
User's Guide

**HP B3080A
Real-Time OS Measurement
Tool for pSOS+**

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

© Copyright 1992, 1994, Hewlett-Packard Company.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

Microtec is a registered trademark of Microtec Research Inc.

pSOS+ and pROBE+ are trademarks of Integrated Systems Inc.

SunOS, SPARCsystem, OpenWindows, and SunView are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Hewlett-Packard

P.O. Box 2197

1900 Garden of the Gods Road

Colorado Springs, CO 80901-2197, U.S.A.

RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304 U.S.A. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Printing History

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

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. 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 B3080-97000, August 1992

Edition 2 B3080-97001, September 1994

Measurements for the pSOS+ Real-Time Operating System

The screenshot displays the Hewlett Packard Emulator/Analyzer interface for the pSOS+ Real-Time Operating System. It is divided into several panes:

- Top Left Pane:** Shows the main execution window with a menu bar (File, Display, Modify, Execution, Breakpoints, Trace, Settings) and a toolbar. The status bar indicates "M68302--Running user program Emulation trace compl".
- Bottom Left Pane:** A task table listing various tasks with their TID, priority, mode, status, and parameters.

Name	TID	Prio	Mode	Status	Susp?	Parameters
'IDLE'	-#00010000	00	2000	Ready		
'ROOT'	-#00020000	FF	0001	Ready	YES	
'iok'	-#00000000	0C	0006	WkAfter		
'recr'	-#000E0000	09	0006	Ready	YES	
'paal'	-#000F0000	0A	0006	Ewait		EVENTS = 00000001
'silk'	-#00100000	03	0006	Wait		Q = 'cssl' -#0004
'bose'	-#00110000	05	0006	Wait		Q = 'csbo' -#0005
'phnx'	-#00120000	05	0006	Wait		Q = 'cspk' -#0006
'losa'	-#00130000	05	0006	Wait		Q = 'phla' -#0009
'prod'	-#00140000	00	0006	Ready		
'cosp'	-#00150000	08	0006	Running		
'idlp'	-#00160000	07	0004	Ready		
- Top Right Pane:** Shows the "Trace List" for the "Real Time Operating System". It includes a menu bar and a toolbar. The status bar indicates "M68302--Running user program Emulation trace complete".
- Bottom Right Pane:** A histogram titled "Histogram: Interval Duration" showing the distribution of task execution times. The x-axis represents time intervals (0s, 6s, 12s, 18s, 24s, 30s) and the y-axis represents the percentage of tasks. The most frequent interval is 0s, accounting for 25.22% of the tasks.

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 service 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 all 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 size and 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 B3080A Real-Time Operating System Measurement Tool for the pSOS+ Operating System from Integrated Systems Inc.

This book assumes you are familiar with the Emulator/Analyzer interface, whether it be the graphical interface or the terminal emulation based softkey interface.

This book is organized into three parts whose chapters are described below.

Part 1. User's Guide

Chapter 1 explains how to prepare your application to use the RTOS measurement tool.

Chapter 2 describes how to make RTOS measurements in the emulator/analyzer interface.

Chapter 3 describes how to make RTOS measurements in the Software Performance Analyzer interface.

Chapter 4 describes how to access the pROBE+ OS Debugger through a simulated I/O window in the emulator/analyzer interface.

Chapter 5 shows you how to customize the RTOS Measurement Tool.

Part 2. Concept Guide

Chapter 6 describes how the RTOS measurement tool works.

Part 3. Installation Guide

Chapter 7 shows you how to install the RTOS emulation product on HP 9000 Series 300/400/700 computers and on Sun SPARCsystem computers.

Contents

Part 1 User's Guide

1 Preparing Your Application for RTOS Measurements

- Step 1: Make a new source directory 16
- Step 2: Retrieve the RTOS source files 17
- Step 3: Add the RTOS measurement files to your application 19
- Step 4: Build the new application file 20
- Step 5: Open the RTOS emulation window 21
- Step 6: Configure the emulator and load the application 22
- Step 7: Test the RTOS measurement tool 23
- Step 8: Test the Software Performance Analyzer 24

2 Making RTOS Measurements with the Emulator/Analyzer

Tracking the Flow of OS Activity 27

- To track all service calls (including device calls) 29
- To track all service calls plus the stack activity 30
- To track all OS calls before an error occurs 31
- To track everything 32

Tracking Particular OS Service Calls 33

- To track all queue calls 34
- To track all queue calls (include task switches) 35
- To track all event calls 36
- To track all event calls (include task switches) 37
- To track all semaphore calls 38
- To track all semaphore calls (include task switches) 39
- To track a single service call 40
- To track two service calls 41

Tracking Particular Tasks 42

- To track a single task and all OS activity within it 43

Contents

To track four tasks and all OS activity within them	44
To track about a specific task switch	45
To track about a specific task sending a message to a specific queue	46
To trace before an event is received by a specific task	47
To track activity after a function is reached	48
To track activity about the access of a variable by a specific task	49
Tracking Accesses to Functions or Variables	50
To track which tasks access a specific function	51
To track which tasks access a specific variable	52
Tracking Dynamic Memory Usage	53
To track only stack data	54
To track all memory calls (include task switches)	56
Displaying Traces	57
To switch to a normal trace display	58
To switch to the RTOS trace display	59
3 Making RTOS Measurements with the SPA	
Making Time Profile Measurements	64
To define SPA events for tasks, service calls, and user events	64
To display a time histogram of task events	65
To show a table of SPA events	66
To display a count histogram of task events	67
To measure only data from a specific task	68
To show a table of service call invocations	69
To show a normal function duration histogram	70
To show a histogram of task and user events	71
Coordinating Measurements with the Emulator	72
To break on task time overflow	72
To disable the SPA trig2	73
Handling Multiple Projects on One Machine	74
To set up unique SPA windows for multiple projects	74

4 Accessing pROBE+ through Simulated I/O

- To prepare your application for simulated I/O access of pROBE+ 77
- To break pSOS+ execution and enter pROBE+ 80
- To exit pROBE+ and return to RTOS measurements 81

5 Customizing the RTOS Measurement Tool

Creating Your Own RTOS Measurements 85

- Data Table Description 85
- Data Table Contents 89
- To set up trace commands to capture RTOS information 91
- To place your measurements in command files 95
- To place your measurements on action keys 96

Limiting the Intrusion Caused by Instrumented Service Calls 98

- To comment out Level 5 (Id-to-name translation) 99
- To comment out Level 4 (Stack tracking) 99
- To comment out Level 3 (SPA support) 99
- To comment out Level 2 (Overhead, intrusion and error returns) 100
- To comment out Level 1 (Task entry/exit and service calls) 100

Part 2 Concept Guide
6 How the RTOS Measurement Tool Works

- Instrumented Code for Real-Time OS Tracking 105
- Service Call Tracking 105
- Task Switch Tracking 107
- Clock Ticks 107
- Selective Tracking 108
- OS Overhead Tracking 108
- Task and Queue Naming 108
- Stack and Memory Tracking 109
- User-Defined Areas 109
- RTOS Symbol Names 110

Contents

The Data Table	111
Extra Memory Locations	112
How OS Service Calls are Captured and Displayed	113
Inverse Assembler	113
Instrumented Library Writes to the Data Table	113
Data Table Writes Captured by Analyzer	114
Parameters Displayed with Mnemonics	114
Service Call Entry and Exit and Task Switches	115
Inverse Assemblers are Tailored to the OS	115

Part 3 Installation Guide

7 Installation

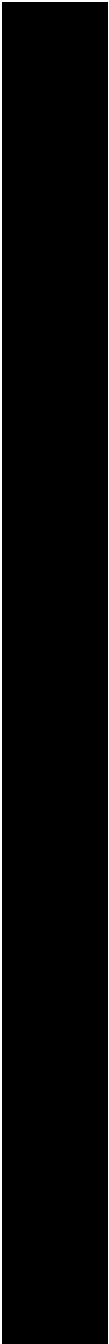
To install HP 9000 software	121
To install Sun SPARCsystem software	123

Part 1

User's Guide


A complete set of task instructions and problem-solving guidelines, with a few basic concepts.

Part 1





Preparing Your Application for RTOS Measurements



Preparing Your Application for RTOS Measurements

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.05.00 or greater.
- Installed the HP B3080 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: **PSOS+ RTOS Measurement Tool ...** and **SPA for pSOS+** If you do not see these new entries, review the installation procedure to make sure it was done correctly, and make sure the /system/B3080/customize script was run. If you still do not see these new entries, contact your Hewlett-Packard representative.

Chapter 1: 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** Add the RTOS measurement files to your application.
- 4** Build the new application file.
- 5** Start the emulator.
- 6** Configure the emulator and load the application.
- 7** Test the RTOS measurement tool.
- 8** 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 `../hertos_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/B3080A directory. If you haven't installed the product, refer to the "Installation" chapter.

During installation, you set the environment variable HP64000 to the directory in which the HP 64000 software has been installed. This directory is "/usr/hp64000" unless you installed the software in a directory other than the root directory.

- 1 Copy the product files into the directory that was created in Step 1. The files are found under \$HP64000/rtos/B3080A. You must copy the following file:

track_os.s

- 2 While in the directory created in Step 1, run the \$HP64000/bin/rtos_edit_psos script.

Doing so creates your application specific "tables.s" file. This assembly language file will contain information that customizes the RTOS tool for your application. This file will be assembled and linked in with your application code. The "rtos_edit_psos" tool asks you whether you wish to edit the file for 16-bit or 32-bit microprocessors.

The "rtos_edit_psos" tool also asks you for the task and queue names in your application. Enter the four letter names of the tasks and message queues you use in your application. These are the names that are defined as parameters to the following OS service calls:

t_create() Create a named task.

q_create() Create a named message queue.

Tables.s allows a "bucket" to be created in memory for each task and message queue entry you define. Information is written to the buckets when task switches and message queue accesses occur.

The "rtos_edit_psos" script also creates a file called "s_init". This is a command file that customizes the Software Performance Analyzer system to your application.

The "rtos_edit_psos" script may be run anytime you wish to add or delete task or queue name information.

Chapter 1: Preparing Your Application for RTOS Measurements

- 3** If you want to access pROBE+ from a simulated I/O window in the emulator/analyzer interface, copy the following files:

io_drivers.c
probe_io.c

And, copy the following files from the \$HP64000/rtos/B3080A/include directory:

simio.h
psos.h

Step 3: Add the RTOS measurement files to your application

- 1 Add "track_os.s" and "tables.s" into your makefile and linker files.

"Track_os.s" contains assembly language code that allows a user to call the pSOS+ OS service call routines from a high-level "C" language. This file also contains special code that writes out RTOS information to the analyzer anytime an OS service call is invoked.

This file *must* replace the pSOS+-to-"C" language interface code previously used in the application.

The data table that resides in "track_os.s" and spans from the symbol "HP_RTOS_TRACK_START" through "HP_RTOS_TRACK_END" only needs to be in an address range that is writeable. Because the data table is never read from, the values written to it don't have to be stored; therefore, no real physical memory is needed.

The pSOS+-to-"C" language interface routines in the file "track_os.s" have been validated with the HP AxLS and the Microtec Research "C" compilers. To use this product with a different compiler, you should edit the "track_os.s" file to match the parameter passing protocol of the desired compiler.

- 2 If you want to make pROBE+ accessible from a simulated I/O window, add "probe_io.c" and "io_drivers.c" to your makefile and use the include files "simio.h" and "psos.h". **Don't forget to change pROBE+ drivers to use the routines in "probe_io.c"**. For more information, refer to the "Accessing pROBE+ through Simulated I/O" chapter.
- 3 Change your pSOS+ configuration table so the task switching callout field, KC_SWITCHCO, has a pointer to the "HPOS_SWITCH_CALLOUT" routine and the task start callout field, KC_STARTCO, has a pointer to "HPOS_START_CALLOUT" routine. (Both routines are defined in "track_os.s".) Refer to your pSOS+ manual for more information on pSOS+ configuration tables.



Step 4: Build the new application file

- Rebuild your application with the new files. The service routines in "track_os.s" have been defined according to the pSOS+ standard so your application should require no changes.

Step 5: Open the RTOS emulation window

- With the emulator/analyzer interface already running, you can open the RTOS emulation window by choosing the **File→Emul700→PSOS+ RTOS Μεασυρεμεντ Τοολ** pulldown menu item.
- If the emulator/analyzer interface is not already running, you can start the RTOS emulation window using the "emulrtos_psos" script found in "\$HP64000/bin". This is a simple script which sets up a few things before calling **emul700** with your given emulator name. The syntax for using this script is:

```
emulrtos_psos [-c <command_file>] PROCESSOR <emulator_name>
```

The PROCESSOR type of your emulator (for example, 68302 or 68020) is needed to run the "emulrtos_psos" script. You can either enter it on the command line or let the script prompt you for it. If you don't want to enter the processor or be prompted for it every time, you may edit the script and assign a value to the variable PROCESSOR.

Opening the RTOS emulation window does several things:

- 1 Action keys are defined for easy "one click" measurements.
- 2 Environment variables are set so the command files related to the action keys are found.
- 3 The PATH variable is set so shell scripts needed by command files will be found.



Step 6: 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.

Step 7: 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 8: 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**→**SPA for PSOS+** pulldown menu item.
- 2 If you wish to make cross-trigger measurements between SPA and the emulation system, make sure the emulation configuration has the following question and answer:

Should Analyzer drive or receive Trig2? receive

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

- 3 In Step 2, when you ran the "rtos_edit_psos" tool, a command file "s_init" should also have been created. If not, rerun "rtos_edit_psos", request only the "s_init" file to be created, and enter the exact task names as given the first time the tool was run.
- 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.



**Making RTOS Measurements with
the Emulator/Analyzer**

Making RTOS Measurements with the Emulator/Analyzer

Action keys for RTOS measurements.

Clock tick.

Service call entry.

Service call exit.

Task switch.

Parameters (decoded if possible).

The screenshot shows the 'Hewlett Packard Emulator/Analyzer: em68302 (m68302)' window. At the top, there is a menu bar with 'File', 'Display', 'Modify', 'Execution', 'Breakpoints', 'Trace', 'Settings', and 'Help'. Below the menu bar is a grid of 'Action keys' for RTOS measurements:

Action keys:	Track OS calls	Track OS +stack	Track Everything	Help RTOS
Only Task X	Only Tsk W,X,Y,Z	Tasks & Queues	Tasks & Events	Tasks & Semaphrs
Only Call X	Only Calls X & Y	Only Queues	Only Events	Only Semaphores
Task switch A->B	Tsk A msg->Que X	Tsk A <- Event X	Task A: FuncX	Task A: VarX
Stack Usage	Before SPA trig2	Trace before Err	? Task: FuncX	? Task: VarX
Memory Usage	Disable SPA trig2	Break to Probe	Disp RTOS Trace	Disp NonRTOS Trc

Below the action keys is a 'Trace List' window. The trace list shows the following entries:

```

Label: Real Time Operating System
Base: with symbols
after NON-RTOS: addr=2EF98 data=00000000
+001 USER DATA #1: data=00000032 ascii=#00000032 820. uS
+003 ++ <CLOCK TICK> 1.2 uS
+004 USER DATA #2: data=00000032 ascii=#00000032 58.64 uS
+005 -> q_send(Qid=00080000:'bopa', 651. uS
      msg = [00000003,00000005,00000001,0003A100])
+018 <- q_send() 157. uS
+020 -> ev_send(Tid=000F0000:'paal', 16.0 uS
      events=B:0001)
+026 STACK BYTES LEFT ON EXIT: Supr 00000100 User 0000010C 149. uS
+034 ---Exiting Task : 'bose'----- 8.76 uS
+036 ---ENTERING TASK: 'paal'----- 11.1 uS
+038 STACK BYTES LEFT ON ENTRY: Supr 00000100 User 000001A8 2.8 uS
+046 <- ev_receive(events=B:0001) 96.0 uS
+050 -> q_receive(qid=00070000:'slpa',NOWAIT) 27.9 uS
+058 <- q_receive( <from 'slpa'> 130. uS
  
```

The status bar at the bottom of the window reads: 'STATUS: M68302--Running user program Emulation trace complete'.

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).

Chapter 2: Making RTOS Measurements with the Emulator/Analyzer Tracking the Flow of OS Activity

Interpreting the measurement output is also very easy. All OS service calls are displayed just as they appear in the OS vendor's manual. Input parameters and return values are decoded into their English language equivalents wherever possible. And, OS specific resources such as task names and mailbox names are decoded into their user-defined ASCII equivalents wherever possible.

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 an event.

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/B3080A/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 value. 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

Chapter 2: Making RTOS Measurements with the Emulator/Analyzer

Tracking the Flow of OS Activity

activity and stops if a failure is ever detected; this helps designers guard against unchecked return values.

This section shows you how to:

- Track all service calls (including device calls).
- Track all service calls plus the stack activity.
- Track all OS calls before an error occurs.
- Track everything.

To track all service calls (including device 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	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=5026 data=00002042	-----
+001	-> rn_create(name='kell', staddr=0003B000, length=000000F0, unit_size=00000020, flags=0[WaitQ=FIFO, delete override=OFF])	627. uS
+011	<- rn_create(rn_id=001B0000, alloc_size=00000060)	234. uS
+017	-> rn_getseg(rn_id=001B0000, size=00000030, NOWAIT)	25.8 uS
+025	<- rn_getseg(seg_addr=0003B0A0)	147. uS
+029	-> q_send(Qid=00090000: 'phla', msg = [00000002, 00000002, 00000001, 0003B0A0])	23.7 uS
+041	<- q_send()	155. uS
+043	-> q_receive(qid=00060000: 'csph', WAIT, FOREVER)	43.84 uS
+051	---Exiting Task : 'phnx'-----	142. uS
+053	---ENTERING TASK: 'losa'-----	11.1 uS
+055	<- q_receive(<from 'phla'> msg=[00000002, 00000002, 00000001, 0003B0A0])	98.7 uS

Service call entry.

Service call exit.

Parameters (decoded if possible).

Task switch.

Return value.

Time stamp.

Note that there are entry and exit arrows on the left of the screen to show when a service call is entered and, on a separate line, to show when a service 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.

As much of the trace information as possible 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 there are a finite number of responses as listed in the OS manual. If the return value at a service call is zero (0), meaning the call was successful, no return value is printed. Any non-zero return values are printed with their English decoding.

To track all service 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	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=prog node_la+B8 data=000065E8	-----
+001	-> q_send(Qid=000A0000: 'lapa', msg = [0000000A, 00000005, 00000001, 0003B0A0])	2.30 mS
+013	<- q_send()	157. uS
+015	-> ev_send(Tid=000F0000: 'paal', events=B:0001)	16.0 uS
+021	STACK BYTES LEFT ON EXIT: Supr 00000100 User 000001C8	149. uS
+029	---Exiting Task : 'losa'-----	8.72 uS
+031	---ENTERING TASK: 'paal'-----	11.1 uS
+033	STACK BYTES LEFT ON ENTRY: Supr 00000100 User 000001A8	2.8 uS
+041	<- ev_receive(events=B:0001)	96.0 uS
+045	-> q_receive(qid=00070000: 'slpa', NOWAIT)	27.9 uS
+053	<- q_receive(<from 'slpa'> msg=[00000000, 00000000, 00000042, 00000011])	131. uS
	** Return code=55: NO PENDING MESSAGE	
+065	-> q_receive(qid=00080000: 'bopa', NOWAIT)	61.36 uS

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 plus show you when the task switches occur and how much stack is left on entering and exiting each task.

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

To track all OS calls before an error occurs

- Click on the **Trace before Err** action key (or run the `e_before_err` command file by entering it on the command line).

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 then can become very difficult to locate.

This command lets you use the analyzer to continuously monitor the system and check if any service call ever fails, even if the developer is not checking that return value.

When the trace completes, you can see the activity that occurred before the failed service call, and the error return value itself is decoded into an easily readable error message as described in the OS kernel manual.

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	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
-056	<- sm_ident(SMid=000C0000)	124. uS
-052	-> sm_v(SMid=000C0000)	12.8 uS
-050	<- sm_v()	106. uS
-048	-> q_send(Qid=00000000:'bopa', msg = [00000006, 00000005, 00000001, 0003A100])	3.18 mS
-036	<- q_send()	158. uS
-034	-> ev_send(Tid=000F0000:'paal', events=B:0001)	16.0 uS
-028	---Exiting Task : 'bose'-----	158. uS
-026	---ENTERING TASK: 'paal'-----	11.1 uS
-024	<- ev_receive(events=B:0001)	98.7 uS
-020	-> q_receive(qid=00070000:'slpa', NOWAIT)	27.8 uS
-012	<- q_receive(<from 'slpa'> msg=[00000000, 00000000, 00000042, 00000011])	130. uS
**	Return code=55: NO PENDING MESSAGE	
before	ERROR CHECK: 55: NO PENDING MESSAGE	22.4 uS

To track everything

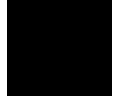
- Click on the **Track Everything** action key (or run the **e_trkall** command file by entering it on the command line).

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=lib lscale+8 data=0000C0C1	-----
+001	USER DATA #1: data=000001F4 ascii=#000001F4	2.34 mS
+003	++ <CLOCK TICK>	1.2 uS
+004	USER DATA #2: data=000001F4 ascii=#000001F4	58.60 uS
+006	-> tm_get()	1.18 mS
+008	<- tm_get(1992 February 14 16:33:14,ticks=231)	98.0 uS
+016	-> q_send(Qid=00060000:'csp', msg = [0000000A,00000005,00000001,00000000])	48.00 uS
+028	STACK BYTES LEFT ON EXIT: Supr 00000100 User 0000019C	145. uS
+036	---Exiting Task : 'cosp'-----	8.76 uS
+038	---ENTERING TASK: 'phnx'-----	11.1 uS
+040	STACK BYTES LEFT ON ENTRY: Supr 00000100 User 000001A8	2.8 uS
+048	<- q_receive(<from 'csp'> msg=[0000000A,00000005,00000001,00000000])	96.0 uS
+060	USER DATA #1: data=000001F4 ascii=#000001F4	1.10 mS
+062	++ <CLOCK TICK>	1.3 uS

This action key is used so that service 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 events, messages, or semaphores. You can also track individual service calls.



This section shows you how to:

- Track all queue calls.
- Track all queue calls (include task switches).
- Track all event calls.
- Track all event calls (include task switches).
- Track all semaphore calls.
- Track all semaphore calls (include task switches).
- Track a single service call.
- Track two service calls.

To track all queue calls

- Click on the **Only Queues** action key (or run the `e_onlyqs` command file by entering it on the command line).

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=prog producer+4A data=0000B880	-----
+001	-> q_send(Qid=00030000:'p2cs', msg = [00000002,00000002,00000001,00000000])	622. uS
+013	<- q_send()	349. uS
+015	-> q_send(Qid=00030000:'p2cs', msg = [00000002,00000002,00000001,00000000])	1.26 mS
+027	<- q_send()	157. uS
+029	-> q_send(Qid=00030000:'p2cs', msg = [00000002,00000002,00000001,00000000])	1.45 mS
+041	<- q_send() ** Return code=53: QUEUE FULL	136. uS
+043	<- q_receive(<from 'p2cs'> msg=[00000001,00000001,00000001,00000000])	249. uS
+055	-> q_send(Qid=00040000:'css1', msg = [00000001,00000001,00000001,00000000])	650. uS
+067	<- q_receive(<from 'css1'>	264. uS

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

To track all queue calls (include task switches)

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

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=prog producer+4A data=0000B880	-----
+001	-> q_send(Qid=00030000:'p2cs', msg = [00000002,00000002,00000001,00000000])	76.36 uS
+013	<- q_send()	157. uS
+015	-> q_send(Qid=00030000:'p2cs', msg = [00000002,00000002,00000001,00000000])	1.45 mS
+027	<- q_send() ** Return code=53: QUEUE FULL	136. uS
+029	---Exiting Task : 'prod'-----	139. uS
+031	---ENTERING TASK: 'cosp'-----	11.2 uS
+033	<- q_receive(<from 'p2cs'> msg=[00000001,00000001,00000001,00000000])	98.7 uS
+045	-> q_send(Qid=00040000:'cssl', msg = [00000001,00000001,00000001,00000000])	649. uS
+057	---Exiting Task : 'cosp'-----	154. uS
+059	---ENTERING TASK: 'sllk'-----	11.1 uS

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

To track all event calls

- Click on the **Only Events** action key (or run the `e_onlyevs` command file by entering it on the command line).

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=2EF9A data=00003AF0	-----
+001	-> ev_send(Tid=000F0000, events=B:0001)	457. uS
+005	<- ev_receive(events=B:0001)	268. uS
+009	-> ev_receive(avs=B:0001, WAIT/AND,FOREVER)	2.33 mS
+015	<- ev_send()	239. uS
+017	-> ev_send(Tid=000F0000, events=B:0001)	2.47 mS
+021	<- ev_receive(events=B:0001)	268. uS
+025	-> ev_receive(avs=B:0001, WAIT/AND,FOREVER)	1.49 mS
+031	<- ev_send()	431. uS
+033	-> ev_send(Tid=000F0000, events=B:0001)	5.85 mS
+037	<- ev_receive(events=B:0001)	268. uS

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

To track all event calls (include task switches)

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

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=lib lscale+8 data=0000C0C1	-----
+001	---Exiting Task : 'prod'-----	33.3 mS
+003	---ENTERING TASK: 'cosp'-----	11.1 uS
+005	---Exiting Task : 'cosp'-----	2.45 mS
+007	---ENTERING TASK: 'sllk'-----	11.1 uS
+009	-> ev_send(Tid=000F0000: 'paal', events=B:0001)	830. uS
+015	---Exiting Task : 'sllk'-----	158. uS
+017	---ENTERING TASK: 'paal'-----	11.2 uS
+019	<- ev_receive(events=B:0001)	98.7 uS
+023	-> ev_receive(avs=B:0001, WAIT/AND, FOREVER)	2.66 mS
+029	---Exiting Task : 'paal'-----	129. uS
+031	---ENTERING TASK: 'sllk'-----	11.1 uS
+033	<- ev_send()	98.5 uS
+035	---Exiting Task : 'sllk'-----	199. uS

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

The display shows that task 'paal' is receiving event signals from the other tasks.

To track all semaphore calls

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

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=2EAA8 data=0000E990	-----
+001	-> sm_ident(name='sem1',node=00000000)	3.28 mS
+005	<- sm_ident(SMid=000C0000)	124. uS
+009	-> sm_p(SMid=000C0000,NOWAIT)	19.2 uS
+015	<- sm_p()	111. uS
	** Return code=66: SEMAPHORE NOT AVAILABLE	
+017	-> sm_ident(name='sem1',node=00000000)	8.85 mS
+021	<- sm_ident(SMid=000C0000)	124. uS
+025	-> sm_p(SMid=000C0000,NOWAIT)	19.2 uS
+031	<- sm_p()	111. uS
	** Return code=66: SEMAPHORE NOT AVAILABLE	
+033	-> sm_ident(name='sem1',node=00000000)	4.27 mS
+037	<- sm_ident(SMid=000C0000)	124. uS
+041	-> sm_p(SMid=000C0000,NOWAIT)	19.2 uS
+047	<- sm_p()	111. uS
	** Return code=66: SEMAPHORE NOT AVAILABLE	

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 & Semaphrs** action key (or run the `e_tracksms` command file by entering it on the command line).

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=lib lscale+C data=00002042	-----
+001	---Exiting Task : 'paal'	470. uS
+003	---ENTERING TASK: 'bose'	11.1 uS
+005	---Exiting Task : 'bose'	298. uS
+007	---ENTERING TASK: 'cosp'	11.2 uS
+009	---Exiting Task : 'cosp'	1.07 mS
+011	---ENTERING TASK: 'sllk'	11.1 uS
+013	---Exiting Task : 'sllk'	627. uS
+015	---ENTERING TASK: 'paal'	11.2 uS
+017	-> sm_ident(name='sem1',node=00000000)	517. uS
+021	<- sm_ident(SMid=000C0000)	124. uS
+025	-> sm_p(SMid=000C0000,NOWAIT)	19.2 uS
+031	<- sm_p()	111. uS
	** Return code=66: SEMAPHORE NOT AVAILABLE	
+033	---Exiting Task : 'paal'	652. uS
+035	---ENTERING TASK: 'sllk'	11.1 uS

This action key is used if you are only concerned about semaphore calls and the task context.

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 in all lower-case characters.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=2F6D0 data=00000000	-----
+001	<- q_send()	1.57 mS
+003	-> q_send(Qid=00040000, msg = [0000000B, 00000005, 00000001, 00000000])	5.84 mS
+013	-> q_send(Qid=00070000, msg = [0000000B, 00000005, 00000001, 00000000])	1.86 mS
+023	<- q_send()	157. uS
+025	<- q_send()	7.42 mS
+027	-> q_send(Qid=00060000, msg = [00000005, 00000001, 00000001, 00000000])	2.84 mS
+037	-> q_send(Qid=00090000, msg = [00000005, 00000001, 00000001, 0003B0A0])	3.43 mS
+047	<- q_send()	155. uS
+049	-> q_send(Qid=000A0000, msg = [00000005, 00000001, 00000001, 0003B0A0])	3.50 mS
+059	<- q_send()	157. uS

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 two service calls

- Click on the **Only Calls X & Y** action key (or run the `e_twocalls` command file by entering it on the command line).

You are prompted for the names of the two service calls you wish to track. Enter the service call names in all lower-case characters.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
sq adv <- q_receive(msg=[00000002,00000005,00000001,00000000])		264. uS
sq adv -> q_send(Qid=00070000, msg = [00000002,00000005,00000001,00000000])		316. uS
sq adv <- q_send()		157. uS
sq adv -> q_receive(qid=00070000, NOWAIT)		444. uS
sq adv <- q_receive(msg=[00000002,00000005,00000001,00000000])		146. uS
sq adv -> q_receive(qid=00040000, WAIT, FOREVER)		1.59 mS
sq adv <- q_send()		269. uS
sq adv -> q_receive(qid=00030000, WAIT, FOREVER)		27.4 uS
sq adv <- q_receive(msg=[00000004,00000005,00000001,00000000])		145. uS
sq adv -> q_send(Qid=00060000, msg = [00000004,00000005,00000001,00000000])		2.19 mS
sq adv <- q_receive(msg=[00000004,00000005,00000001,00000000])		264. uS

You may track just the relationship between two service calls with this action key. For example, the above trace shows who is sending messages with "q_send" and who is receiving them with "q_receive".

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 mailbox; 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 OS activity within it.
- Track four tasks and all OS activity within them.
- Track about a specific task switch.
- Track about a specific task sending a message to a specific queue.
- Trace before an event is received by a specific task.
- Track activity after a function is reached.
- Track activity about the access of a variable by a specific task.

To track a single task and all 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 four letter name of the task you are interested in, or in the graphical interface, by using the cut buffer, you can cut and paste a task name from the screen into the dialog box.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
+001	<- q_receive(<from 'p2cs' msg=[00000001,00000001,00000001,00000000])	98.7 uS
+013	-> tm_get()	503. uS
+015	<- tm_get(1992 February 14 16:40:22,ticks=307)	98.0 uS
+023	-> q_send(Qid=00060000:'csph', msg = [00000001,00000001,00000001,00000000])	47.76 uS
+035	---Exiting Task : 'cosp'-----	154. uS
sq adv	---ENTERING TASK: 'cosp'-----	5.73 mS
+039	<- q_send()	98.5 uS
+041	-> q_receive(qid=00030000:'p2cs',WAIT,FOREVER)	27.4 uS
+049	<- q_receive(<from 'p2cs' msg=[00000001,00000001,00000001,00000000])	145. uS
+061	-> tm_get()	503. uS
+063	<- tm_get(1992 February 14 16:40:22,ticks=309)	98.0 uS
+071	-> q_send(Qid=00050000:'csbo', msg = [00000001,00000001,00000001,00000000])	47.64 uS

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. In this example, the elapsed time was 5.73 milliseconds.

To track four tasks and all 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	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
+001	<- q_send()	98.5 uS
+003	-> q_receive(qid=00030000:'p2cs',WAIT,FOREVER)	27.4 uS
+011	<- q_receive(<from 'p2cs'> msg=[00000006,00000005,00000001,00000000])	145. uS
+023	-> tm_get()	2.98 mS
+025	<- tm_get(1992 February 14 16:42:27,ticks=84)	98.0 uS
+033	-> q_send(Qid=00040000:'css1', msg = [00000006,00000005,00000001,00000000])	48.00 uS
+045	---Exiting Task : 'cosp'-----	154. uS
sq adv	---ENTERING TASK: 'paal'-----	1.44 mS
+049	<- ev_receive(events=B:0001)	98.7 uS
+053	-> q_receive(qid=00070000:'slpa',NOWAIT)	27.9 uS
+061	<- q_receive(<from 'slpa'> msg=[00000006,00000005,00000001,00000000])	146. uS
+073	-> sm_ident(name='sem1',node=00000000)	51.52 uS
+077	<- sm_ident(SMid=000C0000)	124. uS

You can use this command to track 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 "cosp" and "bose", enter "cosp" in the first dialog box and "bose" 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.

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	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
-019	-> q_send(Qid=00030000:'p2cs', msg = [00000001,00000001,00000001,00000001])	656. uS
-007	<- q_send() ** Return code=53: QUEUE FULL	136. uS
-005	-> tm_wkafter(ticks=00000012)	16.5 uS
sq adv	---Exiting Task : 'prod'-----	122. uS
sq adv	---ENTERING TASK: 'cosp'-----	11.2 uS
+001	<- q_receive(<from 'p2cs'> msg=[00000009,00000005,00000001,00000001])	98.7 uS
+013	-> tm_get()	4.54 mS
+015	<- tm_get(1992 February 14 16:43:37,ticks=208)	98.0 uS
+023	-> q_send(Qid=00060000:'csph', msg = [00000009,00000005,00000001,00000001])	48.00 uS
+035	---Exiting Task : 'cosp'-----	154. uS
+037	---ENTERING TASK: 'phnx'-----	11.1 uS
+039	<- q_receive(<from 'csph'>	98.8 uS

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

- Click on the **Tsk A msg->Que X** action key (or run the `e_tsk2queue` 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	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
sq adv	---ENTERING TASK: 'cosp'-----	11.1 uS
-032	<- q_send()	98.5 uS
-030	-> q_receive(qid=00030000:'p2cs',WAIT,FOREVER)	27.4 uS
-022	<- q_receive(<from 'p2cs'> msg=[00000000,00000005,00000001,00000000])	145. uS
-010	-> tm_get()	4.04 mS
-008	<- tm_get(1992 February 14 16:45:18,ticks=299)	98.0 uS
about	-> q_send(Qid=00060000:'csph', msg = [00000000,00000005,00000001,00000000])	47.88 uS
+012	---Exiting Task : 'cosp'-----	154. uS
+014	---ENTERING TASK: 'phnx'-----	11.1 uS
+016	<- q_receive(<from 'csph'> msg=[00000000,00000005,00000001,00000000])	98.8 uS
+028	-> rn_create(name='kell',staddr=0003B000 length=00000F0,unit_size=00000020, flags=0[WaitQ=FIFO,delete override=OFF])	4.17 mS

This measurement is useful if you have a task that sends a message to a specific 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 before an event is received by a specific task

- Click on the **Tsk A <- Event X** action key (or run the `e_tskrcv_ev` command file by entering it on the command line).

You are prompted first for the task name and then for the numeric value designating the event(s). The event number may be entered in decimal, hexadecimal, or binary, the latter two being followed by "h" and "b", respectively. These numeric entries may also include don't care values such as 10XX0X11b.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
	msg = [00000009,00000005,00000001,0003A100]	
-015	<- q_send()	157. uS
-013	-> ev_send(Tid=000F0000:'paal', events=B:0001)	16.0 uS
-007	---Exiting Task : 'bose'-----	158. uS
sq adv	---ENTERING TASK: 'paal'-----	11.1 uS
-003	<- ev_receive(events=B:0001)	98.7 uS
+001	-> q_receive(qid=00070000:'slpa',NOWAIT)	220. uS
+009	<- q_receive(<from 'slpa'> msg=[00000000,00000000,00000042,00000011])	131. uS
	** Return code=55: NO PENDING MESSAGE	
+021	-> q_receive(qid=00080000:'bopa',NOWAIT)	61.36 uS
+029	<- q_receive(<from 'bopa'> msg=[00000009,00000005,00000001,0003A100])	146. uS
+041	-> sm_ident(name='sem1',node=00000000)	51.52 uS
+045	<- sm_ident(SMid=000C0000)	124. uS

This measurement allows you to view the context under which a specific event is received by a specific task. In the above example, we have captured a trace when task "paal" received event 0001.

To track activity after a function is reached

- 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 and then the desired function.

Trace List	Offset=0	More data off screen
Label:	Source Lines Only	time count
Base:		relative
after	##/lsd/rtos/psos/demo_appl/root.c - line 309 thru 318 #	240 nS
	int node;	
	/**/	
	/* A function to be called by each node.	
	/**/	
	{	
+019	##/lsd/rtos/psos/demo_appl/root.c - line 319 thru 321 #	4.76 uS
	static int i = 0, j = 0, k = 0, l = 0, m = 0;	
	if (node == CS_NODE)	
+022	##/lsd/rtos/psos/demo_appl/root.c - line 326 thru 327 #	880 nS
	if (node == SL_NODE)	
+026	##/lsd/rtos/psos/demo_appl/root.c - line 332 thru 333 #	1.4 uS
	if (node == BO_NODE)	

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.

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		Offset=0		More data off screen	
Label:	Address	Opcode or Status w/	Source Lines	time	count
Base:	symbols	mnemonic w/symbols	relative		
#####/lsd/rtos/psos/demo_appl/node_bo.c - line 60 thru 62 #####					
/* Note sending by event */					
ev_send(palocalto_tid, MSG_TO_DEST);					
-009	p node_bo+0000F4	7001	uprog rd word	240	nS
-008	p node_bo+0000F6	2040	uprog rd word	240	nS
-007	p node_bo+0000F8	PER.L [A0]		520	nS
-006	p node_bo+0000FA	MOVEA.L da _palocalto_tid,A0		240	nS
-005	p node_bo+0000FC	0000	uprog rd word	240	nS
-004	02EF98	0000	udata wr word	240	nS
-003	02EF9A	0001	udata wr word	280	nS
-002	p node_bo+0000FE	BB42	uprog rd word	240	nS
-001	p node_bo+000100	PER.L [A0]		240	nS
about	da _palocalto_tid	000F	udata rd word	240	nS
+001	_palocalto+000002	0000	udata rd word	280	nS
+002	p node_bo+000102	JSR	code _ev_send	240	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.

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.
- Track which tasks access a specific variable.

To track which tasks access a specific function

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

You are prompted for the function name.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=probe.pio_consts data=00004E56	-----
+001	NON-RTOS: addr=.common_function data=00004E56	413. uS
pstore	---ENTERING TASK: 'losa'-----	
+004	NON-RTOS: addr=.common_function data=00004E56	4.07 mS
pstore	---ENTERING TASK: 'phnx'-----	
+007	NON-RTOS: addr=.common_function data=00004E56	2.74 mS
pstore	---ENTERING TASK: 'losa'-----	
+010	NON-RTOS: addr=.common_function data=00004E56	4.26 mS
pstore	---ENTERING TASK: 'bose'-----	
+013	NON-RTOS: addr=.common_function data=00004E56	5.30 mS
pstore	---ENTERING TASK: 'sllk'-----	
+016	NON-RTOS: addr=.common_function data=00004E56	3.41 mS
pstore	---ENTERING TASK: 'bose'-----	
+019	NON-RTOS: addr=.common_function data=00004E56	5.31 mS
pstore	---ENTERING TASK: 'sllk'-----	
+022	NON-RTOS: addr=.common_function data=00004E56	3.41 mS

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

To track which tasks access a specific variable

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

You are prompted for the variable name.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=lib lscale+C data=00002042	-----
pstore	---ENTERING TASK: 'cosp'	-----
+003	NON-RTOS: addr=data _p2cs_qid data=00000003	8.62 mS
pstore	---ENTERING TASK: 'cosp'	-----
+006	NON-RTOS: addr=data _p2cs_qid data=00000003	23.7 mS
pstore	---ENTERING TASK: 'cosp'	-----
+009	NON-RTOS: addr=data _p2cs_qid data=00000003	15.9 mS
pstore	---ENTERING TASK: 'cosp'	-----
+012	NON-RTOS: addr=data _p2cs_qid data=00000003	17.9 mS
pstore	---ENTERING TASK: 'cosp'	-----
+015	NON-RTOS: addr=data _p2cs_qid data=00000003	23.4 mS
pstore	---ENTERING TASK: 'prod'	-----
+018	NON-RTOS: addr=data _p2cs_qid data=00000003	6.32 mS
+019	NON-RTOS: addr=data _p2cs_qid data=00000003	6.10 mS
+020	NON-RTOS: addr=data _p2cs_qid data=00000003	6.10 mS
+021	NON-RTOS: addr=data _p2cs_qid data=00000003	812. uS

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

Tracking Dynamic Memory Usage

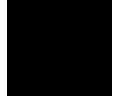
Tracking dynamic memory usage has always been difficult in an embedded design. With these new real-time operating system measurement tools, however, even these debugging headaches become easy to solve.

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 and gives an error if a corrupt pointer is ever detected. This allows you to detect memory allocation problems.

Stack allocation information (that is, size and stack pointer) are 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).

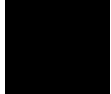
You can enter this command before you run your application from its startup address to capture the initialization of the application which shows you where each local stack is allocated and how large it is.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=1EFFE data=0000100A	-----
+001	STACKS: 'IDLE' Supr base=00039FB6 User base=FFFFFFFF size=00000000 size=00000000	776. mS
+011	STACKS: 'ROOT' Supr base=00035FCA User base=00031FCC size=00000000 size=00000000	482. uS
+021	STACK BYTES LEFT ON EXIT: Supr 00000000 User 00000000	122. uS
+029	---Exiting Task : 'ROOT'-----	8.76 uS
+031	---ENTERING TASK: 'ROOT'-----	11.1 uS
+033	STACK BYTES LEFT ON ENTRY: Supr 00000000 User 00000000	2.8 uS
+041	STACKS: 'iotk' Supr base=00030FCA User base=00030ECC size=00000100 size=00000200	3.59 mS
+051	STACKS: 'recr' Supr base=00030CCA User base=00030BCC size=00000100 size=00001500	815. uS
+061	STACKS: 'paal' Supr base=0002F6CA User base=0002F5CC size=00000100 size=00000200	755. uS
+071	STACKS: 'sllk' Supr base=0002F3CA User base=0002F2CC	838. uS

Chapter 2: Making RTOS Measurements with the Emulator/Analyzer Tracking Dynamic Memory Usage

If you perform this same measurement while the application is running, you see the amount of stack remaining every time a task switch occurs. This gives you a quick indication of potential stack usage problems.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=9 psos_68k+6E6 data=0000630C	-----
+001	STACK BYTES LEFT ON EXIT: Supr 00000100 User 0000019C	62.88 uS
+009	---Exiting Task : 'cosp'-----	8.76 uS
+011	---ENTERING TASK: 'phnx'-----	11.1 uS
+013	STACK BYTES LEFT ON ENTRY: Supr 00000100 User 000001A8	2.7 uS
+021	STACK BYTES LEFT ON EXIT: Supr 00000100 User 000001A8	2.10 mS
+029	---Exiting Task : 'phnx'-----	8.76 uS
+031	---ENTERING TASK: 'losa'-----	11.1 uS
+033	STACK BYTES LEFT ON ENTRY: Supr 00000100 User 000001B0	2.8 uS
+041	STACK BYTES LEFT ON EXIT: Supr 00000100 User 000001C8	1.93 mS
+049	---Exiting Task : 'losa'-----	8.76 uS
+051	---ENTERING TASK: 'paal'-----	11.1 uS
+053	STACK BYTES LEFT ON ENTRY: Supr 00000100 User 000001A8	2.8 uS
+061	STACK BYTES LEFT ON EXIT: Supr 00000100 User 000001A8	2.77 mS
+069	---Exiting Task : 'paal'-----	8.76 uS
+071	---ENTERING TASK: 'losa'-----	11.1 uS



To track all memory calls (include task switches)

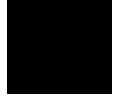
- Click on the **Memory Usage** action key (or run the `e_memory` command file by entering it on the command line).

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=prog mainroute+78 data=00004E71	-----
+001	---Exiting Task : 'cosp'-----	1.53 mS
+003	---ENTERING TASK: 'bose'-----	11.1 uS
+005	-> pt_create(name='horn', Paddr=0003A000 Laddr=00000000, length=00001000, buf_siz=00000100 flags=0[export=LOCAL, delete override=DISABLE])	162. uS
+017	<- pt_create(PTid=00170000, Nbuf=0000000F)	271. uS
+023	-> pt_getbuf(PTid=00170000)	18.2 uS
+025	<- pt_getbuf(Buf_addr=0003A100)	118. uS
+029	---Exiting Task : 'bose'-----	3.29 mS
+031	---ENTERING TASK: 'paal'-----	11.1 uS
+033	-> pt_ident(name='horn', node=00000000)	802. uS
+037	<- pt_ident(PTid=00170000)	124. uS
+041	-> pt_retbuf(PTid=00170000, Buf_addr=0003A100)	15.8 uS
+045	<- pt_retbuf()	124. uS
+047	-> pt_delete(PTid=00170000)	9.76 uS

This command simply tracks all service calls for regions or partitions, giving you an idea of general 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).

Trace List		Offset=0		More data off screen	
Label:	Address	Opcode or Status w/ Source Lines		time count	
Base:	symbols	mnemonic w/symbols		relative	
after	HPOS_T_EXIT_STAC	0002	sdata wr word		
+001	HPOS_T_EX+000002	E4CA	sdata wr word	240	nS
+002	HPOS_T_STACK_VAR	0002	sdata wr word	1.8	uS
+003	HPOS_T_ST+000002	E3CA	sdata wr word	240	nS
+004	HPOS_T_STACK_VAR	0002	sdata wr word	2.5	uS
+005	HPOS_T_ST+000002	E368	sdata wr word	240	nS
+006	HPOS_T_STACK_VAR	0002	sdata wr word	1.8	uS
+007	HPOS_T_ST+000002	E1CC	sdata wr word	240	nS
+008	HP_RTOS_TRACK_ST	636F	sdata wr word	1.8	uS
+009	HP_RTOS_T+000002	7370	sdata wr word	240	nS
+010	HPOS_TASK_ENTRY	626F	sdata wr word	10.9	uS
+011	HPOS_TASK+000002	7365	sdata wr word	240	nS
+012	HPOS_T_ENTRY_STA	0002	sdata wr word	2.5	uS
+013	HPOS_T_EN+000002	F0CA	sdata wr word	240	nS
+014	HPOS_T_STACK_VAR	0002	sdata wr word	1.8	uS
+015	HPOS_T_ST+000002	EFCA	sdata wr word	240	nS

Writes to the data table.

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).

The screenshot shows a trace window with the following content:

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	STACK BYTES LEFT ON EXIT: Supr 00000100 User 0000019C	-----
+008	---Exiting Task : 'cosp'-----	8.76 uS
+010	---ENTERING TASK: 'bose'-----	11.1 uS
+012	STACK BYTES LEFT ON ENTRY: Supr 00000100 User 000001A4	2.8 uS
+020	USER DATA #1: data=00000000 ascii=#0	97.2 uS
+022	++ <CLOCK TICK>	1.2 uS
+023	USER DATA #2: data=00000000 ascii=#0	58.64 uS
+025	<- q_receive(<from 'csbo'> msg=[00000005,00000005,00000001,00000000])	131. uS
+037	-> pt_create(name='horn',Paddr=0003A000 Laddr=00000000,length=00001000,buf_siz=00000100 flags=0[export=LOCAL,delete override=DISABLE])	63.48 uS
+049	<- pt_create(PTid=00220000,Nbuf=0000000F)	271. uS
+055	-> pt_getbuf(PTid=00220000)	18.2 uS
+057	<- pt_getbuf(Buf_addr=0003A100)	118. uS
+061	-> sm_ident(name='sem1',node=00000000)	26.4 uS

Annotations on the left side of the screenshot:

- Task switch. (points to lines +010 and +012)
- Service call entry. (points to line +025)
- Service call exit. (points to line +049)
- Parameters (decoded if possible). (points to the parameters in line +049)
- Return value. (points to the return value in line +049)
- Time stamp. (points to the time stamp in line +049)

Note that there are entry and exit arrows on the left of the screen to show when a service call is entered and, on a separate line, to show when a service 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.

As much of the trace information as possible 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 there are a finite number of responses as listed in the OS manual.

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.



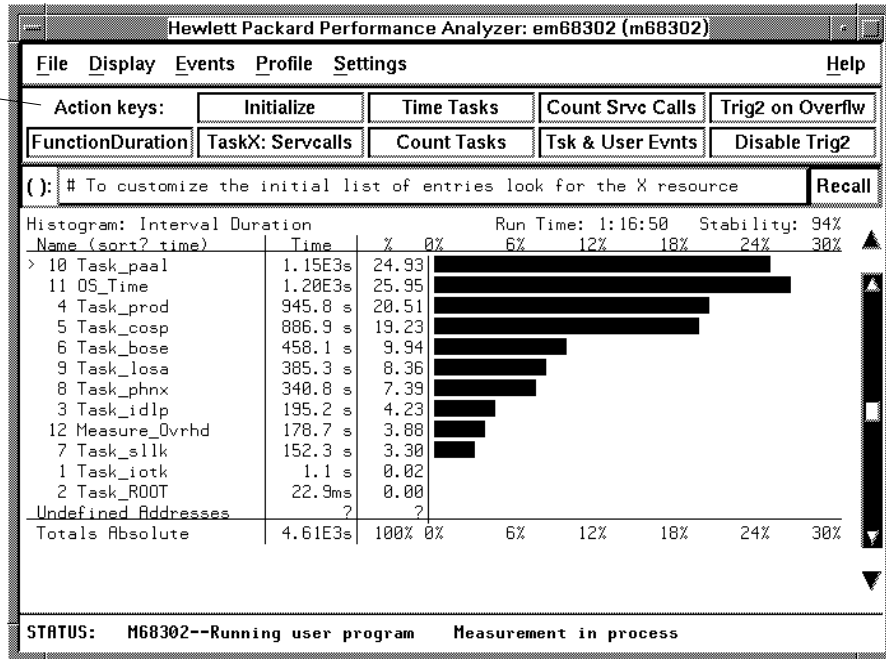
3



**Making RTOS Measurements with
the SPA**

Making RTOS Measurements with the SPA

Action keys for RTOS measurements.



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:

Chapter 3: Making RTOS Measurements with the SPA

- 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 service call invocations.
- Show a normal function duration histogram.
- Show a histogram of task and user events.

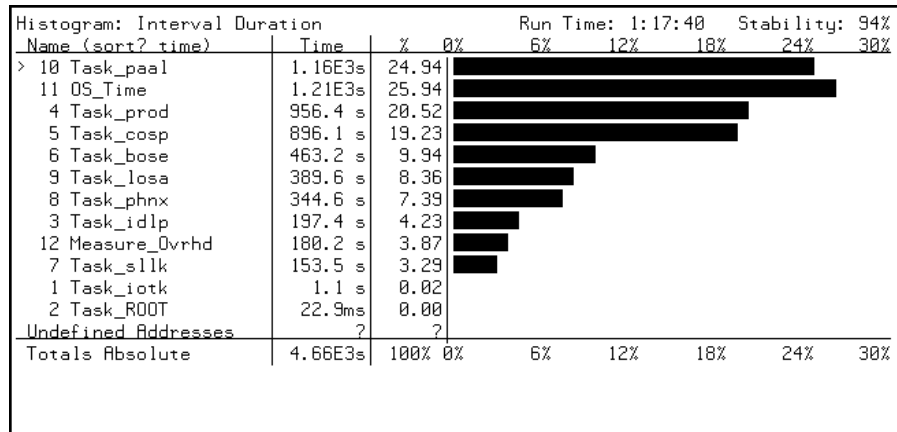
To define SPA events for tasks, service calls, and user events

- 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_psos".

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).



You see that the task names are listed in SPA, and a histogram showing the amount of time each task is taking is being displayed. 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 how much intrusion is caused by the RTOS measurement tool routines. The amount of time spent in measurement overhead caused by the RTOS tool is typically around 1%. The intrusion percentage is controllable by commenting out code in the "track_os.s" file (refer to the "Limiting Intrusion Caused by Instrumented Service Calls" section of the "Customizing the RTOS Measurement Tool" chapter). The example above displays an extreme case where our demo code uses about 4% measurement intrusion overhead time.

To show a table of SPA events

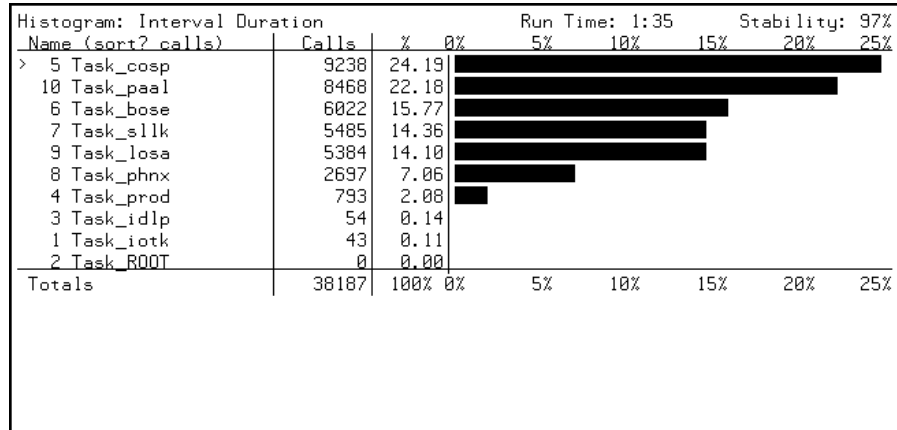
- Choose the **Display**→**Table** pulldown menu item (or enter the **display table** command on the command line).

A raw numbers view of the accumulated data is displayed.

Table: Interval Duration				Run Time: 1:18:27		Stability: 94%	
Name (sort? time)	Calls	Time	Time %	Max	Min	Mean	Std Dev
> 10 Task_paal	406803	1.17E3s	24.94	12.4ms	154.4us	2.9ms	1.7ms
11 OS_Time	1.11E07	1.22E3s	25.94	53.5 s	37.4us	109.9us	16.1ms
4 Task_prod	38638	966.2 s	20.52	60.2ms	154.4us	25.0ms	14.5ms
5 Task_cosp	444736	904.8 s	19.22	53.5 s	154.4us	2.0ms	80.3ms
6 Task_bose	291643	467.9 s	9.94	12.0ms	154.4us	1.6ms	1.8ms
9 Task_losa	256626	393.6 s	8.36	11.6ms	154.4us	1.5ms	1.6ms
8 Task_phnx	128384	348.2 s	7.40	11.8ms	210.0us	2.7ms	1.5ms
3 Task_idlp	2692	199.5 s	4.24	103.2ms	300.5us	74.1ms	17.8ms
12 Measure_Ovrhd	7.13E06	182.6 s	3.88	283.4us	7.2us	25.6us	20.0us
7 Task_sllk	264678	155.6 s	3.30	3.3ms	154.4us	587.8us	377.7us
1 Task_iotk	2345	1.1 s	0.02	198.8ms	266.0us	472.7us	5.8ms
2 Task_ROOT	2	22.9ms	0.00	11.5ms	11.5ms	11.5ms	21.5us
Undefined Addresses	?	?	?				
Totals Absolute	2.01E07	4.71E3s	100%				

To display a count histogram of task events

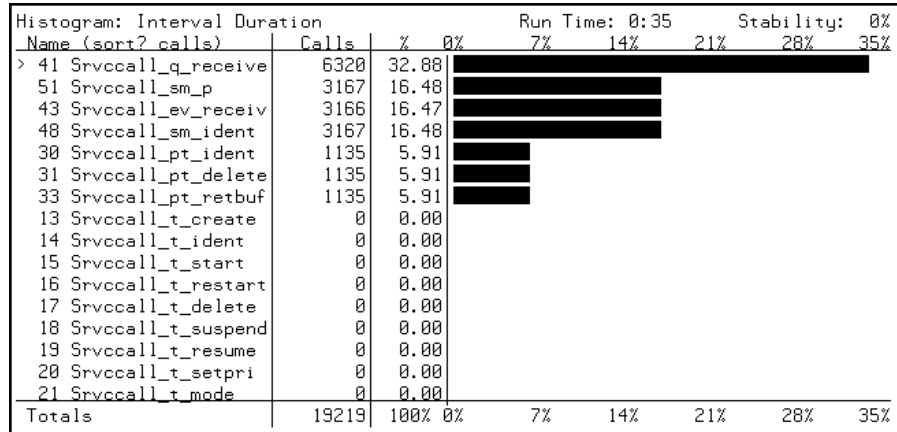
- Click on the **Count Tasks** action key (or run the `s_counttasks` command file by entering it on the command line).



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

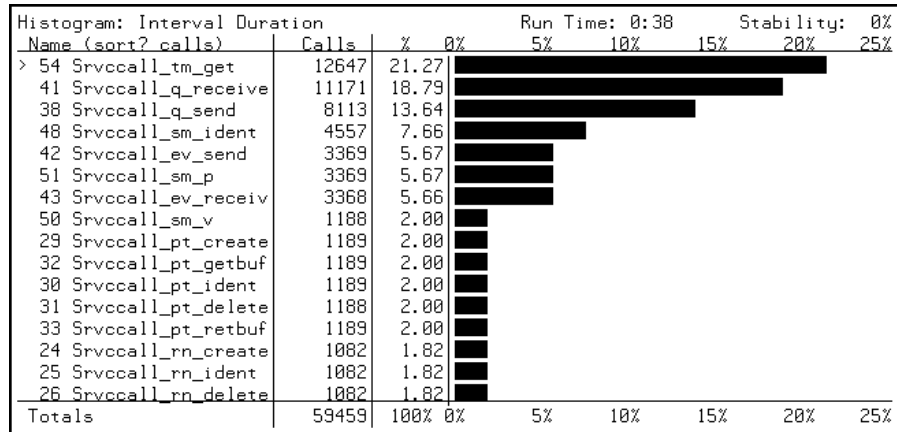
- Click on the **TaskX: Servcalls** action key (or run the `s_taskwindow` command file by entering it on the command line).



This displays a histogram of the number of times each service 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).



This displays a histogram of the number of times each service 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).

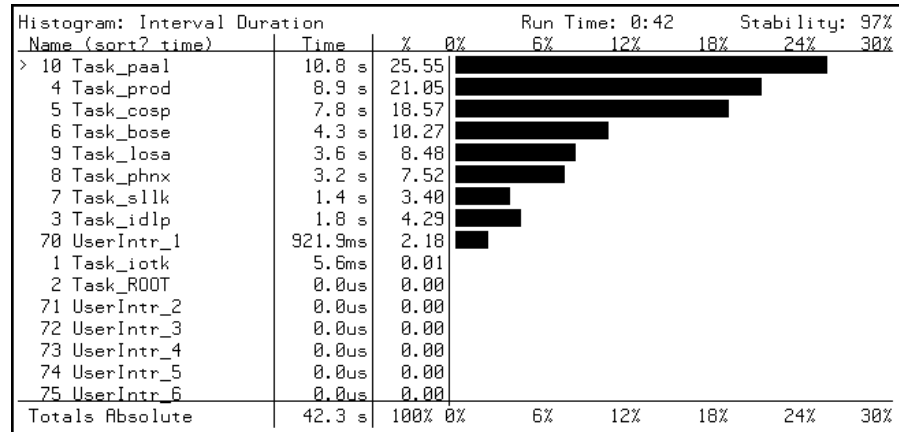


Histogram: Function Duration exclude calls			Run Time: 1:06	Stability: 100%				
Name (sort? time)	Time	%	0%	1%	2%	3%	4%	5%
>104 pio_consts	1.1 s	1.72						
76 atof	0.0us	0.00						
77 strtod	0.0us	0.00						
78 _dbl_to_str	0.0us	0.00						
79 palocalto	0.0us	0.00						
80 _doprnt	0.0us	0.00						
81 _doscan	0.0us	0.00						
82 clear_simio_screen	0.0us	0.00						
83 fill_response_stri	0.0us	0.00						
84 input	0.0us	0.00						
85 read_write	0.0us	0.00						
86 close_driver	0.0us	0.00						
87 nop1	0.0us	0.00						
88 nop2	0.0us	0.00						
89 open_driver	0.0us	0.00						
90 read_driver	0.0us	0.00						
Totals Absolute	66.4 s	100%	0%	1%	2%	3%	4%	5%

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_taskuser` command file by entering it on the command line).



This measurement includes any user-defined events you may have set up. The example above shows that user event "UserIntr_1" uses greater than 1% of the system time.

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 Overflow** action key.

Chapter 3: Making RTOS Measurements with the SPA Coordinating Measurements with the Emulator

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

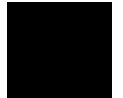
- If more than one project is using the RTOS Measurement Tool, you need to make sure the **Initialize** action key calls a command file specific to your currently loaded application.

There is a semi-automated way to have unique SPA windows which run application specific command files:

- 1 Re-run the \$HP64000/bin/rtos_edit_psos script.
- 2 Answer "y" only to the question "Do you want to create a new 's_init' file?".
- 3 Enter your task names from your application.
- 4 Answer "y" to the question "Do you need to customize the SPA environment?".
- 5 Enter a short unique string (your initials are suggested) for a suffix.
- 6 You must set the environment variable RTOS_UNIQUE to the unique string you just entered. When this environment variable is set, the RTOS tools know to use the specific command file that has been created.

Don't forget to always have RTOS_UNIQUE set in any window in which you run the RTOS product.

4



Accessing pROBE+ through Simulated I/O

Accessing pROBE+ through Simulated I/O

A pSOS+ compatible static OS debugger, called pROBE+, is also available from Integrated Systems, Inc. Supplied with the RTOS product is a connection library that allows you to run pROBE+ through the simulated I/O window in the emulator/analyzer interface, thereby eliminating the need for a separate hardware I/O device to connect to pROBE+.

pROBE+ is a static debugger that is a complementary product with the real-time "dynamic" RTOS measurement tools. With the real-time debugger you can capture flow-of-information in a dynamic, real-time mode. The static OS debugger can be used to browse through the internal OS resource lists such as mailbox contents or task status lists when running in the OS-resident debug monitor.

To help you access pROBE+ from a simulated I/O window, the files "io_drivers.c" and "probe_io.c" (found in the \$HP64000/rtos/B3080A directory) are included with the RTOS measurement tool.

The "io_drivers.c" file contains routines for using simulated I/O in the emulator/analyzer interface.

The "probe_io.c" file contains user-supplied initialization and console procedures that must be linked in with the application and identified in pROBE+'s configuration table.

You also need the include files "simio.h" and "psos.h" (found in the \$HP64000/rtos/B3080A/include directory).

This chapter shows you how to:

- Prepare your application for simulated I/O access of pROBE+.
- Break pSOS+ execution and enter pROBE+.
- Exit pROBE+ and return to RTOS measurements.

To prepare your application for simulated I/O access of pROBE+

To integrate pROBE+ into your application and have it be available through through the simulated I/O window, you must do the following:

- 1 Put the simulated I/O drivers into the application's I/O jump table.

For example:

```

_XDEF      _DRV_TBL
_DRV_TBL:
XREF _simio_init
XREF _open_driver
XREF _close_driver
XREF _read_driver
XREF _write_driver
XREF _simio_clear_screen

DC.L _simio_init      ;init
DC.L _open_driver    ;open
DC.L _close_driver   ;close
DC.L _read_driver    ;read
DC.L _write_driver   ;write
DC.L _simio_clear_screen ;ctrl
DC.L 0                ;reserved
DC.L 0                ;reserved
```

- 2 Have a pointer to the I/O jump table within pSOS+'s configuration table; in other words, set the KC_IOJTABLE to the address of your I/O jump table.
- 3 Include "probe_io.c" and "io_drivers.c" in your application.

Chapter 4: Accessing pROBE+ through Simulated I/O
To prepare your application for simulated I/O access of pROBE+

- 4 Initialize pROBE's configuration table to have pointers to the initialization and console procedures found in "probe_io.c".

For example:

```
static struct s_rc rom_rc =
    {PROBE_CODE, /* Address of pROBE code */
    (INT32)&probe_data[0x800], /* Address of pROBE data */
    (INT32)pio_init, /* Address of I/O init procedure */
    (INT32)pio_consts, /* Address of console status
                        procedure */
    (INT32)pio_conin, /* Address of console input
                      procedure */
    (INT32)pio_conout, /* Address of console output
                       procedure */
    (INT32)0, /* Address of host status procedure */
    (INT32)0, /* Address of host input procedure */
    (INT32)0, /* Address of host output procedure */
    0x4E41, /* Breakpoint opcode */
    ...
```

- 5 Define a simulated I/O buffer.

For example:

```
SECTION iobuf,,D
XDEF _systemio_buf
_systemio_buf
DS.B 512
END
```

Chapter 4: Accessing pROBE+ through Simulated I/O
To prepare your application for simulated I/O access of pROBE+

6 Rebuild your application.

After completing the above tasks, your application will have pROBE+ available in a simulated I/O window. Start your task running within the emulator and depending on how you have the RC_SMODE value set in pROBE+'s configuration, you will now be able to access pROBE+.

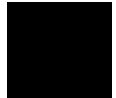
1 If RC_SMODE is set to "0" (meaning normal start-up), you can just enter:

display simulated_io
modify keyboard_to_simio.

2 If RC_SMODE is set to "1" (meaning silent start-up), your application will be running with continuous checks by the kernel for console status. Enter the following commands:

modify memory break_to_probe to 1
display simulated_io
modify keyboard_to_simio

You will now see the "pROBE+>" prompt in the simulated I/O display. Enter commands as you normally would to pROBE+. When done, press the **suspend** softkey and you will return to normal emulator control.



To break pSOS+ execution and enter pROBE+

- 1 Click on the **Break to Probe** action key.

Or, run the `e_brk2probe` command file by entering it on the command line.

- 2 In the simulated I/O window, enter the `modify keyboard_to_simio` command using the command line.

You have now broken into the pROBE+ monitor. Any command can now be given to pROBE+, and the output will appear in the simulated I/O window. For example, you can query the task status list by typing in the pROBE+ "qt" command on the command line and pressing <RETURN>.

```
Simulated I/O display
display is open
-----
Name      TID      Prio Mode Status   Susp?   Parameters          Ticks
-----
`IDLE`    -#00010000 00 2000  Ready
`ROOT`    -#00020000 FF 0001  Ready   YES
`iotk`    -#000D0000 0C 0006  Wkafter
`recr`    -#000E0000 09 0006  Ready   YES
`paal`    -#000F0000 0A 0006  Ewait   EVENTS = 00000001  forever
`silk`    -#00100000 09 0006  Qwait   Q = `cssl` -#00040000  forever
`bose`    -#00110000 09 0006  Qwait   Q = `csbo` -#00050000  forever
`phnx`    -#00120000 09 0006  Qwait   Q = `csph` -#00060000  forever
`losa`    -#00130000 09 0006  Qwait   Q = `phla` -#00090000  forever
`prod`    -#00140000 08 0006  Running
`cosp`    -#00150000 08 0006  Ready
`idlp`    -#00160000 07 0004  Ready
pROBE+>
```

You see that the query task command browses through the internal OS kernel and displays the status of each task. Note that the task names are the same as you see in the emulation window.

If you try to take a real-time trace, for example by clicking on the **Track OS calls** action key, you see that the status line indicates "Emulation trace started", but the trace does not complete because the application is in the pROBE+ monitor with no application tasks running.

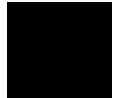
To exit pROBE+ and return to RTOS measurements

- In the simulated I/O window, type in "go" on the command line, and press <RETURN>.

After issuing the "go" command to pROBE+, the application tasks begin running again.

If you started a trace while in pROBE+, the trace becomes complete.

If your application uses the simulated I/O feature of the emulator/analyzer interface, the simulated I/O window returns to displaying the application output instead of the pROBE+ prompt.





5



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. Assembly language instructions in the "instrumented" service call library write values to the data table when:

- Tasks start.
- Tasks switch.
- 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.

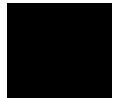
If your measurements will be made often, you can create your own command files and add your own action keys to the emulator/analyzer interface.

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 error checking information.

The part of the "track_os.s" source file that reserves space for the data table is shown below.



Chapter 5: Customizing the RTOS Measurement Tool

Creating Your Own RTOS Measurements

```

*****
*      -- THIS DATA TABLE MUST NOT BE CHANGED IN ANY WAY --
*      -- The interpretation of 'traced' data is dependent --
*      -- on the relative offsets of symbols --
*****
HPOS_Start_Ovrhd      DS.W  1      ;Start of OS interval for SPA
HPOS_Stop_Ovrhd       DS.W  1      ;End of OS interval for SPA

HPOS_Start_Intrusion  DS.W  1      ;Start interval for measuring intrusion
HPOS_Stop_Intrusion   DS.W  1      ;End interval for measuring intrusion

HP_RTOS_TRACK_START   ; The name of this symbol MUST NOT CHANGE!!!
                      ; It is required that the interface find this
                      ; symbol and pass its value to the Interpreter
                      ; so the beginning of this table is known.

HPOS_TASK_EXIT        DS.L  1
HPOS_TASK_ENTRY       DS.L  1

HPOS_t_create_Entry   DS.L  5
HPOS_t_create_Exit    DS.L  2
HPOS_t_ident_Entry    DS.L  2
HPOS_t_ident_Exit     DS.L  2
HPOS_t_start_Entry    DS.L  7
HPOS_t_start_Exit     DS.L  1
HPOS_t_restart_Entry  DS.L  5
HPOS_t_restart_Exit   DS.L  1
HPOS_t_delete_Entry   DS.L  1
HPOS_t_delete_Exit    DS.L  1
HPOS_t_suspend_Entry  DS.L  1
HPOS_t_suspend_Exit   DS.L  1
HPOS_t_resume_Entry   DS.L  1
HPOS_t_resume_Exit    DS.L  1
HPOS_t_setpri_Entry   DS.L  2
HPOS_t_setpri_Exit    DS.L  2
HPOS_t_mode_Entry     DS.L  2
HPOS_t_mode_Exit      DS.L  2
HPOS_t_getreg_Entry   DS.L  2
HPOS_t_getreg_Exit    DS.L  2
HPOS_t_setreg_Entry   DS.L  4
HPOS_t_setreg_Exit    DS.L  1

HPOS_T_TRANS_TID      DS.L  1
HPOS_TTRANS_ERR       DS.L  1
HPOS_TASK_BKT_UNDEF   DS.L  1

HPOS_rn_create_Entry  DS.L  5
HPOS_rn_create_Exit   DS.L  3
HPOS_rn_ident_Entry   DS.L  1
HPOS_rn_ident_Exit    DS.L  2
HPOS_rn_delete_Entry  DS.L  1
HPOS_rn_delete_Exit   DS.L  1
HPOS_rn_getseg_Entry  DS.L  4
HPOS_rn_getseg_Exit   DS.L  2
HPOS_rn_retseg_Entry  DS.L  2
HPOS_rn_retseg_Exit   DS.L  1

```

Chapter 5: Customizing the RTOS Measurement Tool Creating Your Own RTOS Measurements

```
HPOS_pt_create_Entry      DS.L 6
HPOS_pt_create_Exit      DS.L 3
HPOS_pt_ident_Entry     DS.L 2
HPOS_pt_ident_Exit      DS.L 2
HPOS_pt_delete_Entry    DS.L 1
HPOS_pt_delete_Exit     DS.L 1
HPOS_pt_getbuf_Entry    DS.L 1
HPOS_pt_getbuf_Exit     DS.L 2
HPOS_pt_retbuf_Entry    DS.L 2
HPOS_pt_retbuf_Exit     DS.L 1
HPOS_pt_sgetbuf_Entry   DS.L 1
HPOS_pt_sgetbuf_Exit    DS.L 3
```

```
HPOS_q_create_Entry      DS.L 3
HPOS_q_create_Exit      DS.L 2
HPOS_q_ident_Entry     DS.L 2
HPOS_q_ident_Exit      DS.L 2
HPOS_q_delete_Entry    DS.L 1
HPOS_q_delete_Exit     DS.L 1
HPOS_q_send_Entry      DS.L 5
HPOS_q_send_Exit       DS.L 1
HPOS_q_urgent_Entry    DS.L 5
HPOS_q_urgent_Exit     DS.L 1
HPOS_q_broadcast_Entry DS.L 5
HPOS_q_broadcast_Exit  DS.L 2
HPOS_q_receive_Entry   DS.L 3
HPOS_q_receive_Exit    DS.L 5
```

```
HPOS_Q_TRANS_QID        DS.L 1
HPOS_QTRANS_ERR         DS.L 1
```

```
HPOS_ev_send_Entry     DS.L 2
HPOS_ev_send_Exit     DS.L 1
HPOS_ev_receive_Entry  DS.L 3
HPOS_ev_receive_Exit  DS.L 2
```

```
*****
*          --- THIS DATA TABLE MUST NOT BE CHANGED IN ANY WAY ---          *
*          --- The interpretation of 'traced' data is dependent ---          *
*          --- on the relative offsets of symbols ---                          *
*****
```

```
HPOS_as_catch_Entry    DS.L 2
HPOS_as_catch_Exit     DS.L 1
HPOS_as_send_Entry     DS.L 2
HPOS_as_send_Exit      DS.L 1
HPOS_as_return_Entry   DS.L 1
HPOS_as_return_Exit    DS.L 1
```

```
HPOS_sm_create_Entry   DS.L 3
HPOS_sm_create_Exit    DS.L 2
HPOS_sm_ident_Entry    DS.L 2
HPOS_sm_ident_Exit     DS.L 2
HPOS_sm_delete_Entry   DS.L 1
HPOS_sm_delete_Exit    DS.L 1
HPOS_sm_v_Entry        DS.L 1
HPOS_sm_v_Exit         DS.L 1
HPOS_sm_p_Entry        DS.L 3
```



Chapter 5: Customizing the RTOS Measurement Tool

Creating Your Own RTOS Measurements

```

HPOS_sm_p_Exit          DS.L  1

HPOS_tm_tick_Entry     DS.L  1
HPOS_tm_tick_Exit      DS.L  1
HPOS_tm_set_Entry      DS.L  3
HPOS_tm_set_Exit       DS.L  1
HPOS_tm_get_Entry      DS.L  1
HPOS_tm_get_Exit       DS.L  4
HPOS_tm_wkafter_Entry  DS.L  1
HPOS_tm_wkafter_Exit   DS.L  1
HPOS_tm_wkwhen_Entry   DS.L  3
HPOS_tm_wkwhen_Exit    DS.L  1
HPOS_tm_evafter_Entry  DS.L  2
HPOS_tm_evafter_Exit   DS.L  2
HPOS_tm_evevery_Entry  DS.L  2
HPOS_tm_evevery_Exit   DS.L  2
HPOS_tm_evwhen_Entry   DS.L  4
HPOS_tm_evwhen_Exit    DS.L  2
HPOS_tm_cancel_Entry   DS.L  1
HPOS_tm_cancel_Exit    DS.L  1

HPOS_k_fatal_Entry     DS.L  2
HPOS_i_return_Entry    DS.L  1

HPOS_m_ext2int_Entry   DS.L  1
HPOS_m_ext2int_Exit    DS.L  2
HPOS_m_int2ext_Entry   DS.L  1
HPOS_m_int2ext_Exit    DS.L  2

HPOS_SERVICE_CALLS    ; Label to make tracing easier

HPOS_de_init_Entry     DS.L  2
HPOS_de_init_Exit      DS.L  3
HPOS_de_open_Entry     DS.L  2
HPOS_de_open_Exit      DS.L  2
HPOS_de_close_Entry    DS.L  2
HPOS_de_close_Exit     DS.L  2
HPOS_de_read_Entry     DS.L  2
HPOS_de_read_Exit      DS.L  2
HPOS_de_write_Entry    DS.L  2
HPOS_de_write_Exit     DS.L  2
HPOS_de_cntrl_Entry    DS.L  2
HPOS_de_cntrl_Exit     DS.L  2

HPOS_SRVC_DEVICES     ; Label to make tracing easier

HPOS_T_START_NAME      DS.L  1
HPOS_T_ENTRY_STACK     DS.L  1
HPOS_T_EXIT_STACK      DS.L  1
HPOS_T_STACK_VAR1      DS.L  1
HPOS_T_STACK_VAR2      DS.L  1
HPOS_T_STACK_VAR3      DS.L  1
HPOS_T_STACK_VAR4      DS.L  1

HPOS_SRVC_DEV_STACK    ; Label to make tracing easier

HPOS_USER_DEFENTRY1    DS.L  1          ; data entries to be used for
HPOS_USER_DEFEXIT1     DS.L  1          ; either SPA intervals or

```


Chapter 5: Customizing the RTOS Measurement Tool Creating Your Own RTOS Measurements

```
HPOS_USER_DEFENTRY2      DS.L  1      ; for general program tracking
HPOS_USER_DEFEXIT2      DS.L  1
HPOS_USER_DEFENTRY3      DS.L  1
HPOS_USER_DEFEXIT3      DS.L  1
HPOS_USER_DEFENTRY4      DS.L  1
HPOS_USER_DEFEXIT4      DS.L  1
HPOS_USER_DEFENTRY5      DS.L  1
HPOS_USER_DEFEXIT5      DS.L  1
HPOS_USER_DEFENTRY6      DS.L  1
HPOS_USER_DEFEXIT6      DS.B  3

HP_RTOS_TRACK_END      ;End of list indicator
HPOS_END_OF_DATA_AREA  DS.B  1

HPOS_CLOCK_TICK        DS.L  1

HPOS_CHECK_ERRORS      DS.L  1

*****
*      -- THIS DATA TABLE MUST NOT BE CHANGED IN ANY WAY --      *
*      -- The interpretation of 'traced' data is dependent --      *
*      -- on the relative offsets of symbols --                      *
*****
```

Data Table Contents

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

HPOS_TASK_EXIT
HPOS_TASK_ENTRY

The four character ASCII 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 parameters passed to, or returned from, a service call are written to these locations.

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.

Chapter 5: Customizing the RTOS Measurement Tool

Creating Your Own RTOS Measurements

HPOS_T_<stack_info_sym>

Stack information is written to these locations by the task start and task switch callout routines.

When including stack information in the RTOS trace, store writes to the entire range identified by the T_ symbols.

HPOS_CLOCK_TICK

This location is 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_ERRORS

Error return codes are written to this location when service calls exit.

HPOS_USER_DEF[ENTRY|EXIT]n

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 pSOS+ code itself. This can usually be accomplished by storing all activity not in the range of the pSOS+ code (that is, **trace only address not range** <pSOS_start> **thru** <pSOS_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.

When using data qualifiers to identify the entry or exit of a particular task, remember 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. Because 4 ASCII character (32-bit) task names are written to HPOS_TASK_ENTRY and HPOS_TASK_EXIT, you must capture the write of the high-order word or low-order word to identify a particular task when using a

Chapter 5: Customizing the RTOS Measurement Tool

Creating Your Own RTOS Measurements

16-bit processor. (This is the reason the first two and last two characters must be made unique when naming your tasks in the "tables_16.s" file.)

Examples

To track only queue and event service calls:

```
trace only address range HPOS_q_create_Entry thru  
HPOS_as_catch_entry-1 <RETURN>
```

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

To track only queue and event service calls including task switches (for 16-bit processors):

```
trace only address range HPOS_q_create_Entry thru  
HPOS_as_catch_entry-1 or HPOS_TASK_EXIT or  
HPOS_TASK_EXIT+2 or HPOS_TASK_ENTRY or  
HPOS_TASK_ENTRY+2 <RETURN>
```

This captures the same data table writes as the previous command and also the task entries and exits.

To track only queue and event service calls including task switches (for 32-bit processors):

```
trace only address range HPOS_q_create_Entry thru  
HPOS_as_catch_entry-1 or HPOS_TASK_EXIT or  
HPOS_TASK_ENTRY <RETURN>
```

This captures the same data table writes as the previous command, but it is for 32-bit processors.

Chapter 5: Customizing the RTOS Measurement Tool Creating Your Own RTOS Measurements

To track only the "cosp" task and queue service calls (for 16-bit processors)

(note that the hex value for "co" is 636fh and the hex value for "sp" is 7370h):

```
trace enable address HPOS_TASK_ENTRY data 636fh disable  
address HPOS_TASK_EXIT+2 data 7370h only address range  
HPOS_q_create_Entry thru HPOS_ev_send_Entry-1 <RETURN>
```

This trace starts or resumes capturing data when "co" (636FH) is written to the first word of the task entry location and halts data capturing when "sp" (7370H) is written to the task exit location. While enabled to capture data, the only states captured are the data table accesses that correspond to queue service calls.

To track only the "cosp" task and queue service calls (for 32-bit processors)

(note that the hex value for "cosp" is 636f7370h):

```
trace enable address HPOS_TASK_ENTRY data 636f7370h  
disable address HPOS_TASK_EXIT data 636f7370h only  
address range HPOS_q_create_Entry thru  
HPOS_ev_send_Entry-1 <RETURN>
```

This is the same as the previous command, except the starts and halts are done on data of "cosp" since the full 32-bit name is written in one cycle for 32-bit processors.

To trigger before an error return in task "cosp" (for 16-bit processors):

```
trace find_sequence HPOS_TASK_ENTRY data 636fh restart  
HPOS_TASK_EXIT+2 data 7370h trigger before  
HPOS_CHECK_ERRORS data not 0 only address range  
HP_RTOS_TRACK_START thru HPOS_TRACK_END <RETURN>
```

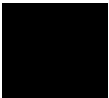
Starting (enabling) and halting (disabling) are done the same way as in previous commands, but now instead of capturing data, a specific event (in this case, a write of something other than zero (0) to HPOS_CHECK_ERRORS) is looked for as the trigger to complete the trace.

Chapter 5: Customizing the RTOS Measurement Tool
Creating Your Own RTOS Measurements

To trigger before an error return in task "cosp" (for 32-bit processors):

```
trace find_sequence HPOS_TASK_ENTRY data 636f7370h  
restart HPOS_TASK_EXIT data 636f7370h trigger before  
HPOS_CHECK_ERRORS data not 0 only address range  
HP_RTOS_TRACK_START thru HPOS_TRACK_END <RETURN>
```

This is the same as the previous command, but it is for 32-bit processors.



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 particular task and all the queue service calls that occur during the task. Notice that this is similar to the provided RTOS measurement that tracks only task X, except that you want to limit the service calls that are stored in the trace to just queue service calls.

First copy the existing command file.

```
$ cp $HP64000/rtos/B3080A/act_keys_302/e_trkltask  
e_trkltsknqs <RETURN>
```

The storage qualifier part of the command you wish to create is:

```
... only address range HPOS_q_create_Entry thru  
HPOS_ev_send_Entry-1 <RETURN>
```

So, edit the "e_trkltsknqs" command file so that only writes to the locations above are stored in the trace.

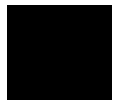
If your command file is placed in the \$HOME/rtoscmdf directory, you should set the HP64KPATH environment variable as follows:

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

- The easiest way to include an RTOS measurement on an action key is to first place the measurement in a command file; then, edit the "emulrtos_psos" script to add an action key label and the name of the command file.

When you open the RTOS emulation window (either by choosing the **File→Emul700→PSOS+ RTOS Measurement Tool** pulldown menu item in the emulator/analyzer interface or by using the "\$HP64000/bin/emulrtos_psos" script), the **emul700** command is issued with the **-xrm** option to set the X resource that defines action keys.

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.

Examples

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 the "emulrtos_psos" script.

```
vi $HP64000/bin/emulrtos_psos <RETURN>
```

Add a line that defines the action key label "Tsk X & Queues" and the location of the command file. In this case, add the line:

```
\ "Tsk X & Queues\"          \ "e_trkltsknqs\" \
```

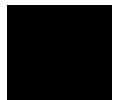
as part of the "keyDefs" resource definition.

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

Chapter 5: Customizing the RTOS Measurement Tool

Creating Your Own RTOS Measurements

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



Limiting the Intrusion Caused by Instrumented Service Calls

Within the "track_os.s" file, some sections/statements are labeled as being at a certain "level". The level indicates certain groupings of measurements that can be made. Even within levels, some subsets of measurement data could possibly be divided up but the levels keep similar measurements together for easier editing and understanding.

Level 1	Service call tracking (entry and return), task switching, clock ticks.
Level 2	Operating System overhead tracking, Intrusion measurement, Tracking Service call error returns
Level 3	SPA support - real time histogram of tasks
Level 4	Stack tracking - creation and dynamic sizes of stacks
Level 5	Task and queue id-to-name translation.

There are approximately 300 total lines of assembly code added within the service call functions and another 150 lines of support routine code. This may be cut down to about 100 lines in the service calls and only two lines of support routine code which will still show all service calls and task switching.

To reduce the amount of intrusion (and correspondingly the amount of measurement data), you may remove any of the levels. When levels are removed (generally by commenting out the relevant code), you should remove complete levels and you may only remove a level when all the levels "ranked" above it are removed. For example, if you don't want the intrusion imposed by the level 3 routines (in other words, task and queue naming), you must also remove levels 4 and 5. This is the recommended method, but you may find other ways which work for you.

This section shows you how to comment out the various levels of RTOS measurement support. Different variations on commenting out the instrumentation code may work but they will not be supported.

To comment out Level 5 (Id-to-name translation)

- 1 Comment out the call to `_SAVE_QUEUE_INFO` in the `q_create` service call.
- 2 Comment out all calls to `WRITE_TASK_NAME` in all service calls. The affected calls are `t_restart`, `t_delete`, `t_suspend`, `t_resume`, `t_setpri`, `t_getreg`, `t_setreg`, `ev_send`, and `as_send`.
- 3 Comment out all calls to `HPOS_WRITE_Q_NAME` in all queue service calls. The affected calls are `q_delete`, `q_send`, `q_urgent`, `q_broadcast`, and `q_receive` (2 calls).

To comment out Level 4 (Stack tracking)

- 1 Comment out the call to `SAVE_STACK_INFO` in the "`_t_create`" service call.
- 2 Comment out all instructions in the `_HPOS_SWITCH_CALLOUT` routine labeled with comments as "HP-RTOS-Level-4".
- 3 Comment out all instructions in the `_HPOS_START_CALLOUT` routine within the section labeled as "HP-RTOS-Level-4".

To comment out Level 3 (SPA support)

- 1 Comment out all instructions in the `_HPOS_SWITCH_CALLOUT` routine EXCEPT the 2 instructions labeled as "HP-RTOS-Level-1" and the 4 instructions labeled as "HP-RTOS-Level-2".
- 2 Remove the `_HPOS_START_CALLOUT` entry from your configuration table so it is no longer invoked when "`t_start()`" is called.

To comment out Level 2 (Overhead, intrusion and error returns)

- Comment out all instructions in "track_os.s" which are commented by the string "HP-RTOS-Level-2".

To comment out Level 1 (Task entry/exit and service calls)

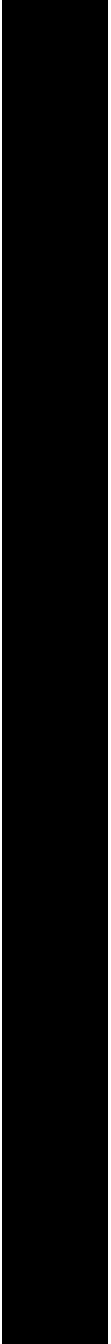
- 1 Remove the `_HPOS_SWITCH_CALLOUT` entry from your configuration table so it is no longer invoked when a task switch is done.
- 2 Comment out all instructions in "track_os.s" which are commented by the string "HP-RTOS-Level-1".
- 3 If instrumented, remove any writes to the data area from your application code.

Part 2

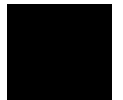
Concept Guide

Topics that explain concepts and apply them to advanced tasks.

Part 2



6



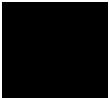
How the RTOS Measurement Tool Works

How the RTOS Measurement Tool Works

The RTOS measurement tool lets you perform a real-time trace of all 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 very limited. The simplest level of RTOS measurements require only two MOVEM assembly language instructions for each service call and a two-instruction task switch callout routine.

Additional RTOS measurements like stack tracking, measurements that include clock ticks, and real-time (no sampling) software performance analysis can be provided by adding a few more instructions to the application program. The level of intrusion is still quite minimal.

If the intrusion introduced becomes a problem, you can comment out some of the added instructions (in the "track_os.s" file) to find the right balance between intrusion and debugging capabilities (see the "Limiting the Intrusion Caused by Instrumented Service Calls" section in the "Customizing the RTOS Measurement Tool" chapter).

Service Call Tracking

Tracking of service calls takes advantage of the fact that there is usually an *interface library* which allows a high-level language application to call an assembly language based RTOS. This library is a set of functions that correspond directly to each routine available from the RTOS. We will refer to these functions as *service calls* of the RTOS.

Each function in the library is accessible via a normal high-level subroutine call. The function is responsible for taking parameters off the stack and placing values into proper registers. A "trap" instruction is then executed to pass control to the RTOS which interprets the registers and determines which of its own functions needs to be run. (The D0 register is usually set in the interface function to arbitrate which function in the RTOS is being requested.)

In order to track service calls, code has been added to each service call in the interface library. This code writes the contents of the registers that are used to specific known locations within a defined data table. The data table has defined offsets within it for each parameter of each function. (For pSOS+, the data table requires about 1000 bytes.)

Chapter 6: How the RTOS Measurement Tool Works

Instrumented Code for Real-Time OS Tracking

So for each function, any register that has been set with a specific value to be passed to the RTOS has its value written to a location unique to that function and parameter. This is accomplished through a simple MOVEM instruction which writes all registers that have been assigned values by the service call to a specific memory location in the data area. One MOVEM is done right before the "trap" instruction and one is done upon return.

When running an application that uses the "instrumented" interface library (that is, the interface library to which code has been added for RTOS measurements), tracing the address range of the defined data table captures all data being passed into and returned from each and every service call.

When trace information is captured, a RTOS specific inverse assembler decodes the information and displays the intimate details of the interaction between an application and a RTOS.

The data table needed for a specific RTOS relates directly to the number of functions available from a RTOS and the number of parameters passed to and returned from such a RTOS. For each function, there is a set of long words associated with the call to the function and a set for the return from the function.

For instance, in the pSOS+ RTOS, there is a function called "t_create()" which creates a task. There are 6 registers which are assumed to be set before trapping to the kernel and 2 output registers which are set by the kernel before it returns. One of the 6 input registers is D0 whose contents, as noted above, specify the function pSOS+ should execute. Because the function is already identified by the data table locations being written to, it is not necessary to write out the value of D0. Consequently, only 5 long words are reserved for register values written when the "t_create" function is called. Upon return, both registers contain information specific to the call; therefore, 2 long words are reserved for the "t_create" return values.

The portion of the code in the "instrumented" interface library for the "t_create" call would look like:

```
MOVEM.L D1-D5,HPOS_t_create_Entry ; write out input data
TRAP #SVCTRAP ; trap to the kernel
MOVEM.L D0-D1,HPOS_t_create_Exit ; write out return data
```

and the respective data area declarations would look like:

```
HPOS_t_create_Entry DS.L 5
HPOS_t_create_Exit DS.L 2
```

Notice that a single MOVEM instruction can move multiple register values to the data area.

Chapter 6: How the RTOS Measurement Tool Works Instrumented Code for Real-Time OS Tracking

Instructions added for service call tracking represent the most minimal intrusion while giving you almost complete knowledge of the interaction between your application and the RTOS kernel. The information that's missing is knowledge about the tasks running and when task switches take place. You can add task information by writing a "task switch callout" routine.

Task Switch Tracking

The *task switch callout* routine is a hook provided by the RTOS vendor. It allows a user to define a routine to be called every time a task switch occurs. Upon calling the routine, two registers are set with pointers to the task control blocks of the task exiting and the task being entered.

For the simplest task switch tracking, the callout routines need only consist of two instructions: one writing out the task name of the task being exited, one writing the task 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. Since each task needs to be measured, each task 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 task ID 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_psos" script creates the file "tables.s" which defines these task buckets.

Clock Ticks

There are two methods for tracking clock ticks. First, if the application uses the TM_TICK OS service call, clock tick information is automatically available since this service call is instrumented.

However, some applications may choose not to use the "C" interface function for this feature and may write the associated interrupt service routine (ISR) directly in assembly language code for speed reasons. In this case, the interrupt service routine should be instrumented with a simple MOVE.W Dx,HPOS_CLOCK_TICK instruction before the trap to pSOS+. (Make sure it is a word write to the HPOS_CLOCK_TICK location.) The memory location corresponding to

Chapter 6: How the RTOS Measurement Tool Works

Instrumented Code for Real-Time OS Tracking

CLOCK_TICK is 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.

Selective Tracking

With the data area for service calls defined, it is possible to selectively trace 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.

OS Overhead Tracking

In order to get some idea of how efficient an application is, that is, to see how much time is spent switching tasks as opposed to executing them, the software performance analyzer can display a dynamic histogram of the time spent in the OS kernel.

This is done, as is the service call tracking, by adding simple MOVE instructions to the service call routines. The first MOVE instruction, executed just before the trap to the kernel, writes to a location that represents the start of the OS interval. The second MOVE instruction, executed just after the return from the trap, writes to a location that represents end of the OS interval. The software performance analyzer measures the time between these writes as time spent in the OS kernel.

Note

Using this method, some kernel time may be missed due to clock ticks. The time spent processing clock ticks is minimal and consistent, so this time is of little consequence. Additional kernel time is missed when task switches occur because the task has used up its time slice. If excessive timeouts occur, the measurement of the kernel's accumulated time will be slightly low.

Task and Queue Naming

Tasks and queues are created with ASCII names but mostly referenced with numbers. If a little more information is written out on each service call, these numbers can be translated into names when displayed in the trace. After the register values are written via the MOVEM instruction (for service call tracking), a short subroutine that translates the ID number into a 32-bit name may be called.

Chapter 6: How the RTOS Measurement Tool Works Instrumented Code for Real-Time OS Tracking

This translation is possible because information was stored earlier when the item was created. When a "`tq_create`" service call is made, the name of the item is written to a table indexed by the ID number. With the name stored, it's easy (and quick) to reference any name if the ID number is known.

If the "translation" routine is called, it will index into the appropriate table and write out the 32-bit name to a designated location. If this location is traced, the RTOS inverse assembler can decode the data and output a name "xyzz" instead of just "id=00030000".

Stack and Memory Tracking

Stack information such as size, pointers, and bytes left on stack can be tracked dynamically as an application runs. The necessary data is mostly 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 "bucket" table must be filled with all the names of the application's tasks. This creates a data area that will be used to save the task's stack values.
- 2 The "`t_create`" service call is instrumented to call a routine that will save each task's two stack sizes in the appropriate bucket.
- 3 The "`t_start`" service call is instrumented to call a routine that will save several data items: the task ID number, the memory locations in the Task Control Block that hold the stack pointer values, the limit for each stack, and the task bucket's address. Also, data is written to a special area in the general data area so the stack creation 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 the previously saved data to keep track of stacks. In the callout routine, the task being pre-empted and the task being started running are found by indexing via the task ID to the saved task bucket's address. This address is used to access stack data. The stack data can then be written out and interpreted by the RTOS inverse assembler to display the stack bytes left on exit from a task and entry to a task.

User-Defined Areas

At the bottom 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.

Chapter 6: How the RTOS Measurement Tool Works

Instrumented Code for Real-Time OS Tracking

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 SPA of the time spent in any interrupt service routine.

If a write is done to any of these locations, the captured data is displayed as a hex number and, if possible, translated to ASCII characters. This allows easier debugging since seeing "Loop" in a display easily reminds you what part of the application you just executed versus seeing "0x4c6f6f70" and trying to mentally translate a number to a location of code.

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 longword 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 symbols 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 most common standard prefix for the data area symbols is "HPOS_". Only four (4) symbols do not use that prefix: HP_RTOS_TRACK_START, HP_RTOS_TRACK_END, _HPOS_START_CALLOUT, and _HPOS_SWITCH_CALLOUT.

Chapter 6: How the RTOS Measurement Tool Works Instrumented Code for Real-Time OS Tracking

The Data Table

Task Entry	(1 long word)
Task Exit	(1 long word)
Service Call 1 Entry	(n1 longs)
Service Call 1 Exit	(n1' longs)
Service Call 2 Entry	(n2 longs)
Service Call 2 Exit	(n2' longs)
Service Call 3 Entry	(n3 longs)
Service Call 3 Exit	(n3' longs)
.	
.	
Service Call N Entry	(nN longs)
Service Call N Exit	(nN' longs)
Clock Tick	(1 word)
Task Name	(1 long)
Queue Name	(1 long)
Semaphore Name	(1 long)
Region Name	(1 long)
Stack Task Name	(1 long)
Stack Supr Size	(1 long)
Stack Supr Ptr	(1 long)
Stack User Size	(1 long)
Stack User Ptr	(1 long)
User Numeric	(1 long)
User Numeric	(1 long)
User Numeric	(1 long)
User Numeric	(1 long)
User Numeric	(1 long)
User Numeric	(1 long)
User Numeric	(1 long)
User Ascii	(1 long)
User Ascii	(1 long)
User Ascii	(1 long)
User Ascii	(1 long)
User Ascii	(1 long)
User Ascii	(1 long)



Chapter 6: How the RTOS Measurement Tool Works Instrumented Code for Real-Time OS Tracking

Extra Memory Locations

```
Kernel Overhead Start (1 word)
Kernel Overhead End (1 word)
Task Buckets (created by macro)
Task_abcd 'abcd'
Enter_Task_abcd (1 long word) ;SPA interval starting address
Exit_Task_abcd (1 long word) ;SPA interval ending address
MStack_Siz_abcd (1 long word) ;Master stack size
MStack_Ptr_abcd (1 long word) ;Master stack ptr
MStack_Lmt_abcd (1 long word) ;Master stack limit
UStack_Siz_abcd (1 long word) ;User stack size
UStack_Ptr_abcd (1 long word) ;User stack ptr
UStack_Lmt_abcd (1 long word) ;User stack limit
Tid_abcd (1 long word) ;Task id number
Task_name_abcd EQU 'name' ;task name symbol
-----
Task_cdef 'cdef'
Enter_Task_cdef (1 long word) ;SPA interval starting address
...
Task_name_cdef EQU 'cdef' ;task name symbol
-----
Task_efgh 'efgh'
Enter_Task_efgh (1 long word) ;SPA interval starting address
...
Task_name_xyzz EQU 'xyzz' ;task name symbol
```


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)*.

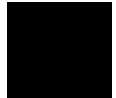
A short example should help. First, let's assume the segment of a user's application that makes an OS service call looks as follows:

```
.  
.  
mailbox = 2;  
message = 1234;  
priority = URGENT;  
return_value = q_send(mailbox, message, priority);  
.  
.
```

The function "q_send()" is an OS service call that sends a message to a specific mailbox.

Instrumented Library Writes to the Data Table

Because the user has substituted our instrumented interface library in place of the original OS interface library, the call to "q_send" causes additional code to 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 value 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:

```
.  
-> q_send(mbox=2, msg=1234, prio=URGENT)  
<- q_send()  
.  
.
```

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 mailbox parameter value of 2 is preceded with a "mbox=". 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.

Also notice that the parameter indicating message priority, "prio=" does not have a numeric value but displays the word "URGENT". Since many OS service call parameters have a finite number of valid input values, we have decoded these parameters directly into their English language equivalents to again make it easy to read the trace without referencing the OS manual.

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 mailboxes, or a task using up a given time slice.

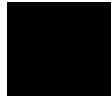
Given this behavior, application code that evokes an OS service call may not immediately return from that service call but may instead begin executing code in another task. For example, when the "q_send()" OS service call in the previous trace example sent a message to the mailbox, if another task of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

```
.  
.  
-> q_send(mbox=2, msg=1234, prio=URGENT)  
--- Exiting Task 'TSK1' -----  
--- Entering Task 'TSK2' -----  
<- q_receive(msg=1234)  
.  
.
```

You can see that task 'TSK1', which sent the message has now exited and task 'TSK2', which had been waiting for a message with the "q_receive()" OS service call, has now started up again. You can also see in the return parameter of the "q_receive()" call that it did indeed receive the same message that was sent.

Inverse Assemblers are Tailored to the OS

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



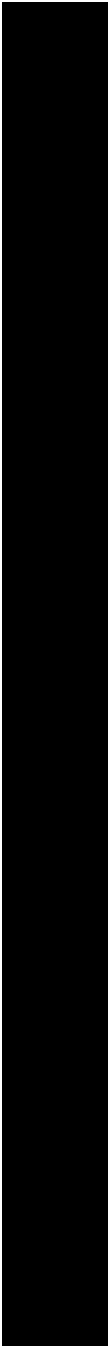


Part 3

Installation Guide

Instructions for installing and configuring the product.

Part 3



7



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: **PSOS+ RTOS Measurement Tool ...** and **SPA for PSOS+** These two entries will, respectively, 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:

Chapter 7: Installation
To install HP 9000 software

`/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`

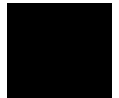
and press `<RETURN>`. It takes about 10 minutes to read the tape.
- 11 When the installation is complete, read `/tmp/update.log` to see the results of the update.

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.

If you are installing a Softkey Interface product, refer to the *How to Use the Softkey Interface on Your SPARCsystem* operating notice for additional software installation instructions.





Glossary

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

callout routine a mechanism provided by the real-time OS that allows you to execute a routine at certain points in the application, for example, when a task starts 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 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 instructions that write to the data table and save information in task and queue buckets.

interface library a library of assembly language routines which allow a high-level language application to call an assembly language based real-time operating system.

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.

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

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



Glossary

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.



Index

- A**
 - about, trace command option, **91**
 - action keys, **96**
 - actionKeys.numColumns, X resource, **96**
 - actionKeysSub.keyDefs, X resource, **96**
 - after, trace command option, **91**

- B**
 - background emulation monitor, **22**
 - before, trace command option, **91**
 - break_on_trigger (in trace command), **31**
 - bucket, **17, 109, 125**
 - buckets, **107**
 - bytes left on stack, **109**

- C**
 - callout routine, **125**
 - callout routines
 - task start, **19**
 - task switch, **19, 105, 107, 109**
 - clock ticks, **32, 85, 90, 105, 107-108**
 - command files, **95**
 - configuration table, pROBE+, **76**
 - console procedures, pROBE+, **76**
 - coordinated measurements, **72-73**
 - count histogram display of task events, **67**
 - custom RTOS measurements, **85-97**
 - customize script, **14**

- D**
 - data bus width, **91**
 - data table, **19, 85, 105, 111, 125**
 - description, **85**
 - device call, **125**
 - device calls, **29**
 - disable, trace command option, **91**
 - duration (function), show histogram, **70**
 - dynamic memory usage, tracking, **53-56**

- E**
 - emul700 command, **21**
 - xrm option, **96**



emulation bus analyzer, **5, 14, 26, 72, 85, 114, 125**
resources of, **108**
emulation monitor, **22**
emulrtos_psos, emulator startup script, **21, 96**
enable, trace command option, **91**
environment variables, **21**
HP64000, **17**
HP64KPATH, **95**
PATH, **21**
PROCESSOR, **21**
RTOS_UNIQUE, **74**
error checking information, **85, 90**
error return, **31**
commenting out, **100**
event calls, **36-37**
event numbers, **47**
event, received by specific task, **47**
events (SPA)
defining for tasks, **64**
table display, **66**
events (task)
count histogram display, **67**
time histogram display, **65**

F files
io_drivers.c, **18, 76**
probe_io.c, **18, 76**
psos.h, **18, 76**
RTOS source, **17**
simio.h, **18, 76**
tables.s, **17, 19**
tables_16.s, **92**
track_os.s, **17, 19, 65, 85, 98, 105**
find_sequence, trace command option, **91**
foreground emulation monitor, **22**
function
any task using a, **51**
specific task using a, **48**
function duration histogram, show normal, **70**

G glossary, **125-126**

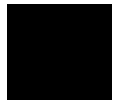
- H** histogram
 - normal function duration, **70**
 - task events, **71**
 - user events, **71**histogram display of task events
 - count, **67**
 - time, **65**HP64000 environment variable, **17**
- HP64KPATH environment variable, **95**
- I** ID-to-name translation, commenting out, **99**
- initialization procedures, pROBE+, **76**
- installation, **120**
 - HP 9000 software, **121-122**
 - Sun SPARCsystem, **123**
- instrumented service call library, **85, 98-100, 106, 113, 125**
- interface library, **105, 113, 125**
- intrusion, **98-100, 105**
- inverse assembler, **85, 104, 106, 109, 113, 115, 125**
- invocations (service call), show table, **69**
- io_drivers.c file, **18, 76**
- L** levels of RTOS measurements, **98**
- M** Measure_Ovrhd in SPA, **65**
- memory calls, **56**
- memory usage, **109**
- memory usage, tracking, **53-56**
- memory, extra locations, **112**
- message queues
 - See* queues
- message, from specific task to specific queue, **46**
- mnemonics in RTOS trace display, **114**
- monitor, emulation, **22**
- N** names, for tasks and queues, **108**
- non-RTOS states, **85, 91**
- O** only, trace command option, **91**
- operating system versions supported, **121**
- OS overhead tracking, **100, 108**
- OS_Time in SPA, **65**
- overflow, task time, **72**

- overhead (OS) tracking, **108**
 - commenting out, **100**
- P**
 - PATH environment variable, **21**
 - prepare for RTOS measurements, **14**
 - pROBE+, **18-19, 76**
 - probe_io.c file, **18, 76**
 - PROCESSOR environment variable, **21**
 - processor type, **21**
 - psos.h file, **18, 76**
- Q**
 - queue calls, **34-35**
 - queue naming, **108**
 - queues
 - defining, **17**
 - naming, **17**
- R**
 - real-time runs, emulator restriction, **22**
 - requirements, **14**
 - RTOS information, trace commands to capture, **91**
 - RTOS measurement tool
 - how it works, **104**
 - overview, **4**
 - testing, **23**
 - RTOS measurements
 - creating your own, **85-97**
 - emulator/analyzer, **26**
 - preparing for, **14**
 - software performance analyzer, **62**
 - RTOS source files, **17**
 - RTOS symbol names, **110**
 - rtos_edit_psos script, **17, 74**
 - RTOS_UNIQUE environment variable, **74**
- S**
 - scripts
 - customize, **14**
 - rtos_edit_psos, **17, 74**
 - selective tracking, **108**
 - semaphore calls, **38-39**
 - service call library (instrumented), **85, 98-100**
 - service calls, **29, 32, 85, 105, 108, 113-115, 125**
 - commenting out, **100**
 - entry and exit, **115**

- parameters, **89**
 - show table of invocations, **69**
 - single call tracking, **40**
 - two call tracking, **41**
 - simio.h file, **18, 76**
 - simulated I/O, **18, 76**
 - software performance analyzer, **5, 14, 62, 105, 107-108, 126**
 - testing, **24**
 - software versions, **121**
 - source files, RTOS, **17**
 - SPA events
 - See* events (SPA)
 - SPA support, commenting out, **99**
 - stack activity, **30, 32**
 - commenting out, **99**
 - stack information, **85, 90**
 - stack pointers, **109**
 - stack size, **109**
 - stack usage, **54, 105, 109**
 - storage qualifiers in trace commands, **91**
 - supported system versions, **121**
 - symbol names, **110**
- T**
- t_create service call, **109**
 - t_create() function, **106**
 - t_start service call, **109**
 - table display of SPA events, **66**
 - table of service call invocations, **69**
 - tables.s file, **17, 19**
 - tables_16.s file, **92**
 - Task Control Block, **109**
 - task events histogram, **71**
 - task naming, **108**
 - task start callout routine, **19, 85**
 - task switch callout routine, **19, 85, 105, 107, 109**
 - task switches, **32, 89, 107-108, 115**
 - commenting out, **100**
 - in event call tracking, **37**
 - in memory call tracking, **56**
 - in queue call tracking, **35**
 - in semaphore call tracking, **39**
 - specific task switch tracking, **45**

task time overflow, **72**
tasks, **125**
 defining, **17**
 four task tracking, **44**
 naming, **17**
 single task tracking, **43**
 SPA data for specific task, **68**
 SPA event definition, **64**
 time interval measurements, **64-71**
time histogram display of task events, **65**
time interval measurements, **64-71**
time overflow, task, **72**
time slice, **108**
time stamp, **27**
trace commands
 about option, **91**
 after option, **91**
 before option, **91**
 disable option, **91**
 enable option, **91**
 find_sequence option, **91**
 only option, **91**
 storage qualifier, **91**
trace commands to capture RTOS information, **91**
trace display
 mnemonics in, **114**
 normal, **58**
 RTOS, **59**
traces, displaying, **57-59**
track_os.s file, **17, 19, 65, 85, 98, 105**
tracking
 memory, **109**
 OS overhead, **108**
 OS overhead, commenting out, **100**
 selective, **108**
 stack, **109**
translation (ID to name) routine, **109**
trig2 break, **73**
trig2 signal, **24, 72**
 disabling, **73**
type of processor, **21**

- U** user events histogram, **71**
 - user-defined areas in data table, **109**
 - user-defined data table locations, **85, 90**
- V** variable
 - any task accessing a, **52**
 - specific task accessing a, **49**
- X** X resources
 - actionKeys.numColumns, **96**
 - actionKeysSub.keyDefs, **96**



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.