User's Guide

HP B3081B
Real-Time OS Measurement Tool for VRTX32 and VRTXsa

Notice

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

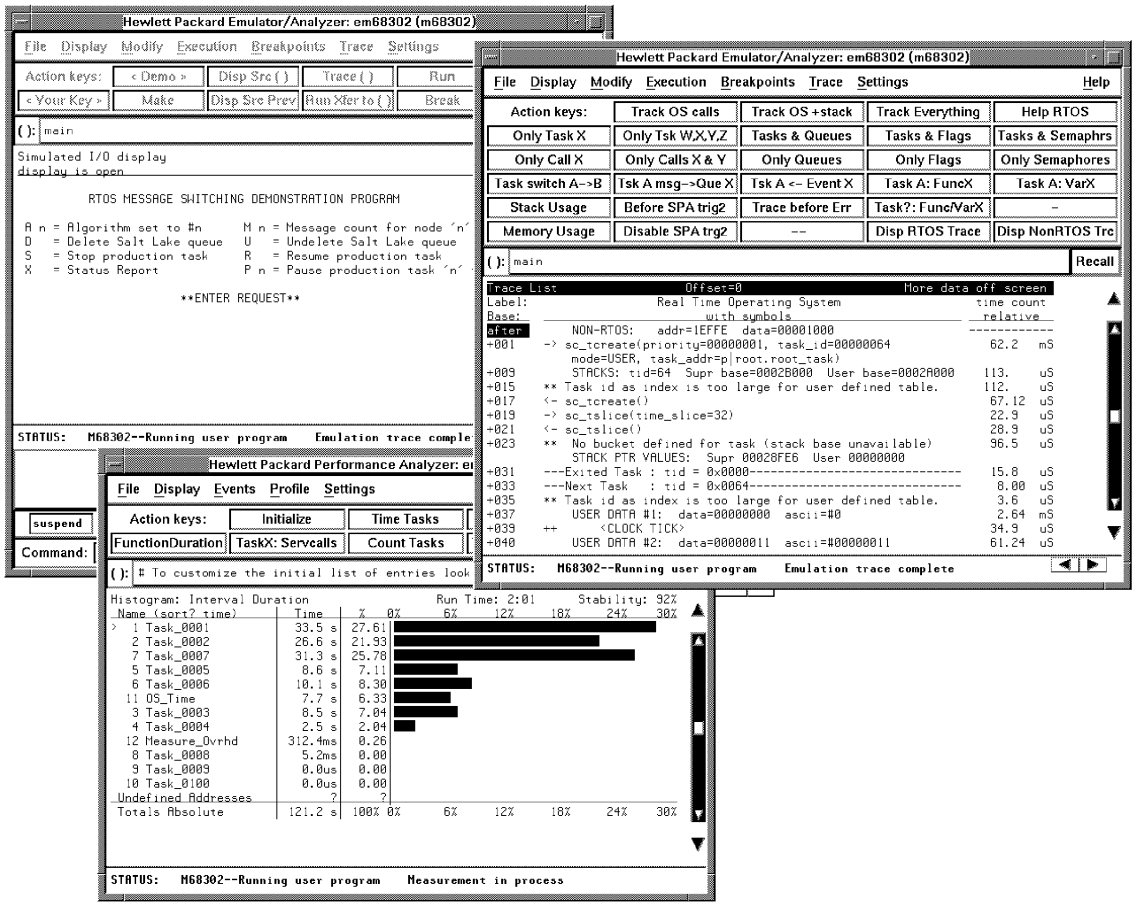
Printing History

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

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	B3081-97001, June 1994
Edition 2	**B3081-97002, August 1995**

Measurements for the VRTX Real-Time Operating System


Hewlett Packard Emulator/Analyzer: em68302 (m68302)
File Display Modify Execution Breakpoints Trace Settings
Action keys: < Demo > Disp Src ( ) Trace ( ) Run
< Your Key > Make Disp Src Prev Run Xfer to ( ) Break
suspend
Command:
Hewlett Packard Emulator/Analyzer: em68302 (m68302)
File Display Modify Execution Breakpoints Trace Settings Help
Action keys: Track OS calls Track OS +stack Track Everything Help RTOS
Only Task X Only Tsk W,X,Y,Z Tasks & Queues Tasks & Flags Tasks & Semaphrs
Only Call X Only Calls X & Y Only Queues Only Flags 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?: Func/VarX -
Memory Usage Disable SPA trg2 -- Disp RTOS Trace Disp NonRTOS Trc
Recall
Hewlett Packard Performance Analyzer: e
File Display Events Profile Settings
Action keys: Initialize Time Tasks
FunctionDuration TaskX: Servcalls Count Tasks

```
( ): main
Simulated I/O display
display is open

          RTOS MESSAGE SWITCHING DEMONSTRATION PROGRAM

 A n = Algorithm set to #n        M n = Message count for node 'n'
 D   = Delete Salt Lake queue     U   = Undelete Salt Lake queue
 S   = Stop production task       R   = Resume production task
 X   = Status Report              P n = Pause production task 'n'

                       **ENTER REQUEST**

STATUS:   M68302--Running user program    Emulation trace comple
```

```
( ): main
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=00001000                 ------------
+001    -> sc_tcreate(priority=00000001, task_id=00000064           62.2   mS
           mode=USER, task_addr=p|root.root_task)
+009       STACKS: tid=64  Supr base=0002B000  User base=0002A000  113.     uS
+015    ** Task id as index is too large for user defined table.   112.     uS
+017    <- sc_tcreate()                                             67.12  uS
+019    -> sc_tslice(time_slice=32)                                 22.9   uS
+021    <- sc_tslice()                                              28.9   uS
+023    **  No bucket defined for task (stack base unavailable)     96.5   uS
           STACK PTR VALUES:  Supr 00028FE6  User 00000000
+031    ---Exited Task : tid = 0x0000------------------------------  15.8   uS
+033    ---Next Task   : tid = 0x0064------------------------------   8.00  uS
+035    ** Task id as index is too large for user defined table.     3.6   uS
+037       USER DATA #1:  data=00000000  ascii=#0                    2.64  mS
+039    ++       <CLOCK TICK>                                       34.9   uS
+040       USER DATA #2:  data=00000011  ascii=#00000011            61.24  uS
STATUS:   M68302--Running user program    Emulation trace complete
```

```
( ): # To customize the initial list of entries look
Histogram: Interval Duration                    Run Time: 2:01      Stability: 92%
 Name (sort? time)      |   Time  |   %  0%       6%      12%     18%     24%     30%
>  1 Task_0001          |  33.5 s | 27.61|
   2 Task_0002          |  26.6 s | 21.93|
   7 Task_0007          |  31.3 s | 25.78|
   5 Task_0005          |   8.6 s |  7.11|
   6 Task_0006          |  10.1 s |  8.30|
  11 OS_Time            |   7.7 s |  6.33|
   3 Task_0003          |   8.5 s |  7.04|
   4 Task_0004          |   2.5 s |  2.04|
  12 Measure_Ovrhd      | 312.4ms |  0.26|
   8 Task_0008          |   5.2ms |  0.00|
   9 Task_0009          |   0.0us |  0.00|
  10 Task_0100          |   0.0us |  0.00|
 Undefined Addresses    |       ? |     ?|
 Totals Absolute        | 121.2 s | 100% 0%       6%      12%     18%     24%     30%
STATUS:   M68302--Running user program    Measurement in process
```

The HP B3081B Real-Time Operating System Measurement Tool for VRTX32 and VRTXsa supports the 68000/68302, 68020/68030/68040, and 68331/68332/68340/68360/68010 VRTX32 and VRTXsa Operating Systems from Ready Systems.

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 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 B3081B Real-Time Operating System Measurement Tool for the VRTX32 and VRTXsa Operating Systems from Ready Systems.

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 shows you how to customize the RTOS Measurement Tool.

Part 2. Concept Guide

Chapter 5 describes how the RTOS measurement tool works.

Part 3. Installation Guide

Chapter 6 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

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

Part 1

1

Preparing Your Application for RTOS Measurements

Preparing Your Application for RTOS Measurements

Requirements

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

- Installed the emulator, emulation bus analyzer, and Graphical User Interface as described in their *User's Guide* manuals. The emulator/analyzer interface software must be version C.05.20 or greater. Note that if you have installed another Graphical User Interface after you installed the HP B3081B Real-Time Operating System Measurement Tool, you must re-run the HP B3081B "customize" script.
- Installed the HP B3081B 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 four new entries under the **File→Emul700** pulldown menu: **VRTX32 Emulator/Analyzer ...**, **VRTXsa Emulator/Analyzer ...**, **VRTX32 Performance Analyzer ...**, and **VRTXsa Performance Analyzer ...**. If you do not see these new entries, review the installation procedure to make sure it was done correctly, and make sure the /system/B3081B/customize script was run. If you still do not see these new entries, contact your Hewlett-Packard representative.

VRTX Versions

This product is compatible with VRTX32 version 1.08 and VRTXsa version 4.0. Note that these version numbers are not the same as the version number of Spectra or Velocity.

Task list control file

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

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

Preparing your application for RTOS measurements

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

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

The remainder of this chapter describes these steps in detail.

Step 1: Make a new source directory

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

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

Step 2: Retrieve the RTOS source files

If you have already installed the RTOS Measurement Tool, source files will be found under the $HP64000/rtos/B3081A (VRTX) or $HP64000/rtos/B3081B (VRTXsa) 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/B3081A (VRTX) or $HP64000/rtos/B3081B (VRTXsa). You must copy the following files:

VRTXsa	VRTX
track_os.s	track_os.s
track_il.c	
HPIL.h	

Step 3: Create the task table

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

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

```
rtos_edit_vrtx -i -tables <task_name_file>
```

or

```
rtos_edit_vrtxsa -i -tables <task_name_file>
```

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

The "rtos_edit_<os>" script asks you for the task IDs in your application. Enter the ID numbers of the tasks you use in your application. These are the IDs that are defined as parameters to the following OS service calls:

sc_tcreate()	Create a task.
sc_tecreatec()	Create a task (extended call).

- **If you have prepared a task list control file, run the $HP64000/bin/rtos_edit_vrtxsa or $HP64000/bin/rtos_edit_vrtx script. Type:**

```
rtos_edit_vrtx -tables <task_name_file>
```

or

```
rtos_edit_vrtxsa -tables <task_name_file>
```

Running the "rtos_edit_<os>" script 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. Tables.s allows a "bucket" to be created in memory for each task entry you define. Information is written to the buckets when task switches occur.

The "rtos_edit_<os>" script may be run any time you wish to add or delete task ID information.

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

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

See Also

Page 28 for instructions on how to add the "rtos_edit_<os>" script to your makefile.

Step 4: Create the Software Performance Analyzer initialization file

1 Create the "s_init" file. Type:

```
rtos_edit_vrtx -s_init <task_name_file>
```

or

```
rtos_edit_vrtxsa -s_init <task_name_file>
```

where <*task_name_file*> is the name of the task list control file which you created in Step 3.

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

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

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

Step 5: Add the RTOS measurement files to your application

For VRTX32:

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 VRTX 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 VRTX-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 VRTX-to"C" language interface routines in the file "track_os.s" have been validated with the HP 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 **Change your VRTX configuration table so the task switching callout field, CFTSWITCH, has a pointer to the "_SWITCH_CALLOUT" routine and the task start callout field, CFSCREATE, has a pointer to "_START_CALLOUT" routine. (Both routines are defined in "track_os.s".) Refer to your VRTX manual for more information on VRTX configuration tables.**

For VRTXsa:

1 Add "track_os.s", "track_il.c" and "tables.s" into your makefile and linker files.

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.

2 Add the header file "HPIL.h" to every .c source file that contains VRTXsa service calls.

This header file redirects the VRTXsa service routine to an HP routine that will provide tracking measurements.

3 Change your VRTXsa configuration table so the task switching callout field, CFTSWITCH, has a pointer to the "_SWITCH_CALLOUT" routine and the task start callout field, CFTSTART, has a pointer to "_START_CALLOUT" routine. (Both routines are defined in "track_os.s".) Refer to your VRTXsa manual for more information on VRTXsa configuration tables.

4 Define the version of VRTXsa you are using.

If you are using version 3.x, add the following directive to your source code:

```
#DEFINE HP_VRTXsa_V3
```

If you are using version 4.x, add the following directive:

```
#DEFINE HP_VRTXsa_V4
```

Step 6: Build the new application file

- Rebuild your application with the new files. The service routines defined in "track_os.s" (for VRTX) or "track_il.c" (for VRTXsa) have been defined according to the VRTXsa standard so your application should require no changes.

Step 7: Start the emulator interface

- **Start the RTOS emulation window using the "emulrtos_vrtx" or "emulrtos_vrtxsa" script found in "$HP64000/bin":**

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

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

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

1 Sets HP64000 if it is not already set.

2 Sets HP64RTOSIAL based on the determined bus width.

3 Defines the environment variable HP64KPATH so the command files related to the action keys are found.

4 Defines the PATH variable so shell scripts needed by command files will be found.

If you have used the **emul700** command to start the emulator/analyzer interface, you can choose the **File→Emul700→VRTX/VRTXsa RTOS Measurement Tool** pulldown menu item to open the RTOS emulation window. This will work only if the $HP64RTOSIAL environment variable has been set. If you need to find out how to set the $HP64RTOSIAL variable, examine the "emulrtos_<os>" script.

Step 8: Configure the emulator and load the application

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

A few notes on the configuration:

1. You MAY set the emulator to be restricted to real-time runs. The RTOS measurements are done without breaking into the emulation monitor.
2. You may use either a foreground or background monitor.

You are now ready to test your application.

See Also

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

Step 9: Test the RTOS measurement tool

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

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

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

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

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

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

Step 10: Test the Software Performance Analyzer

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

1 Bring up SPA window by choosing the **File→Emul700→SPA for VRTX/VRTXsa** pulldown menu item.

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

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

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

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

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

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

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

See Also

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

Suggestions for Easier Software Development

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

To add rtos_edit to your makefile

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

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

```
~/.rtos/vrtx/s_init: <task_file_name>
        rtos_edit_vrtx -s_init <task_file_name>
tables.s: <task_file_name>
        rtos_edit_vrtx -tables <task_file_name>
```

Or, for VRTXsa,

```
~/.rtos/vrtxsa/s_init: <task_file_name>
        rtos_edit_vrtxsa -s_init <task_file_name>
tables.s: <task_file_name>
        rtos_edit_vrtxsa -tables <task_file_name>
```

To write a VRTX32 configuration table

- To simplify writing your configuration table, refer to the following example.

```
#include "demo.h"

#define m133 1

#define USER_MON_TRACE_ENTRY 0x40800

void root();
void os_error_func();
void SWITCH_CALLOUT();
void START_CALLOUT();

/*---------------------------------------------------------------------*/
#define VRTX_CODE ((INT32) 0x0E000)


/*---------------------------------------------------------------------*/
/* Important vectors
/*---------------------------------------------------------------------*/
#define V_BUSERR    2           /* Bus error */
#define V_ADDRERR   3           /* Address error */
#define V_TRACE     9           /* Trace */
#define V_TRAP11    43          /* Trap 11 - VRTX call */
#define VECTOR64    64          /* Vector 64 - address 0x100 */

/*---------------------------------------------------------------------*/
/* Configuration table
/*---------------------------------------------------------------------*/
struct vrtx_conf
    {
    INT32 workspace_addr;   /* VRTX Workspace start address */
    INT32 workspace_size;   /* VRTX Workspace length */
    INT16 sys_stack_size;   /* VRTX system stack size in bytes */
    INT16 int_stack_size;   /* Interrupt stack size in bytes */
    INT16 cntrl_blk_count;  /* Max # of event flag groups & semaphores */
    INT16 reserve1;         /* Reserve 1 */
    INT32 reserve2;         /* Reserve 2 */
    INT16 comp_disable;     /* Disable interrupts value (0 = default = 7) */
    INT16 user_stack_size;  /* User stack size */
    INT32 reserve3;         /* Reserve 3 */
    INT16 user_task_count;  /* Max number of tasks */
    INT16 reserve4;         /* Reserve 4 */
    INT32 txrdy_driver;     /* TXRDY driver address */
    INT32 tcreate_callout;  /* Callout at task creation */
    INT32 tdelete_callout;  /* Callout at task deletion */
    INT32 tswitch_callout;  /* Callout at task switch */
    INT32 comp_vect_tbl;    /* Component vector table */
    } nc;


/*---------------------------------------------------------------------*/
/* Value template for configuration table
/*---------------------------------------------------------------------*/
```

```
static struct vrtx_conf vrtx_conf_values =
    {0x1F000,                 /* VRTX Workspace start address */
     0x20000,                 /* VRTX Workspace length */
     0x01000,                 /* VRTX system stack size in bytes */
     0x00800,                 /* Interrupt stack size in bytes */
     5,                       /* Max # of event flag groups & semaphores */
     0,                       /* Reserve 1 */
     0,                       /* Reserve 2 */
     0,                       /* Disable interrupts value (0 = default = 7) */
     0x1000,                  /* User stack size */
     0,                       /* Reserve 3 */
     11,                      /* Max number of tasks */
     0,                       /* Reserve 4 */
     (INT32)0,                /* TXRDY driver address */
     (INT32) START_CALLOUT,   /* Callout at task creation */
     0,                       /* Callout at task deletion */
     (INT32) SWITCH_CALLOUT,  /* Callout at task switch */
     0                        /* Component vector table */
     };


/*************************************************************************/
/*                                                                       */
/*               Perform initial reset functions                         */
/*                                                                       */
/*************************************************************************/
void init_config()
{
        INT32 *v_page;                      /* Pointer to vector page */

#ifdef M68340
        v_page = (INT32 *)0x38000;                        /* Base vector page at
0x38000 */
#else
        v_page = (INT32 *)0;                              /* Base vector page at
zero */
#endif

        /*---------------------------------------------------------------------*/
        /* Set up the vector page
        /*---------------------------------------------------------------------*/
        v_page[V_TRACE]  = USER_MON_TRACE_ENTRY;
        v_page[V_TRAP11] = VRTX_CODE;
        v_page[VECTOR64] = (INT32) &vrtx_conf_values;
}
```

To write a VRTXsa configuration table

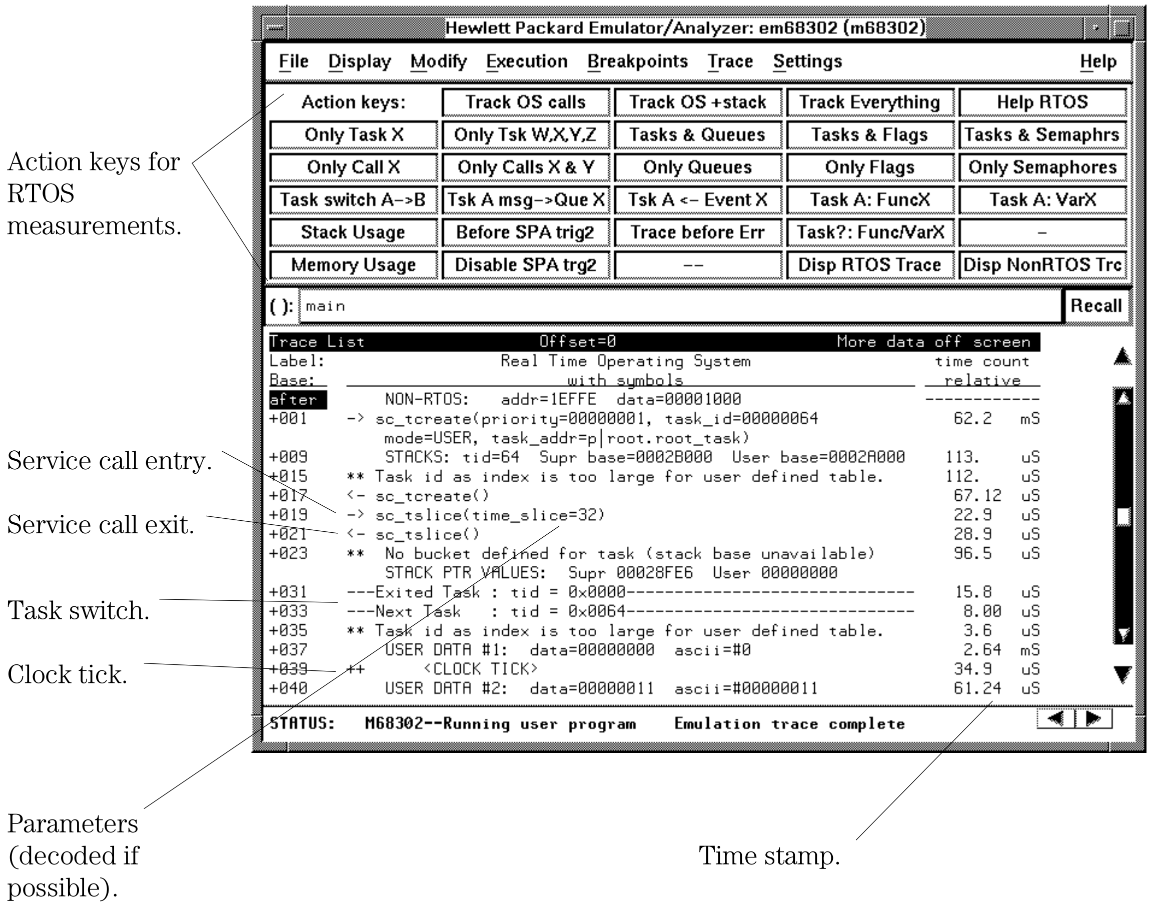
- To simplify writing your configuration table, refer to the following example.

```
/* #define M68K 1
*/

#include "demo.h"
#include "target.h"
#include "vrtxvisi.h"

#define m133 1

#define USER_MON_TRACE_ENTRY 0x40800

extern void SWITCH_CALLOUT();
extern void START_CALLOUT();
extern void     init_application();

extern CFTBL v32_configuration_table;


/*----------------------------------------------------------------------*/
/* Important vectors
/*----------------------------------------------------------------------*/
#define V_BUSERR    2           /* Bus error */
#define V_ADDRERR   3           /* Address error */
#define V_TRACE     9           /* Trace */
#define V_TRAP11    43          /* Trap 11 - VRTX call */
#define VECTOR64    64          /* Vector 64 - address 0x100 */

/*----------------------------------------------------------------------*/
/* Configuration table
/*----------------------------------------------------------------------*/
/* defined in file vrtxvisi.h */


/*----------------------------------------------------------------------*/
/* Value template for configuration table
/*----------------------------------------------------------------------*/
struct CFTBL vrtx_conf_values =
        {(unsigned char *) 0x15400,  /* VRTX Workspace start address */
     0x09FFF,                   /* VRTX Workspace length */
     0x00700,                   /* VRTX system stack size in bytes */

#ifdef M68020
     0x00100,                  /* Interrupt stack size in bytes */
#endif
#ifdef M68302
     0,                                  /* Interrupt stack size in bytes */
#endif

     5,                        /* Control block count */
     3,                        /* number of memory partitions */
     256,                      /* idle task stack size */
          10,                                                     /* number of
queues */
     0,                        /* Disable interrupts value (0 = default = 7) */
     0x0070,                   /* User stack size */
     11,                       /* Max allowed task ID */

#ifdef M68020
     target_mc68020,           /* target type from file target.h */
#endif
#ifdef M68302
     target_mc68302,
#endif

     11,                       /* Max number of tasks */
          0,                                                          /*
configuration options */
     0,                        /* TXRDY driver address */
     &START_CALLOUT,    /* Callout at task creation */
     0,                        /* Callout at task deletion */
     &SWITCH_CALLOUT,   /* Callout at task switch */
     (unsigned char *) 0,      /* Component vector table */
          0,                                                          /* Reserved */
          0,                                                          /* Reserved */
          0,                                                          /* Reserved */
          0                                                           /* Reserved */
     };


/************************************************************************/
/*                                                                      */
/*              Perform initial reset functions                         */
/*                                                                      */
/************************************************************************/
void init_config()
{
        int err;
        INT32 *v_page;                    /* Pointer to vector page */

#ifdef M68340
        v_page = (INT32 *)0x38000;                          /* Base vector page at
0x38000 */
#endif
#ifdef M68020
        v_page = (INT32 *)0;                                /* Base vector page at
zero */
#endif
#ifdef M68302
        v_page = (INT32 *)0;                                /* Base vector page at
zero */
#endif

        /*----------------------------------------------------------------------*/
        /* Set up the vector page
        /*----------------------------------------------------------------------*/
        v_page[VECTOR64] = (INT32) &vrtx_conf_values;

        v32_configuration_table = vrtx_conf_values;
        vrtx_init(&err);
        init_application();
}
```

2

Making RTOS Measurements with the Emulator/Analyzer

Making RTOS Measurements with the Emulator/Analyzer



```
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=00001000                ------------
+001     -> sc_tcreate(priority=00000001, task_id=00000064          62.2   mS
            mode=USER, task_addr=p|root.root_task)
+009        STACKS: tid=64  Supr base=0002B000  User base=0002A000  113.    uS
+015     ** Task id as index is too large for user defined table.   112.    uS
+017     <- sc_tcreate()                                             67.12  uS
+019     -> sc_tslice(time_slice=32)                                 22.9   uS
+021     <- sc_tslice()                                              28.9   uS
+023     **  No bucket defined for task (stack base unavailable)     96.5   uS
            STACK PTR VALUES:  Supr 00028FE6  User 00000000
+031     ---Exited Task : tid = 0x0000------------------------------  15.8   uS
+033     ---Next Task   : tid = 0x0064------------------------------   8.00  uS
+035     ** Task id as index is too large for user defined table.     3.6   uS
+037        USER DATA #1:  data=00000000  ascii=#0                    2.64  mS
+039     ++       <CLOCK TICK>                                       34.9   uS
+040        USER DATA #2:  data=00000011  ascii=#00000011            61.24  uS
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).

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.

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

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

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/B3081B/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 activity and stops if a failure is ever detected; this helps designers guard against unchecked return values.

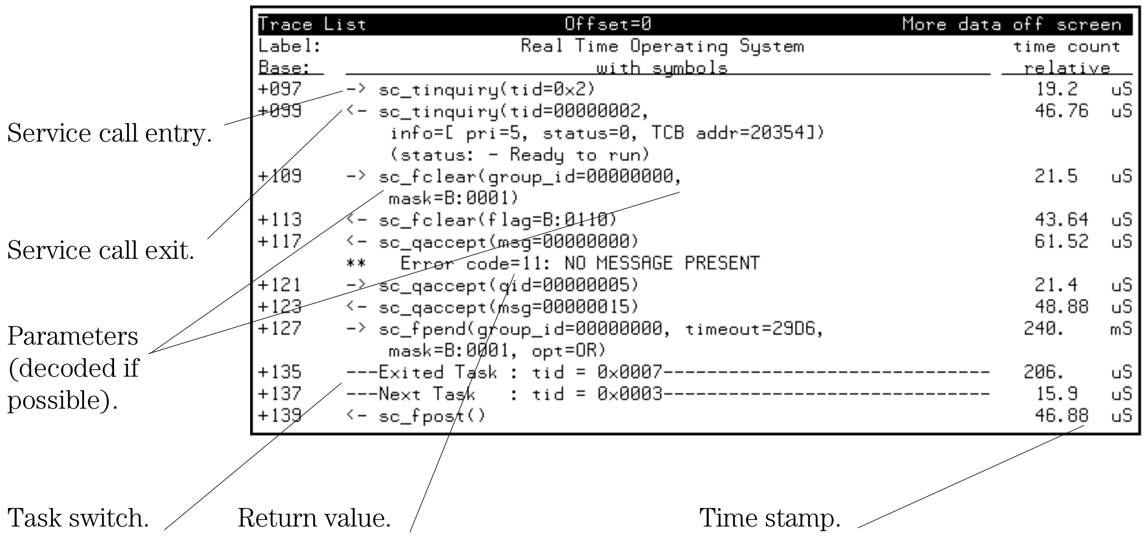
This section shows you how to:

- 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.



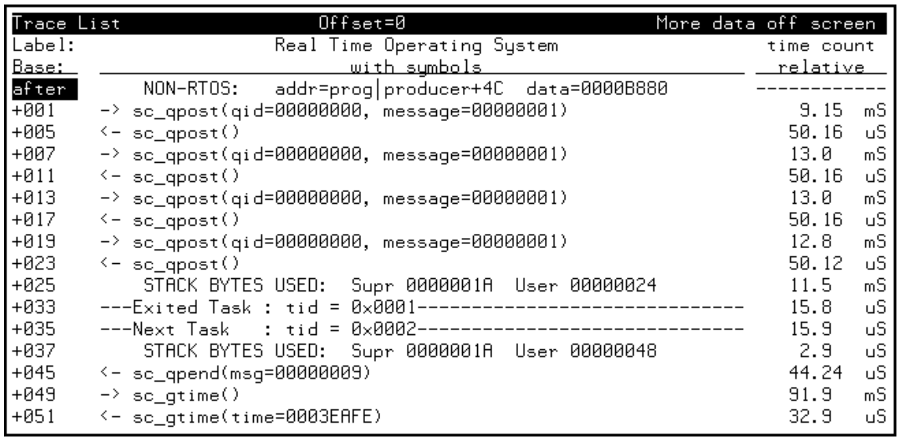
```
Trace List                    Offset=0                    More data off screen
Label:                 Real Time Operating System                   time count
Base:                         with symbols                           relative
+097     -> sc_tinquiry(tid=0x2)                                     19.2   uS
+099     <- sc_tinquiry(tid=00000002,                                46.76  uS
            info=[ pri=5, status=0, TCB addr=20354])
            (status: - Ready to run)
+109     -> sc_fclear(group_id=00000000,                             21.5   uS
            mask=B:0001)
+113     <- sc_fclear(flag=B:0110)                                   43.64  uS
+117     <- sc_qaccept(msg=00000000)                                 61.52  uS
         **   Error code=11: NO MESSAGE PRESENT
+121     -> sc_qaccept(qid=00000005)                                 21.4   uS
+123     <- sc_qaccept(msg=00000015)                                 48.88  uS
+127     -> sc_fpend(group_id=00000000, timeout=29D6,               240.    mS
            mask=B:0001, opt=OR)
+135     ---Exited Task : tid = 0x0007------------------------------ 206.    uS
+137     ---Next Task   : tid = 0x0003------------------------------  15.9   uS
+139     <- sc_fpost()                                               46.88  uS
```

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|producer+4C  data=0000B880        ------------
+001    -> sc_qpost(qid=00000000, message=00000001)                   9.15  mS
+005    <- sc_qpost()                                                50.16  uS
+007    -> sc_qpost(qid=00000000, message=00000001)                  13.0   mS
+011    <- sc_qpost()                                                50.16  uS
+013    -> sc_qpost(qid=00000000, message=00000001)                  13.0   mS
+017    <- sc_qpost()                                                50.16  uS
+019    -> sc_qpost(qid=00000000, message=00000001)                  12.8   mS
+023    <- sc_qpost()                                                50.12  uS
+025       STACK BYTES USED:  Supr 0000001A  User 00000024           11.5   mS
+033    ---Exited Task : tid = 0x0001------------------------------  15.8   uS
+035    ---Next Task   : tid = 0x0002------------------------------  15.9   uS
+037       STACK BYTES USED:  Supr 0000001A  User 00000048            2.9   uS
+045    <- sc_qpend(msg=00000009)                                    44.24  uS
+049    -> sc_gtime()                                                91.9   mS
+051    <- sc_gtime(time=0003EAFE)                                   32.9   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 used 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

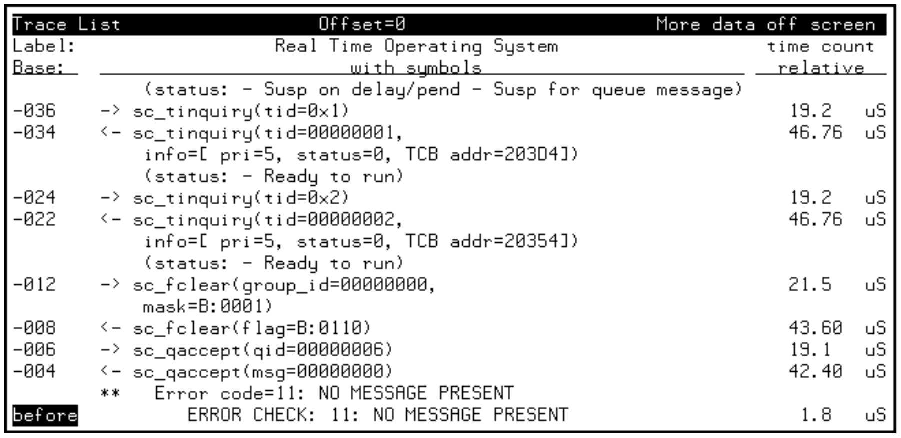
- Click on the **Trace before Err** action key (or run the **e_before_err** command file by entering it on the command line).

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.

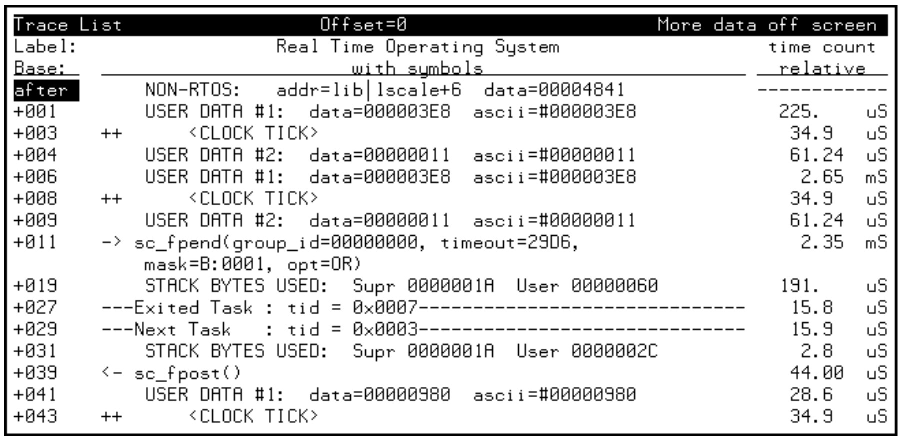
When the trace completes, you can see the activity that occurred before the failed service call, and the error return value itself is decoded into an easily readable error message as described in the OS kernel manual.

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
            (status: - Susp on delay/pend - Susp for queue message)
-036     -> sc_tinquiry(tid=0x1)                                     19.2   uS
-034     <- sc_tinquiry(tid=00000001,                                46.76  uS
            info=[ pri=5, status=0, TCB addr=203D4])
            (status: - Ready to run)
-024     -> sc_tinquiry(tid=0x2)                                     19.2   uS
-022     <- sc_tinquiry(tid=00000002,                                46.76  uS
            info=[ pri=5, status=0, TCB addr=20354])
            (status: - Ready to run)
-012     -> sc_fclear(group_id=00000000,                             21.5   uS
            mask=B:0001)
-008     <- sc_fclear(flag=B:0110)                                   43.60  uS
-006     -> sc_qaccept(qid=00000006)                                 19.1   uS
-004     <- sc_qaccept(msg=00000000)                                 42.40  uS
         **   Error code=11: NO MESSAGE PRESENT
before           ERROR CHECK:  11: NO MESSAGE PRESENT                 1.8   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+6  data=00004841          ------------
+001        USER DATA #1:  data=000003E8  ascii=#000003E8            225.    uS
+003    ++      <CLOCK TICK>                                          34.9   uS
+004        USER DATA #2:  data=00000011  ascii=#00000011             61.24  uS
+006        USER DATA #1:  data=000003E8  ascii=#000003E8              2.65  mS
+008    ++      <CLOCK TICK>                                          34.9   uS
+009        USER DATA #2:  data=00000011  ascii=#00000011             61.24  uS
+011    -> sc_fpend(group_id=00000000, timeout=29D6,                   2.35  mS
            mask=B:0001, opt=OR)
+019        STACK BYTES USED:  Supr 0000001A  User 00000060          191.    uS
+027    ---Exited Task : tid = 0x0007------------------------------   15.8   uS
+029    ---Next Task   : tid = 0x0003------------------------------   15.9   uS
+031        STACK BYTES USED:  Supr 0000001A  User 0000002C            2.8   uS
+039    <- sc_fpost()                                                 44.00  uS
+041        USER DATA #1:  data=00000980  ascii=#00000980             28.6   uS
+043    ++      <CLOCK TICK>                                          34.9   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

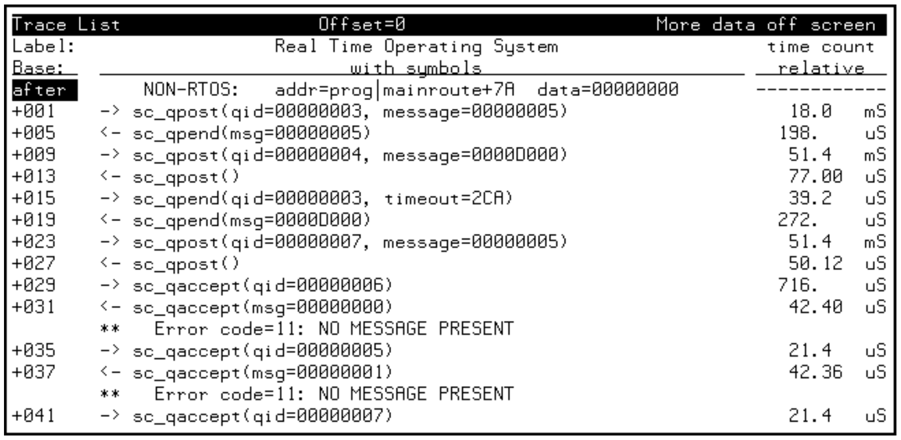
There are also RTOS measurements provided to track particular types of service call activity or OS resources such as flags, messages, or semaphores. You can also track individual service calls.

This section shows you how to:

- Track all queue calls.
- Track all queue calls (include task switches).
- Track all flag calls.
- Track all flag 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).

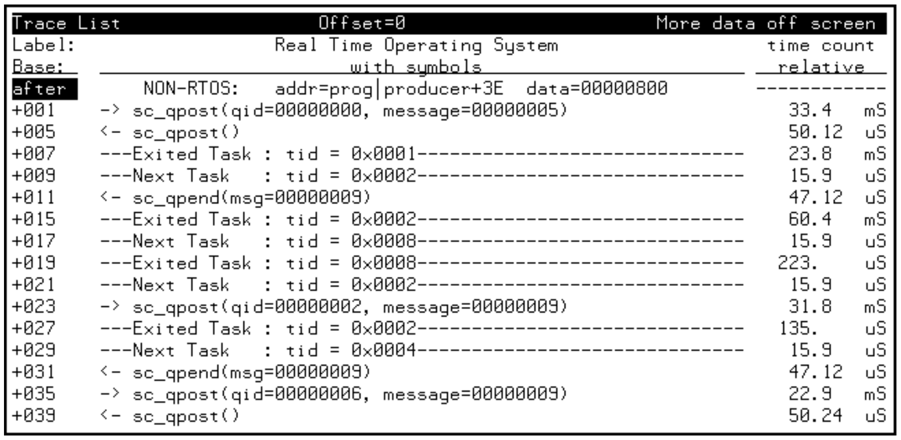


```
Trace List                    Offset=0                      More data off screen
Label:                   Real Time Operating System                 time count
Base:                           with symbols                         relative
after       NON-RTOS:   addr=prog|mainroute+7A  data=00000000       ------------
+001     -> sc_qpost(qid=00000003, message=00000005)                   18.0   mS
+005     <- sc_qpend(msg=00000005)                                    198.    uS
+009     -> sc_qpost(qid=00000004, message=0000D000)                   51.4   mS
+013     <- sc_qpost()                                                 77.00  uS
+015     -> sc_qpend(qid=00000003, timeout=2CA)                        39.2   uS
+019     <- sc_qpend(msg=0000D000)                                    272.    uS
+023     -> sc_qpost(qid=00000007, message=00000005)                   51.4   mS
+027     <- sc_qpost()                                                 50.12  uS
+029     -> sc_qaccept(qid=00000006)                                  716.    uS
+031     <- sc_qaccept(msg=00000000)                                   42.40  uS
         **   Error code=11: NO MESSAGE PRESENT
+035     -> sc_qaccept(qid=00000005)                                   21.4   uS
+037     <- sc_qaccept(msg=00000001)                                   42.36  uS
         **   Error code=11: NO MESSAGE PRESENT
+041     -> sc_qaccept(qid=00000007)                                   21.4   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).

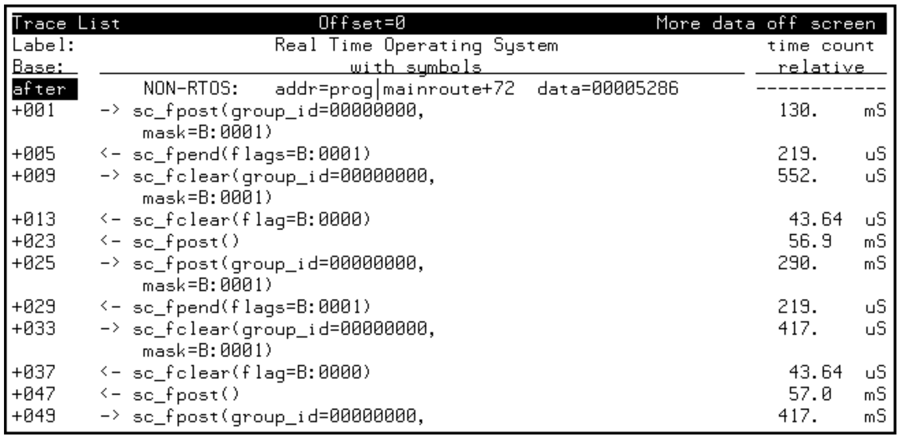


```
Trace List                  Offset=0                  More data off screen
Label:                  Real Time Operating System                time count
Base:                         with symbols                         relative
after       NON-RTOS:    addr=prog|producer+3E  data=00000800     ------------
+001     -> sc_qpost(qid=00000000, message=00000005)                33.4   mS
+005     <- sc_qpost()                                              50.12  uS
+007     ---Exited Task : tid = 0x0001------------------------------ 23.8   mS
+009     ---Next Task   : tid = 0x0002------------------------------ 15.9   uS
+011     <- sc_qpend(msg=00000009)                                  47.12  uS
+015     ---Exited Task : tid = 0x0002------------------------------ 60.4   mS
+017     ---Next Task   : tid = 0x0008------------------------------ 15.9   uS
+019     ---Exited Task : tid = 0x0008------------------------------ 223.    uS
+021     ---Next Task   : tid = 0x0002------------------------------ 15.9   uS
+023     -> sc_qpost(qid=00000002, message=00000009)                31.8   mS
+027     ---Exited Task : tid = 0x0002------------------------------ 135.    uS
+029     ---Next Task   : tid = 0x0004------------------------------ 15.9   uS
+031     <- sc_qpend(msg=00000009)                                  47.12  uS
+035     -> sc_qpost(qid=00000006, message=00000009)                22.9   mS
+039     <- sc_qpost()                                              50.24  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 flag calls

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

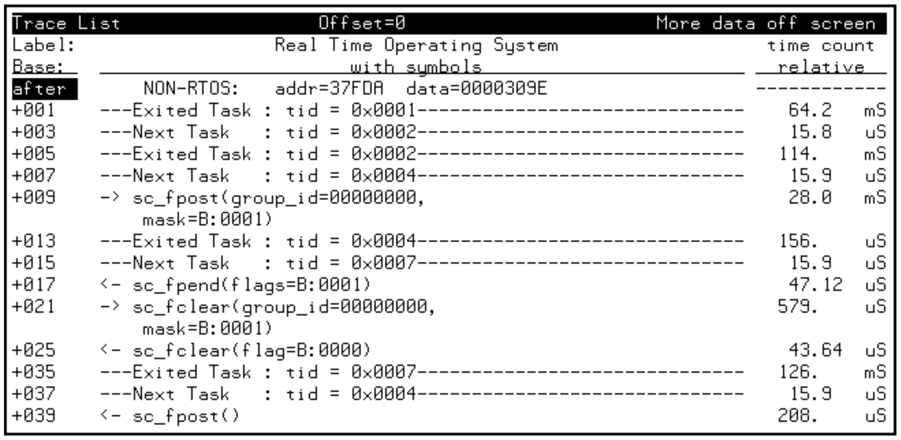


```
Trace List                  Offset=0                    More data off screen
Label:              Real Time Operating System                 time count
Base:                      with symbols                          relative
after        NON-RTOS:    addr=prog|mainroute+72  data=00005286   ------------
+001     -> sc_fpost(group_id=00000000,                           130.     mS
             mask=B:0001)
+005     <- sc_fpend(flags=B:0001)                                219.     uS
+009     -> sc_fclear(group_id=00000000,                          552.     uS
             mask=B:0001)
+013     <- sc_fclear(flag=B:0000)                                 43.64   uS
+023     <- sc_fpost()                                             56.9    mS
+025     -> sc_fpost(group_id=00000000,                           290.     mS
             mask=B:0001)
+029     <- sc_fpend(flags=B:0001)                                219.     uS
+033     -> sc_fclear(group_id=00000000,                          417.     uS
             mask=B:0001)
+037     <- sc_fclear(flag=B:0000)                                 43.64   uS
+047     <- sc_fpost()                                             57.0    mS
+049     -> sc_fpost(group_id=00000000,                           417.     mS
```

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

To track all flag calls (include task switches)

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



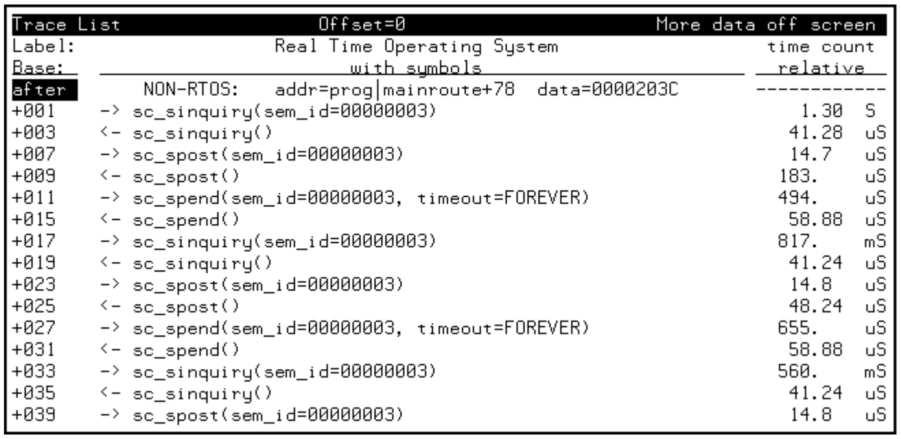
```
Trace List                  Offset=0                    More data off screen
Label:                Real Time Operating System                  time count
Base:                        with symbols                          relative
after      NON-RTOS:    addr=37FDA  data=0000309E                ------------
+001     ---Exited Task : tid = 0x0001------------------------------   64.2   mS
+003     ---Next Task   : tid = 0x0002------------------------------   15.8   uS
+005     ---Exited Task : tid = 0x0002------------------------------  114.    mS
+007     ---Next Task   : tid = 0x0004------------------------------   15.9   uS
+009     -> sc_fpost(group_id=00000000,                                28.0   mS
            mask=B:0001)
+013     ---Exited Task : tid = 0x0004------------------------------  156.    uS
+015     ---Next Task   : tid = 0x0007------------------------------   15.9   uS
+017     <- sc_fpend(flags=B:0001)                                     47.12  uS
+021     -> sc_fclear(group_id=00000000,                              579.    uS
            mask=B:0001)
+025     <- sc_fclear(flag=B:0000)                                     43.64  uS
+035     ---Exited Task : tid = 0x0007------------------------------  126.    mS
+037     ---Next Task   : tid = 0x0004------------------------------   15.9   uS
+039     <- sc_fpost()                                                208.    uS
```

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

The display shows that task 7 is receiving flag signals from the other tasks.

To track all semaphore calls

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

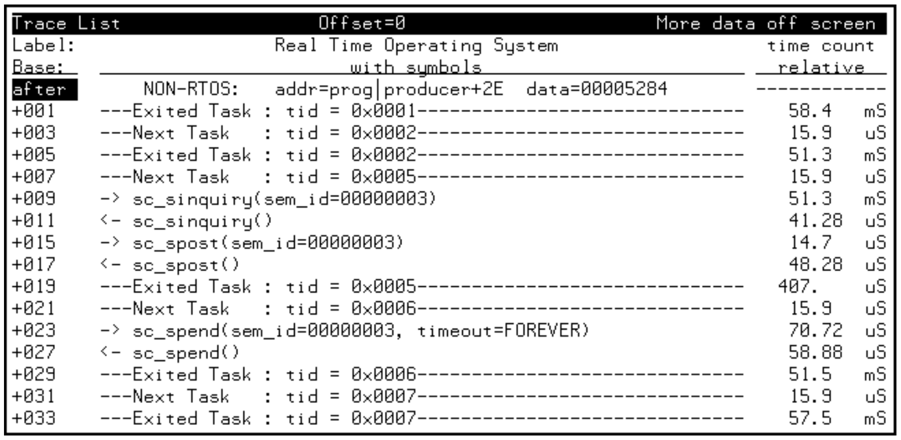


```
Trace List                    Offset=0                        More data off screen
Label:                  Real Time Operating System                    time count
Base:                          with symbols                            relative
after       NON-RTOS:   addr=prog|mainroute+78  data=0000203C       ------------
+001    -> sc_sinquiry(sem_id=00000003)                                 1.30  S
+003    <- sc_sinquiry()                                               41.28  uS
+007    -> sc_spost(sem_id=00000003)                                   14.7   uS
+009    <- sc_spost()                                                 183.    uS
+011    -> sc_spend(sem_id=00000003, timeout=FOREVER)                 494.    uS
+015    <- sc_spend()                                                  58.88  uS
+017    -> sc_sinquiry(sem_id=00000003)                               817.    mS
+019    <- sc_sinquiry()                                               41.24  uS
+023    -> sc_spost(sem_id=00000003)                                   14.8   uS
+025    <- sc_spost()                                                  48.24  uS
+027    -> sc_spend(sem_id=00000003, timeout=FOREVER)                 655.    uS
+031    <- sc_spend()                                                  58.88  uS
+033    -> sc_sinquiry(sem_id=00000003)                               560.    mS
+035    <- sc_sinquiry()                                               41.24  uS
+039    -> sc_spost(sem_id=00000003)                                   14.8   uS
```

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).

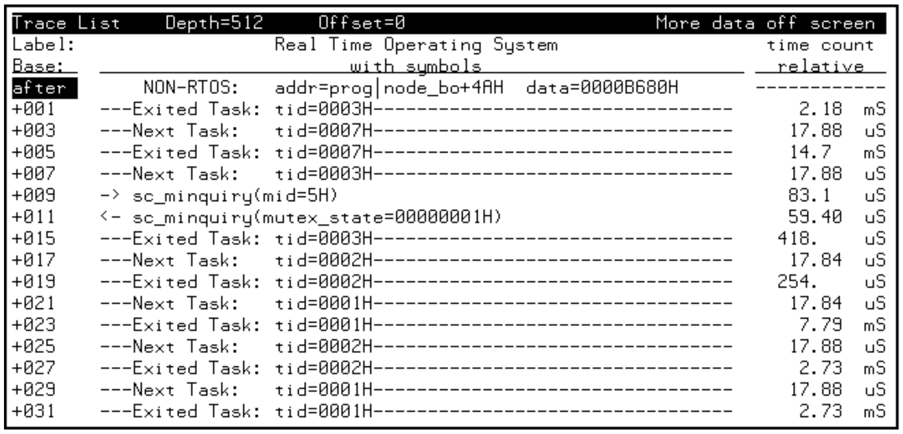


```
Trace List                 Offset=0                     More data off screen
Label:                 Real Time Operating System                time count
Base:                         with symbols                         relative
after        NON-RTOS:   addr=prog|producer+2E  data=00005284     ------------
+001     ---Exited Task : tid = 0x0001------------------------------   58.4   mS
+003     ---Next Task   : tid = 0x0002------------------------------   15.9   uS
+005     ---Exited Task : tid = 0x0002------------------------------   51.3   mS
+007     ---Next Task   : tid = 0x0005------------------------------   15.9   uS
+009     -> sc_sinquiry(sem_id=00000003)                               51.3   mS
+011     <- sc_sinquiry()                                              41.28  uS
+015     -> sc_spost(sem_id=00000003)                                  14.7   uS
+017     <- sc_spost()                                                 48.28  uS
+019     ---Exited Task : tid = 0x0005------------------------------  407.    uS
+021     ---Next Task   : tid = 0x0006------------------------------   15.9   uS
+023     -> sc_spend(sem_id=00000003, timeout=FOREVER)                 70.72  uS
+027     <- sc_spend()                                                 58.88  uS
+029     ---Exited Task : tid = 0x0006------------------------------   51.5   mS
+031     ---Next Task   : tid = 0x0007------------------------------   15.9   uS
+033     ---Exited Task : tid = 0x0007------------------------------   57.5   mS
```

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

To track all mutex calls (include task switches)

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



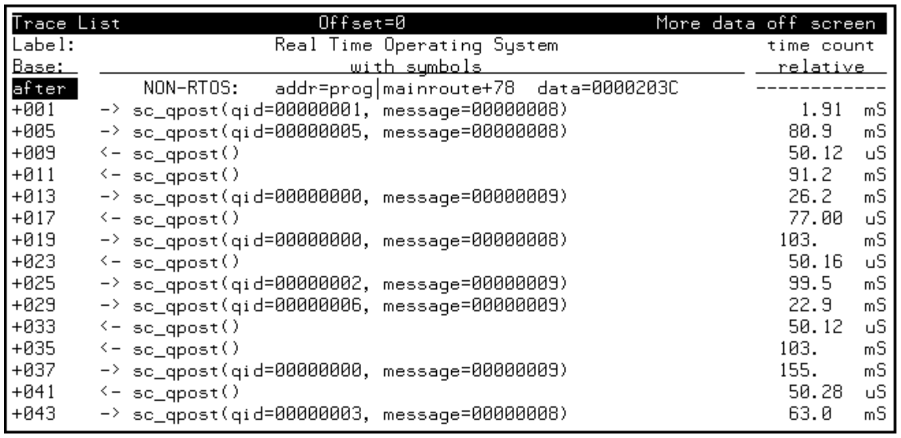
```
Trace List     Depth=512     Offset=0                     More data off screen
Label:                   Real Time Operating System                 time count
Base:                          with symbols                          relative
after       NON-RTOS:    addr=prog|node_bo+4AH  data=0000B680H     ------------
+001    ---Exited Task: tid=0003H---------------------------------     2.18  mS
+003    ---Next Task:   tid=0007H---------------------------------    17.88  uS
+005    ---Exited Task: tid=0007H---------------------------------    14.7   mS
+007    ---Next Task:   tid=0003H---------------------------------    17.88  uS
+009    -> sc_minquiry(mid=5H)                                        83.1   uS
+011    <- sc_minquiry(mutex_state=00000001H)                         59.40  uS
+015    ---Exited Task: tid=0003H---------------------------------   418.    uS
+017    ---Next Task:   tid=0002H---------------------------------    17.84  uS
+019    ---Exited Task: tid=0002H---------------------------------   254.    uS
+021    ---Next Task:   tid=0001H---------------------------------    17.84  uS
+023    ---Exited Task: tid=0001H---------------------------------     7.79  mS
+025    ---Next Task:   tid=0002H---------------------------------    17.88  uS
+027    ---Exited Task: tid=0002H---------------------------------     2.73  mS
+029    ---Next Task:   tid=0001H---------------------------------    17.88  uS
+031    ---Exited Task: tid=0001H---------------------------------     2.73  mS
```

This action key is used if you are only concerned about mutex 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.



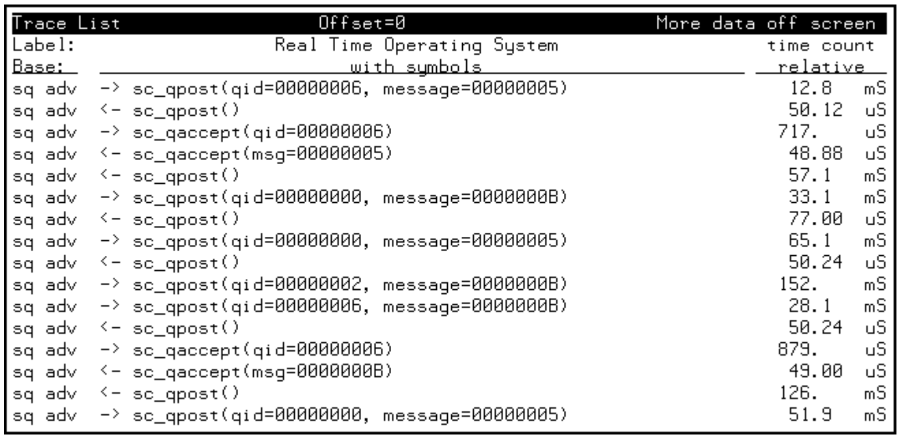
```
Trace List                   Offset=0                      More data off screen
Label:                 Real Time Operating System                 time count
Base:   ___________________with symbols______________________     relative
after        NON-RTOS:   addr=prog|mainroute+78  data=0000203C    ------------
+001    -> sc_qpost(qid=00000001, message=00000008)                  1.91  mS
+005    -> sc_qpost(qid=00000005, message=00000008)                 80.9   mS
+009    <- sc_qpost()                                               50.12  uS
+011    <- sc_qpost()                                               91.2   mS
+013    -> sc_qpost(qid=00000000, message=00000009)                 26.2   mS
+017    <- sc_qpost()                                               77.00  uS
+019    -> sc_qpost(qid=00000000, message=00000008)                103.    mS
+023    <- sc_qpost()                                               50.16  uS
+025    -> sc_qpost(qid=00000002, message=00000009)                 99.5   mS
+029    -> sc_qpost(qid=00000006, message=00000009)                 22.9   mS
+033    <- sc_qpost()                                               50.12  uS
+035    <- sc_qpost()                                              103.    mS
+037    -> sc_qpost(qid=00000000, message=00000009)                155.    mS
+041    <- sc_qpost()                                               50.28  uS
+043    -> sc_qpost(qid=00000003, message=00000008)                 63.0   mS
```

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  -> sc_qpost(qid=00000006, message=00000005)                  12.8   mS
sq adv  <- sc_qpost()                                                50.12  uS
sq adv  -> sc_qaccept(qid=00000006)                                 717.    uS
sq adv  <- sc_qaccept(msg=00000005)                                  48.88  uS
sq adv  <- sc_qpost()                                                57.1   mS
sq adv  -> sc_qpost(qid=00000000, message=0000000B)                  33.1   mS
sq adv  <- sc_qpost()                                                77.00  uS
sq adv  -> sc_qpost(qid=00000000, message=00000005)                  65.1   mS
sq adv  <- sc_qpost()                                                50.24  uS
sq adv  -> sc_qpost(qid=00000002, message=0000000B)                 152.    mS
sq adv  -> sc_qpost(qid=00000006, message=0000000B)                  28.1   mS
sq adv  <- sc_qpost()                                                50.24  uS
sq adv  -> sc_qaccept(qid=00000006)                                 879.    uS
sq adv  <- sc_qaccept(msg=0000000B)                                  49.00  uS
sq adv  <- sc_qpost()                                               126.    mS
sq adv  -> sc_qpost(qid=00000000, message=00000005)                  51.9   mS
```

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 "sc_qpost" and who is receiving them with "sc_qaccept".

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.

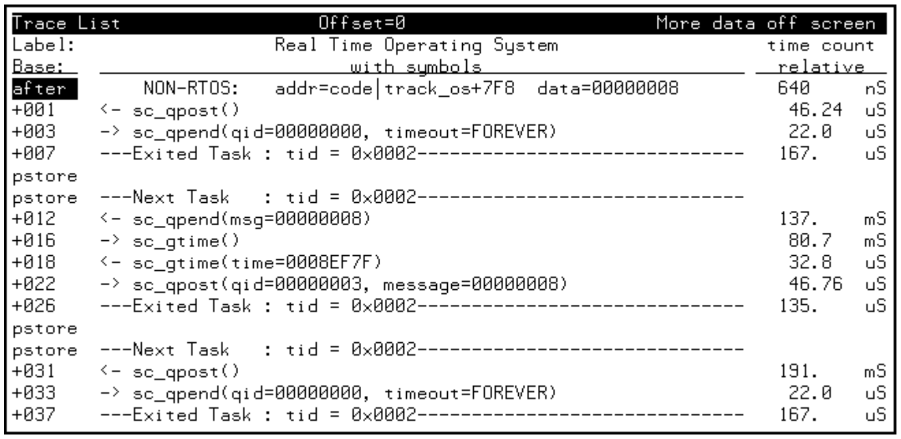
This section shows you how to:

- 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 display task and queue names.

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

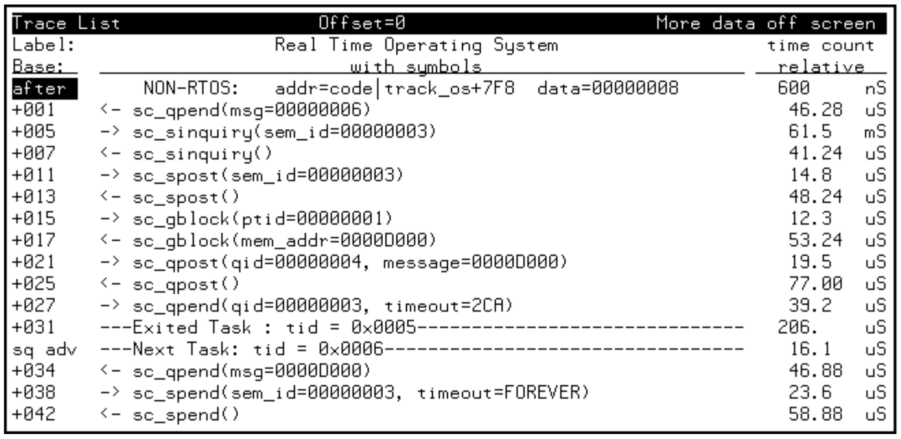


```
Trace List                  Offset=0                      More data off screen
Label:                 Real Time Operating System                  time count
Base:                         with symbols                          relative
after       NON-RTOS:   addr=code|track_os+7F8  data=00000008      640     nS
+001    <- sc_qpost()                                                46.24  uS
+003    -> sc_qpend(qid=00000000, timeout=FOREVER)                   22.0   uS
+007    ---Exited Task : tid = 0x0002------------------------------  167.    uS
pstore
pstore  ---Next Task   : tid = 0x0002------------------------------
+012    <- sc_qpend(msg=00000008)                                    137.    mS
+016    -> sc_gtime()                                                 80.7   mS
+018    <- sc_gtime(time=0008EF7F)                                    32.8   uS
+022    -> sc_qpost(qid=00000003, message=00000008)                   46.76  uS
+026    ---Exited Task : tid = 0x0002------------------------------  135.    uS
pstore
pstore  ---Next Task   : tid = 0x0002------------------------------
+031    <- sc_qpost()                                                191.    mS
+033    -> sc_qpend(qid=00000000, timeout=FOREVER)                    22.0   uS
+037    ---Exited Task : tid = 0x0002------------------------------  167.    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 191 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
after       NON-RTOS:   addr=code|track_os+7F8  data=00000008        600    nS
+001     <- sc_qpend(msg=00000006)                                    46.28  uS
+005     -> sc_sinquiry(sem_id=00000003)                              61.5   mS
+007     <- sc_sinquiry()                                             41.24  uS
+011     -> sc_spost(sem_id=00000003)                                 14.8   uS
+013     <- sc_spost()                                                48.24  uS
+015     -> sc_gblock(ptid=00000001)                                  12.3   uS
+017     <- sc_gblock(mem_addr=0000D000)                              53.24  uS
+021     -> sc_qpost(qid=00000004, message=0000D000)                  19.5   uS
+025     <- sc_qpost()                                                77.00  uS
+027     -> sc_qpend(qid=00000003, timeout=2CA)                       39.2   uS
+031     ---Exited Task : tid = 0x0005------------------------------ 206.    uS
sq adv   ---Next Task: tid = 0x0006---------------------------------  16.1   uS
+034     <- sc_qpend(msg=0000D000)                                    46.88  uS
+038     -> sc_spend(sem_id=00000003, timeout=FOREVER)                23.6   uS
+042     <- sc_spend()                                                58.88  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 IDs. For example, if you wanted to track only tasks 2 and 3, enter 2 in the first dialog box and 3 in the remaining dialog boxes.

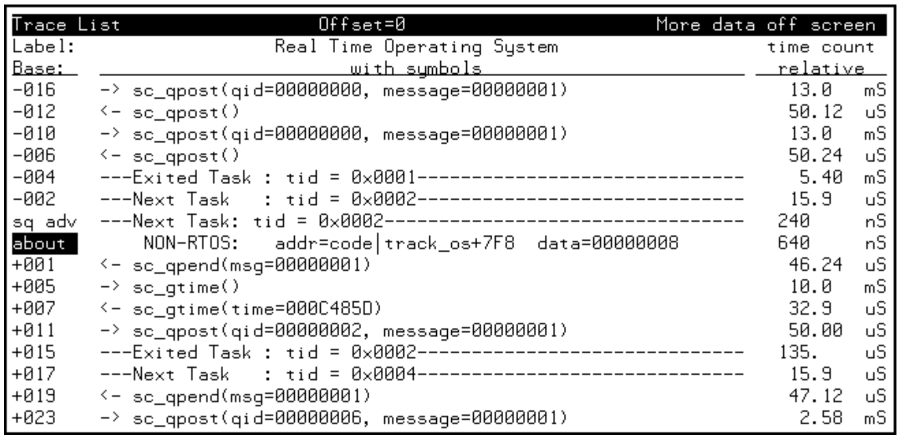
You can also edit the command file to create two new command files which would be used specifically for tracking two or three tasks.

To track about a specific task switch

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

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

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

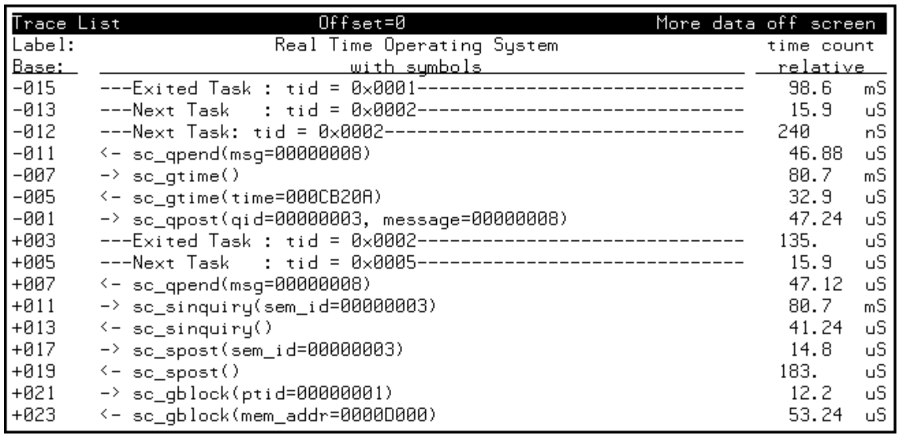


```
Trace List                  Offset=0                        More data off screen
Label:                 Real Time Operating System                  time count
Base:                         with symbols                          relative
-016     -> sc_qpost(qid=00000000, message=00000001)                 13.0   mS
-012     <- sc_qpost()                                               50.12  uS
-010     -> sc_qpost(qid=00000000, message=00000001)                 13.0   mS
-006     <- sc_qpost()                                               50.24  uS
-004     ---Exited Task : tid = 0x0001------------------------------  5.40  mS
-002     ---Next Task   : tid = 0x0002------------------------------ 15.9   uS
sq adv   ---Next Task: tid = 0x0002--------------------------------- 240    nS
about       NON-RTOS:   addr=code|track_os+7F8  data=00000008       640    nS
+001     <- sc_qpend(msg=00000001)                                   46.24  uS
+005     -> sc_gtime()                                               10.0   mS
+007     <- sc_gtime(time=000C485D)                                  32.9   uS
+011     -> sc_qpost(qid=00000002, message=00000001)                 50.00  uS
+015     ---Exited Task : tid = 0x0002------------------------------ 135.   uS
+017     ---Next Task   : tid = 0x0004------------------------------ 15.9   uS
+019     <- sc_qpend(msg=00000001)                                   47.12  uS
+023     -> sc_qpost(qid=00000006, message=00000001)                  2.58  mS
```

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 ID and then for the queue ID to which the task sends a message.



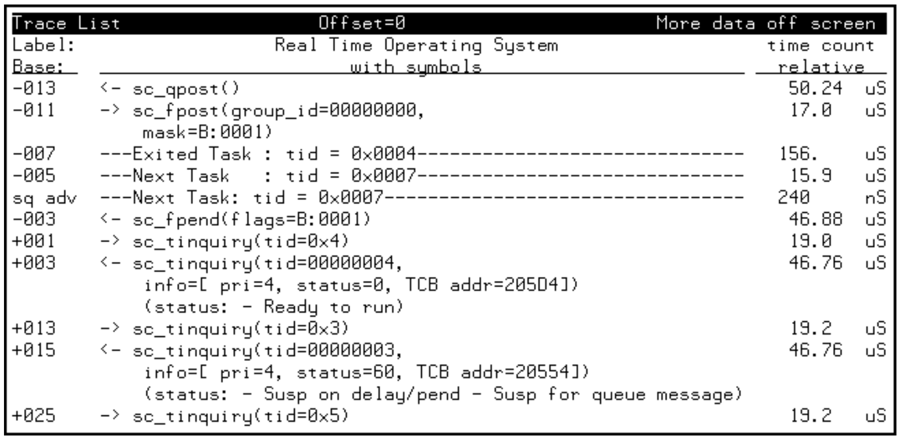
```
Trace List                   Offset=0                      More data off screen
Label:                Real Time Operating System                 time count
Base:                        with symbols                          relative
-015     ---Exited Task : tid = 0x0001------------------------------   98.6   mS
-013     ---Next Task   : tid = 0x0002------------------------------   15.9   uS
-012     ---Next Task: tid = 0x0002---------------------------------  240     nS
-011     <- sc_qpend(msg=00000008)                                     46.88  uS
-007     -> sc_gtime()                                                 80.7   mS
-005     <- sc_gtime(time=000CB20A)                                    32.9   uS
-001     -> sc_qpost(qid=00000003, message=00000008)                   47.24  uS
+003     ---Exited Task : tid = 0x0002------------------------------  135.    uS
+005     ---Next Task   : tid = 0x0005------------------------------   15.9   uS
+007     <- sc_qpend(msg=00000008)                                     47.12  uS
+011     -> sc_sinquiry(sem_id=00000003)                               80.7   mS
+013     <- sc_sinquiry()                                              41.24  uS
+017     -> sc_spost(sem_id=00000003)                                  14.8   uS
+019     <- sc_spost()                                                183.    uS
+021     -> sc_gblock(ptid=00000001)                                   12.2   uS
+023     <- sc_gblock(mem_addr=0000D000)                               53.24  uS
```

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 a flag is received by a specific task

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

You are prompted first for the task ID and then for the numeric value designating the flag(s). The flag 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
-013     <- sc_qpost()                                                   50.24  uS
-011     -> sc_fpost(group_id=00000000,                                  17.0   uS
             mask=B:0001)
-007     ---Exited Task : tid = 0x0004------------------------------    156.    uS
-005     ---Next Task   : tid = 0x0007------------------------------     15.9   uS
sq adv   ---Next Task: tid = 0x0007---------------------------------    240     nS
-003     <- sc_fpend(flags=B:0001)                                       46.88  uS
+001     -> sc_tinquiry(tid=0x4)                                         19.0   uS
+003     <- sc_tinquiry(tid=00000004,                                    46.76  uS
             info=[ pri=4, status=0, TCB addr=205D4])
             (status: - Ready to run)
+013     -> sc_tinquiry(tid=0x3)                                         19.2   uS
+015     <- sc_tinquiry(tid=00000003,                                    46.76  uS
             info=[ pri=4, status=60, TCB addr=20554])
             (status: - Susp on delay/pend - Susp for queue message)
+025     -> sc_tinquiry(tid=0x5)                                         19.2   uS
```

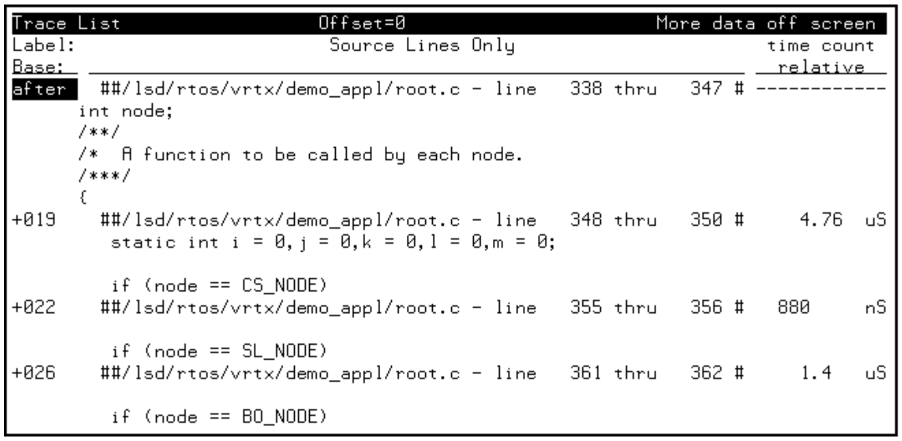
This measurement allows you to view the context under which a specific flag is received by a specific task. In the above example, we have captured a trace when task 7 received flag 0001.

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.



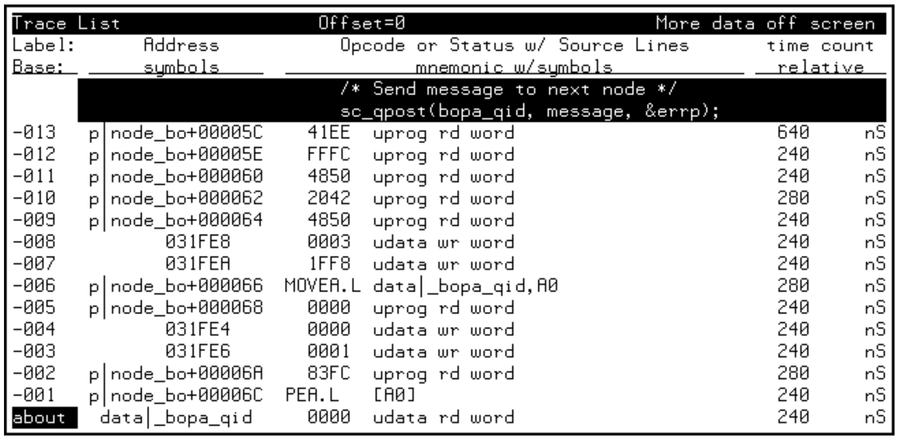
```
Trace List                     Offset=0                         More data off screen
Label:                          Source Lines Only                           time count
Base:   _______________________________________________________________   relative
after    ##/lsd/rtos/vrtx/demo_appl/root.c - line   338 thru   347 # ------------
      int node;
      /**/
      /*  A function to be called by each node.
      /***/
      {
+019     ##/lsd/rtos/vrtx/demo_appl/root.c - line   348 thru   350 #      4.76  uS
         static int i = 0, j = 0,k = 0,l = 0,m = 0;

         if (node == CS_NODE)
+022     ##/lsd/rtos/vrtx/demo_appl/root.c - line   355 thru   356 #   880      nS

         if (node == SL_NODE)
+026     ##/lsd/rtos/vrtx/demo_appl/root.c - line   361 thru   362 #      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 ID 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
                                  /* Send message to next node */
                                  sc_qpost(bopa_qid, message, &errp);
-013    p|node_bo+00005C    41EE  uprog rd word                       640     nS
-012    p|node_bo+00005E    FFFC  uprog rd word                       240     nS
-011    p|node_bo+000060    4850  uprog rd word                       240     nS
-010    p|node_bo+000062    2042  uprog rd word                       280     nS
-009    p|node_bo+000064    4850  uprog rd word                       240     nS
-008           031FE8       0003  udata wr word                       240     nS
-007           031FEA       1FF8  udata wr word                       240     nS
-006    p|node_bo+000066  MOVEA.L data|_bopa_qid,A0                   280     nS
-005    p|node_bo+000068    0000  uprog rd word                       240     nS
-004           031FE4       0000  udata wr word                       240     nS
-003           031FE6       0001  udata wr word                       240     nS
-002    p|node_bo+00006A    83FC  uprog rd word                       280     nS
-001    p|node_bo+00006C  PEA.L   [A0]                                240     nS
about    data|_bopa_qid     0000  udata rd word                       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.

To display task and queue names

1 Place task and queue names in the absolute file.

If you are using the HP cross compiler, do this by defining symbols using an "ASM" pragma in the C source file.

For example,

```
/* Place the task and queue names in the symbol table */
#pragma ASM
XDEF            ROOT_TASK
XDEF            QUEUE1
ROOT_TASK       EQU     $A1
QUEUE1          EQU     $B0
#pragma END_ASM
```

2 Define the USE_USER_MAPPING symbol.

This symbol can be created by including the following line in the linker command file:

```
public   USE_USER_MAPPING=$03FF
```

Code containing OS calls can be difficult to read because tasks, queues, and other objects are identified by the numeric value. This difficulty is overcome by using C define statements to attach names (symbols) to tasks and queues.

Symbolic task and queue names can be displayed for VRTX32 only.

See also "Symbolic Task and Queue Names in RTOS Traces" in the chapter, "How the RTOS Measurement Tool Works."

Tracking Accesses to Functions or Variables

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

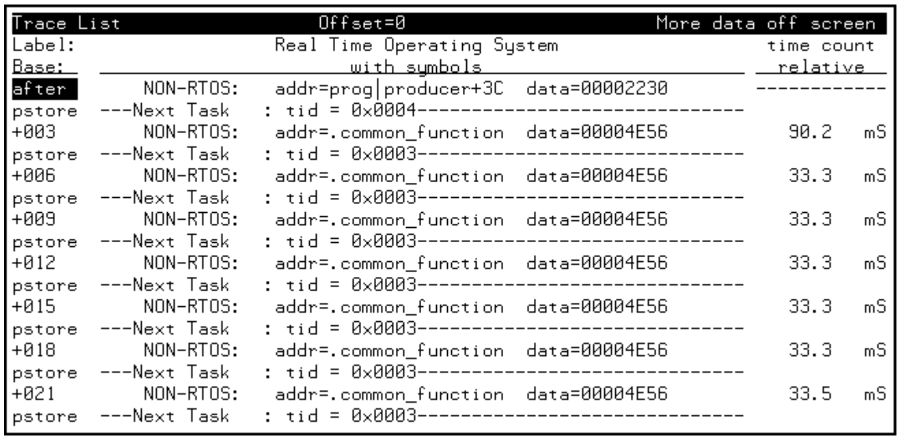
This section shows you how to:

- Track which tasks access a specific function or variable.

To track which tasks access a specific function or variable

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

You are prompted for a function or variable name.



```
Trace List                    Offset=0                    More data off screen
Label:                   Real Time Operating System                 time count
Base:                           with symbols                          relative
after       NON-RTOS:   addr=prog|producer+3C  data=00002230      ------------
pstore  ---Next Task   : tid = 0x0004------------------------------
+003        NON-RTOS:   addr=.common_function  data=00004E56           90.2   mS
pstore  ---Next Task   : tid = 0x0003------------------------------
+006        NON-RTOS:   addr=.common_function  data=00004E56           33.3   mS
pstore  ---Next Task   : tid = 0x0003------------------------------
+009        NON-RTOS:   addr=.common_function  data=00004E56           33.3   mS
pstore  ---Next Task   : tid = 0x0003------------------------------
+012        NON-RTOS:   addr=.common_function  data=00004E56           33.3   mS
pstore  ---Next Task   : tid = 0x0003------------------------------
+015        NON-RTOS:   addr=.common_function  data=00004E56           33.3   mS
pstore  ---Next Task   : tid = 0x0003------------------------------
+018        NON-RTOS:   addr=.common_function  data=00004E56           33.3   mS
pstore  ---Next Task   : tid = 0x0003------------------------------
+021        NON-RTOS:   addr=.common_function  data=00004E56           33.5   mS
pstore  ---Next Task   : tid = 0x0003------------------------------
```

All tasks that call a specific function or 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, stack pointer) is also provided. With this information, you can use the analyzer to monitor for stack overflow conditions.

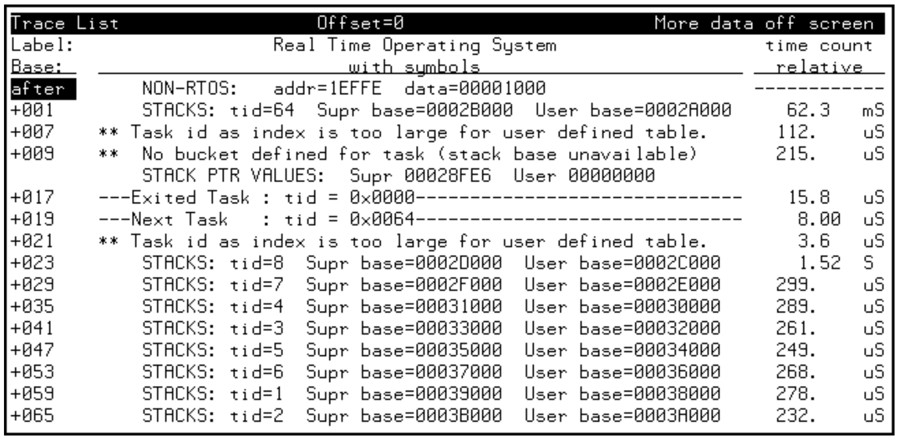
This section shows you how to:

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

To track only stack data

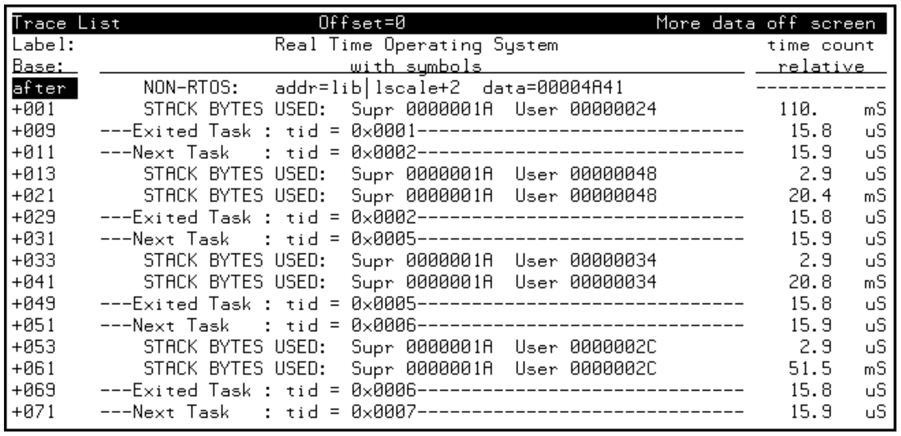
- Click on the **Stack Usage** action key (or run the **e_stack** command file by entering it on the command line).

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



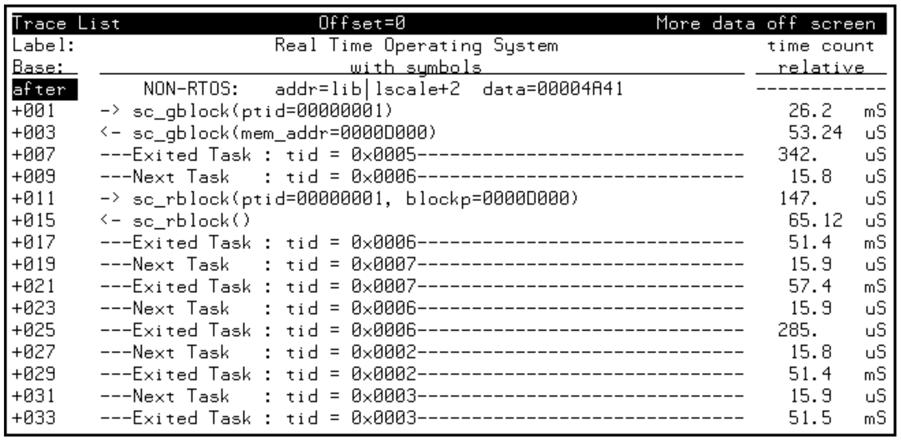
```
Trace List                  Offset=0                      More data off screen
Label:                 Real Time Operating System                   time count
Base:                         with symbols                           relative
after        NON-RTOS:   addr=1EFFE  data=00001000               ------------
+001         STACKS: tid=64  Supr base=0002B000  User base=0002A000     62.3   mS
+007    ** Task id as index is too large for user defined table.     112.    uS
+009    **   No bucket defined for task (stack base unavailable)     215.    uS
             STACK PTR VALUES:  Supr 00028FE6  User 00000000
+017    ---Exited Task : tid = 0x0000------------------------------   15.8   uS
+019    ---Next Task   : tid = 0x0064------------------------------    8.00  uS
+021    ** Task id as index is too large for user defined table.       3.6   uS
+023         STACKS: tid=8  Supr base=0002D000  User base=0002C000     1.52  S
+029         STACKS: tid=7  Supr base=0002F000  User base=0002E000    299.    uS
+035         STACKS: tid=4  Supr base=00031000  User base=00030000    289.    uS
+041         STACKS: tid=3  Supr base=00033000  User base=00032000    261.    uS
+047         STACKS: tid=5  Supr base=00035000  User base=00034000    249.    uS
+053         STACKS: tid=6  Supr base=00037000  User base=00036000    268.    uS
+059         STACKS: tid=1  Supr base=00039000  User base=00038000    278.    uS
+065         STACKS: tid=2  Supr base=0003B000  User base=0003A000    232.    uS
```

If you perform this same measurement while the application is running, you see the amount of stack used 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=lib|lscale+2  data=00004A41                 ------------
+001        STACK BYTES USED:  Supr 0000001A  User 00000024                110.    mS
+009    ---Exited Task : tid = 0x0001------------------------------        15.8    uS
+011    ---Next Task   : tid = 0x0002------------------------------        15.9    uS
+013        STACK BYTES USED:  Supr 0000001A  User 00000048                 2.9    uS
+021        STACK BYTES USED:  Supr 0000001A  User 00000048                20.4    mS
+029    ---Exited Task : tid = 0x0002------------------------------        15.8    uS
+031    ---Next Task   : tid = 0x0005------------------------------        15.9    uS
+033        STACK BYTES USED:  Supr 0000001A  User 00000034                 2.9    uS
+041        STACK BYTES USED:  Supr 0000001A  User 00000034                20.8    mS
+049    ---Exited Task : tid = 0x0005------------------------------        15.8    uS
+051    ---Next Task   : tid = 0x0006------------------------------        15.9    uS
+053        STACK BYTES USED:  Supr 0000001A  User 0000002C                 2.9    uS
+061        STACK BYTES USED:  Supr 0000001A  User 0000002C                51.5    mS
+069    ---Exited Task : tid = 0x0006------------------------------        15.8    uS
+071    ---Next Task   : tid = 0x0007------------------------------        15.9    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=lib|lscale+2  data=00004A41         ------------
+001     -> sc_gblock(ptid=00000001)                               26.2   mS
+003     <- sc_gblock(mem_addr=0000D000)                           53.24  uS
+007     ---Exited Task : tid = 0x0005------------------------------ 342.     uS
+009     ---Next Task   : tid = 0x0006------------------------------  15.8   uS
+011     -> sc_rblock(ptid=00000001, blockp=0000D000)             147.     uS
+015     <- sc_rblock()                                             65.12  uS
+017     ---Exited Task : tid = 0x0006------------------------------  51.4   mS
+019     ---Next Task   : tid = 0x0007------------------------------  15.9   uS
+021     ---Exited Task : tid = 0x0007------------------------------  57.4   mS
+023     ---Next Task   : tid = 0x0006------------------------------  15.9   uS
+025     ---Exited Task : tid = 0x0006------------------------------ 285.     uS
+027     ---Next Task   : tid = 0x0002------------------------------  15.8   uS
+029     ---Exited Task : tid = 0x0002------------------------------  51.4   mS
+031     ---Next Task   : tid = 0x0003------------------------------  15.9   uS
+033     ---Exited Task : tid = 0x0003------------------------------  51.5   mS
```

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

Displaying Traces

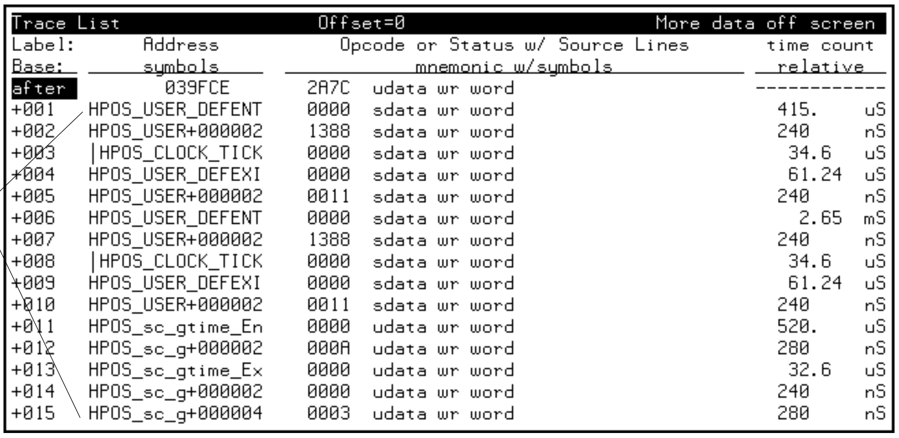
The normal "C" source code tracing is still available whenever you need to see your actual application code. You can switch between the normal "C" source code display and the RTOS measurements display with a simple click of an action key or by entering a display trace command.

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).

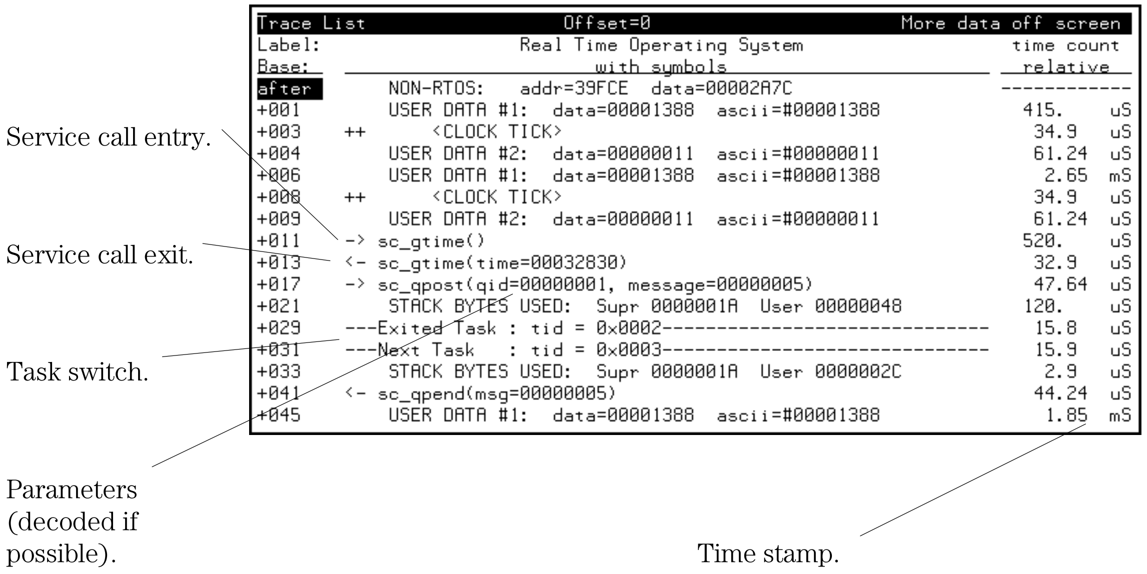


Writes to the data table.

```
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        039FCE       2A7C  udata wr word                      ------------
+001   HPOS_USER_DEFENT   0000  sdata wr word                        415.    uS
+002   HPOS_USER+000002   1388  sdata wr word                        240     nS
+003   |HPOS_CLOCK_TICK   0000  sdata wr word                         34.6   uS
+004   HPOS_USER_DEFEXI   0000  sdata wr word                         61.24  uS
+005   HPOS_USER+000002   0011  sdata wr word                        240     nS
+006   HPOS_USER_DEFENT   0000  sdata wr word                          2.65  mS
+007   HPOS_USER+000002   1388  sdata wr word                        240     nS
+008   |HPOS_CLOCK_TICK   0000  sdata wr word                         34.6   uS
+009   HPOS_USER_DEFEXI   0000  sdata wr word                         61.24  uS
+010   HPOS_USER+000002   0011  sdata wr word                        240     nS
+011   HPOS_sc_gtime_En   0000  udata wr word                        520.    uS
+012   HPOS_sc_g+000002   000A  udata wr word                        280     nS
+013   HPOS_sc_gtime_Ex   0000  udata wr word                         32.6   uS
+014   HPOS_sc_g+000002   0000  udata wr word                        240     nS
+015   HPOS_sc_g+000004   0003  udata wr word                        280     nS
```

To switch to the RTOS trace display

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



```
Trace List                  Offset=0                        More data off screen
Label:                    Real Time Operating System                time count
Base:                          with symbols                          relative
after       NON-RTOS:    addr=39FCE  data=00002A7C                  ------------
+001        USER DATA #1:  data=00001388  ascii=#00001388              415.    uS
+003   ++       <CLOCK TICK>                                            34.9   uS
+004        USER DATA #2:  data=00000011  ascii=#00000011              61.24  uS
+006        USER DATA #1:  data=00001388  ascii=#00001388               2.65  mS
+008   ++       <CLOCK TICK>                                            34.9   uS
+009        USER DATA #2:  data=00000011  ascii=#00000011              61.24  uS
+011   -> sc_gtime()                                                  520.    uS
+013   <- sc_gtime(time=00032830)                                      32.9   uS
+017   -> sc_qpost(qid=00000001, message=00000005)                     47.64  uS
+021        STACK BYTES USED:  Supr 0000001A  User 00000048           120.    uS
+029   ---Exited Task : tid = 0x0002------------------------------    15.8   uS
+031   ---Next Task   : tid = 0x0003------------------------------    15.9   uS
+033        STACK BYTES USED:  Supr 0000001A  User 0000002C             2.9   uS
+041   <- sc_qpend(msg=00000005)                                       44.24  uS
+045        USER DATA #1:  data=00001388  ascii=#00001388               1.85  mS
```

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.

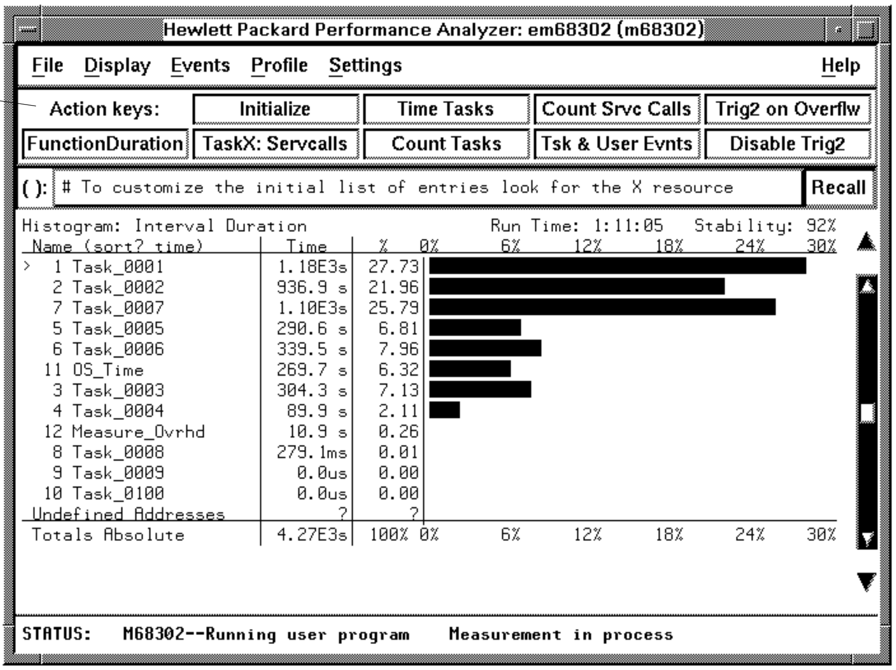
You may have noticed that the line numbers in the first column of the display are not sequential. This is because several trace states may be disassembled for each line in the RTOS trace display.

3

Making RTOS Measurements with the SPA

Making RTOS Measurements with the SPA

Action keys for RTOS measurements.



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

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

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

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

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

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

These tasks are grouped into the following sections:

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

Making Time Profile Measurements

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

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

This section shows you how to:

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

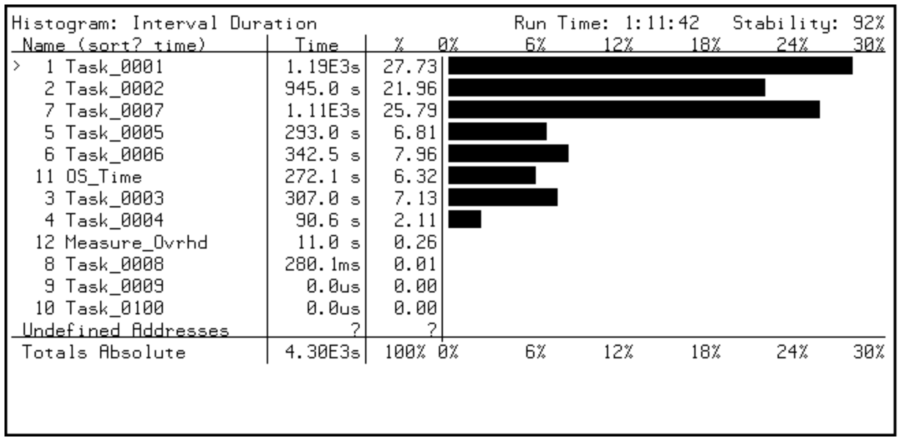
To define SPA events for tasks, service calls, and user events

- **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 generated an **s_init** command file by running the tool "rtos_edit_vrtx" or "rtos_edit_vrtxsa".

To display a time histogram of task events

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



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

You see that the task IDs 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.

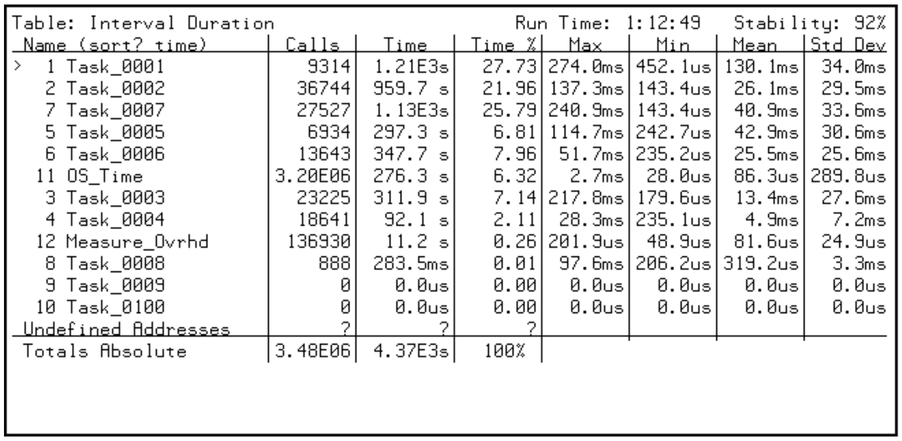
Note that one line of the histogram is labeled "OS_Time". This indicates how much time the application is spending in the OS kernel itself. This OS overhead measurement has some limitations however. Refer to the "OS Overhead Tracking" section in the "How the RTOS Measurement Tool Works" chapter for more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

Another line is labeled "Measure_Ovrhd". This indicates approximately 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 less than 1%.

To show a table of SPA events

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

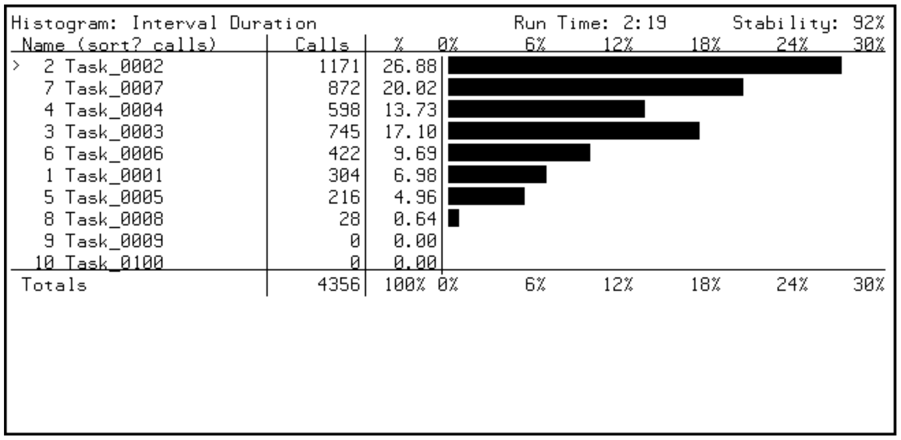
A raw numbers view of the accumulated data is displayed.



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

To display a count histogram of task events

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

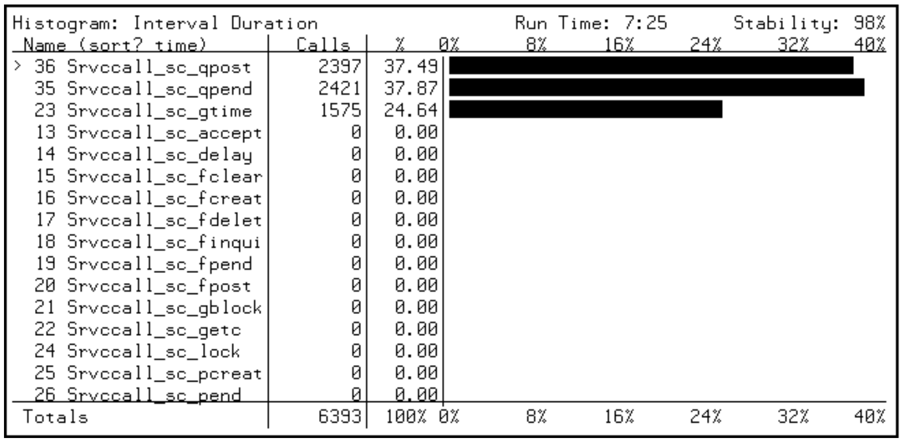


```
Histogram: Interval Duration                  Run Time: 2:19      Stability: 92%
 Name (sort? calls)      |   Calls |   %   0%      6%     12%     18%     24%     30%
>  2 Task_0002           |    1171 | 26.88|
   7 Task_0007           |     872 | 20.02|
   4 Task_0004           |     598 | 13.73|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

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

To measure only data from a specific task

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

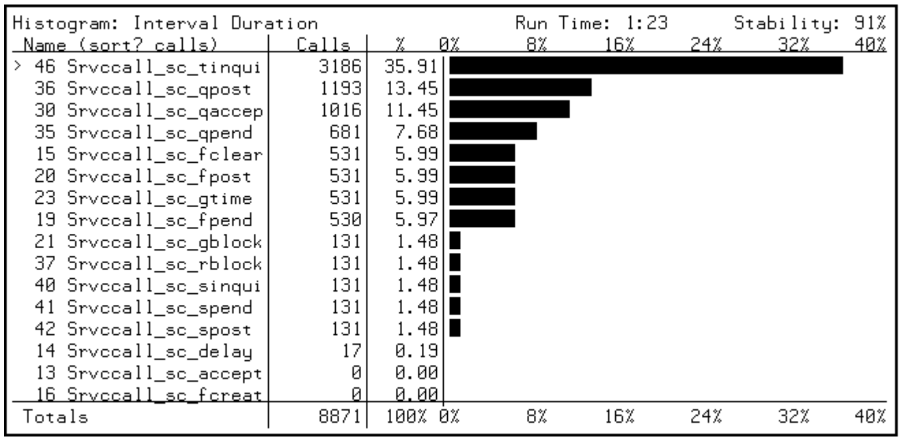


```
Histogram: Interval Duration                  Run Time: 7:25      Stability: 98%
 Name (sort? time)       |  Calls |   %   0%      8%      16%      24%      32%     40%
> 36 Srvccall_sc_qpost   |   2397 | 37.49|
  35 Srvccall_sc_qpend   |   2421 | 37.87|
  23 Srvccall_sc_gtime   |   1575 | 24.64|
  13 Srvccall_sc_accept  |      0 |  0.00|
  14 Srvccall_sc_delay   |      0 |  0.00|
  15 Srvccall_sc_fclear  |      0 |  0.00|
  16 Srvccall_sc_fcreat  |      0 |  0.00|
  17 Srvccall_sc_fdelet  |      0 |  0.00|
  18 Srvccall_sc_finqui  |      0 |  0.00|
  19 Srvccall_sc_fpend   |      0 |  0.00|
  20 Srvccall_sc_fpost   |      0 |  0.00|
  21 Srvccall_sc_gblock  |      0 |  0.00|
  22 Srvccall_sc_getc    |      0 |  0.00|
  24 Srvccall_sc_lock    |      0 |  0.00|
  25 Srvccall_sc_pcreat  |      0 |  0.00|
  26 Srvccall_sc_pend    |      0 |  0.00|
 Totals                  |   6393 | 100% 0%      8%      16%      24%      32%     40%
```

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).

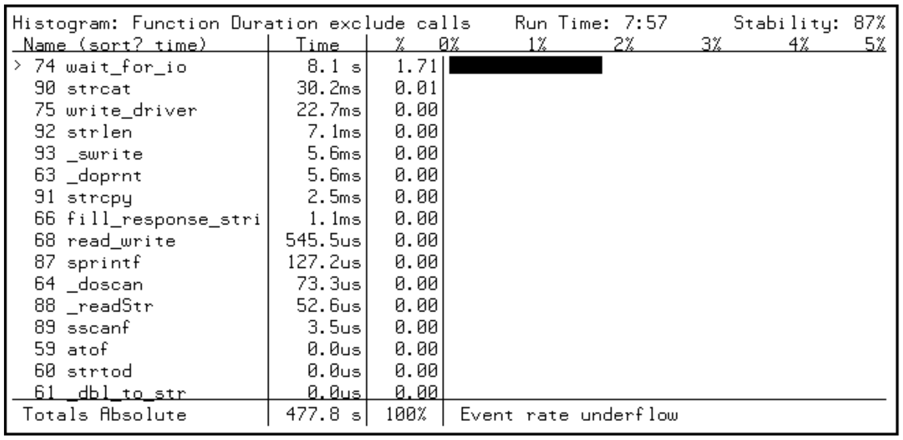


```
Histogram: Interval Duration                    Run Time: 1:23      Stability: 91%
 Name (sort? calls)      |  Calls |  %    0%      8%     16%     24%     32%    40%
> 46 Srvccall_sc_tinqui  |   3186 | 35.91|
  36 Srvccall_sc_qpost   |   1193 | 13.45|
  30 Srvccall_sc_qaccep  |   1016 | 11.45|
  35 Srvccall_sc_qpend   |    681 |  7.68|
  15 Srvccall_sc_fclear  |    531 |  5.99|
  20 Srvccall_sc_fpost   |    531 |  5.99|
  23 Srvccall_sc_gtime   |    531 |  5.99|
  19 Srvccall_sc_fpend   |    530 |  5.97|
  21 Srvccall_sc_gblock  |    131 |  1.48|
  37 Srvccall_sc_rblock  |    131 |  1.48|
  40 Srvccall_sc_sinqui  |    131 |  1.48|
  41 Srvccall_sc_spend   |    131 |  1.48|
  42 Srvccall_sc_spost   |    131 |  1.48|
  14 Srvccall_sc_delay   |     17 |  0.19|
  13 Srvccall_sc_accept  |      0 |  0.00|
  16 Srvccall_sc_fcreat  |      0 |  0.00|
 Totals                  |   8871 | 100% 0%      8%     16%     24%     32%    40%
```

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).

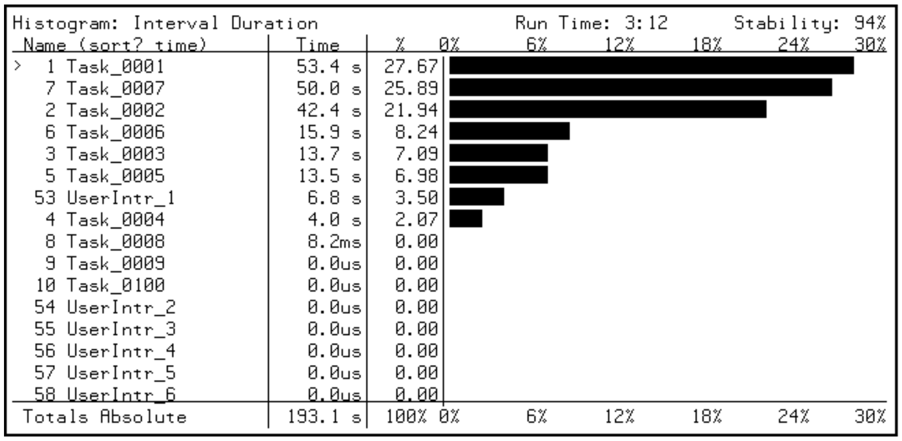


```
Histogram: Function Duration exclude calls    Run Time: 7:57      Stability: 87%
 Name (sort? time)      |   Time   |  %   0%      1%      2%      3%      4%      5%
> 74 wait_for_io        |   8.1 s  | 1.71|██████████████
  90 strcat             |  30.2ms  | 0.01|
  75 write_driver       |  22.7ms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

This performs a normal function duration profile measurement.

To show a histogram of task and user events

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



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

This measurement includes any user-defined events you may have set up. The example above shows that user event "UserIntr_1" uses 3.50% 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 Overflw action key.

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

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

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

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

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

To disable the SPA trig2

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

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

Note

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

Handling Multiple Projects on One Machine

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

To set up unique SPA windows for multiple projects

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

1 Run the $HP64000/bin/rtos_edit_vrtx or $HP64000/bin/rtos_edit_vrtxsa script.

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

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

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

Here are two ways to set an X resource:

- Edit the $HOME/.HP64_schemes/Softkey.Label file, as described on page 98.
- Place the X resource definition in a file, and run "xrdb -merge <filename>".

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

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

4

Customizing the RTOS Measurement Tool

Customizing the RTOS Measurement Tool

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

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

These tasks are grouped into the following sections:

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

Creating Your Own RTOS Measurements

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

```
********************************************************************************
*       -=- THIS DATA TABLE MUST NOT BE CHANGED IN ANY WAY -=-                 *
*      -=- The interpretation of 'traced' data is dependent -=-                *
*            -=- on the relative offsets of symbols -=-                        *
********************************************************************************
                      ; The name of this symbol MUST NOT CHANGE!!!
HP_RTOS_TRACK_START   ; It is required that the interface find this
                      ; symbol and pass it's value to the Interpreter
                      ; so the beginning of this table is known.
_HPOS_TASK_EXIT           DS.L  1
_HPOS_TASK_ENTRY          DS.L  1

**- Task Management -**
_HPOS_sc_tcreate_Entry         DS.L   3
_HPOS_sc_tcreate_Exit          DS.L   2
_HPOS_sc_tecreate_Entry        DS.L   8         ;VRTXsa
_HPOS_sc_tecreate_Exit         DS.L   2         ;VRTXsa
_HPOS_sc_tdelete_Entry         DS.L   2
_HPOS_sc_tdelete_Exit          DS.L   1
_HPOS_sc_tsuspend_Entry        DS.L   2
_HPOS_sc_tsuspend_Exit         DS.L   1
_HPOS_sc_tresume_Entry         DS.L   2
_HPOS_sc_tresume_Exit          DS.L   1
_HPOS_sc_tpriority_Entry       DS.L   2
_HPOS_sc_tpriority_Exit        DS.L   1
_HPOS_sc_tinquiry_Entry        DS.L   1
_HPOS_sc_tinquiry_Exit         DS.L   5
_HPOS_sc_lock_Entry            DS.L   1
_HPOS_sc_lock_Exit             DS.L   1
_HPOS_sc_unlock_Entry          DS.L   1
_HPOS_sc_unlock_Exit           DS.L   1
_HPOS_sc_delay_Entry           DS.L   1
_HPOS_sc_delay_Exit            DS.L   1
**- Memory Allocation -**
_HPOS_sc_gblock_Entry          DS.L   1
_HPOS_sc_gblock_Exit           DS.L   2
_HPOS_sc_rblock_Entry          DS.L   2
_HPOS_sc_rblock_Exit           DS.L   1
_HPOS_sc_pcreate_Entry         DS.L   4
_HPOS_sc_pcreate_Exit          DS.L   2
_HPOS_sc_pdelete_Entry         DS.L   2         ;VRTXsa
_HPOS_sc_pdelete_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_pextend_Entry         DS.L   3
_HPOS_sc_pextend_Exit          DS.L   1
_HPOS_sc_pinquiry_Entry        DS.L   1         ;VRTXsa
_HPOS_sc_pinquiry_Exit         DS.L   4         ;VRTXsa
_HPOS_sc_halloc_Entry          DS.L   2         ;VRTXsa
_HPOS_sc_halloc_Exit           DS.L   2         ;VRTXsa
_HPOS_sc_hcreate_Entry         DS.L   3         ;VRTXsa
_HPOS_sc_hcreate_Exit          DS.L   2         ;VRTXsa
_HPOS_sc_hdelete_Entry         DS.L   2         ;VRTXsa
_HPOS_sc_hdelete_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_hfree_Entry           DS.L   2         ;VRTXsa
_HPOS_sc_hfree_Exit            DS.L   1         ;VRTXsa
_HPOS_sc_hinquiry_Entry        DS.L   1         ;VRTXsa
_HPOS_sc_hinquiry_Exit         DS.L   4         ;VRTXsa
**- Communication & Synchronization -**
_HPOS_sc_post_Entry            DS.L   2
_HPOS_sc_post_Exit             DS.L   1
_HPOS_sc_pend_Entry            DS.L   2
```

```
_HPOS_sc_pend_Exit             DS.L   2
_HPOS_sc_accept_Entry          DS.L   1
_HPOS_sc_accept_Exit           DS.L   2
_HPOS_sc_maccept_Entry         DS.L   1         ;VRTXsa
_HPOS_sc_maccept_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_mcreate_Entry         DS.L   1         ;VRTXsa
_HPOS_sc_mcreate_Exit          DS.L   2         ;VRTXsa
_HPOS_sc_mdelete_Entry         DS.L   2         ;VRTXsa
_HPOS_sc_mdelete_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_minquiry_Entry        DS.L   1         ;VRTXsa
_HPOS_sc_minquiry_Exit         DS.L   2         ;VRTXsa
_HPOS_sc_mpend_Entry           DS.L   2         ;VRTXsa
_HPOS_sc_mpend_Exit            DS.L   1         ;VRTXsa
_HPOS_sc_mpost_Entry           DS.L   1         ;VRTXsa
_HPOS_sc_mpost_Exit            DS.L   1         ;VRTXsa

_HPOS_sc_qpost_Entry           DS.L   2
_HPOS_sc_qpost_Exit            DS.L   1
_HPOS_sc_qjam_Entry            DS.L   2
_HPOS_sc_qjam_Exit             DS.L   1
_HPOS_sc_qpend_Entry           DS.L   2
_HPOS_sc_qpend_Exit            DS.L   2
_HPOS_sc_qaccept_Entry         DS.L   1
_HPOS_sc_qaccept_Exit          DS.L   2
_HPOS_sc_qbrdcst_Entry         DS.L   2         ;VRTXsa
_HPOS_sc_qbrdcst_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_qcreate_Entry         DS.L   2
_HPOS_sc_qcreate_Exit          DS.L   2
_HPOS_sc_qdelete_Entry         DS.L   2         ;VRTXsa
_HPOS_sc_qdelete_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_qecreate_Entry        DS.L   3
_HPOS_sc_qecreate_Exit         DS.L   2
_HPOS_sc_qinquiry_Entry        DS.L   1
_HPOS_sc_qinquiry_Exit         DS.L   3

_HPOS_sc_fcreate_Entry         DS.L   1
_HPOS_sc_fcreate_Exit          DS.L   2
_HPOS_sc_fdelete_Entry         DS.L   2
_HPOS_sc_fdelete_Exit          DS.L   1
_HPOS_sc_fpost_Entry           DS.L   2
_HPOS_sc_fpost_Exit            DS.L   1
_HPOS_sc_fpend_Entry           DS.L   4
_HPOS_sc_fpend_Exit            DS.L   2
_HPOS_sc_fclear_Entry          DS.L   2
_HPOS_sc_fclear_Exit           DS.L   2
_HPOS_sc_finquiry_Entry        DS.L   1
_HPOS_sc_finquiry_Exit         DS.L   2

_HPOS_sc_saccept_Entry         DS.L   1         ;VRTXsa
_HPOS_sc_saccept_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_screate_Entry         DS.L   2
_HPOS_sc_screate_Exit          DS.L   2
_HPOS_sc_sdelete_Entry         DS.L   2
_HPOS_sc_sdelete_Exit          DS.L   1
_HPOS_sc_spost_Entry           DS.L   1
_HPOS_sc_spost_Exit            DS.L   1
_HPOS_sc_spend_Entry           DS.L   2
_HPOS_sc_spend_Exit            DS.L   1
_HPOS_sc_sinquiry_Entry        DS.L   1
_HPOS_sc_sinquiry_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 -=-                        *
********************************************************************************

**- Real-Time Clock -**
_HPOS_sc_adelay_Entry         DS.L   2         ;VRTXsa
_HPOS_sc_adelay_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_gclock_Entry         DS.L   1         ;VRTXsa
_HPOS_sc_gclock_Exit          DS.L   4         ;VRTXsa
_HPOS_sc_sclock_Entry         DS.L   3         ;VRTXsa
_HPOS_sc_sclock_Exit          DS.L   1         ;VRTXsa
_HPOS_sc_gtime_Entry          DS.L   1
_HPOS_sc_gtime_Exit           DS.L   1
_HPOS_sc_stime_Entry          DS.L   1
_HPOS_sc_stime_Exit           DS.L   1
_HPOS_sc_tslice_Entry         DS.L   1
_HPOS_sc_tslice_Exit          DS.L   1
_HPOS_ui_timer_Entry          DS.L   1
_HPOS_ui_timer_Exit           DS.L   1

**- Character I/O -**
_HPOS_sc_getc_Entry           DS.L   1
_HPOS_sc_getc_Exit            DS.L   1
_HPOS_sc_putc_Entry           DS.L   1
_HPOS_sc_putc_Exit            DS.L   1
_HPOS_sc_waitc_Entry          DS.L   1
_HPOS_sc_waitc_Exit           DS.L   1
_HPOS_sc_acceptc_Entry        DS.L   1         ;VRTXsa
_HPOS_sc_acceptc_Exit         DS.L   2         ;VRTXsa
_HPOS_ui_rxchr_Entry          DS.L   1
_HPOS_ui_rxchr_Exit           DS.L   1
_HPOS_ui_txrdy_Entry          DS.L   1
_HPOS_ui_txrdy_Exit           DS.L   2

**- Initialization -**
_HPOS_vrtx_init_Entry         DS.L   1
_HPOS_vrtx_init_Exit          DS.L   1
_HPOS_vrtx_go_Entry           DS.L   1
_HPOS_vrtx_go_Exit            DS.L   1
_HPOS_sc_gversion_Entry       DS.L   1         ;VRTXsa
_HPOS_sc_gversion_Exit        DS.L   1         ;VRTXsa

_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_TASK_BKT_UNDEF          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
_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 ID number 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.

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 VRTX code itself. This can usually be accomplished by storing all activity not in the range of the VRTX code (that is, **trace only address not range** <VRTX_start> **thru** <VRTX_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 long word (32-bit) task IDs 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 16-bit processor.

Examples

To track only queue and flag service calls:

```
trace only address range HPOS_sc_qpost_Entry thru
HPOS_sc_screate_Entry-1 <RETURN>
```

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

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

```
trace only address range HPOS_sc_qpost_Entry thru
HPOS_sc_screate_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_sc_qpost_Entry thru
HPOS_sc_screate_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.

To track only task 2 and queue service calls (for 16-bit processors):

```
trace enable address HPOS_TASK_ENTRY+2 data 2h disable
address HPOS_TASK_EXIT+2 data 2h only address range
HPOS_sc_qpost_Entry thru HPOS_sc_fcreate_Entry-1 or
HPOS_TASK_EXIT <RETURN>
```

This trace starts or resumes capturing data when 0002H is written to the second word of the task entry location and halts data capturing when 0002H is written to the second word of the task exit location. While enabled to capture data, the only states captured are the data table accesses that correspond to queue service calls or the first word of the task exit location.

To track only task 2 and queue service calls (for 32-bit processors):

```
trace enable address HPOS_TASK_ENTRY data 2h disable
address HPOS_TASK_EXIT data 2h only address range
HPOS_sc_qpost_Entry thru HPOS_sc_fcreate_Entry-1
<RETURN>
```

This is the same as the previous command, except the starts and halts are done on the HPOS_TASK_ENTRY and HPOS_TASK_EXIT locations since the full 32-bit ID is written in one cycle for 32-bit processors.

To trigger before an error return in task 3 (for 16-bit processors):

```
trace find_sequence HPOS_TASK_ENTRY+2 data 3h restart
HPOS_TASK_EXIT+2 data 3h trigger before
HPOS_CHECK_ERRORS data not 0 only address range
HP_RTOS_TRACK_START thru HP_RTOS_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.

To trigger before an error return in task 3 (for 32-bit processors):

```
trace find_sequence HPOS_TASK_ENTRY data 3h restart
HPOS_TASK_EXIT data 3h trigger before HPOS_CHECK_ERRORS
data not 0 only address range HP_RTOS_TRACK_START thru
HP_RTOS_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/B3081B/action_keys_302/e_trk1task
e_trk1tsknqs <RETURN>
```

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

```
... only address range HPOS_sc_qpost_Entry thru
HPOS_sc_fcreate_Entry-1 <RETURN>
```

So, edit the "e_trk1tsknqs" 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

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

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

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

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

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

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

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

The system-wide scheme file is stored in the directory "$HP64000/inst/rtos/<os>/HP64_schemes". Copy the "Softkey.App" file.

- **Edit the action key definitions.**

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

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

Earlier versions of the "emulrtos_<os>" script used the **emul700 -xrm** option to set up the action keys. You can still use this method if you wish, by using a **-xrm** option with the "emulrtos_<os>" script, but you must be sure to define all of the action keys you want to use. This is because all of the action key definitions are part of a *single* X resource string.

Example

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

Edit your "Softkey.Label" file.

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

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

```
\"Tsk X & Queues\"     \"e_trk1tsknqs\" \
```

as part of the "keyDefs" resource definition.

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

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

See Also

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

Part 2

Concept Guide

Topics that explain concepts and apply them to advanced tasks.

Part 2

5

How the RTOS Measurement Tool Works

How the RTOS Measurement Tool Works

The RTOS measurement tool lets you perform a real-time trace of 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 minimal, typically 3 to 5 data write instructions.

Service Call Tracking (for VRTX Assembly Based RTOS)

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 (such as VRTX). 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 VRTX, the data table requires about 1000 bytes.)

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 VRTX RTOS, there is a function called "sc_tcreate()" which creates a task. There are 5 registers which are assumed to be set before trapping to the kernel and 1 output register which is set by the kernel before it returns. One of the 5 input registers is D0 whose contents, as noted above, specify the function VRTX 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 4 long words are reserved for register values written when the "sc_tcreate" function is called. Upon return, the register contains information specific to the call; therefore, 1 long word is reserved for the "sc_tcreate" return value.

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

```
MOVEM.L  D1-D3/A0,HPOS_sc_tcreate_Entry   ; write out input data
TRAP     #VRTXTRAP                        ; trap to the kernel
MOVEM.L  D0-D1,HPOS_sc_tcreate_Exit       ; write out return data
```

and the respective data area declarations would look like:

```
HPOS_sc_tcreate_Entry      DS.L  4
HPOS_sc_tcreate_Exit       DS.L  1
```

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

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.

Service Call Tracking (for VRTXsa "C" Based RTOS)

For real-time operating systems whose main calling mechanism is through high-level function calls like "C", the above instrumentation method is slightly different. In this case, a ".h" header file is used to call an HP supplied function in place of the RTOS function in the user's code. This HP function writes out the necessary information to the data table using high-level language assignments and then calls the real OS function in a "daisy-chain" fashion. An example of an HP supplied function is:

```
int
HPIL_sc_create(void (*task)(void*), int tid, int pri, int *errp)
{
int   retval;

   HPOS_sc_tcreate_Entry[0] = (int) task;
   HPOS_sc_tcreate_Entry[1] = (int) tid;
   HPOS_sc_tcreate_Entry[2] = (int) pri;
   retval = sc_tcreate(task, tid, pri, errp);
   HPOS_sc_tcreate_Exit[0] = (int) *errp;
   HPOS_sc_tcreate_Exit[1] = (int) retval;

   HPOS_CHECK_ERRORS = (int) *errp;

   return(retval);
}
```

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 ID of the task being exited, one writing the task ID 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 IDs. The "rtos_edit_vrtx" or "rtos_edit_vrtxsa" 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 sc_tslice() 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 VRTX. (Make sure it is a word write to the HPOS_CLOCK_TICK location.) The memory location corresponding to 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 the 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.

Stack and Memory Tracking

Stack information such as pointers and bytes used 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 IDs of the application's tasks. This creates a data area that will be used to save the task's stack values.

2 The task start callout routine will save several data items: the task ID number, the memory locations in the Task Control Block that hold the stack pointer values, 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 used 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.

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_". The only symbols which do not use that prefix are _START_CALLOUT, and _SWITCH_CALLOUT.

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 Ascii              (1 long)
User Ascii              (1 long)
User Ascii              (1 long)
User Ascii              (1 long)
User Ascii              (1 long)
User Ascii              (1 long)
```

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
```

Symbolic Task and Queue Names in RTOS Traces

Code containing OS calls can be difficult to read because tasks, queues, and other objects are identified by the numeric value.

For VRTX32, this difficulty can be overcome by using C define statements to attach names (symbols) to tasks and queues.

```
/* defines allow C code to use symbolic task and queue names */

#define ROOT_TASK       0xA1
#define QUEUE1          0xB0

/********************************************************************/
/* FUNCTION */
void init_application()
/**/
/*  DESCRIPTION  Create the root task which will create everything else.
/*
/*  RETURN       Nothing
/**/
{
        static int errp;
        /* start the root task */
        sc_tcreate(&root_task, ROOT_TASK, 10, &errp);

        /* create a 1st queue */
        sc_qecreate(QUEUE1,3,FIFO,&errp);

        /* Remove this task (self) since it is no longer needed */
        sc_tsuspend(0, 0, &errp);

}
```

The symbolic association provided by the #define statments only exist during compilation. The symbols defined by #define during compilation do not exist in the absolute file.

In following trace fragment, tasks and queues are identified by the numerical value assigned to them. The trace would be more easily understood if the names assigned by the #define statments were displayed.

```
+001    - sc_tcreate(task_addr=1CE4,
400.    mS
            tid=A1, priority=A)
+004    <- sc_tcreate()
65.80  uS
+005    -> sc_qecreate(qid=B0, qsize=3, opt=FIFO)
7.00  uS
+008    <- sc_qecreate()
24.76  uS
+009    -> sc_tsuspend(SELF)
5.76  uS
+010    <- sc_tsuspend()
18.48  uS
+011    ---Exited Task : Tid = 00  -------------------------------
31.80  uS
```

```
+012    ---Begin Task  : Tid = A1  -------------------------------
1.36  uS
```

Displaying task and queue names in the RTOS trace

The RTOS trace will contain task and queue names when:

- the absolute contains task and queue names
- the absolute contains the symbol USE_USER_MAPPING

How to place task and queue names in the absolute

The task and queue names can be placed in the absolute file with several lines of assembly code. The use of the HP cross-compiler "ASM" pragma allow the assembly code to be placed in the C source file.

```
/* Place the task and queue names in the symbol table */
#pragma ASM
XDEF            ROOT_TASK
XDEF            QUEUE1
ROOT_TASK       EQU     $A1
QUEUE1          EQU     $B0
#pragma END_ASM
```

The USE_USER_MAPPING symbol

The RTOS inverse assembler has been modified to decode the user defined queue and task names. The definition of the symbol "USE_USER_MAPPING" enables the the decoding of queue and task names. This symbol can be created by including the following line in the linker command file.

```
public   USE_USER_MAPPING=$03FF    ; causes IAL to map names to user
defined symbols
public   USE_USER_MAPPING=$03FF    ; causes IAL to map names to user
defined symbols
```

The RTOS trace with symbolic task and queue names is much easier to understand.

```
+001    - sc_tcreate(task_addr=p|demo.root_task,
341.    mS
            tid=A1 :ROOT_TASK, priority=A)
+004    <- sc_tcreate()
65.84  uS
+005    -> sc_qecreate(qid=B0 :QUEUE1, qsize=3, opt=FIFO)
7.00  uS
+008    <- sc_qecreate()
24.76  uS
+009    -> sc_tsuspend(SELF)
5.76  uS
+010    <- sc_tsuspend()
18.48  uS
+011    ---Exited Task : Tid = 00  -------------------------------
```

```
31.80  uS
+012    ---Begin Task  : Tid = A1  :ROOT_TASK----------------------
1.36  uS
```

Limitations

User-defined names are not displayed when using the softkey interface.

The user-defined names are imposed upon the program name space. The user-defined names are mapped to address that may conflict with other symbols in the programs address map.

Symbols can be mapped to either a single address or to an address range. Symbols mapped to address ranges will not conflict to task and queue names which are mapped to a single address.

The symbol USE_USER_SYMBOLS is mapped to 0x3ff. This is the last byte in the vector table of the Motorola 68000 family processors. This address was chosen because it is uncommon to map this address to a symbol. The symbolic task and queue names will not be displayed unless you define this symbol. The task and queue symbols are mapped to the numbers used to represent them. It is up to the user to choose numbers that do not represent address to which other symbols have been mapped.

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:

```
.
.
queue_id = 2;
message = 1234;
sc_qpost(queue_id, message, &return_value);
.
.
```

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

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 "sc_qpost" 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:

```
.
.
->  sc_qpost(qid=00000002, message=00001234)
<-  sc_qpost()
.
.
```

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 queue ID parameter value of 2 is preceded with a "qid=". 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.

Service Call Entry and Exit and Task Switches

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

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

Given this behavior, application code that 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 "sc_qpost()" OS service call in the previous trace example sent a message to the queue, if another task of higher priority was waiting for a message at that same queue, then that task would now resume executing and the trace would look something like the following:

```
.
.
->  sc_qpost(qid=00000002, message=00001234)
--- Exited Task : tid = 0x0004 -------------
--- Next Task   : tid = 0x0007 -------------
<-  sc_qaccept(msg=00001234)
.
.
```

You can see that task 4, which sent the message has now exited and task 7, which had been waiting for a message with the "sc_qaccept()" OS service call, has now started up again. You can also see in the return parameter of the "sc_qaccept()" 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 VRTX real-time OS. Each RTOS Measurement Tool has a unique inverse assembler that is tailored to the particular real-time OS.

Part 3

Installation Guide

Instructions for installing and configuring the product.

Part 3

6

Installation

Installation

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

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

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

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

This chapter shows you how to:

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

When the Real-Time OS Measurement Tool is installed, you will have an enhanced emulation window with four additional entries available in the **File→Emul700** pulldown menu: **VRTX Emulator/Analyzer...** and **VRTX Performance Analyzer ...** for both the VRTX32 and VRTXsa products. These 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.

Note

If you have installed another Graphical User Interface after you installed the HP B3081B Real-Time Operating System Measurement Tool, you must re-run the HP B3081B "customize" script.

To install HP 9000 software

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

1 Check the HP-UX operating system version

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

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

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

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

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

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

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

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

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

```
/etc/update
```

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

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

   `/` (for the destination directory)

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

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

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

    ```
    y
    ```

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

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

To install Sun SPARCsystem software

Refer to the *Software Installation Guide* operating notice for instructions on installing software on Sun SPARCsystem computers.

Refer to the Graphical User Interface *User's Guide* for additional instructions on installing the Emulator/Analyzer interface software.

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 captures writes to the data table and decodes the stored trace information in an easy-to-read display.

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

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

instrumented service call library an interface library with callout routines and 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.

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

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.