8A58H  end=E8A50H       520
```

This action key is used so that tooled calls, task switches, clock ticks, stack activity, and user-defined events are all tracked and displayed in the trace.

Tracking Particular OS Service Calls

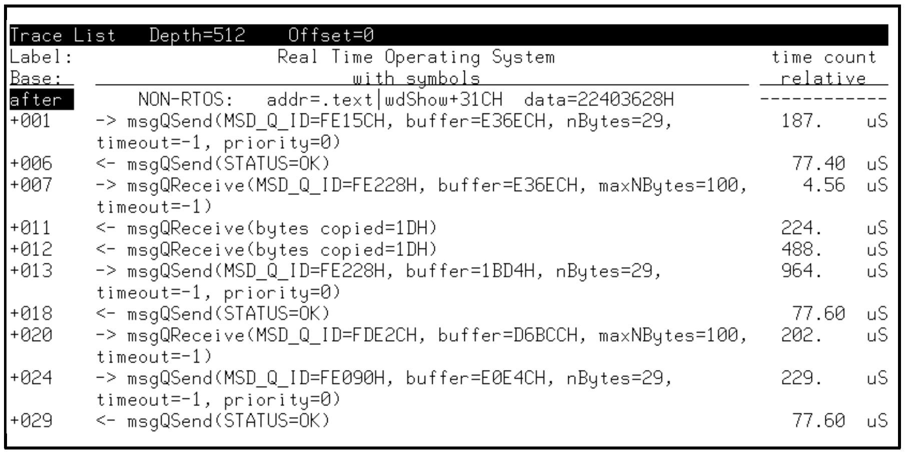
There are also RTOS measurements provided to track particular types of service call activity or OS resources such as semaphores, messages, or semaphores. You can also track individual service calls.

This section shows you how to:

- Track all messages.
- Track all message calls (include task switches).
- Track all semaphore calls.
- Track all semaphore calls (include task switches).
- Track a single service call.
- Track multiple service calls.

To track all messages

- Click on the **Only Messages** action key (or run the **e_onlymsgs** command file by entering it on the command line).

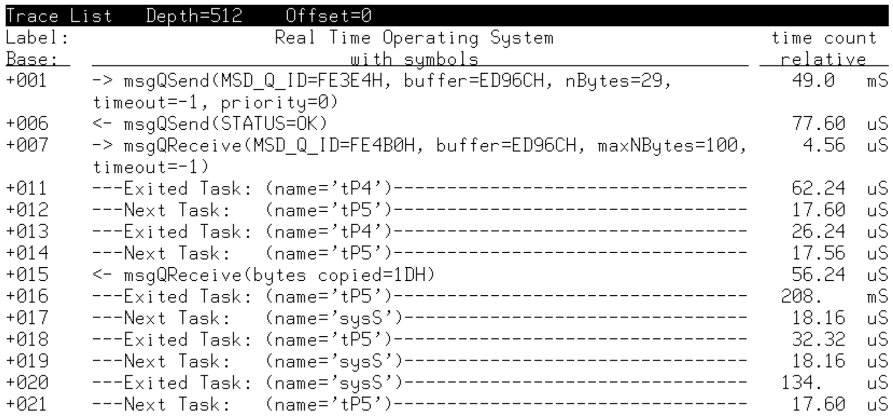


```
Trace List   Depth=512    Offset=0
Label:                    Real Time Operating System                    time count
Base:                            with symbols                            relative
after       NON-RTOS:   addr=.text|wdShow+31CH  data=22403628H          ------------
+001     -> msgQSend(MSD_Q_ID=FE15CH, buffer=E36ECH, nBytes=29,          187.    uS
         timeout=-1, priority=0)
+006     <- msgQSend(STATUS=OK)                                            77.40  uS
+007     -> msgQReceive(MSD_Q_ID=FE228H, buffer=E36ECH, maxNBytes=100,     4.56  uS
         timeout=-1)
+011     <- msgQReceive(bytes copied=1DH)                                224.    uS
+012     <- msgQReceive(bytes copied=1DH)                                488.    uS
+013     -> msgQSend(MSD_Q_ID=FE228H, buffer=1BD4H, nBytes=29,           964.    uS
         timeout=-1, priority=0)
+018     <- msgQSend(STATUS=OK)                                            77.60  uS
+020     -> msgQReceive(MSD_Q_ID=FDE2CH, buffer=D6BCCH, maxNBytes=100,   202.    uS
         timeout=-1)
+024     -> msgQSend(MSD_Q_ID=FE090H, buffer=E0E4CH, nBytes=29,          229.    uS
         timeout=-1, priority=0)
+029     <- msgQSend(STATUS=OK)                                            77.60  uS
```

This action key is used if you are interested in all message activity. No other types of calls are tracked (neither are task switches).

To track all message calls (include task switches)

- Click on the **Tasks & Messages** action key (or run the **e_trackmsgs** command file by entering it on the command line).

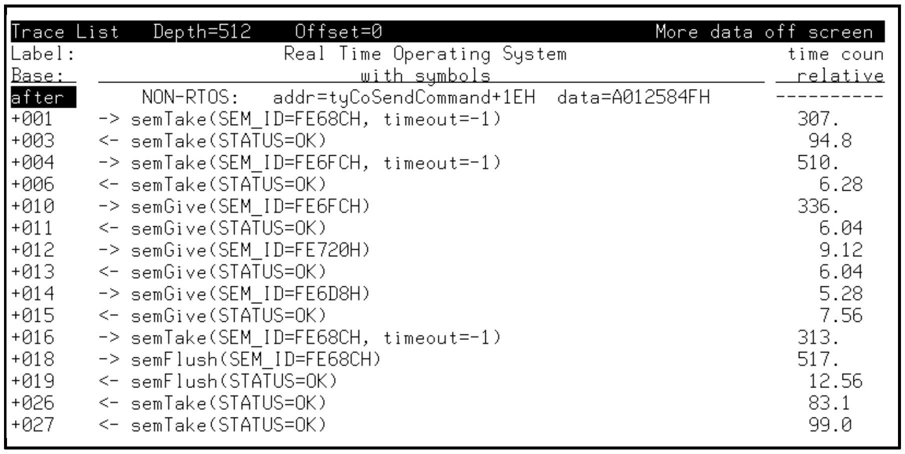


```
Trace List    Depth=512     Offset=0
Label:                    Real Time Operating System                      time count
Base:                              with symbols                             relative
+001     -> msgQSend(MSD_Q_ID=FE3E4H, buffer=ED96CH, nBytes=29,             49.0   mS
         timeout=-1, priority=0)
+006     <- msgQSend(STATUS=OK)                                             77.60  uS
+007     -> msgQReceive(MSD_Q_ID=FE4B0H, buffer=ED96CH, maxNBytes=100,       4.56  uS
         timeout=-1)
+011     ---Exited Task: (name='tP4')---------------------------------     62.24  uS
+012     ---Next Task:   (name='tP5')---------------------------------     17.60  uS
+013     ---Exited Task: (name='tP4')---------------------------------     26.24  uS
+014     ---Next Task:   (name='tP5')---------------------------------     17.56  uS
+015     <- msgQReceive(bytes copied=1DH)                                   56.24  uS
+016     ---Exited Task: (name='tP5')---------------------------------    208.    mS
+017     ---Next Task:   (name='sysS')--------------------------------     18.16  uS
+018     ---Exited Task: (name='tP5')---------------------------------     32.32  uS
+019     ---Next Task:   (name='sysS')--------------------------------     18.16  uS
+020     ---Exited Task: (name='sysS')--------------------------------    134.    uS
+021     ---Next Task:   (name='tP5')---------------------------------     17.60  uS
```

This action key is used if you are only interested in message activity but want to know the task context also.

To track all semaphore calls

- Click on the **Only Semaphores** action key (or run the **e_onlysems** command file by entering it on the command line).

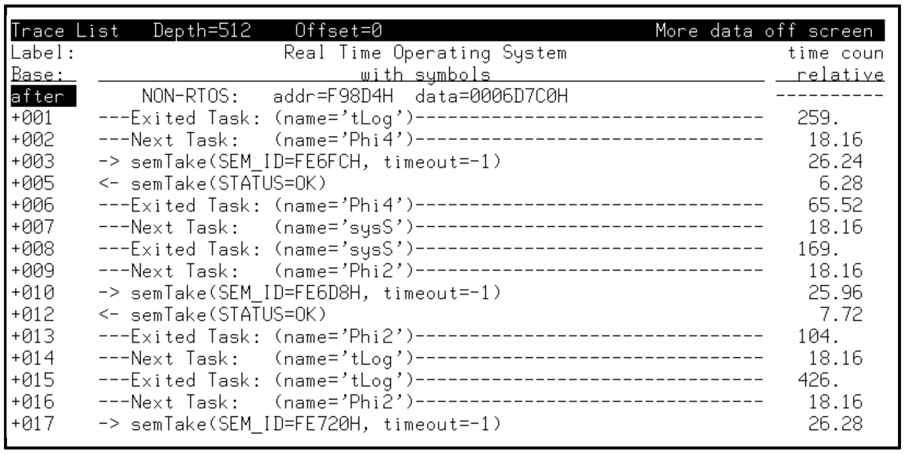


```
Trace List   Depth=512    Offset=0                           More data off screen
Label:                   Real Time Operating System                      time coun
Base:                          with symbols                              relative
after       NON-RTOS:   addr=tyCoSendCommand+1EH  data=A012584FH        ----------
+001    -> semTake(SEM_ID=FE68CH, timeout=-1)                             307.
+003    <- semTake(STATUS=OK)                                              94.8
+004    -> semTake(SEM_ID=FE6FCH, timeout=-1)                             510.
+006    <- semTake(STATUS=OK)                                               6.28
+010    -> semGive(SEM_ID=FE6FCH)                                         336.
+011    <- semGive(STATUS=OK)                                               6.04
+012    -> semGive(SEM_ID=FE720H)                                           9.12
+013    <- semGive(STATUS=OK)                                               6.04
+014    -> semGive(SEM_ID=FE6D8H)                                           5.28
+015    <- semGive(STATUS=OK)                                               7.56
+016    -> semTake(SEM_ID=FE68CH, timeout=-1)                             313.
+018    -> semFlush(SEM_ID=FE68CH)                                        517.
+019    <- semFlush(STATUS=OK)                                             12.56
+026    <- semTake(STATUS=OK)                                              83.1
+027    <- semTake(STATUS=OK)                                              99.0
```

This action key is used if you are interested in all semaphore activity. No other types of calls are tracked (neither are task switches).

To track all semaphore calls (include task switches)

- Click on the **Tasks & Semaphores** action key (or run the **e_tracksems** command file by entering it on the command line).



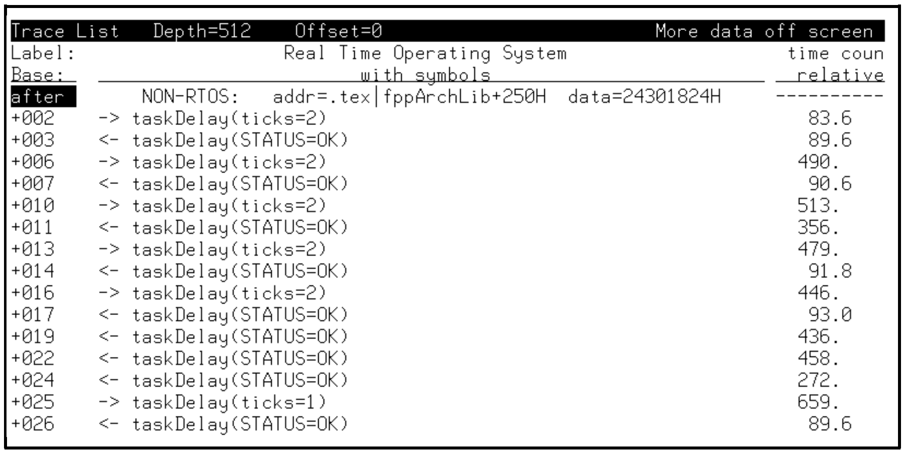
```
Trace List   Depth=512    Offset=0                             More data off screen
Label:                   Real Time Operating System                      time coun
Base:                          with symbols                               relative
after        NON-RTOS:   addr=F98D4H  data=0006D7C0H                     ----------
+001     ---Exited Task: (name='tLog')--------------------------------   259.
+002     ---Next Task:   (name='Phi4')--------------------------------    18.16
+003     -> semTake(SEM_ID=FE6FCH, timeout=-1)                            26.24
+005     <- semTake(STATUS=OK)                                             6.28
+006     ---Exited Task: (name='Phi4')--------------------------------    65.52
+007     ---Next Task:   (name='sysS')--------------------------------    18.16
+008     ---Exited Task: (name='sysS')--------------------------------   169.
+009     ---Next Task:   (name='Phi2')--------------------------------    18.16
+010     -> semTake(SEM_ID=FE6D8H, timeout=-1)                            25.96
+012     <- semTake(STATUS=OK)                                              7.72
+013     ---Exited Task: (name='Phi2')--------------------------------   104.
+014     ---Next Task:   (name='tLog')--------------------------------    18.16
+015     ---Exited Task: (name='tLog')--------------------------------   426.
+016     ---Next Task:   (name='Phi2')--------------------------------    18.16
+017     -> semTake(SEM_ID=FE720H, timeout=-1)                            26.28
```

The command above traces only semaphores and task switches so you can see what tasks use semaphores and how they effect system flow.

To track a single service call

- Click on the **Only Call X** action key (or run the **e_onecall** command file by entering it on the command line).

You are prompted for the name of the service call you wish to track. Enter the service call name.



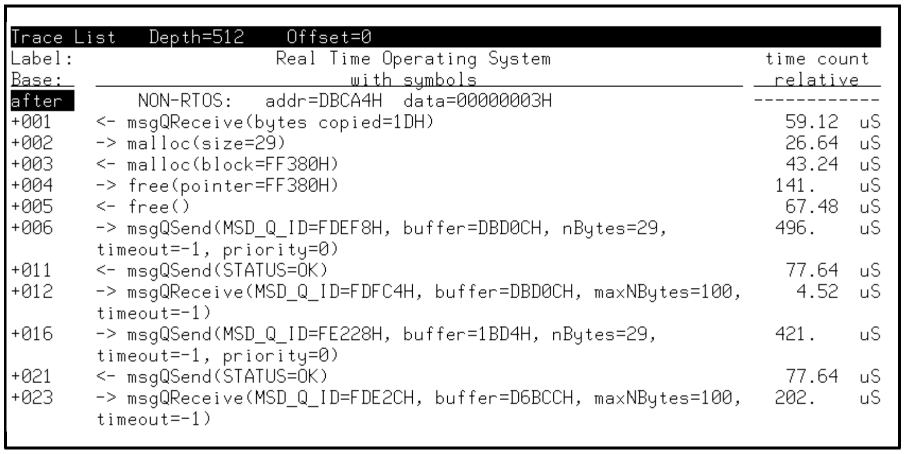
```
Trace List   Depth=512    Offset=0                         More data off screen
Label:                  Real Time Operating System                    time coun
Base:                           with symbols                           relative
after       NON-RTOS:   addr=.tex|fppArchLib+250H  data=24301824H    ----------
+002    -> taskDelay(ticks=2)                                           83.6
+003    <- taskDelay(STATUS=OK)                                         89.6
+006    -> taskDelay(ticks=2)                                          490.
+007    <- taskDelay(STATUS=OK)                                         90.6
+010    -> taskDelay(ticks=2)                                          513.
+011    <- taskDelay(STATUS=OK)                                        356.
+013    -> taskDelay(ticks=2)                                          479.
+014    <- taskDelay(STATUS=OK)                                         91.8
+016    -> taskDelay(ticks=2)                                          446.
+017    <- taskDelay(STATUS=OK)                                         93.0
+019    <- taskDelay(STATUS=OK)                                        436.
+022    <- taskDelay(STATUS=OK)                                        458.
+024    <- taskDelay(STATUS=OK)                                        272.
+025    -> taskDelay(ticks=1)                                          659.
+026    <- taskDelay(STATUS=OK)                                         89.6
```

This action key is used if you have a specific service call you want to track and have no need of the context in which the calls are made.

To track multiple service calls

- Click on the **Only Calls W,X,Y,Z** action key (or run the **e_trk4call** command file by entering it on the command line).

You are prompted for the names of the service calls you wish to track. Enter the service call names.



```
Trace List   Depth=512    Offset=0
Label:                    Real Time Operating System                       time count
Base:                            with symbols                               relative
after       NON-RTOS:   addr=DBCA4H  data=00000003H                       ------------
+001    <- msgQReceive(bytes copied=1DH)                                    59.12  uS
+002    -> malloc(size=29)                                                  26.64  uS
+003    <- malloc(block=FF380H)                                             43.24  uS
+004    -> free(pointer=FF380H)                                            141.    uS
+005    <- free()                                                           67.48  uS
+006    -> msgQSend(MSD_Q_ID=FDEF8H, buffer=DBD0CH, nBytes=29,             496.    uS
        timeout=-1, priority=0)
+011    <- msgQSend(STATUS=OK)                                              77.64  uS
+012    -> msgQReceive(MSD_Q_ID=FDFC4H, buffer=DBD0CH, maxNBytes=100,        4.52  uS
        timeout=-1)
+016    -> msgQSend(MSD_Q_ID=FE228H, buffer=1BD4H, nBytes=29,              421.    uS
        timeout=-1, priority=0)
+021    <- msgQSend(STATUS=OK)                                              77.64  uS
+023    -> msgQReceive(MSD_Q_ID=FDE2CH, buffer=D6BCCH, maxNBytes=100,      202.    uS
        timeout=-1)
```

You may track just the relationship between several tooled calls with this action key.

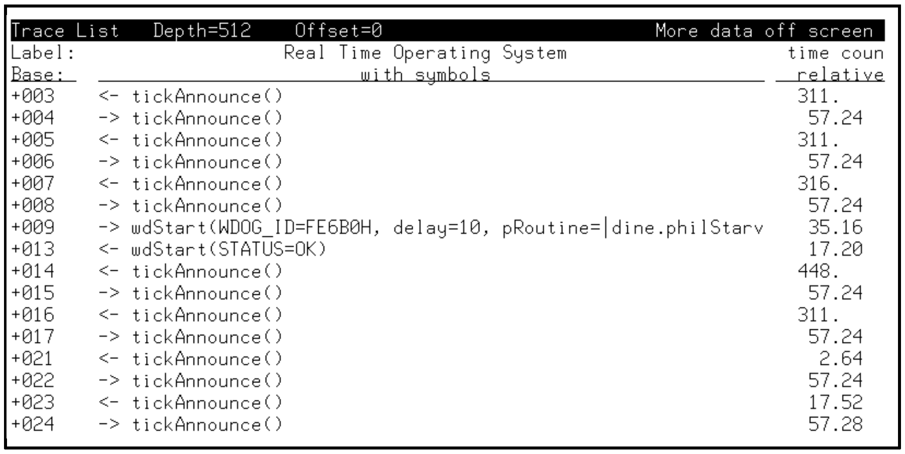
For example, the trace above shows calls to free(), malloc(), msgQSend(), and msgQReceive().

If you are using a 16-bit processor, this action key will trace only two service calls.

To track clocks and timers

- Click on the **Clocks & Timers** action key (or run the **e_onlyclktmr** command file by entering it on the command line).

You may track just the clock ticks and tooled timers with this action key.



```
Trace List   Depth=512    Offset=0                            More data off screen
Label:                  Real Time Operating System                       time coun
Base:                           with symbols                             relative
+003    <- tickAnnounce()                                                  311.
+004    -> tickAnnounce()                                                   57.24
+005    <- tickAnnounce()                                                  311.
+006    -> tickAnnounce()                                                   57.24
+007    <- tickAnnounce()                                                  316.
+008    -> tickAnnounce()                                                   57.24
+009    -> wdStart(WDOG_ID=FE6B0H, delay=10, pRoutine=|dine.philStarv      35.16
+013    <- wdStart(STATUS=OK)                                               17.20
+014    <- tickAnnounce()                                                  448.
+015    -> tickAnnounce()                                                   57.24
+016    <- tickAnnounce()                                                  311.
+017    -> tickAnnounce()                                                   57.24
+021    <- tickAnnounce()                                                    2.64
+022    -> tickAnnounce()                                                   57.24
+023    <- tickAnnounce()                                                   17.52
+024    -> tickAnnounce()                                                   57.28
```

If the **Track OS Calls** action key were to track the tickAnnounce() call, the display would be dominated by the frequent clock ticks. Therefore, a separate action key is provided to track clocks and timers.

Tracking Particular Tasks

Using the powerful sequence triggering capability of the HP 64700 series emulation bus analyzers, several RTOS measurements allow you to capture a very specific sequence of events or very rare events. For example, one point-and-click measurement watches for a user-defined message being sent to a specific message queue; this could help detect a very rare message occurrence. Another point-and-click sequence measurement triggers only when 4 (or less) specific tasks are switched into and out of in any order.

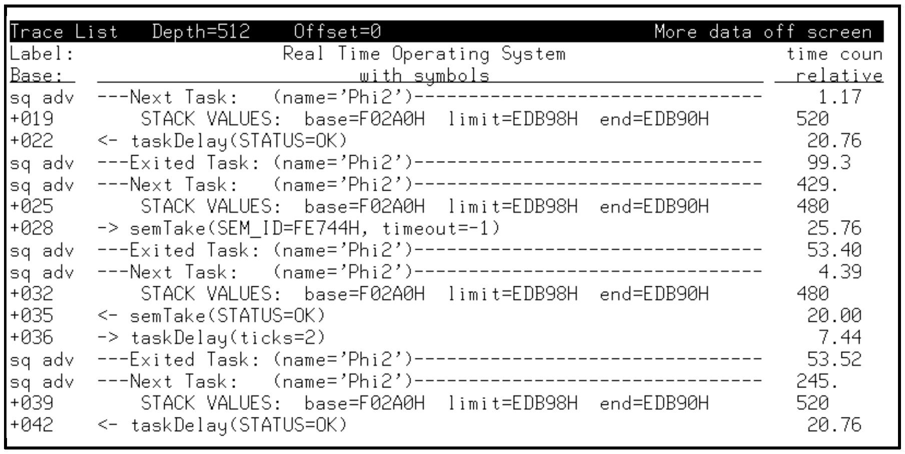
This section shows you how to:

- Track a single task and all tooled OS activity within it.
- Track four tasks and all tooled OS activity within them.
- Track about a specific task switch.
- Track about a specific task sending a message to a specific message queue.
- Trace semTake activity by a specific task.
- Track activity after a function is reached from a specific task.
- Track activity about the access of a variable by a specific task.

To track a single task and all tooled OS activity within it

- Click on the **Only Task X** action key (or run the **e_trk1task** command file by entering it on the command line).

You are prompted for the name of the task that you want to trace. You can type in the name of the task you are interested in, or in the graphical interface, by using the cut buffer, you can cut and paste a taskname from the screen into the dialog box.



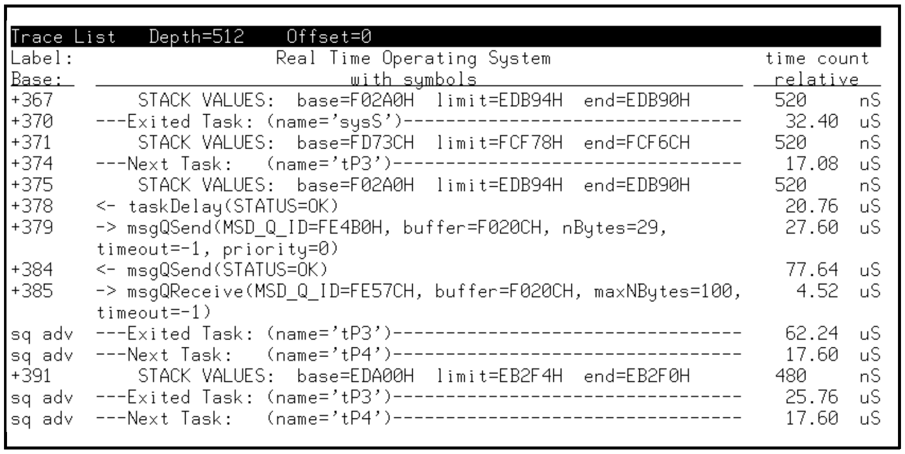
```
Trace List   Depth=512    Offset=0                          More data off screen
Label:                   Real Time Operating System                    time coun
Base:                           with symbols                            relative
sq adv  ---Next Task:   (name='Phi2')--------------------------------    1.17
+019        STACK VALUES:  base=F02A0H  limit=EDB98H  end=EDB90H       520
+022    <- taskDelay(STATUS=OK)                                         20.76
sq adv  ---Exited Task: (name='Phi2')--------------------------------   99.3
sq adv  ---Next Task:   (name='Phi2')--------------------------------  429.
+025        STACK VALUES:  base=F02A0H  limit=EDB98H  end=EDB90H       480
+028    -> semTake(SEM_ID=FE744H, timeout=-1)                           25.76
sq adv  ---Exited Task: (name='Phi2')--------------------------------   53.40
sq adv  ---Next Task:   (name='Phi2')--------------------------------    4.39
+032        STACK VALUES:  base=F02A0H  limit=EDB98H  end=EDB90H       480
+035    <- semTake(STATUS=OK)                                           20.00
+036    -> taskDelay(ticks=2)                                            7.44
sq adv  ---Exited Task: (name='Phi2')--------------------------------   53.52
sq adv  ---Next Task:   (name='Phi2')--------------------------------  245.
+039        STACK VALUES:  base=F02A0H  limit=EDB98H  end=EDB90H       520
+042    <- taskDelay(STATUS=OK)                                         20.76
```

In the example above, only the OS activity in task "Phi2" is being traced.

Notice that the time stamp on the right hand side of the screen gives a useful indication of the time between task exit and the next entry into this same task.

To track four tasks and all tooled OS activity within them

- Click on the **Only Tsk W,X,Y,Z** action key (or run the **e_trk4task** command file by entering it on the command line).



```
Trace List   Depth=512    Offset=0
Label:                  Real Time Operating System                      time count
Base:                           with symbols                              relative
+367        STACK VALUES:  base=F02A0H  limit=EDB94H  end=EDB90H       520     nS
+370     ---Exited Task: (name='sysS')---------------------------------  32.40  uS
+371        STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       520     nS
+374     ---Next Task:   (name='tP3')----------------------------------  17.08  uS
+375        STACK VALUES:  base=F02A0H  limit=EDB94H  end=EDB90H       520     nS
+378     <- taskDelay(STATUS=OK)                                         20.76  uS
+379     -> msgQSend(MSD_Q_ID=FE4B0H, buffer=F020CH, nBytes=29,          27.60  uS
         timeout=-1, priority=0)
+384     <- msgQSend(STATUS=OK)                                          77.64  uS
+385     -> msgQReceive(MSD_Q_ID=FE57CH, buffer=F020CH, maxNBytes=100,    4.52  uS
         timeout=-1)
sq adv   ---Exited Task: (name='tP3')----------------------------------  62.24  uS
sq adv   ---Next Task:   (name='tP4')----------------------------------  17.60  uS
+391        STACK VALUES:  base=EDA00H  limit=EB2F4H  end=EB2F0H       480     nS
sq adv   ---Exited Task: (name='tP3')----------------------------------  25.76  uS
sq adv   ---Next Task:   (name='tP4')----------------------------------  17.60  uS
```

You can use this command to track tooled OS activity within up to four tasks. One, two, or three tasks can also be tracked by entering duplicate names. For example, if you wanted to track only tasks t2 and t3, enter t2 in the first dialog box and t3 in the remaining dialog boxes.

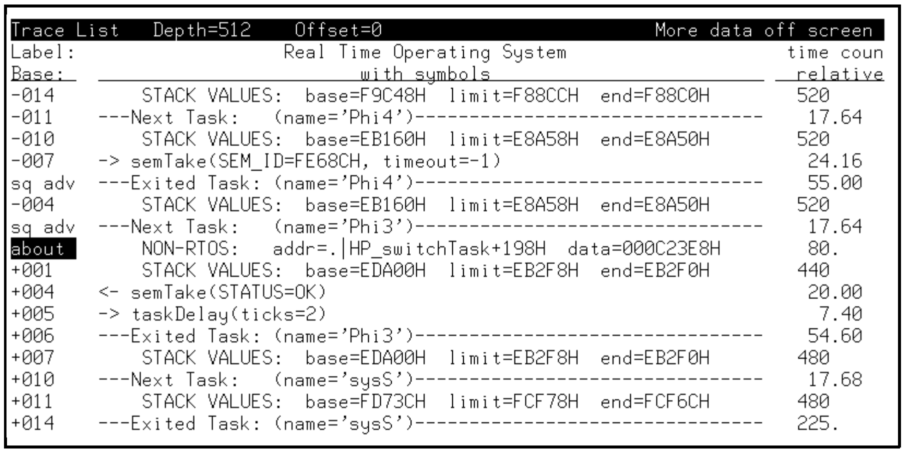
You can also edit the command file to create two new command files which would be used specifically for tracking two or three tasks.

To track about a specific task switch

- Click on the **Task switch A->B** action key (or run the **e_AthenB** command file by entering it on the command line).

This measurement will trace when the kernel switches from one desired task immediately into another desired task. The dialog box first prompts for the task that is being switched out of then prompts again for the task that is being switched into.

When the trace completes, you can see the activity before and after the task switch occurred. This type of measurement may lead you to a problem surrounding a task switch.



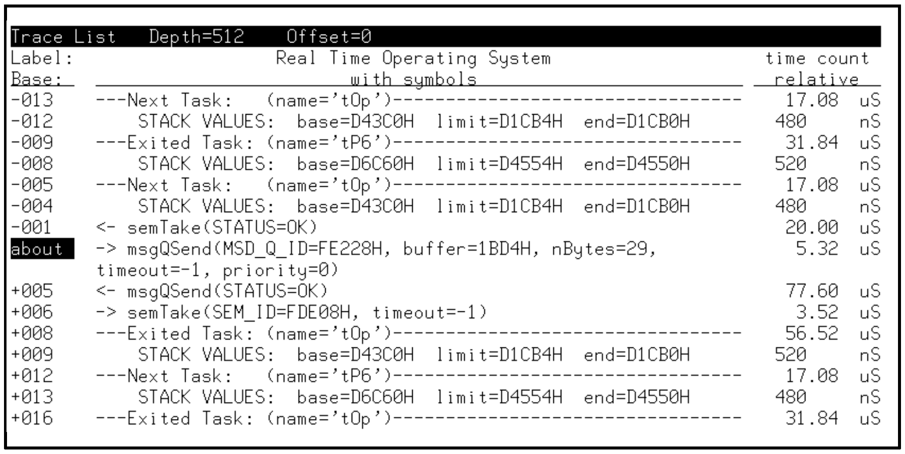
```
Trace List   Depth=512    Offset=0                          More data off screen
Label:                   Real Time Operating System                    time coun
Base:                          with symbols                             relative
-014         STACK VALUES:  base=F9C48H  limit=F88CCH  end=F88C0H       520
-011     ---Next Task:   (name='Phi4')--------------------------------   17.64
-010         STACK VALUES:  base=EB160H  limit=E8A58H  end=E8A50H       520
-007     -> semTake(SEM_ID=FE68CH, timeout=-1)                           24.16
sq adv   ---Exited Task: (name='Phi4')--------------------------------   55.00
-004         STACK VALUES:  base=EB160H  limit=E8A58H  end=E8A50H       520
sq adv   ---Next Task:   (name='Phi3')--------------------------------   17.64
about        NON-RTOS:   addr=.|HP_switchTask+198H  data=000C23E8H       80.
+001         STACK VALUES:  base=EDA00H  limit=EB2F8H  end=EB2F0H       440
+004     <- semTake(STATUS=OK)                                           20.00
+005     -> taskDelay(ticks=2)                                            7.40
+006     ---Exited Task: (name='Phi3')--------------------------------   54.60
+007         STACK VALUES:  base=EDA00H  limit=EB2F8H  end=EB2F0H       480
+010     ---Next Task:   (name='sysS')--------------------------------   17.68
+011         STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       480
+014     ---Exited Task: (name='sysS')--------------------------------  225.
```

In the example above, "Phi4" was entered in the first dialog box and "Phi3" was entered in the second dialog box, thus triggering a trace around this switch from task "Phi4" to task "Phi3".

To track about a specific task sending a message to a specific message queue

- Click on the **Tsk A->MsgQue X** action key (or run the **e_tsk2msgque** command file by entering it on the command line).

You are prompted first for the task name and then for the queue name to which the task sends a message.



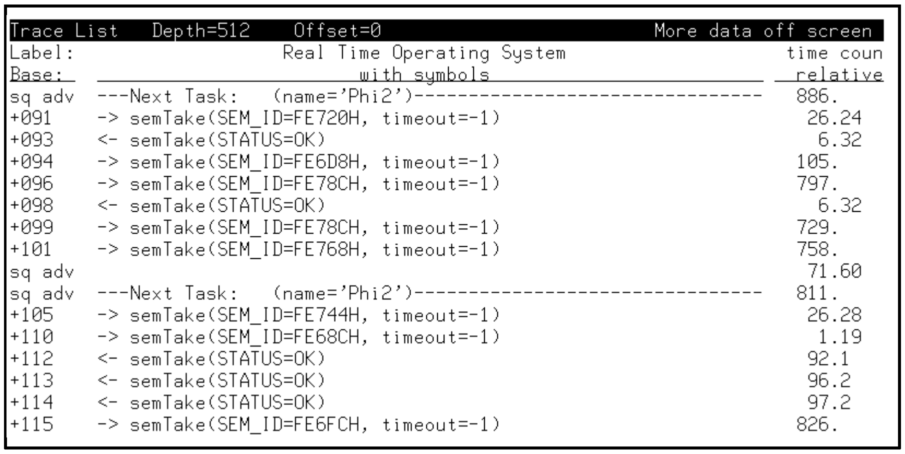
```
Trace List   Depth=512    Offset=0
Label:                   Real Time Operating System                      time count
Base:                           with symbols                              relative
-013     ---Next Task:   (name='tOp')---------------------------------    17.08  uS
-012         STACK VALUES:  base=D43C0H  limit=D1CB4H  end=D1CB0H        480     nS
-009     ---Exited Task: (name='tP6')---------------------------------    31.84  uS
-008         STACK VALUES:  base=D6C60H  limit=D4554H  end=D4550H        520     nS
-005     ---Next Task:   (name='tOp')---------------------------------    17.08  uS
-004         STACK VALUES:  base=D43C0H  limit=D1CB4H  end=D1CB0H        480     nS
-001     <- semTake(STATUS=OK)                                            20.00  uS
about    -> msgQSend(MSD_Q_ID=FE228H, buffer=1BD4H, nBytes=29,             5.32  uS
         timeout=-1, priority=0)
+005     <- msgQSend(STATUS=OK)                                           77.60  uS
+006     -> semTake(SEM_ID=FDE08H, timeout=-1)                             3.52  uS
+008     ---Exited Task: (name='tOp')---------------------------------    56.52  uS
+009         STACK VALUES:  base=D43C0H  limit=D1CB4H  end=D1CB0H        520     nS
+012     ---Next Task:   (name='tP6')---------------------------------    17.08  uS
+013         STACK VALUES:  base=D6C60H  limit=D4554H  end=D4550H        480     nS
+016     ---Exited Task: (name='tOp')---------------------------------    31.84  uS
```

This measurement is useful if you have a task that sends a message to a specific message queue intermittently and you either want to verify that the message gets sent or you want to see the service call context under which the message is sent.

To trace semTake activity by a specific task

- Click on the **Tsk A <- sem** action key (or run the **e_tasksemtake** command file by entering it on the command line).

You are prompted first for the task name.



```
Trace List   Depth=512    Offset=0                          More data off screen
Label:                 Real Time Operating System                      time coun
Base:                         with symbols                              relative
sq adv  ---Next Task:   (name='Phi2')--------------------------------   886.
+091    -> semTake(SEM_ID=FE720H, timeout=-1)                            26.24
+093    <- semTake(STATUS=OK)                                             6.32
+094    -> semTake(SEM_ID=FE6D8H, timeout=-1)                           105.
+096    -> semTake(SEM_ID=FE78CH, timeout=-1)                           797.
+098    <- semTake(STATUS=OK)                                             6.32
+099    -> semTake(SEM_ID=FE78CH, timeout=-1)                           729.
+101    -> semTake(SEM_ID=FE768H, timeout=-1)                           758.
sq adv                                                                   71.60
sq adv  ---Next Task:   (name='Phi2')--------------------------------   811.
+105    -> semTake(SEM_ID=FE744H, timeout=-1)                            26.28
+110    -> semTake(SEM_ID=FE68CH, timeout=-1)                             1.19
+112    <- semTake(STATUS=OK)                                            92.1
+113    <- semTake(STATUS=OK)                                            96.2
+114    <- semTake(STATUS=OK)                                            97.2
+115    -> semTake(SEM_ID=FE6FCH, timeout=-1)                           826.
```

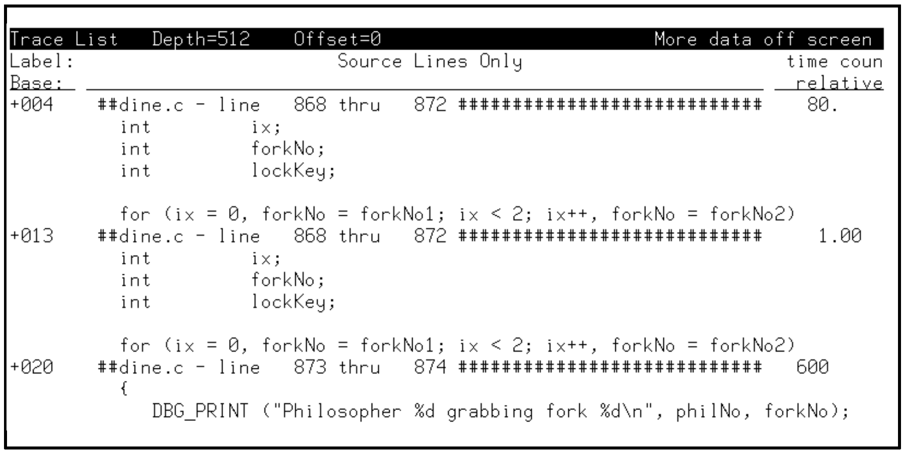
This measurement allows you to view the context under which a semTake occurs for a specific task. Because there is no indicator of which semaphore is being granted on a semTake return, all activity is shown.

To track activity after a function is reached from a specific task

- **Click on the Task A: FuncX action key (or run the e_afterfunc command file by entering it on the command line).**

The normal "C" source code tracing is still available whenever you need to see your actual application code. In fact you can use an RTOS trigger point to then capture source code activity.

This command will trace into a source code function but only when it has been called from a certain task. You are first prompted for the calling task name and then the desired function.



```
Trace List   Depth=512    Offset=0                         More data off screen
Label:                          Source Lines Only                      time coun
Base:                                                                   relative
+004    ##dine.c - line   868 thru   872 ############################    80.
          int         ix;
          int         forkNo;
          int         lockKey;

          for (ix = 0, forkNo = forkNo1; ix < 2; ix++, forkNo = forkNo2)
+013    ##dine.c - line   868 thru   872 ############################     1.00
          int         ix;
          int         forkNo;
          int         lockKey;

          for (ix = 0, forkNo = forkNo1; ix < 2; ix++, forkNo = forkNo2)
+020    ##dine.c - line   873 thru   874 ############################   600
          {
             DBG_PRINT ("Philosopher %d grabbing fork %d\n", philNo, forkNo);
```

This example shows calls to the forkGrab() function from task Phi2.

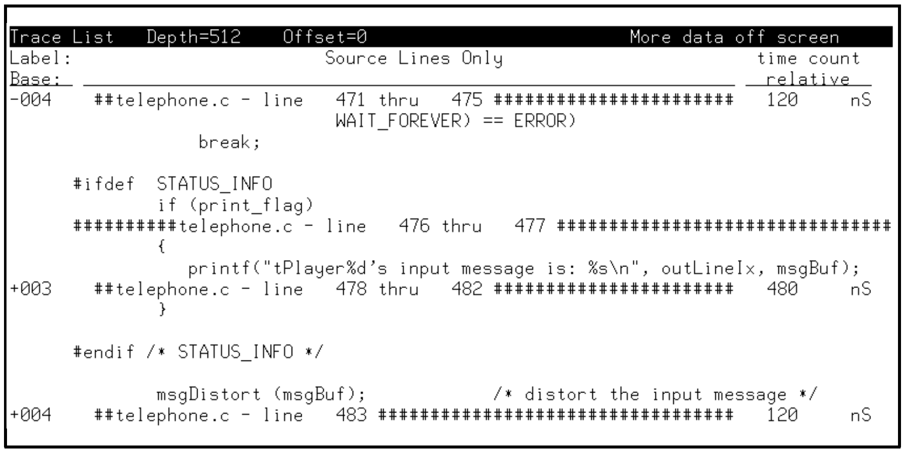
You can easily return to the RTOS trace display by clicking on the **Disp RTOS Trace** action key (or by entering the **display trace real_time_os** command on the command line) and making another RTOS measurement.

The function must be in the global symbol table. That is, no function can be traced if its address is local.

To track activity about the access of a variable by a specific task

- Click on the **Task A: VarX** action key (or run the **e_aftervar** command file by entering it on the command line).

You are prompted first for the task name and then for the variable name which the task accesses.



```
Trace List   Depth=512    Offset=0                        More data off screen
Label:                        Source Lines Only                         time count
Base:                                                                    relative
-004    ##telephone.c - line   471 thru   475 #######################   120     nS
                               WAIT_FOREVER) == ERROR)
                  break;

      #ifdef  STATUS_INFO
              if (print_flag)
      ##########telephone.c - line   476 thru   477 ################################
              {
                 printf("tPlayer%d's input message is: %s\n", outLineIx, msgBuf);
+003    ##telephone.c - line   478 thru   482 #######################   480     nS
              }

      #endif /* STATUS_INFO */

              msgDistort (msgBuf);            /* distort the input message */
+004    ##telephone.c - line   483 ##################################   120     nS
```

This measurement allows you to see when a specific variable is accessed by a specific task and the source code context under which the variable is accessed.

The function must be in the global symbol table. That is, no function can be traced if its address is local.

Tracking Accesses to Functions or Variables

Another useful RTOS measurement identifies which tasks are accessing a shared global variable or calling a shared function.

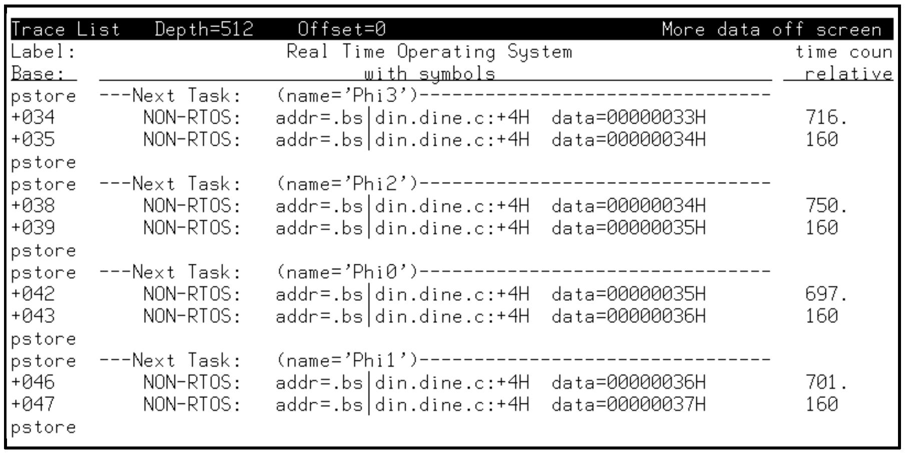
This section shows you how to:

- Track which tasks access a specific function or variable.

To track which tasks access a specific function or variable

- Click on the **Task?: Func/VarX** action key (or run the **e_qtskfunc** command file by entering it on the command line).

You are prompted for a function or variable name.



```
Trace List   Depth=512    Offset=0                          More data off screen
Label:                   Real Time Operating System                      time coun
Base:                          with symbols                               relative
pstore  ---Next Task:   (name='Phi3')--------------------------------
+034        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000033H        716.
+035        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000034H        160
pstore
pstore  ---Next Task:   (name='Phi2')--------------------------------
+038        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000034H        750.
+039        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000035H        160
pstore
pstore  ---Next Task:   (name='Phi0')--------------------------------
+042        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000035H        697.
+043        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000036H        160
pstore
pstore  ---Next Task:   (name='Phi1')--------------------------------
+046        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000036H        701.
+047        NON-RTOS:   addr=.bs|din.dine.c:+4H  data=00000037H        160
pstore
```

All tasks that call a specific function or access a specific variable can be tracked with this measurement.

The function must be in the global symbol table. That is, no function can be traced if its address is local.

Tracking Dynamic Memory Usage

Tracking dynamic memory usage has always been difficult in an embedded design. With these new real-time operating system measurement tools, however, even these debugging headaches become easy to solve.

The basic measurement set displays the size and location of a memory segment whenever the system allocates a new block of memory. The system also reports whenever a previously allocated block of memory is freed.

Stack allocation information is also provided. With this information, you can use the analyzer to monitor for stack overflow conditions.

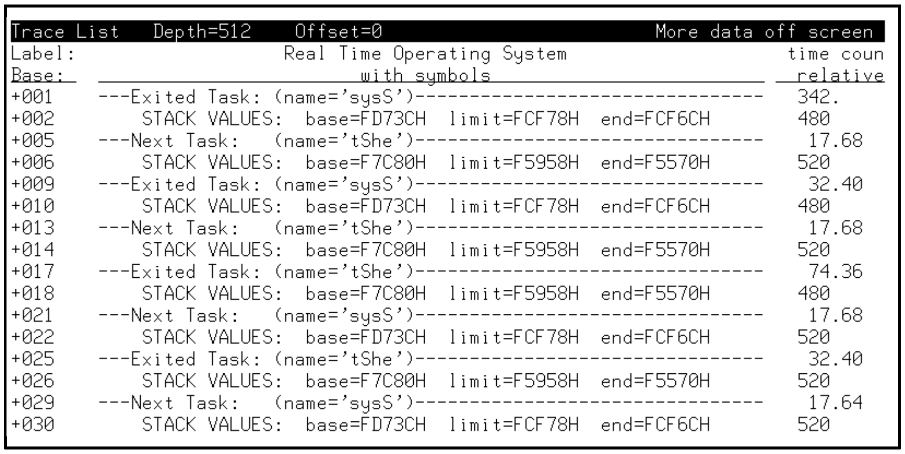
This section shows you how to:

- Track only stack data.
- Track all memory calls (include task switches).

To track only stack data

- Click on the **Stack Usage** action key (or run the **e_stack** command file by entering it on the command line).

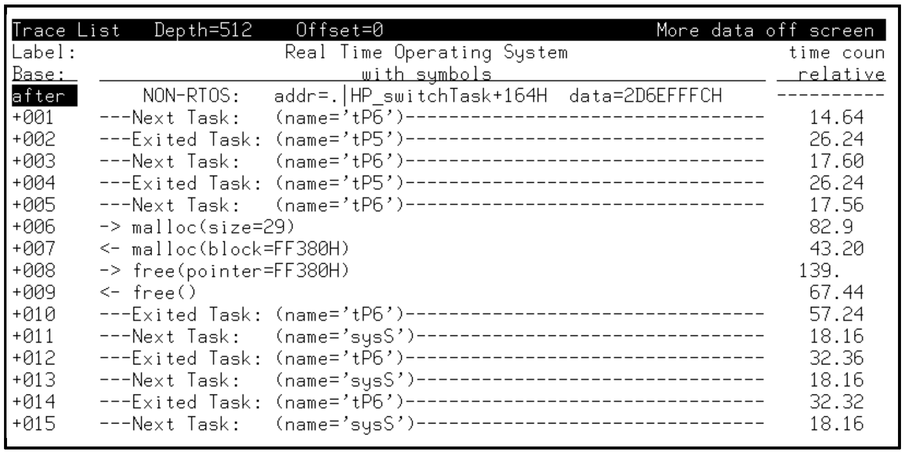
You can enter this command before you run your application from its startup address to capture the initialization of the application which shows you where each local stack is allocated.



```
Trace List   Depth=512    Offset=0                         More data off screen
Label:                   Real Time Operating System                  time coun
Base:                          with symbols                           relative
+001    ---Exited Task: (name='sysS')--------------------------------  342.
+002        STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       480
+005    ---Next Task:   (name='tShe')--------------------------------   17.68
+006        STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       520
+009    ---Exited Task: (name='sysS')--------------------------------   32.40
+010        STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       480
+013    ---Next Task:   (name='tShe')--------------------------------   17.68
+014        STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       520
+017    ---Exited Task: (name='tShe')--------------------------------   74.36
+018        STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       480
+021    ---Next Task:   (name='sysS')--------------------------------   17.68
+022        STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       520
+025    ---Exited Task: (name='tShe')--------------------------------   32.40
+026        STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       520
+029    ---Next Task:   (name='sysS')--------------------------------   17.64
+030        STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       520
```

To track all tooled memory calls (include task switches)

- Click on the **Memory Usage** action key (or run the **e_memory** command file by entering it on the command line).



```
Trace List   Depth=512    Offset=0                         More data off screen
Label:                  Real Time Operating System                   time coun
Base:                          with symbols                           relative
after       NON-RTOS:   addr=.|HP_switchTask+164H  data=2D6EFFFCH   ----------
+001    ---Next Task:   (name='tP6')---------------------------------   14.64
+002    ---Exited Task: (name='tP5')---------------------------------   26.24
+003    ---Next Task:   (name='tP6')---------------------------------   17.60
+004    ---Exited Task: (name='tP5')---------------------------------   26.24
+005    ---Next Task:   (name='tP6')---------------------------------   17.56
+006    -> malloc(size=29)                                              82.9
+007    <- malloc(block=FF380H)                                         43.20
+008    -> free(pointer=FF380H)                                        139.
+009    <- free()                                                       67.44
+010    ---Exited Task: (name='tP6')---------------------------------   57.24
+011    ---Next Task:   (name='sysS')--------------------------------   18.16
+012    ---Exited Task: (name='tP6')---------------------------------   32.36
+013    ---Next Task:   (name='sysS')--------------------------------   18.16
+014    ---Exited Task: (name='tP6')---------------------------------   32.32
+015    ---Next Task:   (name='sysS')--------------------------------   18.16
```

This command simply tracks all tooled calls for memory allocation, giving you an idea of memory usage.

Displaying Traces

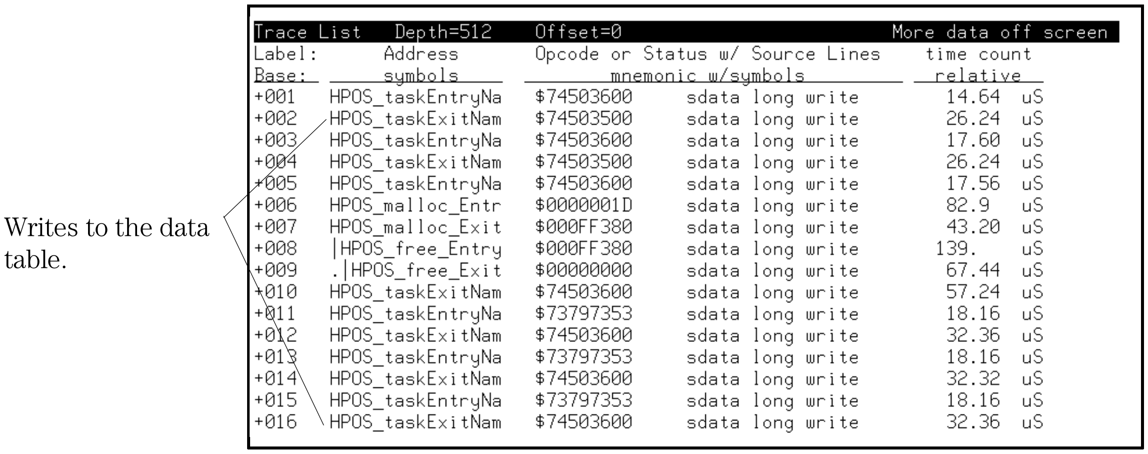
The normal "C" source code tracing is still available whenever you need to see your actual application code. You can switch between the normal "C" source code display and the RTOS measurements display with a simple click of an action key or by entering a display trace command.

This section shows you how to:

- Switch to a normal trace display.
- Switch to the RTOS trace display.

To switch to a normal trace display

- Click on the **Disp NonRTOS Trc** action key (or run the **e_normtrace** command file by entering it on the command line, or enter the **display trace mnemonic** command on the command line).

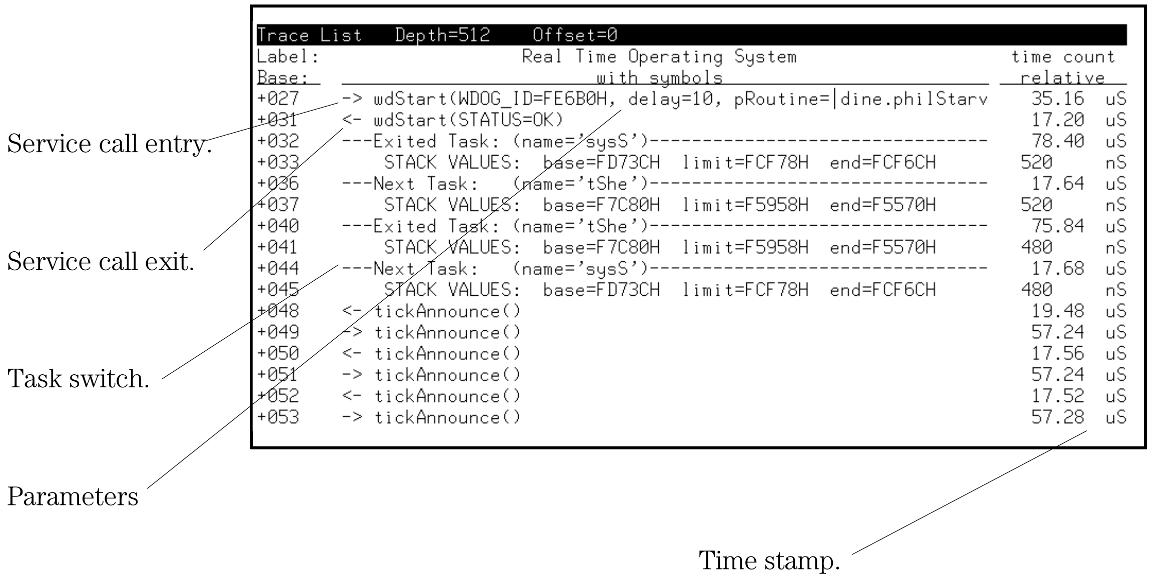


Writes to the data table.

```
Trace List   Depth=512    Offset=0                         More data off screen
Label:      Address      Opcode or Status w/ Source Lines    time count
Base:       symbols             mnemonic w/symbols            relative
+001   HPOS_taskEntryNa  $74503600     sdata long write       14.64  uS
+002   HPOS_taskExitNam  $74503500     sdata long write       26.24  uS
+003   HPOS_taskEntryNa  $74503600     sdata long write       17.60  uS
+004   HPOS_taskExitNam  $74503500     sdata long write       26.24  uS
+005   HPOS_taskEntryNa  $74503600     sdata long write       17.56  uS
+006   HPOS_malloc_Entr  $0000001D     sdata long write       82.9   uS
+007   HPOS_malloc_Exit  $000FF380     sdata long write       43.20  uS
+008   |HPOS_free_Entry  $000FF380     sdata long write      139.    uS
+009   .|HPOS_free_Exit  $00000000     sdata long write       67.44  uS
+010   HPOS_taskExitNam  $74503600     sdata long write       57.24  uS
+011   HPOS_taskEntryNa  $73797353     sdata long write       18.16  uS
+012   HPOS_taskExitNam  $74503600     sdata long write       32.36  uS
+013   HPOS_taskEntryNa  $73797353     sdata long write       18.16  uS
+014   HPOS_taskExitNam  $74503600     sdata long write       32.32  uS
+015   HPOS_taskEntryNa  $73797353     sdata long write       18.16  uS
+016   HPOS_taskExitNam  $74503600     sdata long write       32.36  uS
```

To switch to the RTOS trace display

- Click on the **Disp RTOS Trace** action key (or enter the **display trace real_time_os** command on the command line).



```
Trace List   Depth=512    Offset=0
Label:                   Real Time Operating System                    time count
Base:                           with symbols                            relative
+027     -> wdStart(WDOG_ID=FE6B0H, delay=10, pRoutine=|dine.philStarv    35.16  uS
+031     <- wdStart(STATUS=OK)                                            17.20  uS
+032     ---Exited Task: (name='sysS')--------------------------------    78.40  uS
+033          STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       520     nS
+036     ---Next Task:   (name='tShe')--------------------------------    17.64  uS
+037          STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       520     nS
+040     ---Exited Task: (name='tShe')--------------------------------    75.84  uS
+041          STACK VALUES:  base=F7C80H  limit=F5958H  end=F5570H       480     nS
+044     ---Next Task:   (name='sysS')--------------------------------    17.68  uS
+045          STACK VALUES:  base=FD73CH  limit=FCF78H  end=FCF6CH       480     nS
+048     <- tickAnnounce()                                                19.48  uS
+049     -> tickAnnounce()                                                57.24  uS
+050     <- tickAnnounce()                                                17.56  uS
+051     -> tickAnnounce()                                                57.24  uS
+052     <- tickAnnounce()                                                17.52  uS
+053     -> tickAnnounce()                                                57.28  uS
```

Note that there are entry and exit arrows on the left of the screen to show when a tooled call is entered and, on a separate line, to show when a tooled call is exited. This is important since an OS service call may switch to another task while in the OS and NOT return to the calling service call for a long time, if ever.

The OS service calls are decoded into the same mnemonics that appear in the OS manual. The parameters and return values that are associated with tooled calls are displayed. The parameter variable names also appear as they do in the OS manual decoded into their English mnemonics. Some of the parameter values and all return values are also decoded.

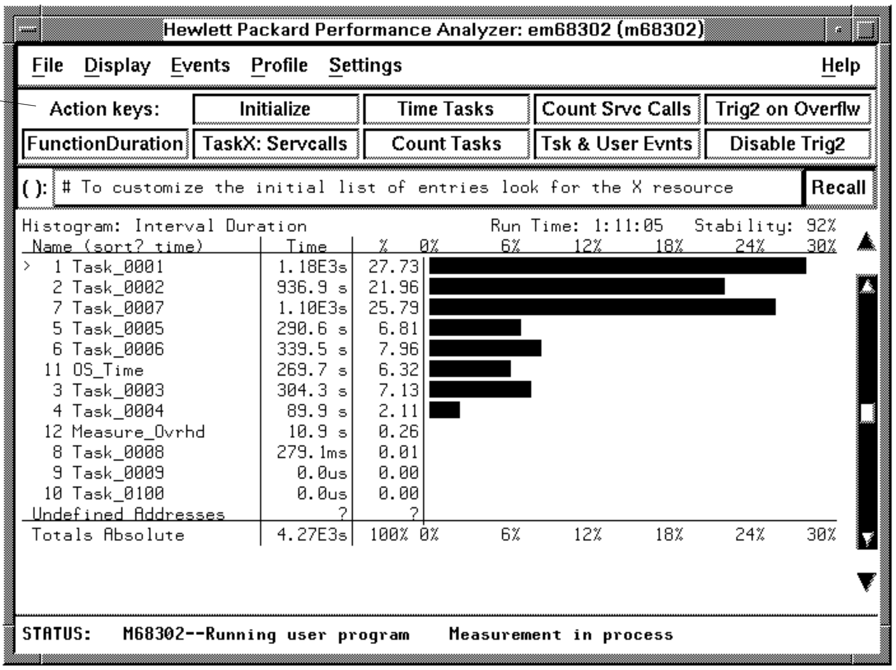
You may have noticed that the line numbers in the first column of the display are not sequential. This is because several trace states may be disassembled for each line in the RTOS trace display.

3

Making RTOS Measurements with the SPA

Making RTOS Measurements with the SPA

Action keys for RTOS measurements.



```
# To customize the initial list of entries look for the X resource

Histogram: Interval Duration                    Run Time: 1:11:05   Stability: 92%
 Name (sort? time)          Time       %    0%      6%      12%      18%      24%     30%
>  1 Task_0001             1.18E3s   27.73
   2 Task_0002             936.9 s   21.96
   7 Task_0007             1.10E3s   25.79
   5 Task_0005             290.6 s    6.81
   6 Task_0006             339.5 s    7.96
  11 OS_Time               269.7 s    6.32
   3 Task_0003             304.3 s    7.13
   4 Task_0004              89.9 s    2.11
  12 Measure_Ovrhd          10.9 s    0.26
   8 Task_0008             279.1ms    0.01
   9 Task_0009               0.0us    0.00
  10 Task_0100               0.0us    0.00
 Undefined Addresses             ?       ?
 Totals Absolute           4.27E3s    100% 0%      6%      12%      18%      24%     30%

STATUS:   M68302--Running user program    Measurement in process
```

The HP 64708A Software Performance Analyzer (SPA), a plug-in card for the HP 64700 emulation system, provides valuable OS-level profiling measurements. This makes finding bottlenecks simple. In addition, the number of times each task is called can be displayed, providing valuable information on system "thrashing". Also, the number of times each OS service call is invoked from your application can be tracked, helping to isolate bottlenecks from over-utilized system features.

The Software Performance Analyzer can also detect when a task has exceeded a maximum preset time duration. When combined with the cross triggering capabilities of the emulation system, you are able to capture a historical trace showing the sequence of events leading up to the overflow and/or the system can be halted to allow browsing through the current state of the system.

If you have multiple projects on one machine, you'll need to set up unique SPA windows for each project.

These tasks are grouped into the following sections:

- Making time profile measurements.
- Coordinating measurements with the emulator.
- Handling multiple projects on one machine.

Making Time Profile Measurements

By measuring the time between writes made to task entry and exit locations, the Software Performance Analyzer (SPA) can provide time interval measurements for the tasks in your application as well as for the OS.

The time duration of each task can be displayed in an easy to read histogram. Cumulative, maximum, and minimum time spent in each task can be displayed in a table.

This section shows you how to:

- Define SPA events for tasks, service calls, and user events.
- Display a time histogram of task events.
- Show a table of SPA events.
- Display a count histogram of task events.
- Measure only data from a specific task.
- Show a table of tooled call invocations.
- Show a normal function duration histogram.
- Show a histogram of task and user events.

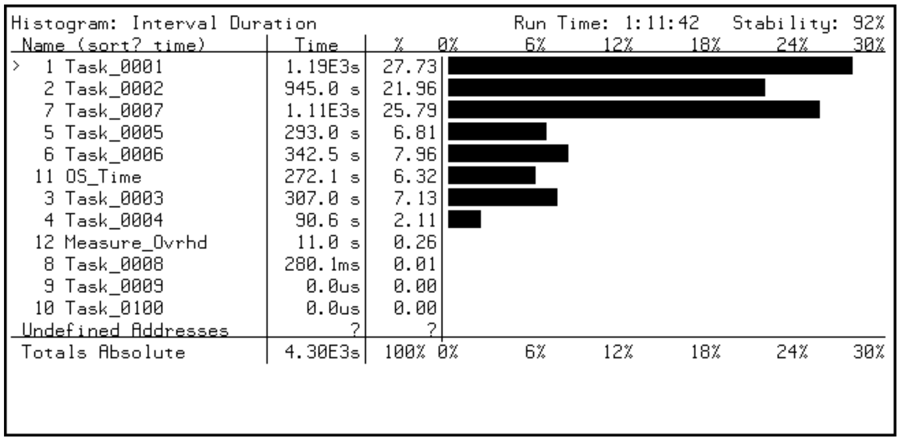
To define SPA events for tasks, tooled calls, and user events

- **Click on the Initialize action key (or run the s_init command file by entering it on the command line).**

These instructions assume you have edited the **s_init** command file by running the tool "rtos_edit_vxworks".

To display a time histogram of task events

- Click on the **Time Tasks** action key (or run the **s_timetasks** command file by entering it on the command line).



```
Histogram: Interval Duration                  Run Time: 1:11:42   Stability: 92%
 Name (sort? time)      |   Time   |  %   0%      6%     12%     18%     24%    30%
>  1 Task_0001          | 1.19E3s | 27.73|
   2 Task_0002          | 945.0 s | 21.96|
   7 Task_0007          | 1.11E3s | 25.79|
   5 Task_0005          | 293.0 s |  6.81|
   6 Task_0006          | 342.5 s |  7.96|
  11 OS_Time            | 272.1 s |  6.32|
   3 Task_0003          | 307.0 s |  7.13|
   4 Task_0004          |  90.6 s |  2.11|
  12 Measure_Ovrhd      |  11.0 s |  0.26|
   8 Task_0008          | 280.1ms |  0.01|
   9 Task_0009          |   0.0us |  0.00|
  10 Task_0100          |   0.0us |  0.00|
 Undefined Addresses    |       ? |     ?|
 Totals Absolute        | 4.30E3s |  100% 0%      6%     12%     18%     24%    30%
```

The histogram shows how much time each task is taking. This is very useful for detecting system bottlenecks.

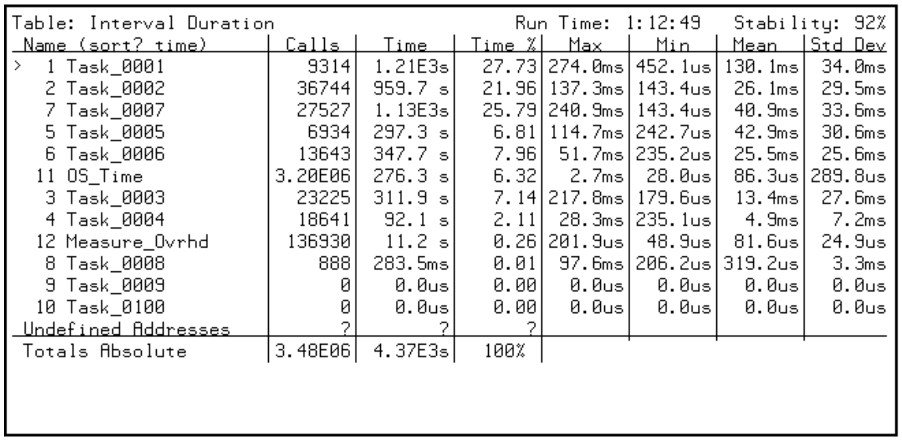
Note that one line of the histogram is labeled "OS_Time". This indicates how much time the application is spending in the OS kernel itself. This OS overhead measurement has some limitations however. Refer to the "OS Overhead Tracking" section in the "How the RTOS Measurement Tool Works" chapter for more information.

Another line is labeled "Measure_Ovrhd". This indicates approximately how much intrusion is caused by the RTOS measurement tool routines.

To show a table of SPA events

- Choose the **Display→Table** pulldown menu item (or enter the **display table** command on the command line).

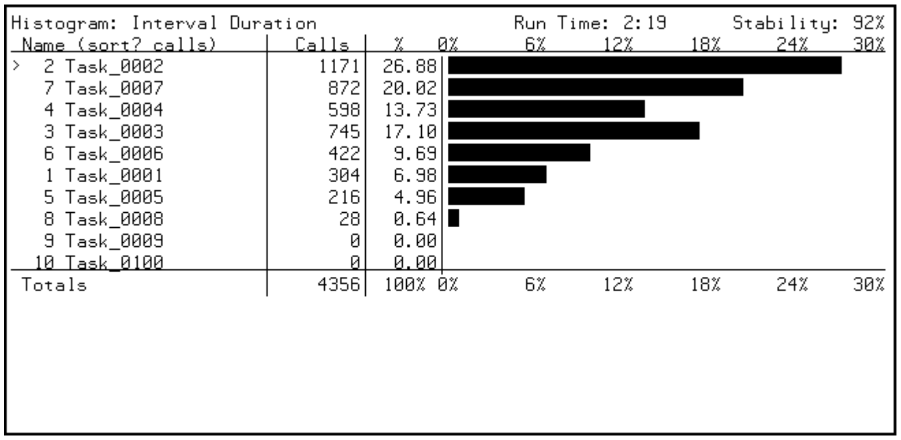
A raw numbers view of the accumulated data is displayed.



```
Table: Interval Duration                          Run Time: 1:12:49   Stability: 92%
 Name (sort? time)        | Calls  |  Time   | Time %|  Max    |  Min    | Mean    |Std Dev
>   1 Task_0001           |    9314| 1.21E3s |  27.73|274.0ms  |452.1us  |130.1ms  | 34.0ms
    2 Task_0002           |   36744| 959.7 s |  21.96|137.3ms  |143.4us  | 26.1ms  | 29.5ms
    7 Task_0007           |   27527| 1.13E3s |  25.79|240.9ms  |143.4us  | 40.9ms  | 33.6ms
    5 Task_0005           |    6934| 297.3 s |   6.81|114.7ms  |242.7us  | 42.9ms  | 30.6ms
    6 Task_0006           |   13643| 347.7 s |   7.96| 51.7ms  |235.2us  | 25.5ms  | 25.6ms
   11 OS_Time             | 3.20E06| 276.3 s |   6.32|  2.7ms  | 28.0us  | 86.3us  |289.8us
    3 Task_0003           |   23225| 311.9 s |   7.14|217.8ms  |179.6us  | 13.4ms  | 27.6ms
    4 Task_0004           |   18641|  92.1 s |   2.11| 28.3ms  |235.1us  |  4.9ms  |  7.2ms
   12 Measure_Ovrhd       |  136930|  11.2 s |   0.26|201.9us  | 48.9us  | 81.6us  | 24.9us
    8 Task_0008           |     888| 283.5ms |   0.01| 97.6ms  |206.2us  |319.2us  |  3.3ms
    9 Task_0009           |       0|   0.0us |   0.00|  0.0us  |  0.0us  |  0.0us  |  0.0us
   10 Task_0100           |       0|   0.0us |   0.00|  0.0us  |  0.0us  |  0.0us  |  0.0us
 Undefined Addresses      |       ?|       ? |      ?|         |         |         |
 Totals Absolute          | 3.48E06| 4.37E3s |  100% |
```

To display a count histogram of task events

- Click on the **Count Tasks** action key (or run the **s_counttasks** command file by entering it on the command line).

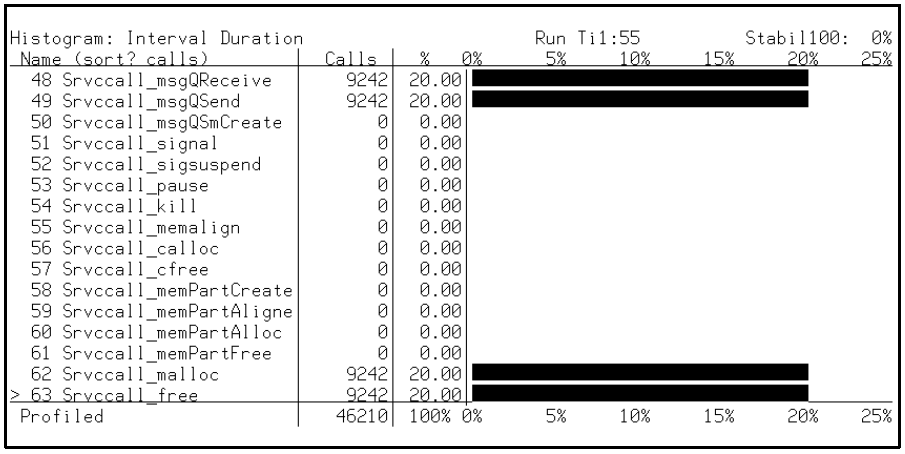


```
Histogram: Interval Duration                    Run Time: 2:19      Stability: 92%
 Name (sort? calls)    |  Calls |  %   0%      6%      12%      18%      24%     30%
>  2 Task_0002         |   1171 | 26.88|
   7 Task_0007         |    872 | 20.02|
   4 Task_0004         |    598 | 13.73|
   3 Task_0003         |    745 | 17.10|
   6 Task_0006         |    422 |  9.69|
   1 Task_0001         |    304 |  6.98|
   5 Task_0005         |    216 |  4.96|
   8 Task_0008         |     28 |  0.64|
   9 Task_0009         |      0 |  0.00|
  10 Task_0100         |      0 |  0.00|
 Totals                |   4356 | 100% 0%      6%      12%      18%      24%     30%
```

The histogram shows the the number of times each task is entered (and exited). This can be very useful for detecting system "thrashing" between tasks.

To measure only data from a specific task

- Click on the **TaskX: Servcalls** action key (or run the **s_taskwindow** command file by entering it on the command line).

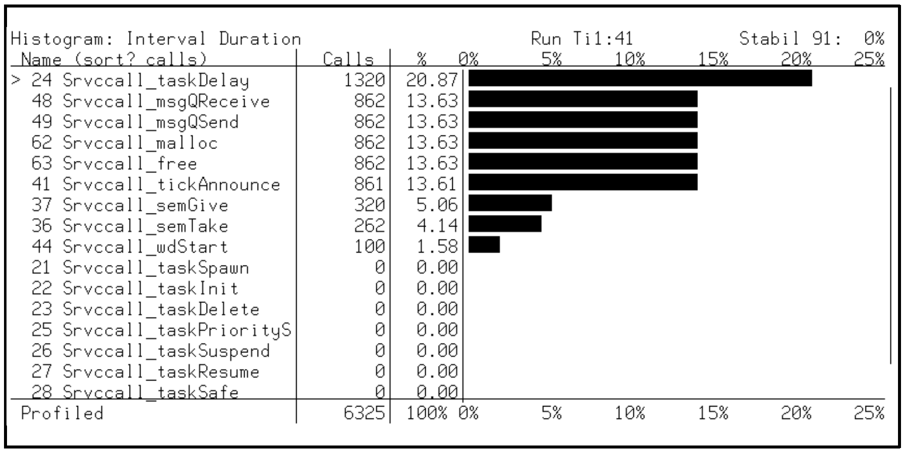


```
Histogram: Interval Duration                    Run Ti1:55        Stabil100:  0%
 Name (sort? calls)         |  Calls |  %   0%      5%     10%     15%     20%     25%
  48 Srvccall_msgQReceive   |   9242| 20.00|████████████████████████████████
  49 Srvccall_msgQSend      |   9242| 20.00|████████████████████████████████
  50 Srvccall_msgQSmCreate  |      0|  0.00|
  51 Srvccall_signal        |      0|  0.00|
  52 Srvccall_sigsuspend    |      0|  0.00|
  53 Srvccall_pause         |      0|  0.00|
  54 Srvccall_kill          |      0|  0.00|
  55 Srvccall_memalign      |      0|  0.00|
  56 Srvccall_calloc        |      0|  0.00|
  57 Srvccall_cfree         |      0|  0.00|
  58 Srvccall_memPartCreate |      0|  0.00|
  59 Srvccall_memPartAligne |      0|  0.00|
  60 Srvccall_memPartAlloc  |      0|  0.00|
  61 Srvccall_memPartFree   |      0|  0.00|
  62 Srvccall_malloc        |   9242| 20.00|████████████████████████████████
> 63 Srvccall_free          |   9242| 20.00|████████████████████████████████
 Profiled                   |  46210| 100% 0%      5%     10%     15%     20%     25%
```

This displays a histogram of the number of times each tooled call is invoked from a single task.

To show a table of service call invocations

- Click on the **Count Srvc Calls** action key (or run the **s_countsrvcls** command file by entering it on the command line).

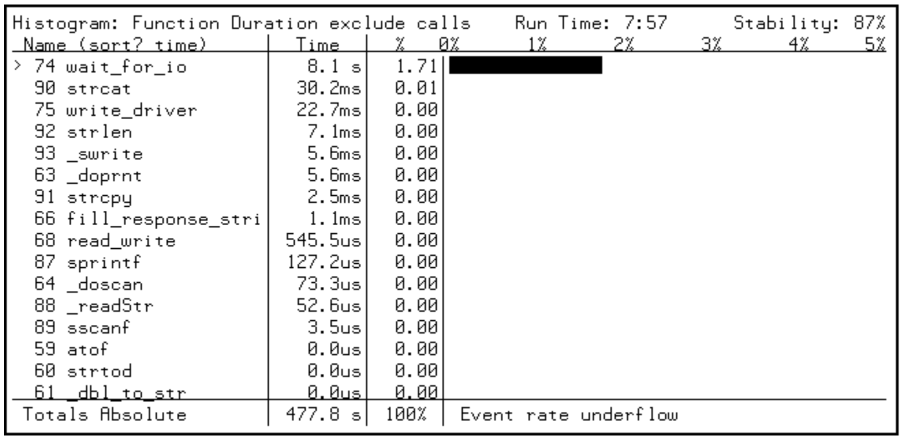


```
Histogram: Interval Duration                       Run Ti1:41        Stabil 91:  0%
 Name (sort? calls)          | Calls |  %   0%      5%      10%     15%     20%    25%
> 24 Srvccall_taskDelay      |  1320| 20.87|
  48 Srvccall_msgQReceive    |   862| 13.63|
  49 Srvccall_msgQSend       |   862| 13.63|
  62 Srvccall_malloc         |   862| 13.63|
  63 Srvccall_free           |   862| 13.63|
  41 Srvccall_tickAnnounce   |   861| 13.61|
  37 Srvccall_semGive        |   320|  5.06|
  36 Srvccall_semTake        |   262|  4.14|
  44 Srvccall_wdStart        |   100|  1.58|
  21 Srvccall_taskSpawn      |     0|  0.00|
  22 Srvccall_taskInit       |     0|  0.00|
  23 Srvccall_taskDelete     |     0|  0.00|
  25 Srvccall_taskPriorityS  |     0|  0.00|
  26 Srvccall_taskSuspend    |     0|  0.00|
  27 Srvccall_taskResume     |     0|  0.00|
  28 Srvccall_taskSafe       |     0|  0.00|
 Profiled                    |  6325| 100% 0%      5%      10%     15%     20%    25%
```

This displays a histogram of the number of times each tooled call is invoked from all tasks.

To show a normal function duration histogram

- Click on the **FunctionDuration** action key (or run the **s_funcdur** command file by entering it on the command line).

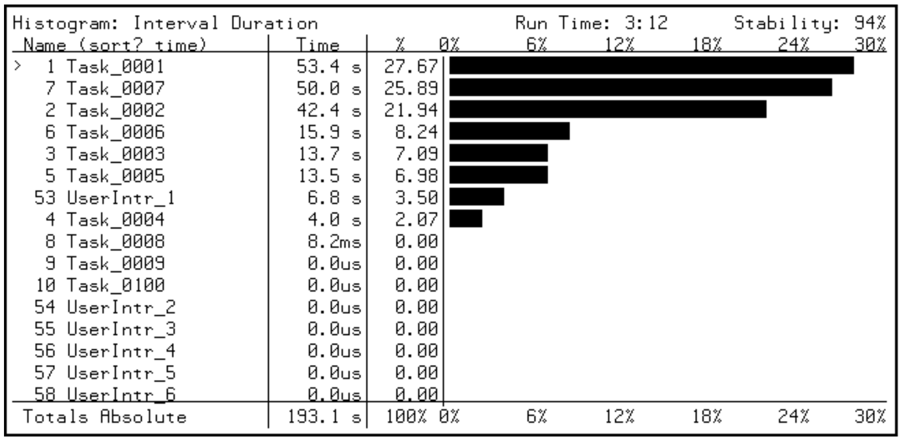


```
Histogram: Function Duration exclude calls    Run Time: 7:57      Stability: 87%
 Name (sort? time)       |  Time   |  %   0%      1%       2%       3%       4%      5%
> 74 wait_for_io         |   8.1 s |  1.71|██████████████
  90 strcat              |  30.2ms |  0.01|
  75 write_driver        |  22.7ms |  0.00|
  92 strlen              |   7.1ms |  0.00|
  93 _swrite             |   5.6ms |  0.00|
  63 _doprnt             |   5.6ms |  0.00|
  91 strcpy              |   2.5ms |  0.00|
  66 fill_response_stri  |   1.1ms |  0.00|
  68 read_write          | 545.5us |  0.00|
  87 sprintf             | 127.2us |  0.00|
  64 _doscan             |  73.3us |  0.00|
  88 _readStr            |  52.6us |  0.00|
  89 sscanf              |   3.5us |  0.00|
  59 atof                |   0.0us |  0.00|
  60 strtod              |   0.0us |  0.00|
  61 _dbl_to_str         |   0.0us |  0.00|
 Totals Absolute         | 477.8 s |  100%| Event rate underflow
```

This performs a normal function duration profile measurement.

To show a histogram of task and user events

- Click on the **Tsk & User Evnts** action key (or run the **s_tasknuser** command file by entering it on the command line).



```
Histogram: Interval Duration                    Run Time: 3:12      Stability: 94%
 Name (sort? time)       |  Time   |   %   0%      6%      12%     18%     24%     30%
>   1 Task_0001          |  53.4 s | 27.67|
    7 Task_0007          |  50.0 s | 25.89|
    2 Task_0002          |  42.4 s | 21.94|
    6 Task_0006          |  15.9 s |  8.24|
    3 Task_0003          |  13.7 s |  7.09|
    5 Task_0005          |  13.5 s |  6.98|
   53 UserIntr_1         |   6.8 s |  3.50|
    4 Task_0004          |   4.0 s |  2.07|
    8 Task_0008          |   8.2ms |  0.00|
    9 Task_0009          |   0.0us |  0.00|
   10 Task_0100          |   0.0us |  0.00|
   54 UserIntr_2         |   0.0us |  0.00|
   55 UserIntr_3         |   0.0us |  0.00|
   56 UserIntr_4         |   0.0us |  0.00|
   57 UserIntr_5         |   0.0us |  0.00|
   58 UserIntr_6         |   0.0us |  0.00|
 Totals Absolute         | 193.1 s | 100% 0%      6%      12%     18%     24%     30%
```

This measurement includes any user-defined events you may have set up.

Coordinating Measurements with the Emulator

During a Software Performance Analyzer duration measurement, the SPA can generate a trig2 signal if the event being measured executes for too long a period of time. This signal can be used by the emulator to stop the application program, or it can be used by the emulation analyzer to trace activity up to that point.

This combination of events allows you to stop the application program when a task exceeds a certain amount of continuous execution time and/or track activity that leads up to the break.

This section shows you how to:

- Break on task time overflow.
- Disable the SPA trig2.

To break on task time overflow

You can also set up a coordinated measurement between the software performance analyzer and the emulation bus analyzer. For example, you might like to capture a trace and then break into the emulation monitor if a certain task ever takes longer than a specified maximum time. Tracing before the time overflow will show a history of what led up to the time overrun.

1 In the emulation window, click on the **Before SPA trig2** action key.

Or (in the emulation window), run the **e_spatrig** command file by entering it on the command line.

You have now set up the analyzer to capture a trace when a signal is received from SPA. Note that the trace has started but has not completed because it is waiting for the trig2 signal as its trigger point.

2 In the SPA window, click on the **Trig2 on Overflw** action key.

You can now set up SPA to detect the time overflow and then send the appropriate signal to the emulation window. The dialog box again prompts you for specific information. The first box prompts you for a task name.

3 In the dialog box, type the name of the task; then, click the "OK" pushbutton.

Another dialog box now appears asking you for the maximum time limit to be watching for. Type in the number of milliseconds that is the maximum time you want the given task to ever continuously execute.

4 In the dialog box, type in the limit; then, click the "OK" pushbutton.

After a while you see that the emulator is running in monitor due to a time overflow break from SPA. The status line of the emulation window shows a "trig2 break" which came from SPA. The trace has completed and shows you a historical trace of what led up to the time overflow. Notice that the application has just entered the task which you specified.

To disable the SPA trig2

- In the SPA window, click on the **Disable Trig2** action key.

This action key must be pressed whenever cross-trigger measurements to the emulator are no longer desired.

Note

Until the trig2 signal from SPA is disabled, the signal will be continually sent to the emulation system. This may result in unexpected behavior such as continually breaking into the monitor or traces being started but not completing.

Handling Multiple Projects on One Machine

In order to run multiple sessions—one for each unique application—of the RTOS product on one machine, a couple of changes need to be made. These changes are required because a command file for the Software Performance Analyzer contains application specific commands that set up intervals for each task.

To set up unique SPA windows for multiple projects

- **If more than one project is using the RTOS Measurement Tool, you need to make sure the Initialize action key calls a command file specific to your currently loaded application.**

1 Run the $HP64000/bin/rtos_edit_vxworks script.

2 Rename the s_init file which was generated by the script.

Repeat steps 1 and 2 above for all of your projects.

3 Before you start the emulator window for a given project, set the perf.Vrtx*actionKeysSub.keyDefs X resource so that the **Initialize** action key calls the appropriate s_init file.

Here are two ways to set an X resource:

- Edit the $HOME/.HP64_schemes/Softkey.Label file, as described on page 86.
- Place the X resource definition in a file, and run "xrdb -merge <filename>".

Note that all of the action keys are set in a single X resource, so you need to set all of the Software Performance Analyzer action keys along with the changed **Initialize** action key.

If you are using several different real-time operating systems, and a project is the only one which uses a particular operating system, you do not need to make any changes for that project.

4

Customizing the RTOS Measurement Tool

Customizing the RTOS Measurement Tool

You can customize the RTOS Measurement Tool to create your own RTOS measurements. You can set up your own trace commands that capture particular writes to the data table, put these commands in command files, and set up action keys that run these command files.

Though the level of intrusion introduced by the "instrumented" service call library is very limited, you can customize the RTOS Measurement Tool to further limit the intrusion if it becomes a problem.

These tasks are grouped into the following sections:

- Creating your own RTOS measurements.
- Limiting the intrusion caused by instrumented service calls.

Creating Your Own RTOS Measurements

Real-time OS measurements in the emulator/analyzer interface are made by using the emulation bus analyzer to capture writes made to a data table. Instructions in the "track_il.c" and "callout.c" files write values to the data table when:

Tasks start.
Tasks switch.
Tasks are deleted.
Service calls are entered and exited.

Any states captured by the emulation bus analyzer outside the range of the data table are interpreted as non-RTOS states.

When you display the RTOS trace, the inverse assembler looks at the information written to the data table, and, since it knows how these locations are defined, it interprets the information and presents it in an easy to read form on the trace display.

In order to understand how to make your own RTOS measurements, you must understand what writes to each of the locations in the data table mean. Once you understand this, you will be able to enter your own trace commands to capture the RTOS information you're looking for.

If your measurements will be made often, you can create your own command files and add your own action keys to the emulator/analyzer interface.

Data Table Description

The data table reserves space for information saved when tasks start, when tasks switch, and when service call functions are entered or exited.

There are also locations for device service call, stack, user-defined, clock tick, and return value checking information.

The "track_os.c" source file reserves space for the data table.

Data Table Contents

The types of values that are written to the data table are:

HPOS_taskExitName
HPOS_taskEntryName

The name of the task being exited or entered is written to these locations. By triggering on specific data values written to these locations, you can trigger on a particular task's entry or exit.

HPOS_<svc_call_sym>_Entry
HPOS_<svc_call_sym>_Exit

The first parameter passed to, or returned from, a service call is written to these locations. HPOS_AdditionalParameters is used for parameters other than the first.

When creating your own RTOS trace commands, be sure to store writes through the full range of the symbol; once the inverse assembler sees the first word written to these locations, it expects an exact number of subsequent writes to follow.

HPOS_tickAnnouce_Entry
HPOS_tickAnnounce_Exit

These locations are written to as system clock ticks are sent into the OS kernel. You have to instrument your clock interrupt service routine (ISR) to see this functionality.

HPOS_CHECK_RETURNS

Return codes are written to this location when service calls exit.

HPOS_User*n*_[Entry|Exit]

These locations are reserved for tracking user-defined activity. For more information, refer to the "How the RTOS Measurement Tool Works" chapter.

To set up trace commands to capture RTOS information

- **Use the "only" syntax of the trace command to specify the storage qualifier.**

The most basic thing to realize about capturing RTOS information with the emulation bus analyzer is that you only want to store writes to the data table. Any other stored state will be displayed in the RTOS trace display as a non-RTOS state.

Virtually all the trace commands you enter to capture RTOS information will specify that "only" a range of locations in the data table or "only" a range and other specific locations in the data table are to be stored in the trace. (If you wish to look at all code execution you will store all states.)

One exception to this guideline is the ability to capture both writes to the data table and your application code execution excluding execution of the actual VxWorks code itself. This can usually be accomplished by storing all activity not in the range of the VxWorks code (that is, **trace only address not range** <VxWorks_start> **thru** <VxWorks_end>). Once the analyzer has captured this data, you may find it helpful to use two emulation windows simultaneously: one to display the normal source code trace, and the other to display the RTOS trace.

- **Use the "after", "about", or "before" syntax of the trace command if you wish to trigger the analyzer on a certain event or occurrence in your program. The option you choose specifies the position of the trigger point in trace memory.**

- **Use the "find_sequence" syntax of the trace command if you wish to trigger the analyzer on a certain sequence of events or occurrences in your program.**

- Use the "enable" and "disable" syntax of the trace command to capture only certain parts (in other words, windows) of program execution.

You can identify the entry or exit of a particular task, by using data qualifiers. The HP Real-Time Operating System Measurement Tool uses integer (32-bit) task names, corresponding to the first 4 characters of the name assigned as the task was spawned. The integer task names are written to HPOS_taskExitName and HPOS_taskEntryName.

Note that the emulation bus analyzer captures 16 bits of data per state when used with 16-bit processors and 32 bits of data per state when used with 32-bit processors. If you are using a 16-bit processor, you must capture the write of the high-order word or low-order word to identify a particular task.

- Use the provided action key measurements as models.

Some of the action key measurements use shell scripts or utility programs which you may find useful. For example, the taskToInt32 program is used by some of the action keys to convert full task names to integer task names.

Example

To track only message queue and semaphore service calls:

```
trace only address range HPOS_semBCreate_Entry thru
HPOS_msgQSmCreate_Exit or HPOS_AdditionalParameters
<RETURN>
```

This captures all writes to the data table that correspond to any semaphore or queue service calls.

To place your measurements in command files

1 If your measurement is similar to a measurement that already exists on the action keys (and therefore in a command file), the best way to create the new command file is to copy and modify the similar command file.

2 Add the directory that contains your custom command files to the HP64KPATH environment variable.

Examples

Suppose you want to create a command file for an RTOS measurement that tracks a 2 tasks. Notice that this is similar to the provided RTOS measurement that tracks 4 tasks.

First copy the existing command file. Assuming you are using 32-bit VxWorks, type"

```
$ cp $HP64000/rtos/B3084A/action_keys_32/e_trk4task
e_trk2task <RETURN>
```

Edit the "e_trk2task" command file so that only 2 tasks are tracked.

If your command file is placed in the $HOME/rtoscmdf directory, you should set the HP64KPATH environment variable as follows:

If you're using "sh" or "ksh":

```
$ HP64KPATH=$HP64KPATH:$HOME/rtoscmdf; export HP64KPATH
<RETURN>
```

If you're using "csh":

```
$ setenv HP64KPATH ${HP64KPATH}:$HOME/rtoscmdf <RETURN>
```

To place your measurements on action keys

You may redefine, add, or delete action keys.

- **Save the measurement in a command file.**

  Follow the instructions in the previous "To place your measurements in command files" section

- **Create a "$HOME/.HP64_schemes" directory:**

  ```
  $ cd <RETURN>
  $ mkdir .HP64_schemes <RETURN>
  $ cd .HP64_schemes <RETURN>
  ```

  This directory must be in your home directory. Note the dot (.) in the ".HP64_schemes" directory name.

- **Copy the system-wide X resources "scheme" file to "Softkey.Label" in the directory you just created:**

  ```
  $ cp
  $HP64000/inst/rtos/vxworks/HP64_schemes/Softkey.App
  Softkey.Label <RETURN>
  ```

- **Edit the action key definitions.**

  The "actionKeysSub.keyDefs" X resource defines a list of paired strings. The first string defines the text that appears on the action key pushbutton. The second string defines the command or, in the case of the RTOS measurement tool, the command file that should be sent to the command line area and executed when the action key is pushed.

  The command files associated with action keys typically set up trace commands that capture real-time OS activity. If parameters are required, the command files prompt you for them. Also, some command files have commands that extract information from memory.

Example

Suppose you wish to create an action key for the command file created in the previous "To place your measurements in command files" section.

Edit your "Softkey.Label" file.

```
vi $HOME/.HP64_schemes/Softkey.Label
```

Add a line that defines the action key label "Only Task X,Y" and the location of the command file. In this case, add the line:

```
\"Only Task X,Y\"     \"e_trk2task\" \
```

as part of the "keyDefs" resource definition.

You may also set the "actionKeys.numColumns" resource to manage the number of rows of action keys.

The next time you start the emulator/analyzer interface, the new action key will appear. Clicking on the new action key will cause the associated command file to be run.

See Also

Your HP *Emulator/Analyzer Graphical Interface User's Guide* or *Debugger/Emulator User's Guide* for more information on setting X resources.

Part 2

Concept Guide

Topics that explain concepts and apply them to advanced tasks.

Part 2

5

How the RTOS Measurement Tool Works

How the RTOS Measurement Tool Works

The RTOS measurement tool lets you perform a real-time trace of selected calls and returns between your application and a Real-Time Operating System (RTOS). The RTOS measurement tool works with the HP 64700 series emulation bus analyzer and includes a specially developed inverse assembler. The trace display is easily readable and includes a fully interpreted display of all parameters passed into and returned from the RTOS along with possibly other pertinent data.

The following topics are discussed in this chapter:

- Instrumented code for real-time OS tracking.
- How OS service calls are captured and displayed.

Instrumented Code for Real-Time OS Tracking

In order to make RTOS measurements, a few instructions must be added to the application program. The level of intrusion introduced by these instructions is minimal, typically under 50 µs per call.

Service Call Tracking

A ".h" header file is used to redefine a regular service call to point to an HP supplied function in place of the RTOS function in your code. This HP function writes information to a data table using high-level language assignments and then calls the real OS function in a "daisy-chain" fashion. An example of an HP supplied function is:

```
/*****************************************************************
* semTake
*****************************************************************/
STATUS HPIL_semTake
   (
    SEM_ID semId,
    int timeout
    )
{
   STATUS       retval;

#ifdef MEASURE_OS_TIME
   HPOS_Start_Ovrhd=(short int) 1;   /* Start OS overhead;HP-RTOS-Level-2*/
#endif /* MEASURE_OS_TIME */
   HPOS_semTake_Entry = (int) semId;
   HPOS_AdditionalParameters = timeout;
   retval =semTake(semId,timeout);
   HPOS_semTake_Exit = (int) retval;
   HPOS_CHECK_RETURNS = (int) retval;
#ifdef MEASURE_OS_TIME
   HPOS_Stop_Ovrhd=  (short int) 2;  /*Stop OS overhead;HP-RTOS-Level-2*/
#endif /* MEASURE_OS_TIME */

   return(retval);
}
```

Notice that information about the parameters for the service calls is written to defined memory locations in a data table. When the application is run, tracing the address range of the data table captures all data being passed into and returned from all the tooled service calls.

When trace information is captured by the emulator, an RTOS-specific inverse assembler decodes the information and displays the results. For each tooled service call, a 32-bit word is associated with the entry (first parameter, if any), and the exit (first return parameter, if any). Any additional parameters are written to a third fixed memory location. Note that these written values do not have to be stored permanently–the emulator captures the values as they are written. Also note that since multiple data values may be written to the same location, it is imperative that track_il.c be compiled *without optimization*.

Task Creation, Switching, and Deletion Tracking

The routines in the taskHookLib library are provided by the RTOS vendor. They allow a user to define a *callout* routine to be called every time tasks are created, switched, or deleted. Upon calling the task switch routine, for example, two parameters are set with pointers to the task control blocks of the task being exited and the task being entered.

For the simplest task switch tracking, the callout routine need only consist of two operations: one writing out the name of the task being exited, one writing the name of the task being entered. This means the data area must have two positions for task entry and exit.

For software performance analysis support, a little more needs to be done. The software performance analyzer needs separate memory locations for the start and end of each interval it is measuring. Each task to be measured must have its own unique start and end memory locations. The callout routine must write to these unique locations depending on which tasks are switching. In the callout routine, the pointer value in a spare location in the task control block is used as an index to a special task data *buckets* area where there is a unique location for every task's exit and entry. This data area is application dependent and must be modified with the application's task names. The "rtos_edit_vxworks" script creates the file "tables.c" which defines these task buckets.

Clock Ticks

In order to track clock ticks, it is assumed that the application uses the tickAnnounce() OS service call. Clock tick information is automatically available since this service call is instrumented.

The memory locations for tickAnnounce are placed at the end of the data table so it may be simply included or excluded from the range of memory accesses stored in the trace. This is done because of the large amount of emulator trace depth that might be consumed tracking every clock tick.

Selective Tracking

With the data area for service calls defined, it is possible to selectively track certain functions. The only limiting factors are the resources of the emulation bus analyzer which allow you to track any range (of any size) along with any 8 distinct memory locations. The 8 locations may be consecutive which, in essence, provides another range for needed cases. The calls must be ones which are tooled.

OS Overhead Tracking

In order to get an estimate of an application's efficiency, that is, to see how much time is spent switching tasks as opposed to executing them, the HP Software Performance Analyzer can display a dynamic histogram of the approximate time spent in the OS kernel.

This is done by addding an instruction in the entry to each tooled service call that writes to a location that represents the start of the OS interval. A second instruction, executed after the return from the service call, writes to a location that represents the end of the OS interval. The HP Software Performance Analyzer measures the time between these writes as time spent in the OS kernel.

Since service calls may be preemtpted by other service calls or tasks, the OS interval is also ended whenever a task creation, switch, or deletion occurs. It is assumed that any current service call has been preempted at this point, and that a task will subsequently execute, which should not be counted as OS time.

Note

The OS_Time measurement thus shown in the Software Performance Analyzer is an approximation of the actual value, and its accuracy cannot be guaranteed. To decide on the appropriateness of the measurement for your application, carefully read the measurement procedure described above.

Stack and Memory Tracking

Stack information on a per-task basis can be tracked dynamically as an application runs. The necessary data is written out during the task switch callout routine. For this to work, there are several things that must be done before the application is running and switching tasks:

1. The "task buckets" in the task table must be filled with the names of any of application’s tasks that are to be tracked. This informs the task switch callout routine to save the task’s stack values.
2. The task start callout routine will save several data items: the task name, the memory locations in the Task Control Block that hold the stack pointer values, and the task bucket’s address. The data is written to a special area in the general data area so the dynamic stack information can be captured and seen in the trace display at startup time.

Once the application is switching tasks, the task switch callout routine uses data in the task control block to determine various stack values. The stack data can then be written out and interpreted by the RTOS inverse assembler to display the stack information on exit from a task and entry to a task.

User-Defined Areas

Near the end of the general data table is a set of user-definable locations. There are twelve locations which an application can use in any way. These locations are intended to allow you to track other parts of an application while simultaneously following the kernel activity.

A good example use of this facility would be to instrument the entry and exit of your application’s interrupt service routines. By doing this, you could get a histogram in the HP Software Performance Analyzer of the time spent in any interrupt service routine.

If a write is done to any of these locations, the location is identified (user number and entry or exit), and the captured data is displayed as a hex number and, if possible, translated to ASCII characters. This allows easier debugging.

Note If you are capturing on a range that includes any of the 12 user-defined locations, all of these locations must be written to with integer writes in order for the trace display to work correctly.

RTOS Symbol Names

When your application includes the instrumented service calls, the data area included has many global symbol names. In order to keep these names from conflicting with your application’s symbol names, the symbols all have one of three standard prefixes: "HPOS_", "HP_RTOS_" or "_HPOS_".

The data table contents may be examined in detail by viewing the file "track_os.c". The task table contents may be examined in detail by viewing the file "tables.c". This file will be created in the working directory from which "rtos_edit_vxworks" is executed.

How OS Service Calls are Captured and Displayed

The RTOS Measurement Tool uses the emulation bus analyzer and software performance analyzer to capture operating system software activity in real-time. The captured data is actually a series of memory writes to a data table. These writes can contain encoded information about an OS service call that was just executed or a task switch that just occurred.

When an RTOS action key is pressed in the emulator/analyzer interface, a command file sets up the analyzer to capture the writes to the data table. By setting up the analyzer to capture only writes to selected areas of the data table, you can track specific OS activity or look for a specific sequence of activity.

Inverse Assembler

In the same way that bus cycle information is decoded into assembly language mnemonics in a normal trace display, writes to the data table are decoded into OS service call mnemonics in the RTOS trace display. The software mechanism that decodes information captured by the emulation bus analyzer is called an *Inverse Assembler* (IA).

A short example should help. First, let’s assume the segment of a user’s application that makes an OS service call looks as follows:

```
.
.
SEM_ID my_semaphore;
my_timeout = FOREVER;
.
.
semTake(my_semaphore, my_timeout);
.
.
```

The function "semTake()" is an OS service call that requests a specific semaphore.

Instrumented Library Writes to the Data Table

Because the user has substituted the HP instrumented interface library in place of the original C calls, additional code will execute. This code simply writes information to the data table that identifies the OS service call being executed, the parameters being passed into it, and upon return, writes out the return values from the OS kernel.

Data Table Writes Captured by Analyzer

By clicking on an action key (or running a command file), the emulation bus analyzer is automatically set up to capture memory writes to the data table. The captured data represents the flow of activity into and out of the OS kernel through OS service calls. For the example above, the inverse assembler would decode the captured data and display it as:

```
.
.
-> semTake(SEM_ID=FE68CH, timeout=-1)
<- semTake(STATUS=OK)
.
.
```

Parameters Displayed with Mnemonics

Using the example above, a few more details of inverse assembly can be described. First, you can see that the actual parameter values were captured by the analyzer and are displayed in the trace. Note further that each parameter is preceded by a mnemonic that indicates what the parameter is. The semaphore ID parameter value is preceded with a " SEM_ID =". These are the same parameter mnemonics that the OS vendor uses in their OS manual. This allows very easy interpretation of the trace parameters without needing to reference the OS manual.

MACRO definitions are generally not decoded. (In this example, this status is of such frequent usage that it is decoded mnemonically). Negative decimal numbers are decoded into a number with a minus (-) sign. (That is, a wait forever would be displayed as a -1 rather than a FFFFFFFFH).

Service Call Entry and Exit and Task Switches

Another point of interest is the entry (->) and exit (<-) arrows. This is where an RTOS trace most greatly differs from a normal source code trace.

Since a real-time OS is used in part to manage application execution at a higher level, it has the capability to switch execution from one task to another whenever any OS service call is executed. This may happen for any number of reasons based on changing task priorities, the sending and waiting for messages at queues, or a task using up a given time slice.

Given this behavior, application code that calls an OS service call may not immediately return from that service call but may instead begin executing code in another task. For example, when the "semTake()" OS service call in the previous trace example requested a semaphore, if another task of higher priority was waiting for the same semaphore, then that task would now resume executing and the trace would look something like the following:

```
----Next Task:   (name='Phi4')--------------------------------
-> semTake(SEM_ID=FE68CH, timeout=-1)
---Exited Task: (name='Phi4')--------------------------------
---Next Task:   (name='Phi0')--------------------------------
<- semTake(STATUS=OK)
-> taskDelay(ticks=2)
```

You can see that task Phi4, which sent the semaphore request has now exited and task Phi0, which had been waiting for a semaphore with the "semTake()" OS service call, has now started up again.

This example illustrates a difficulty with tracking semaphores in VxWorks. Since the semTake() return does not identify which semaphore was taken, you need to look at the *call* to semTake() which was made by task Phi0. The semTake() return to a task must be paired up with the corresponding semTake() call for that same task. It should not be assumed that it is the same semaphore that task Phi4 was waiting for. The RTOS displays the activity in the correct time sequence, but it cannot identify the semaphore from the data in the return parameter.

Inverse Assemblers are Tailored to the OS

Note that the examples above use the inverse assembler for the VxWorks real-time OS. Each RTOS Measurement Tool has a unique inverse assembler that is tailored to the particular real-time OS.

Part 3

Installation Guide

Instructions for installing and configuring the product.

Part 3

6

Installation

Installation

This chapter describes the installation of RTOS emulation software that runs on UNIX‰ workstations.

The RTOS emulation product is an extension to the HP 64700 Series emulator and Graphical User Interface (or Softkey Interface) products.

If you have ordered the emulator, interface, and RTOS emulation products together (or just the interface product and the RTOS emulation product), the software products are on the same media. In this case, refer to the installation instructions in your Graphical User Interface *User's Guide*.

If you have ordered the emulator interface and RTOS emulation products separately, install the emulator interface first. Then, install the RTOS emulation product using the instructions in this chapter.

This chapter shows you how to:

- Install HP 9000 software.
- Install Sun SPARCsystem software.

When the Real-Time OS Measurement Tool is installed, you will have an enhanced emulation window with two additional entries available in the **File→Emul700** pulldown menu: **VxWorks Emulator/Analyzer ...** and **VxWorks Performance Analyzer ...**. These entries will bring up a new emulation window and bring up a Performance Analyzer window, each with RTOS action keys defined. You can do anything in these windows that you would normally do.

To install HP 9000 software

Perform the following steps to install HP 64700 Series software on the HP 9000 Workstation:

1 Check the HP-UX operating system version

HP 64700 Series software requires an HP-UX operating system version of 7.03 or greater. To determine the version of your HP-UX operating system, enter the command:

```
# uname -a <RETURN>
```

If the version number of the HP-UX operating system is less than 7.03, you must update the operating system to 7.03 or higher before you can use the RTOS emulation product.

Refer to the "Updating HP-UX" chapter of the *HP-UX System Administration Tasks* manual for detailed information on updating your system.

2 Become the root user on the system you want to update.

3 Make sure the tape's write-protect screw points to SAFE.

4 Put the "HP 64700 Series Products" update tape in the tape drive that will be the "source device".

5 Be sure that the tape drive BUSY and PROTECT lights are on. If either the PROTECT or BUSY light is off, check the tape's write-protect screw or the tape drive for proper operation. The tape drive will condition the tape for about three minutes or less for shorter tapes.

6 When the BUSY light stays off for at least 10 seconds, start the update program by typing:

```
/etc/update
```

7 When the HP-UX Update Utility Main Menu screen appears, make sure that the source and destination devices are correct. The defaults are:

`/dev/update.src` (for Series 300 and 400 Workstations)

`/` (for the destination directory)

8 If you do not use the defaults, change the "source device" and/or "destination directory" as appropriate.

9 Select `Load Everything from Source Media` when your source and destination directories are correct.

10 To begin the update, press the softkey `<Select Item>`. At the next menu, press the softkey `<Select Item>` again. Answer the last prompt with

```
y
```

and press <RETURN>. It takes about 10 minutes to read the tape.

11 When the installation is complete, read /tmp/update.log to see the results of the update.

To install Sun SPARCsystem software

Refer to the *Software Installation Guide* operating notice (included with this binder) for instructions on installing software on Sun SPARCsystem computers.

If you are installing a Graphical User Interface product, refer to the Graphical User Interface *User's Guide* for additional software installation instructions.

Glossary

bucket a portion of a memory area to which information about a particular task is saved.

callout routine a mechanism ("hook") provided by the real-time OS that allows you to execute a routine at certain points in the application, for example, when a task is created or deleted, or when a task switch occurs.

data table the table to which real-time OS information is written while the application executes in real time. The emulation bus analyzer captures writes to the data table and decodes the stored trace information in an easy-to-read display.

device call a service call that communicates with an I/O device.

emulation bus analyzer the analyzer that captures information on the processor bus as programs execute. This analyzer is used to capture writes to the data table which are then decoded to provide RTOS measurement information.

instrumented service call library an interface library with callout routines and wrapper routines that write to the data table and save information in task buckets.

inverse assembler software that decodes hexadecimal machine code values into mnemonics that are easy to read. In the case of the RTOS measurement tool, writes to the data table are decoded into real-time OS mnemonics.

RTOS real-time operating system.

service call a call, made by a task, to a function in the real-time OS kernel.

software performance analyzer an instrument that records information about events that occur during program execution. The software performance analyzer is used to compare time spent in different program modules.

task an independent program or process that executes under the real-time operating system.

Index

Certification and Warranty

Certification

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

Warranty

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.

Exclusive Remedies

The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.