
Developer's Guide

**HP B3082A
Custom Real-Time OS
Measurement Tool**

Notice

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

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

© Copyright 1992, Hewlett-Packard Company.

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

HP is a trademark of Hewlett-Packard Company.

Microtec is a registered trademark of Microtec Research Inc.

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

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

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

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

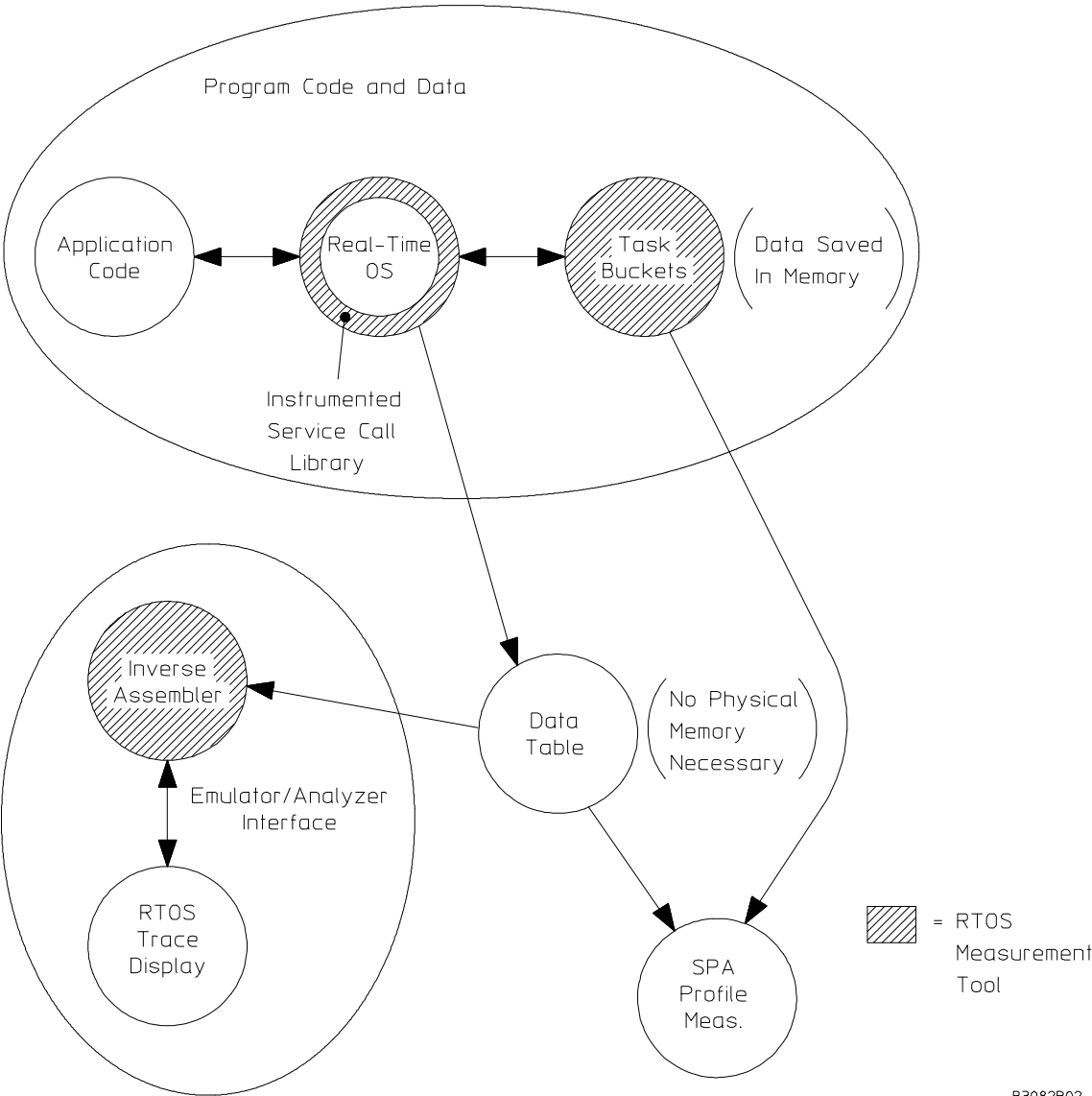
Printing History

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

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition 1 B3082-97000, December 1992

Creating a Measurement Tool for a Custom Real-Time OS



B3082B02

This product lets you create an HP 64700 emulation/analysis-based measurement tool for a custom real-time operating system.

An RTOS measurement tool lets users view program execution in the context of the real-time OS. For example, users can view service calls and their parameters, task switches, clock ticks, and dynamic memory usage.

In order to provide real-time OS measurements, you must "instrument" the real-time OS code, for example, by inserting instructions that write to a data table when service calls enter and exit.

With the HP 64700 emulation bus analyzer, the user captures writes to the data table. When the user displays the real-time OS trace, a special inverse assembler that you create decodes the captured information and displays it in an easy-to-read format.

You can also "instrument" real-time OS code so that users can measure time taken by tasks with the software performance analyzer.

You can make RTOS measurements easy for the user by creating command files that run when action keys are clicked.

In This Book

This book describes the HP B3082A Custom Real-Time Operating System Measurement Tool. It 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 five parts:

- Part 1. Quick Start Guide
- Part 2. User's Guide
- Part 3. Reference
- Part 4. Concept Guide
- Part 5. Installation Guide

If you ...	Then go to ...
Need to install the Custom RTOS Measurement Tool.	Part 5. The "Installation" chapter describes how to install the Custom RTOS Measurement Tools software.
Are unfamiliar with RTOS measurement tools.	Part 1. The Custom RTOS Measurement Tool comes with a demo. The first chapter shows you how to use the demo RTOS measurement tool, and the second chapter shows you how to modify the demo RTOS measurement tool to add support for a new OS service call.
Are familiar with the demo RTOS measurement tool and are ready to create your own RTOS measurement tool.	Part 2. The chapters in this part show you how to modify your OS code, create an inverse assembler, and set up Emulator/Analyzer and Software Performance Analyzer commands to perform real-time OS measurements.
Need information on the inverse assembly language (IAL) instructions.	Part 3. This reference information includes descriptions of the IAL instructions and the error messages that can occur when building the inverse assembler.
Would like more information about the demo RTOS measurement tool.	Part 4. The chapter in this part provides more detail about how the demo RTOS measurement tools works.

Contents

Part 1 Quick Start Guide

1 Using the Demo RTOS Measurement Tool

RTOS Measurement Tools — At a Glance 18

Product Overview 20

Making RTOS Measurements with the Emulator/Analyzer 24

Step 1: Change to the demo directory 24

Step 2: Start the Emulator/Analyzer interface 25

Step 3: Load the emulator configuration 26

Step 4: Load the real-time OS demo program 26

Step 5: Trace data table writes and display the real-time OS trace 27

Step 6: Run the program 27

Step 7: Use other action keys 28

RTOS Measurements with the SPA 29

Step 7: Start the Software Performance Analyzer Interface 29

Step 8: Define events for OS tasks 30

Step 9: Execute a time profile SPA measurement 31

Step 10: Use other action keys 32

Exiting and Releasing the HP 64700 Interfaces 33

Step 11: Exit and release the HP 64700 interfaces 33

2 Modifying the Demo RTOS Measurement Tool

Instrumenting OS Code for a New Service Call 37

- Step 1: Copy the demo OS files 37
- Step 2: Add an entry in the data table 38
- Step 3: Instrument the service call 39
- Step 4: Re-build your application 40

Modifying the Inverse Assembler 41

- Step 5: Copy the demo inverse assembler files 41
- Step 6: Add code to the inverse assembler 42
- Step 7: Re-build the inverse assembler 45
- Step 8: Install the inverse assembler 45

Modifying Related Scripts and Command Files 47

- Step 9: Add an entry in "create_12_call" and "create_12_call32" 47
- Step 10: Add an entry in "spabasecmd" 48
- Step 11: Create a new "s_init" file 49

Part 2 User's Guide

3 Modifying a Custom OS for Real-Time Measurements

- Emulation Bus Analyzer Measurements 54
- Software Performance Analyzer Measurements 55

Guidelines for OS Code Instrumentation 57

- Comment Instrumented Code 57
- Use Descriptive Symbol Names 58

Instrumenting Code for Real-Time OS Tracking	59
To track service calls	60
To track task switches	63
To track clock ticks	67
To track OS overhead	69
To track stack and memory	72
Organizing the Data Table	73
Group Locations for Easy Tracking	74
Create User-Defined Areas	74
Organizing the Task Data Buckets	75
Re-Building the OS and the Application	77
4 Writing the RTOS Inverse Assembler Code	
The Demo RTOS Inverse Assembler	81
Writing IAL Code	82
IAL Instructions	82
IAL Operands	83
IAL Program Control	85
Writing RTOS Inverse Assembler Code	87
To define strings, variables, and number formats	88
To assign data table locations to variables	90
To decode address information from the trace	93
To decode data information and output to the trace display	95
Building and Installing the RTOS Inverse Assembler	101
To build the RTOS inverse assembler	101
To install the RTOS inverse assembler	102

5 Making RTOS Measurements with the Emulator/Analyzer

Analyzer Resource Limitations	104
Trace Command Overview	105
Tracing Writes to the Data Table	107
To track everything	109
To track task switches and service calls	109
To track groups of service calls	110
To track a single service call	110
To include task switches when selectively tracking	111
To track two service calls	112
To track a single task and all OS activity within it	113
To track four tasks and all OS activity within them	114
To track about a specific task switch	116
To track about a specific task sending a message to a specific queue	117
To track activity after a function is reached	118
To track activity about the access of a variable by a specific task	119
To track which tasks access a specific function or variable	120
To track all memory calls (including task switches)	121
Displaying Traces	122
To switch to a normal trace display	123
To switch to the RTOS trace display	124

6 Making RTOS Measurements with the SPA

SPA Data Table Requirements	127
SPA Command Overview	127
Demo Data Table Entries for SPA	128
Making Time Profile Measurements	129
To define SPA events for tasks, service calls, and user events	130
To display a time histogram of task events	133
To show a table of SPA events	134
To display a count histogram of task events	134
To measure only data from a specific task	135
To show a table of service call invocations	136
To show a normal function duration histogram	137
To show a histogram of task and user events	139

Coordinating Measurements with the Emulator	140
To configure the emulation analyzer to receive trig2	140
To break on task time overflow	141
To disable the SPA trig2	143

7 Automating RTOS Measurements

Demo Action Keys and Related Command Files	146
Using Command Files	149
To place your measurements in command files	149
To use command file parameters to pass in variables	150
To use shell scripts from within command files	151
To define command file search directories	152
Using Action Keys	154
To place your measurements on action keys	154
To modify interface startup scripts	157

8 Installing New Custom OS Product Files

To answer install_rtos questions	161
To reinstall the original HP Custom RTOS product	162

Part 3 Reference

9 Inverse Assembler Language (IAL) Instructions

Instruction Set Summary	166
Executable Instructions	166
Pseudo Instructions	168
Predefined Communication Variables	169

Contents

Instruction Descriptions	170
ABORT - Leave inverse assembler	171
ADD - Add to accumulator	172
AND - Logical AND with accumulator	173
ASCII/ASC (Pseudo) - Define ASCII string	174
CALL - Transfer program control to label w/RETURN	175
CASE_OF - Conditional testing of variable or accumulator	176
COMPLEMENT - One's complement on accumulator	178
CONSTANT/CONST (Pseudo) - Define constant	179
DECREMENT - Decrement variable	180
DEFAULT_WIDTH (Pseudo) - Default width of display field	181
EXCLUSIVE_OR - Exclusive OR with accumulator	182
EXTRACT_BIT - Extract from accumulator	183
FETCH_POSITION - Get column number	184
FORMAT (Pseudo) - Format accumulator	185
GOTO - Transfer program control, no RETURN	186
IF - Compare operands	187
IF_NOT_MAPPED - Check for symbol in default map	188
INCLUSIVE_OR - Logical OR with accumulator	190
INCREMENT - Increment variable	191
INPUT - Input data	192
LOAD - Load accumulator	194
MAPPED_WIDTH (Pseudo) - Define maximum width of display	195
MARK_STATE - Analysis state display control	196
MAX_INSTRUCTION (Pseudo) - Limit instruction execution	197
NEW_LINE - Begin generating a new output line	198
NOP - No operation	199
OUTPUT - Output to output buffer	200
POSITION - Position column pointer	201
QUALIFY_MASK/_VALUE (Pseudos) - Set qualify specifications	202
RETURN - Return	203
ROTATE - Rotate accumulator contents	204
SEARCH_LIMIT (Pseudo) - Limit analysis search	205
SET - Set variable	206
STORE - Store value in accumulator	207
SUBTRACT - Subtract from accumulator	208
TAG_WITH - Flag analysis states	209
TWOS_COMPLEMENT - Two's complement on accumulator	210
VARIABLE/VAR (Pseudo) - Define and initialize a variable	211

10 IAL Builder Error Messages

Part 4 Concept Guide

11 Demo RTOS Measurement Tool Details

The "rtos_edit" Script 221

The "rtos_emul" Startup Script 230

The "rtos_spa" Startup Script 237

Scripts Run by the Action Key Command Files 241

 create_12_call 241

 create_12_call32 243

 get_task_number 246

Part 5 Installation Guide

12 Installation

To install HP 9000 software 251

To install Sun SPARCsystem software 253

Glossary

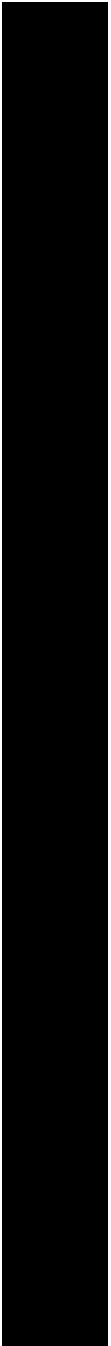
Index

Part 1

Quick Start Guide

A one-glance overview of the product and a few task instructions to help you get comfortable.

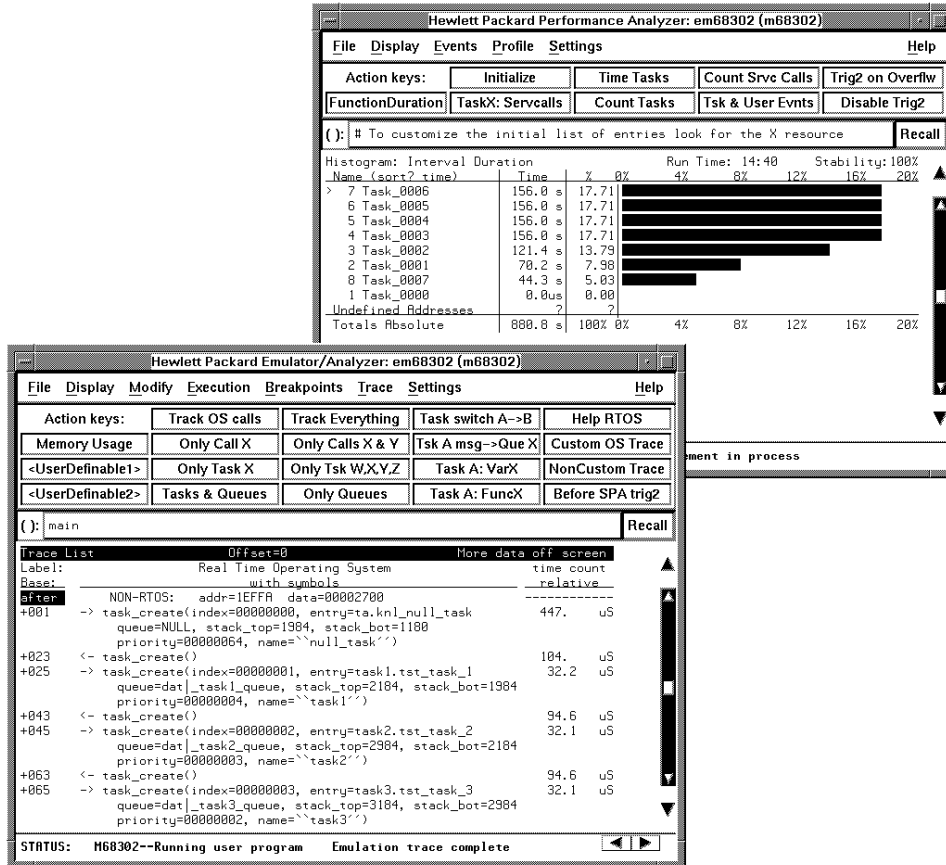
Part 1





Using the Demo RTOS Measurement Tool

RTOS Measurement Tools — At a Glance



A Real-Time Operating System (RTOS) Measurement Tool uses the HP 64700 emulation bus analyzer and software performance analyzer (SPA) to capture operating system software activity in real-time. It includes a specially developed inverse assembler that uses the trace display to show program flow information. The captured and displayed data is actually a series of memory writes to a data table. The histograms displayed by SPA are derived from special *task buckets*, which are groups of memory locations for each known task, that are written to by *callout routines* that run when tasks switch. The inverse assembler, modification of code to do the memory writes, data table,

Chapter 1: Using the Demo RTOS Measurement Tool RTOS Measurement Tools — At a Glance

task buckets, and callout routines are the items developers must customize to create their own RTOS Measurement Tool. Once *action keys*, which make common measurements available at the click of a button, are defined, a complete, easy to use, OS knowledgeable tool will be created.



When using an RTOS measurement tool 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.

When using an RTOS measurement tool with a 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.

Getting Started

The HP B3082A Custom Real-Time OS Measurement Tool comes with a small custom OS and a demo RTOS measurement tool. After installing the product, you can start the emulator/analyzer interface, load the custom OS and application code, and perform real-time OS measurements.

If you have installed the Software Performance Analyzer (SPA) HP 64708 into the HP 64700A Card Cage and you have installed the HP B1487B SPA Interface on your computer, you can perform profile measurements on RTOS tasks and service calls.

Once you have seen the types of measurements you can perform with a RTOS measurement tool, the next chapter shows you how to modify the demo RTOS measurement tool to include support for an additional OS service call.

These tutorial tasks are described in the following sections:

- Making RTOS Measurements with the Emulator/Analyzer
- Making RTOS Measurements with the SPA

Note

The demo OS code is for 68000 family processors. You must rewrite assembly language code in the demo if you're using other microprocessors.

Product Overview

The Custom RTOS Measurement Tool is designed to help someone with an application connected to some Real-Time Operating System understand the interaction between the those two pieces of code.

What the tool presents to the user is an ordered list of events along with time stamps for each of the events. Each event consists of either a function call or return from a service call (in other words, a call from the application to the OS) or a task switch. This list of events is presented to the user in the "trace" window of the emulator. Each "event" has been captured by the analyzer and

the data captured is interpreted by a special inverse assembler which displays the data in an OS unique form.



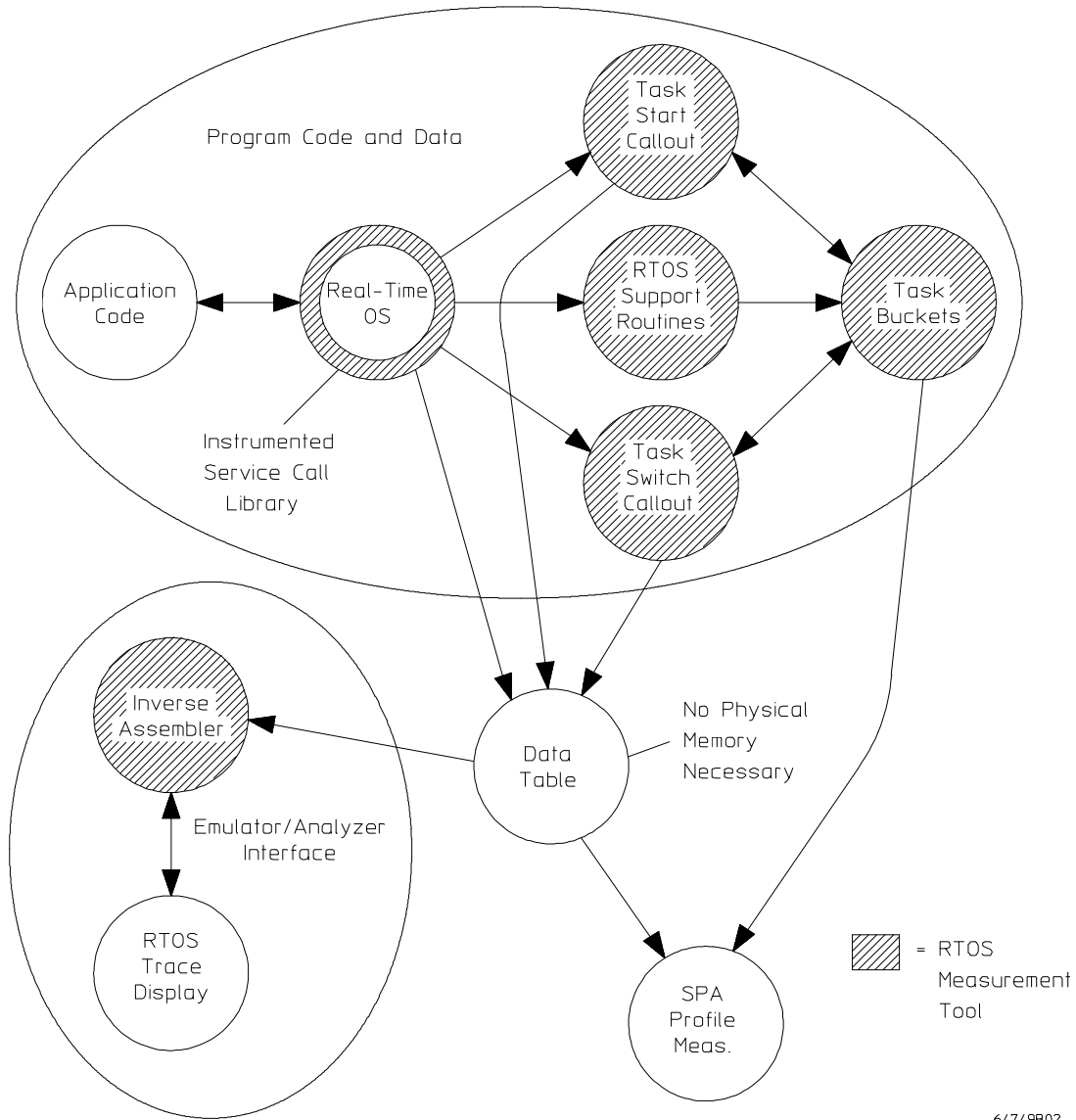
By viewing the specially displayed trace, a user is able to see what parameters were passed to the OS from the application, what values were returned, when the application switched to another task, which task it switched to, and what was the first OS interaction of the new task. All of this is repeated for the whole trace and a better understanding of the flow program is easily obtained.

All of this capability is done solely by "instrumenting" code with a few memory writes. Since it is just "data writes" that create the special trace, the flow of the data is explained below.

Normally, when an application calls the OS, it passes parameters to the OS, and the OS interprets the parameters depending on the function (that is, service call) being used. For the RTOS measurement tool, right before control is passed to the OS, all of the parameters are written to a unique location in a data table. (There is a "unique" location for every function call and return defined for the OS.)

A trace has been set up to capture any writes to this data table so the analyzer saves all of the parameters being passed to the OS along with the relative time at which they are being passed. Since the address is part of the information saved by the analyzer and the address was a unique location, it is known which function has been called (via the address), what parameter values were passed (via the data), and at what relative time this call occurred (via the time stamp). On return from the function, a similar thing occurs and the return function, parameters values, and relative time are also known.

Now, in between the call and the return, the OS may decide the some other task needs to be run. If the new task originally called the OS through some service call, it is going to return through that call. In the same manner as described before, all return data will be captured by the analyzer. The problem is that, although the call and returns of functions are now known, the task that is currently making those calls is not known. This is where the "switch callout" routine is used. Before the return is done, a routine defined as the switch callout will be called first. This routine has the task IDs of the two tasks being switched and writes these two IDs to two more unique locations in the data table. Since the IDs are unique and the locations are unique, the information stored by the analyzer will tell which task ended, which task started anew, and at what relative time the switch occurred. This gives all of the basic information needed for knowledge of the application/task-OS interaction.



64749B02

Some extra work may be done in the switch callout routine to also write the task IDs out to locations unique to the task ID (not just unique to whether the task was ending or starting). This gives a start and end point for "intervals" to be used by SPA in determining the length of time that each task executes.

Now that the interaction data is captured, how can it be displayed? A special inverse assembler can be written and installed which, when controlling the trace display, will interpret the analyzer data in an OS specific manner. Going back to the data, values have been written to specific locations in the data table. Since the inverse assembler is written, and must be written, with complete knowledge of the data table layout, all the inverse assembler needs is the starting address of the table to know the addresses of all other locations. Now, if the captured data is given to the special inverse assembler, it will match any captured addresses to the addresses it knows as data table locations. If a match occurs, the assembler knows it has data/parameter values for a specific function. The inverse assembler can then output the function name, the parameter names with their associated values, and the relative time. Similarly, for task switching, so long as the task switch locations for entry and exit are part of the data table, the inverse assembler will output some message like "Task Ending: task id = x" where "x" is the ID written by the switch callout routine to the entry/exit data table locations.

This is generally how the RTOS measurement tool works. There are a few extra details that need to be known, but these are covered in the manual's chapters. The RTOS measurement tool simply interprets special data writes to a table and displays them in the trace window with ASCII text annotating where the data came from.



Making RTOS Measurements with the Emulator/Analyzer

Before you can use the demo RTOS measurement tool, you should have already:

- Installed the emulator, emulation bus analyzer, and Graphical User Interface as described in their *User's Guide* manuals. The emulator/analyzer interface software must be version C.05.00 or greater.
- Installed the HP B30xx Real-Time Operating System Measurement Tool as outlined in the "Installation" chapter of this manual.

It's helpful if you are already familiar with your emulator and its interface before using the demo RTOS measurement tool.

Step 1: Change to the demo directory

- Change to the directory that contains the demo OS code.

If you have already installed the Custom RTOS Measurement Tool, the demo files will be in the \$HP64000/rtos/B3082A/src directory. If you haven't installed the product, refer to the "Installation" chapter.

```
$ cd $HP64000/rtos/B3082A/src <RETURN>
```

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.

Step 2: Start the Emulator/Analyzer interface

- With the emulator/analyzer interface already running, you can open the RTOS emulation window by choosing the **File→Emul700→Custom RTOS Emul** pulldown menu item.

If you do not see the **Custom RTOS Emul ...** and **Custom RTOS SPA ...** entries under the **File→Emul700** pulldown menu, review the installation procedure to make sure it was done correctly, and make sure the `/system/B3082/customize` script was run. If you still do not see these new entries, contact your Hewlett-Packard representative.

- If the emulator/analyzer interface is not already running, you can start the RTOS emulation window using the "rtos_emul" script found in "\$HPP64000/bin". This is a simple script which sets up a few things before calling **emul700** with your given emulator name. The syntax for using this script is:

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

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

The "emulator_name" is the logical emulator name given in the HP 64700 emulator device table file (`$HPP64000/etc/64700tab.net`).

Opening the RTOS emulation window does several things:

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

Step 3: Load the emulator configuration

- Choose **File→Load→Emulator Config...** from the pulldown menu, and use the File Selection dialog box to specify the configuration file to be loaded.

Or:

- Using the command line, enter the **load configuration < FILE>** command.

For example, if you're using the 68302 emulator:

```
load configuration config_302.EA <RETURN>
```

A few notes on configuring emulators for RTOS measurements:

- 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.

Step 4: Load the real-time OS demo program

- Choose **File→Load→Executable** and use the dialog box to select the absolute file.
- Using the command line, enter the **load < absolute_file>** command.

For example, if you're using the 68302 emulator:

```
load test302.x <RETURN>
```

Step 5: Trace data table writes and display the real-time OS trace

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

This runs the "e_trkcalls" command file which contains the following Emulator/Analyzer commands:

```
trace only address range HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1  
display trace real_time_os
```

The service call portion of the data table in the demo OS code begins at the address HP_RTOS_TRACK_START and ends at the address before HPOS_USER_DEFENTRY.

The **display trace real_time_os** command causes the RTOS inverse assembler to be used to decode the data captured by the analyzer. Information about the operation of the real-time OS is displayed in an easy-to-read format.

Step 6: Run the program

- Position the mouse pointer in the entry buffer and enter the address "main"; then, choose **Execution**→**Run**→**from** ().

Or:

- Using the command line, enter the following command:

```
run from main <RETURN>
```

Chapter 1: Using the Demo RTOS Measurement Tool
Making RTOS Measurements with the Emulator/Analyzer

A real-time OS trace similar to the following will be shown.

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=1EFFFF data=00002700	-----
+001	-> task_create(index=00000000, entry=ta.knl_null_task queue=NULL, stack_top=1984, stack_bot=1180 priority=00000064, name=`null_task`)	447. uS
+023	<- task_create()	104. uS
+025	-> task_create(index=00000001, entry=task1.tst_task_1 queue=dat _task1_queue, stack_top=2184, stack_bot=1984 priority=00000064, name=`task1`)	32.2 uS
+043	<- task_create()	94.6 uS
+045	-> task_create(index=00000002, entry=task2.tst_task_2 queue=dat _task2_queue, stack_top=2984, stack_bot=2184 priority=00000003, name=`task2`)	32.1 uS
+063	<- task_create()	94.6 uS
+065	-> task_create(index=00000003, entry=task3.tst_task_3 queue=dat _task3_queue, stack_top=3184, stack_bot=2984 priority=00000002, name=`task3`)	32.1 uS

Step 7: Use other action keys

- Click other action keys to view other sample RTOS measurements.

When the sample RTOS measurements require parameters, you will be prompted for task or queue IDs (for example, 1, 2, 3, etc.) or service call names (for example, send_message, get_message, etc.). In the graphical interface, enter the parameters and click the OK button. In the terminal or terminal emulation based softkey interface, enter the parameters on the command line.

Note that the "Memory Usage" action key will not work until you have completed the changes described in the tutorial in the "Modifying the Demo RTOS Measurement Tool" chapter.

RTOS Measurements with the SPA

If you wish to make profile measurements on RTOS tasks and service calls, in addition to installing the emulator/analyzer interface and the HP B3082 product, 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 the software performance analyzer and its interface before using the demo RTOS measurement tool.

Step 7: Start the Software Performance Analyzer Interface

- Bring up SPA window by choosing the **File**→**Emul700**→**Custom RTOS SPA** pulldown menu item.

You can also bring up the SPA window using the "rtos_spa" script found in "\$HP64000/bin". This is a simple script which sets up a few things before calling **emul700 -u xperf** with your given emulator name. The syntax for using this script is:

```
rtos_spa [-c <command_file>] <emulator_name>
```

Step 8: Define events for OS tasks

- Click the **Initialize** action key.

This runs the "s_init" command file which contains the following SPA commands:

```
define single_event named Task_0001 interval HPOS_Tenter_0001 thru HPOS_Texit_0001
define single_event named Task_0002 interval HPOS_Tenter_0002 thru HPOS_Texit_0002
define single_event named Task_0003 interval HPOS_Tenter_0003 thru HPOS_Texit_0003
define single_event named Task_0004 interval HPOS_Tenter_0004 thru HPOS_Texit_0004
define single_event named Task_0005 interval HPOS_Tenter_0005 thru HPOS_Texit_0005
define single_event named Task_0006 interval HPOS_Tenter_0006 thru HPOS_Texit_0006
define single_event named Task_0007 interval HPOS_Tenter_0007 thru HPOS_Texit_0007
#
define single_event named OS_Time interval HPOS_Start_Ovrhd thru HPOS_Stop_Ovrhd
define single_event named Measure_Ovrhd interval HPOS_Start_Intrusion thru \
HPOS_Stop_Intrusion
#
define single_event named Srvccall_task_create interval HPOS_task_create_Entry thru \
HPOS_task_create_Exit
define single_event named Srvccall_send_message interval HPOS_send_message_Entry thru \
HPOS_send_message_Exit
define single_event named Srvccall_get_message interval HPOS_get_message_Entry thru \
HPOS_get_message_Exit
define single_event named Srvccall_alloc_message interval HPOS_alloc_message_Entry \
thru HPOS_alloc_message_Exit
define single_event named Srvccall_free_message interval HPOS_free_message_Entry thru \
HPOS_free_message_Exit
#
define single_event named UserIntr_1 interval HPOS_USER_DEFENTRY thru \
HPOS_USER_DEFENTRY+3h
define single_event named UserIntr_2 interval HPOS_USER_DEFENTRY+4 thru \
HPOS_USER_DEFENTRY+7h
define single_event named UserIntr_3 interval HPOS_USER_DEFENTRY+8 thru \
HPOS_USER_DEFENTRY+0bh
define single_event named UserIntr_4 interval HPOS_USER_DEFENTRY+0ch thru \
HPOS_USER_DEFENTRY+0fh
define single_event named UserIntr_5 interval HPOS_USER_DEFENTRY+10h thru \
HPOS_USER_DEFENTRY+13h
define single_event named UserIntr_6 interval HPOS_USER_DEFENTRY+14h thru \
HPOS_USER_DEFENTRY+17h
```

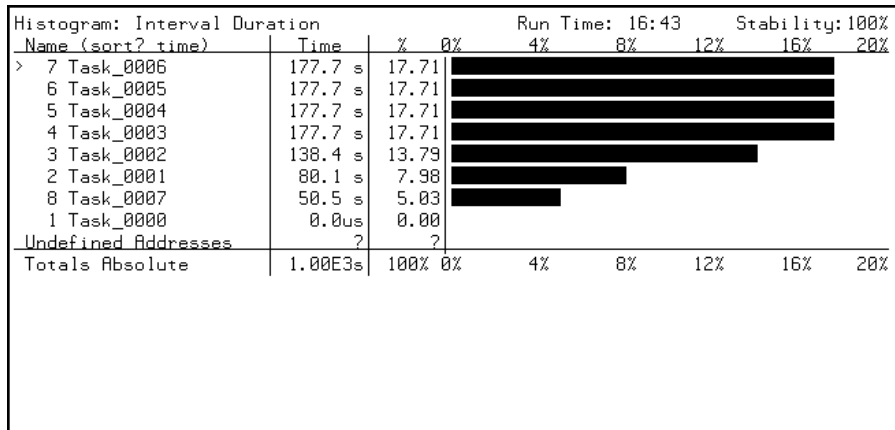
Step 9: Execute a time profile SPA measurement

- Click the **Time Tasks** action key.

This runs the "s_timetasks" command file which contains the following SPA commands:

```
stop_profile
select_events matching "Task_*"
setup_measurement enable off
setup_measurement disable off
profile interval_duration
display histogram data time
wait 2
display histogram sort_events time
display histogram rescale current_max
```

A time profile measurement similar to the following will be shown.





Step 10: Use other action keys

- Click other action keys to view other sample RTOS measurements.

Exiting and Releasing the HP 64700 Interfaces

The Emulator/Analyzer and SPA interfaces are exited and released in the same way.

This section shows you how to:

- Exit and release the HP 64700 interfaces.

Step 11: Exit and release the HP 64700 interfaces

- To exit the interface and release the emulator for access by other users, choose **File**→~~Exit~~→**Released**.

Or:

- In the emulator/analyzer interface command line, enter the following command:

```
end release_system <RETURN>
```





**Modifying the Demo RTOS
Measurement Tool**

Modifying the Demo RTOS Measurement Tool

There is a service call in the demo OS which hasn't been instrumented in any of the files. This chapter shows you how to instrument that service call in the demo OS code and modify the demo inverse assembler.

This section helps you better understand all the connections needed in producing a full measurement tool for your own custom real-time operating system.

These tutorial tasks are described in the following sections:

- Instrumenting OS Code for a New Service Call
- Modifying the Inverse Assembler
- Modifying Related Command Files and Scripts

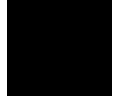
Note

The demo OS code is for 68000 family processors. You must rewrite assembly language code in the demo if you're using other microprocessors.

Instrumenting OS Code for a New Service Call

This section shows you how to:

- Copy the demo OS files.
- Add an entry in the data table.
- Instrument the service call.
- Re-build your application.



Step 1: Copy the demo OS files

- 1 Make a new directory to hold the custom OS files.

For example:

```
$ mkdir $HOME/rtos_demo <RETURN>
$ mkdir $HOME/rtos_demo/src <RETURN>
```

- 2 Change to the OS demo directory you just created.

For example:

```
$ cd $HOME/rtos_demo/src <RETURN>
```

- 3 Copy the demo files to the current directory.

```
$ cp $HP64000/rtos/B3082A/src/* . <RETURN>
```

- 4 Change permissions on the copied files.

```
$ chmod 644 * <RETURN>
```

Chapter 2: Modifying the Demo RTOS Measurement Tool

Instrumenting OS Code for a New Service Call

When installed, these files are given read-only permissions to prevent them from accidentally being removed. You must change the permissions so that you are able edit the files and save your changes (that is, read and write the files).

Step 2: Add an entry in the data table

- In the data table file, "table.c", add locations for the instrumentation of the "alloc_memory()" call.

Locations are needed for both the entry and exit of the routine. Because two parameters are passed in, the location for the entry needs two positions. For the return, there is one return value but there is also the return code of the function. Therefore, we also need two locations for the exit. The code added should look like:

```
long HPOS_alloc_mem_Entry[2];  
long HPOS_alloc_mem_Exit[2];
```

The position of the entries is relevant. Please note where you placed them. (Suggested is immediately after the "free_message" entries.)

Step 3: Instrument the service call

- 1 The information being passed into and returned from a function needs to be written to the data table. You do this by "instrumenting" the system calls in "mikeos.c".

If you find it necessary, you can write out all three inputs so you can see the value of the memory pointer also. The example below only instruments for the two parameters that actually have "data". For the routine "allocate_memory()", the instrumentation should look like:

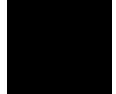
```
long allocate_memory(int region_id, int size, char **mem_area)
{
    long ret_value = NO_ERROR_RET;

    /* INSTRUMENTATION */
    HPOS_alloc_mem_Entry[0] = region_id;
    HPOS_alloc_mem_Entry[1] = size;
    .
    .
    /* INSTRUMENTATION */
    HPOS_alloc_mem_Exit[0] = ret_value;
    HPOS_alloc_mem_Exit[1] = (long) *mem_area;

    return(ret_value);
}
```

- 2 Don't forget to add definitions of the two symbols:

```
extern long HPOS_alloc_mem_Entry[];
extern long HPOS_alloc_mem_Exit[];
```



Step 4: Re-build your application

- Use the **make** command to re-build your application.

The "Makefile" included with the demo OS code includes the targets "test302", "test340", and "test020" for the 68302, 68340, and 68020 emulators, respectively. The different targets are necessary because different compilers are used when generating code for the different emulators. For example, if you are using the 68302 emulator, enter the command:

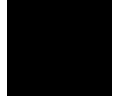
```
$ make test302 <RETURN>
```

This will create the demo executable and include all of the changes you just made to the application. (The demo make file assumes you have the appropriate HP AxLS C compiler.)

Modifying the Inverse Assembler

This section shows you how to:

- Copy the demo inverse assembler files.
- Add code to the inverse assembler.
- Re-build the inverse assembler.
- Install the inverse assembler.



Step 5: Copy the demo inverse assembler files

- 1 Make a new directory to hold the inverse assembler language source files.

For example:

```
$ mkdir $HOME/rtos_demo/interpreter <RETURN>
```

(The \$HOME/rtos_demo directory was created earlier in Step 1.)

- 2 Change to the inverse assembler source directory you just created.

For example:

```
$ cd $HOME/rtos_demo/interpreter <RETURN>
```

- 3 Copy the inverse assembler files to the current directory.

```
$ cp $HP64000/rtos/B3082A/interpreter/* . <RETURN>
```

- 4 Change permissions on the copied files.

```
$ chmod 644 * <RETURN>
```

Chapter 2: Modifying the Demo RTOS Measurement Tool

Modifying the Inverse Assembler

When installed, these files are given read-only permissions to prevent them from accidentally being removed. You must change the permissions so that you are able edit the files and save your changes (that is, read and write the files).

Step 6: Add code to the inverse assembler

- 1 Add a string definition for the function call in the "ial.S" source file.

Instead of defining strings, you could simply output the string in the function; however, it's good to have all the strings defined in a common place for the sake of consistency and to make modification of the inverse assembler easy. Add the following line with the rest of the string definitions.

```
ALLOC_MEM_STR      ASCII    "alloc_memory"
```

The ASCII instruction associates a name with a string. Inverse assembler language instructions that follow can use the name instead of a string operand.

- 2 Add an entry and exit variable for the function.

These variables are needed to keep the actual addresses of the table entries. Once set, these variables will be used to compare with every trace state to see if the state begins a set of states that describes an "allocate memory" event (either an entry or an exit of the function "allocate_memory().")

```
ALLOC_MEM_ENTRY    VARIABLE  0
ALLOC_MEM_EXIT     VARIABLE  0
```

The VARIABLE instruction defines and initializes an internal 32-bit storage location that can be used by the inverse assembler.

- 3 Add the IAL code that will set the variable's values to the actual addresses where the table entries are located within memory.

It is imperative that this code be added in the same relative place that the table entries appear in the table. This is because the code starts with the address of the beginning of the table and the rest of the entries addresses are calculated by just adding offsets from the previous address value. Since the "allocate

Chapter 2: Modifying the Demo RTOS Measurement Tool Modifying the Inverse Assembler

memory" table entry was placed after the "free message" entry, you must add code immediately after the calculation of the latter's address. Therefore, add:

```

      .
      .
      .
      STORE  FREE_MESSAGE_EXIT
      ADD    4
Added code:      STORE  ALLOC_MEM_ENTRY
Added code:      ADD    8
Added code:      STORE  ALLOC_MEM_EXIT
Added code:      ADD    8

      STORE  STRING_ARRAY
      ADD    12
      .
      .
      .
```

The STORE instruction copies the contents of the accumulator to the variable identified in the instruction operand. The ADD instruction adds the operand value to the accumulator and leaves the result in the accumulator. The accumulator is a 32-bit storage location through which arithmetic and logical operations are performed.

4 Add the IAL code that checks for the "allocate memory" data dump.

Whenever the inverse assembler is called, it looks at the address of the state passed in, compares it with the known addresses of the table entries and decides if the first part of a known "function data dump" has occurred.

In order to keep consistency, add the code that checks for "allocate memory" data right after the code that checks for "free message" data. The code checks for both entry and exit since they are separate events:

```

      .
      .
      .
      IF 31,0 = FREE_MESSAGE_EXIT THEN GOTO FREE_MESSAGE_RTN
Added code: IF 31,0 = ALLOC_MEM_ENTRY THEN GOTO ALLOC_MEMORY_FNC
Added code: IF 31,0 = ALLOC_MEM_EXIT THEN GOTO ALLOC_MEMORY_RTN

      IF 31,0 = USER_NUMERIC1 THEN GOTO USER_DEFINED_FNC
      .
      .
      .
```

The IF instructions above compare bits 31 through 0 of the accumulator (which has been loaded with an address value captured by the analyzer) with the values previously stored in the variables. If the accumulator and variable

Chapter 2: Modifying the Demo RTOS Measurement Tool

Modifying the Inverse Assembler

contents are equal, program control goes to the instruction label specified; if they're not equal program control goes to the instruction that follows.

- 5 Write the `ALLOC_MEMORY_FNC` and `ALLOC_MEMORY_RTN` IAL routines that display the captured data.

Once it is determined that a "function data dump" has occurred for the "allocate_memory()" function, the captured trace data needs to be interpreted and displayed in a readable form. The IAL routine "ALLOC_MEMORY_FNC" will do this for this entry and "ALLOC_MEMORY_RTN" will do this for the exit.

The first state of the captured trace will hold the "region_id" data and the next state will hold the "size" data. A string is output telling the parameter's name before the actual value is output. So the function will be:

```
ALLOC_MEMORY_FNC
    OUTPUT    CALL_STRING
    OUTPUT    ALLOC_MEM_STR
    OUTPUT    "(region_id="
    CALL      OUT_REL0_HEX
    OUTPUT    ", size="
    CALL      OUT_REL1_HEX
    GOTO      END_CALL
```

The OUTPUT instruction copies information to the "Real Time Operating System" column in the Emulator/Analyzer trace display. The operand of the OUTPUT instruction can be a string or a name previously defined for a string with an ASCII instruction. The CALL instruction transfers program control to the instruction label specified.

Notice that the routines "OUT_REL0_HEX" and "OUT_REL1_HEX" have been used. These are common routines that have been defined for both the version of the inverse assembler for 16-bit processors, "ial16.S", and the version of the inverse assembler for 32-bit processors, "ial32.S".

For the "exit" display, only the parameters being returned will be displayed by the IAL function-specific function. The return value will be displayed by the code associated with "ERROR_CODE_END".

```
ALLOC_MEMORY_RTN
    OUTPUT    RETURN_STRING
    OUTPUT    ALLOC_MEM_STR
    OUTPUT    "(memarea_ptr="
    CALL      OUT_REL1_HEX
    GOTO      ERROR_CODE_END
```

These are all the changes necessary. The IAL needs to be reassembled and installed, and your application needs to be rebuilt with the changed service call library.



Step 7: Re-build the inverse assembler

- Make either the 16- or 32-bit version of the inverse assembler.

The "Makefile" included with the demo inverse assembler code concatenates the "ial.S" source file with either "ial16.S" or "ial32.S" (versions of the common routines for 16- or 32-bit processors) before assembling the combined file to the inverse assembler.

To make the inverse assembler for 16-bit processors:

```
$ make ial16 <RETURN>
```

To make the inverse assembler for 32-bit processors:

```
$ make ial32 <RETURN>
```

Step 8: Install the inverse assembler

- 1 Become the root user.
- 2 Enter the **make** command with the appropriate target.

The make file included with the demo inverse assembler code also provides targets to install the inverse assembler in the appropriate emulator interface directory.

To install the inverse assembler for the 68302 emulator interface:

```
$ make install302 <RETURN>
```

Chapter 2: Modifying the Demo RTOS Measurement Tool

Modifying the Inverse Assembler

To install the inverse assembler for the 68020 emulator interface:

```
$ make install020 <RETURN>
```

To install the inverse assembler for the 68340 emulator interface:

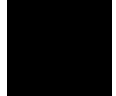
```
$ make install340 <RETURN>
```

Now, if you start the Emulator/Analyzer interface, load the demo OS program, and trace data table writes as described earlier in this chapter you'll be able to see the "alloc_memory()" service call in the RTOS trace display.

Modifying Related Scripts and Command Files

This section shows you how to:

- Add an entry in "create_12_call" and "create_12_call32".
- Add an entry in "spabasecmd".
- Create a new "s_init" file.



Step 9: Add an entry in "create_12_call" and "create_12_call32"

- 1 Change to the \$HP64000/rtos/B3082A directory.

```
$ cd $HP64000/rtos/B3082A <RETURN>
```

- 2 Edit either the file for 16-bit processors or the file for 32-bit processors.

If you have a 16-bit microprocessor, edit the "create_12_call" script. Change the lines:

```
free_message)  funcname=HPOS_free_message
                nextfunc=HPOS_String_array
                ;;
```

To:

```
free_message)  funcname=HPOS_free_message
                nextfunc=HPOS_alloc_mem      # changed
                ;;
alloc_memory)  funcname=HPOS_alloc_mem      # added
                nextfunc=HPOS_String_array  # added
                ;;
```

Chapter 2: Modifying the Demo RTOS Measurement Tool

Modifying Related Scripts and Command Files

Or, if you have a 32-bit microprocessor, edit the "create_12_call32" script.
Change the lines:

```
free_message) funcname=HPOS_free_message
               dataspace=2
               nextfunc=HPOS_String_array
               ; ;
```

To:

```
free_message) funcname=HPOS_free_message
               dataspace=2
               nextfunc=HPOS_alloc_mem      # changed
               ; ;
alloc_memory)  funcname=HPOS_alloc_mem      # added
               dataspace=4                 # added
               nextfunc=HPOS_String_array   # added
               ; ;                         # added
```

These two scripts are used by command files to create trace specifications that will track 1 or 2 specific functions.

Step 10: Add an entry in "spabasecmd"

- 1 Change to the \$HPP64000/rtos/B3082A/action_keys directory.

```
$ cd $HPP64000/rtos/B3082A/action_keys <RETURN>
```

- Edit the "spabasecmd" file.

You must edit this file so the setup of the SPA intervals will include the "allocate_memory" interval. To do this, insert the following line:

```
define single_event named Srvccall_alloc_mem interval \
HPOS_alloc_mem_Entry thru HPOS_alloc_mem_Exit
```

The "spabasecmd" file is used by the "rtos_edit" script when creating the "s_init" command file.

Step 11: Create a new "s_init" file

- Run the `$HP64000/bin/rtos_edit` script to create *only* a new "s_init" file.

```
$ $HP64000/bin/rtos_edit <RETURN>
```

When prompted, enter the task numbers 1 through 8. Don't request a special user environment.

After the script has completed and you have started the Emulator/Analyzer and SPA interfaces, the action keys "Only Call X" and "Only Calls X & Y" in the Emulator/Analyzer interface will work for the "allocate_memory" function, and the function will show up in the SPA histograms.





Part 2

User's Guide

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

Part 2



3



**Modifying a Custom OS for
Real-Time Measurements**

Modifying a Custom OS for Real-Time Measurements

Because the Emulation Bus Analyzer has only 8 state qualifier resources and 1 range qualifier resource, it would be impossible for the analyzer to capture OS activity at discrete locations for each service call, task switch, etc.

However, if the OS code is modified so that it writes information about service calls, task switches, etc. to a single data area, the analyzer can capture this OS information using only the single range qualifier resource.

The Software Performance Analyzer makes time profile measurements by keeping track of time between accesses of two locations. Because tasks don't always enter and exit at the same point, you can modify OS code so that pairs of locations are written to when tasks enter and exit. You can also modify the OS code if you wish to measure other things like OS overhead or RTOS measurement tool intrusion.

This OS code modification for real-time measurements is also known as *instrumenting* the OS code.

This chapter describes the following aspects of modifying a custom OS for real-time measurements:

- Guidelines for OS Code Instrumentation
- Instrumenting Code for Real-Time OS Tracking
- Organizing the Data Table
- Organizing the Task Data Buckets
- Re-Building the OS and the Application

Emulation Bus Analyzer Measurements

Because the Emulation Bus Analyzer captures information on the microprocessor's address and data bus, you must add code that causes the

microprocessor to write data to certain locations when certain events occur in the OS operation.

For example, when OS service calls are made, a user wants to see the parameter values. Likewise, a user wants to see the return values when service calls exit. Therefore, you must modify the OS code so that the microprocessor writes parameters and return values when service calls enter and exit.

Also, a user wants to see when tasks in the application code start and when they switch. Therefore, you must modify the OS code so that the microprocessor writes task ID information when tasks start or switch.

Because the Emulation Bus Analyzer captures information on the microprocessor address and data bus, it's not important that the information actually be saved; therefore, no real physical memory is needed. However, the locations that are written to must be in an address range that is writable.

The locations that are written to when service calls are entered or exited or when tasks start or switch should be located in the same address block to make trace commands as simple as possible and to make inverse assembly easier. This block of locations is called the *data table*.

Software Performance Analyzer Measurements

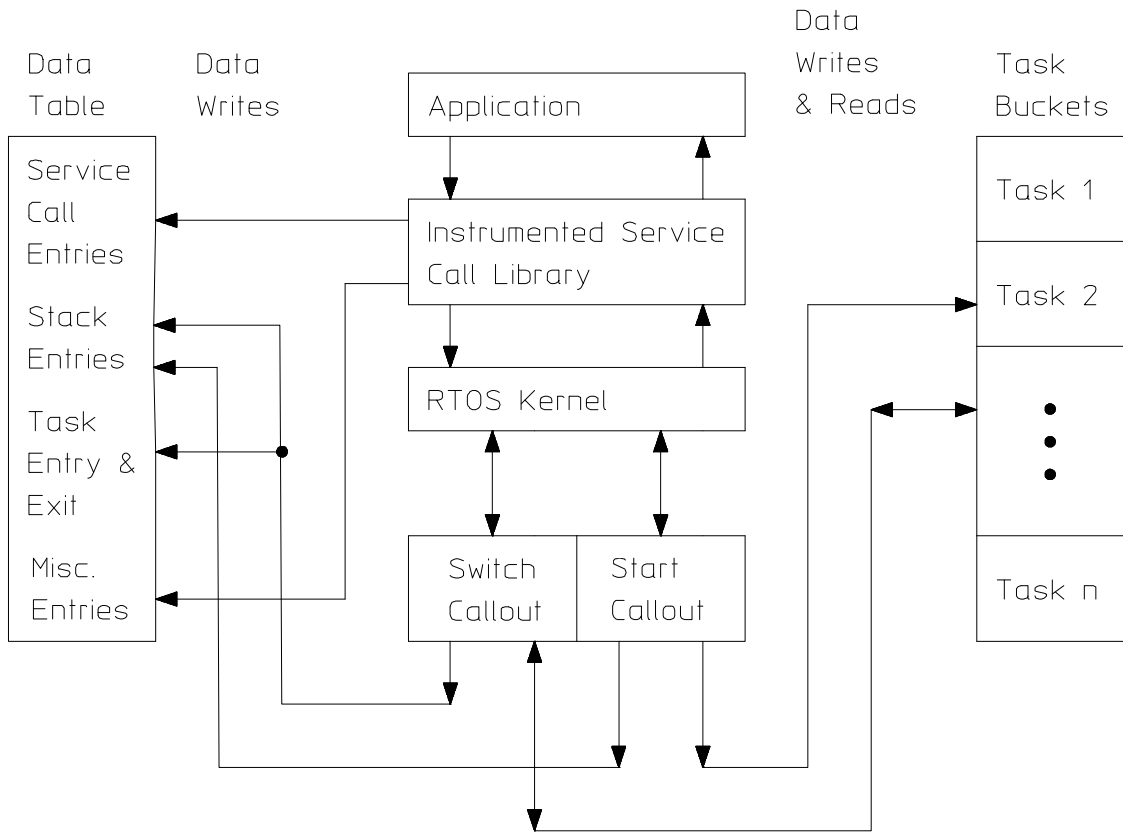
Because the Software Performance Analyzer measures time between accesses of different locations, you must modify the OS code so that the microprocessor accesses one location at the beginning of an event and another location at the end of an event.

For example, you can perform time profile measurements on service calls by measuring between accesses of the entry and exit locations in the data table. These locations are also used for service call tracking by the Emulation Bus Analyzer.

To perform time profile measurements on tasks, you must set up additional locations that are written to when tasks enter and exit. These locations may be in the data table. It's also common to place them with other locations set up for each task, called a task's *data bucket*, in which information like stack pointer values or task names are saved. A task's data bucket is different than the data table because physical memory is required if any of the information is to be saved.



Chapter 3: Modifying a Custom OS for Real-Time Measurements

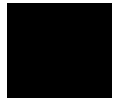


B3082B01

Guidelines for OS Code Instrumentation

It will be easier to identify and change the code added for real-time OS measurements if you:

- Comment instrumented code.
- Use descriptive symbol names.



Comment Instrumented Code

Though the level of intrusion introduced by "instrumented" OS code is very limited, it's best to comment added statements (or put them within conditional assembly or compilation directives) so they're easy to identify if it becomes necessary to reduce the intrusion.

Also, it's good to identify the different types, or levels, of real-time OS measurements that the instrumented code is for so that instrumentation for certain types of measurements can be identified if necessary. For example, you could define the following levels:

Level 1	Service call tracking (entry and return), task switching, clock ticks.
Level 2	Operating System overhead tracking, intrusion measurement, tracking service call error returns.
Level 3	SPA support - real time histogram of tasks.
Level 4	Stack tracking - creation and dynamic sizes of stacks.

You could comment instrumented code for these levels of measurements with `/* INSTRUMENTATION, HP-RTOS-Level-n */`.

Even within these levels, some subsets of measurement data could possibly be divided up. At any rate, the levels keep similar measurements together for easier editing and understanding.

Generally, when reducing the amount of intrusion, complete levels and any higher levels are removed (by commenting out the instrumented code or by placing it within conditional assembly or compilation directives).

Use Descriptive Symbol Names

It's imperative that the data table begin with the symbol `HP_RTOS_TRACK_START`. When you display the real-time OS trace, the address of this symbol is passed to the RTOS inverse assembler where it is used to calculate the addresses of the rest of the locations in the data table.

Also, it's best to use symbol names that start with the same first few characters so that symbols for instrumented code are not confused with the existing OS code symbols. For example, for all data table locations you could use names that begin with "HPOS_".

Instrumenting Code for Real-Time OS Tracking

In order to make RTOS measurements, a few instructions must be added to the OS code. The level of intrusion introduced by these instructions is very limited. The simplest level of RTOS measurements require only two 68xxx MOVEM assembly language instructions for each service call and two-instructions for each task switch. This level provides for basic tracking of the service call input and output parameters and task switching.

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

It's important that instrumented code be commented (or placed within conditional assembly or compilation directives) so that if the intrusion introduced becomes a problem, you can comment out some of the added instructions to find the right balance between intrusion and debugging capabilities.

This section shows you how to:

- Track service calls.
- Track task switches.
- Track clock ticks.
- Track OS overhead.
- Track stack and memory.



To track service calls

- Modify OS code to copy parameters and return values to the data table when service calls enter and exit.

Generally, you must do the following things:

- 1 Declare the data locations as externals.
- 2 Instrument the service call.
- 3 Add locations to the data table.

There are a couple methods of instrumenting real-time OS code to track service calls:

- If your real-time OS is written in assembly language, there is usually an *interface library* that allows high-level language applications to call the assembly language based OS service routines. Interface libraries are a good place to instrument code for real-time OS tracking. (This is the best method because there is less intrusion with one MOVEM instruction.)
- If your real-time OS is written in C, you can modify the code itself by adding statements at the beginning and end of service call functions that copy parameters and return values to an array in the data table. (This method was used in the demo shown in the "Quick Start Guide" part of the manual.)

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 real-time OS kernel. (These instructions, however, do not give you any knowledge about the tasks that are running or when tasks switch.)

Instrumenting an Interface Library

An interface library is a set of functions that correspond directly to each routine available from the real-time OS. These functions are called *service calls* of the real-time OS.

Each function in the interface 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

Chapter 3: Modifying a Custom OS for Real-Time Measurements Instrumenting Code for Real-Time OS Tracking

executed to pass control to the real-time OS 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.)

To instrument the interface library, you would add code that writes the contents of the registers used to specific locations in the data table. A simple 68xxx MOVEM instruction can be used to write the contents of multiple registers to the data table. One 68xxx MOVEM is done right before the "trap" instruction and one is done upon return.

Examples

To instrument real-time OS code written in C:

First, declare the data table locations as externals:

```
extern long HPOS_alloc_mem_Entry[];
extern long HPOS_alloc_mem_Exit[];
```

Then, instrument the service call so that parameters and return values are written on entry and exit:

```
/* _____ */
**
** PROCEDURE NAME:  allocate_memory()
**
** DESCRIPTION:
**
** _____ */
*/
long allocate_memory(int region_id, int size, char **mem_area)
{
    long ret_value = NO_ERROR_RET;

    /* INSTRUMENTATION */
    HPOS_alloc_mem_Entry[0] = region_id;
    HPOS_alloc_mem_Entry[1] = size;

    /* Initalize the return pointer */
    *mem_area = NULL;

    if ((size + memory_index) > MEM_AREA)
    {
        ret_value = REQ_TOO_LARGE;
    }

    if (memory_index == MEM_AREA)
    {
        ret_value = NO_MORE_MEMORY;
    }

    if (region_id != 1)
    {
        ret_value = BAD_REGION_ID;
    }
}
```

Chapter 3: Modifying a Custom OS for Real-Time Measurements

Instrumenting Code for Real-Time OS Tracking

```
    if (ret_value == NO_ERROR_RET)
    {
        *mem_area = &memory_block[memory_index];
        memory_index += size;
    }

    /* INSTRUMENTATION */
    HPOS_alloc_mem_Exit[0] = ret_value;
    HPOS_alloc_mem_Exit[1] = (long) *mem_area;

    return(ret_value);
}
```

Finally, add the locations to the data table:

```
long HPOS_alloc_mem_Entry[2];
long HPOS_alloc_mem_Exit[2];
```

To instrument an interface library for real-time OS code written in assembly language:

```
*****
*   void sc_qcreate (int qid, int qsize, int *errp);
*
*****
XDEF      _sc_qcreate
_sc_qcreate:
    LINK      A6,#0
    MOVE.L    D2,-(SP)

    MOVE.L    PARM0(A6),D1      ;qid
    MOVE.L    PARM1(A6),D2      ;qsize

    MOVEQ     #SC_QCREATE,D0
    MOVEM.L   D1-D2,HPOS_sc_qcreate_Entry      ;HP-RTOS-Level-1
    TRAP      #VRTXTRAP
    MOVE.L    D0,HPOS_sc_qcreate_Exit          ;HP-RTOS-Level-1

    MOVEA.L   PARM2(A6),A0
    MOVE.L    D0,(A0)

    MOVE.L    (SP)+,D2
    UNLK     A6
    RTS
```

Add locations to the data table:

```
HPOS_sc_qcreate_Entry    DS.L    2
HPOS_sc_qcreate_Exit    DS.L    1
```

To track task switches

- Modify OS code to write the task identifiers to the data table and/or task buckets when tasks switch.

For the emulation task switch tracking, OS code modification 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.

Because the software performance analyzer needs separate memory locations for the start and end of each interval it is measuring, each task must have its own unique start and end memory locations. The OS code must be modified to write to these unique locations depending on which tasks are switching. In the instrumented code, the task ID is used as an index to the data table or 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.

If you're creating an RTOS measurement tool for a vendor's real-time OS, you may not be able to instrument the OS code to track task switches. However, the vendor may allow for a task switch callout routine.

Writing a Task Switch Callout Routine

The *task switch callout* routine is a feature provided by the RTOS vendor. It allows a user to define a routine to be called every time a task switch occurs.

Typically, the OS code somehow makes task ID information available to the callout routine. This ID information can be used as an index to the task data buckets area to make the algorithm faster. If consistent and contiguous task IDs are not available, a simple linear search can be done on the task buckets.

Chapter 3: Modifying a Custom OS for Real-Time Measurements

Instrumenting Code for Real-Time OS Tracking

Examples

The demo task switch routine below shows how an OS task switch routine can be directly modified to incorporate instrumentation.

```
** _____ *
**
** PROCEDURE NAME:  knl_start_next_task()
**
** DESCRIPTION:  this procedure is called to switch tasks!  It does this
** by saving the current processor context in the task structure pointed
** to by curr_ptr.  Then it loads a new processor context from the task
** structure pointed to by next_ptr.  From this point it returns to the
** task just 'reloaded' as if it had just called this procedure.
** _____ *
**
** void knl_start_next_task(
** struct _task  *curr_ptr,
** struct _task  *next_ptr)
** {
**
**     XDEF  _knl_start_next_task
**     XREF  _HPOS_TASK_ENTRY, _HPOS_TASK_EXIT
**     XREF  _HPOS_TaskTable
**
_knl_start_next_task
    LINK  A6, #-0                ; setup our BP

S_curr_ptr  SET  8                ; parm2 location on stk
S_next_ptr  SET  12               ; parm1 location on stk

**
** save current-task context
**
** (NOTE: assumes is always called from current task!!!)
**
    MOVE.L  A0, (_saved_register)    ; clear A0 for use
    MOVEA.L (S_curr_ptr, A6), A0     ; get 'curr' task pointer
    MOVE.L  (_saved_register), (R_task_a0, A0) ; save proper A0
    MOVE.L  A7, (R_task_a7, A0)     ; save task's A7
    MOVE.L  A1, (R_task_a1, A0)     ; save task's A1
    LEA    (R_reg_store_hi, A0), A1 ; point SP to reg save area
    MOVEM.L D0-D7/A2-A6, -(A1)     ; save task's regs
    MOVE   SR, (R_task_sr, A0)     ; save proper SR

    ** EMUL INSTRUMENTATION **
    MOVE.L  (R_task_index, A0), _HPOS_TASK_EXIT ;** Write index to exit **
    ** SPA INSTRUMENTATION **
    ** Write to task specific exit point: HPOS_TaskTable[index] + 4
    MOVE.L  (R_task_index, A0), D1   ;** move index into D1
    LSL    #2, D1                    ;** multiply index(D1) by 4
    ADD.L  #4, D1                    ;** add 4 to index(D1)
    ADD.L  #_HPOS_TaskTable, D1     ;** Add start to index-offset
    MOVE.L  D1, A1                   ;** move addr to address reg
    MOVE   #1, (A1)                 ;** write to tasks's exit pt.

**
** restore 'next' context
**
    MOVEA.L (S_next_ptr, A6), A0     ; get 'next' task pointer

    ** EMUL INSTRUMENTATION **
    MOVE.L  (R_task_index, A0), _HPOS_TASK_ENTRY ;** Write index to entry
    ** SPA INSTRUMENTATION **
```


Chapter 3: Modifying a Custom OS for Real-Time Measurements Instrumenting Code for Real-Time OS Tracking

```

** Write to task specific entry point: HPOS_TaskTable[index] + 2
MOVE.L (R_task_index,A0),D1          ;** move index into D1
LSL    #2,D1                          ;** multiply index(D1) by 4
ADD.L  #2,D1                          ;** add 2 to index(D1)
ADD.L  #_HPOS_TaskTable,D1           ;** add start to index-offset
MOVE.L D1,A1                          ;** move addr to address reg
MOVE.W #2,(A1)                       ;** write to tasks's entry pt.

MOVE   (R_task_sr,A0),SR              ; restore proper SR
LEA   (R_reg_store_lo,A0),A1         ; point SP to reg save area
MOVEM.L (A1)+,D0-D7/A2-A6            ; restore task's regs
MOVEA.L (R_task_a1,A0),A1            ; restore task's A1
MOVEA.L (R_task_a7,A0),A7            ; restore task's A7
MOVEA.L (R_task_a0,A0),A0            ; restore proper A0

UNLK   A6                            ; un-setup BP
RTS                                         ; return to caller

```

When tasks switch, the ID of the task being exited is written to HPOS_TASK_EXIT in the data table and the ID of the task being entered is written to HPOS_TASK_ENTRY. The SPA instrumentation must calculate addresses of the entry and exit locations for the particular tasks before writing values to those locations.

Here is an example of an assembly language task switch callout routine. This routine will be called from within an OS's task switch area. The whole routine is instrumented code.

```

*****
*****                               SECTION III                               *****
***** SWITCH CALLOUT ROUTINE *****
***** LEVELS 1, 2, 4 & 5 *****
SECTION code
XDEF  _SWITCH_CALLOUT
XREF  HPOS_Task_Count,HPOS_Task_Offset
XREF  HPOS_Start_Task_List
XREF  HPOS_Queue_Count,HPOS_Queue_Names

TCB_IDNUM    EQU    $0D                ;Offset to task id# in TCB
TCB_SSP     EQU    $38                ;Offset to SSP in TCB
TCB_USP     EQU    $3C                ;Offset to USP in TCB
TCB_STACK   EQU    $40                ;Offset to original stk ptr in TCB

TASK_START  EQU    4                  ;Offset to task start address
TASK_END    EQU    6                  ;Offset to task end address
MSTACK_PTR  EQU    8                  ;Offset to master stack pointer
USTACK_PTR  EQU    12                 ;Offset to user stack pointer
BUCKET_SIZE EQU    16                 ;Size of task buckets in template file

UNKNOWN_TASK EQU    '????'

_SWITCH_CALLOUT
*-- Routine is "Level-4" Intrusion unless otherwise indicated -----
*-- Note: The whole routine except the two noted instructions may be -----
*--      commented out if the Performance Analyzer measurements are -----
*--      not wanted and stack information is not required. -----

```

Chapter 3: Modifying a Custom OS for Real-Time Measurements

Instrumenting Code for Real-Time OS Tracking

```

; A1 holds pointer to old TCB
; A2 holds pointer to new TCB

*
* SPA measurement writes
MOVE.W #1,HPOS_Start_Intrusion ;Start intrusion count ;HP-RTOS-Level-2
MOVE.W #2,HPOS_Stop_Ovrhd ;Stop OS overhead ;HP-RTOS-Level-2

MOVEM.L D0/A0/A3,-(SP) ;Save registers on stack
*
* Find which task has been preempted
*
CLR.L D0 ;Get tid for old task
MOVE.B TCB_IDNUM(A1),D0
CMP.L #HPOS_Task_Count,D0 ;Check if index to large
BGT HPOS_BAD_TID1
LSL #2,D0 ;x4 for long offset
MOVE.L D0,A3
MOVE.L HPOS_Task_Offset(A3),A0 ;Get task's bucket offset
BRA OLD_TASK_FOUND

HPOS_BAD_TID1 ; index out of user defined table's range
MOVE.L D0,HPOS_TASK_EXIT ;Note task exit HP-RTOS-Level-1
MOVE.L #$FFFFFFF,HPOS_TASK_BKT_UNDEF ;Warning: table too small
BRA NO_OLD_MATCH

* Write to the task's 'stop' location
OLD_TASK_FOUND
MOVE.W #1,TASK_END(A0) ;Write to SPA interval end

*-- Commands for trace display -----
;Output 'Exit' Task's stack data
MOVE.L MSTACK_PTR(A0),HPOS_T_EXIT_STACK ;Master stk base HP-RTOS-Level-4
MOVE.L TCB_SSP(A1),HPOS_T_STACK_VAR1 ;Mastr stk value HP-RTOS-Level-4
MOVE.L USTACK_PTR(A0),HPOS_T_STACK_VAR2 ;User stk base HP-RTOS-Level-4
MOVE.L TCB_USP(A1),HPOS_T_STACK_VAR3 ;User stk value HP-RTOS-Level-4

; *** ONLY THESE THREE INSTRUCTIONS ARE NEEDED FOR TASK EXIT TRACKING ***
CLR.L D0 ;HP-RTOS-Level-1
MOVE.B TCB_IDNUM(A1),D0 ;Get tid for old task ;HP-RTOS-Level-1
MOVE.L D0,HPOS_TASK_EXIT ;Note task exit ;HP-RTOS-Level-1
*-- End commands for trace display -----

NO_OLD_MATCH
*
* Find which task will start running
*
CLR.L D0 ;Get tid for old task
MOVE.B TCB_IDNUM(A2),D0
CMP.L #HPOS_Task_Count,D0 ;Check if index to large
BGT HPOS_BAD_TID2
LSL #2,D0 ;x4 for long offset
MOVE.L D0,A3
MOVE.L HPOS_Task_Offset(A3),A0 ;Get task's bucket offset

* Write to the task's 'start' location
NEW_TASK_FOUND
MOVE.W #2,TASK_START(A0) ;Write to SPA interval start

*-- Commands for trace display -----
; *** ONLY THESE THREE INSTRUCTIONS ARE NEEDED FOR TASK ENTRY TRACKING ***

```

Chapter 3: Modifying a Custom OS for Real-Time Measurements Instrumenting Code for Real-Time OS Tracking

```

CLR.L    D0                                ;HP-RTOS-Level-1
MOVE.B   TCB_IDNUM(A2),D0                  ;Get tid for new task ;HP-RTOS-Level-1
MOVE.L   D0,HPOS_TASK_ENTRY                ;Note task entry   ;HP-RTOS-Level-1

;Output 'Exit' Task's stack data
MOVE.L   MSTACK_PTR(A0),HPOS_T_ENTRY_STACK ;Mstr stk base   HP-RTOS-Level-4
MOVE.L   TCB_SSP(A2),HPOS_T_STACK_VAR1    ;Mstr stk value  HP-RTOS-Level-4
MOVE.L   USTACK_PTR(A0),HPOS_T_STACK_VAR2 ;Usr stk base   HP-RTOS-Level-4
MOVE.L   TCB_USP(A2),HPOS_T_STACK_VAR3    ;Usr stk value  HP-RTOS-Level-4
*-- End commands for trace display -----
BRA      END_SWITCH

HPOS_BAD_TID2    ; index out of user defined table's range
MOVE.L   D0,HPOS_TASK_ENTRY                ;Note task entry ;HP-RTOS-Level-1
MOVE.L   #$FFFFFFF,HPOS_TASK_BKT_UNDEF    ;Warning: table too small

END_SWITCH
MOVEM.L (SP)+,D0/A0/A3                    ;Restore registers from stack
MOVE.W   #1,HPOS_Start_Ovrhd              ;Start OS overhead timer ;HP-RTOS-Level-2
MOVE.W   #2,HPOS_Stop_Intrusion          ;Stop intrusion count ;HP-RTOS-Level-2
RTS

*****                               END SECTION III *****
*****                               SWITCH CALLOUT ROUTINE *****
*****                               LEVELS 1, 2, 4 & 5 *****
*****                               *****

```

To track clock ticks

- Modify OS code to write to a "clock tick" location in the data table.

A clock tick is a unit of time used by the OS for the purpose of scheduling tasks or processes. The length of time is determined by a periodic interrupt which is handled by a special interrupt service routine that lets the OS know a clock tick should occur. The OS may switch tasks that have specified "time slices" to "blocked" after a certain number of clock ticks.

There are two methods for tracking clock ticks.

- If the real-time OS has a "time slice" service call that is used by applications, the service call can be instrumented to make clock tick information available.
- Some applications may choose not to use the "time slice" service call and may have an associated interrupt service routine (ISR) written directly in assembly language code for speed reasons. In this case, the interrupt service routine should be instrumented with a simple

Chapter 3: Modifying a Custom OS for Real-Time Measurements

Instrumenting Code for Real-Time OS Tracking

MOVE.W D_x,HPOS_CLOCK_TICK instruction before the trap to the OS. (Make sure the instruction 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.

Examples

Here is an example of instrumentation in an interface library for tracking clock ticks:

```
*****
*
* ANNOUNCE A CLOCK TICK TO pSOS+
*
* rc = tm_tick();
*
* This routine may not be used as is but may be incorporated
* into a user's own clock-interrupt service routine.
*****
      XDEF      _tm_tick
_tm_tick:
      LINK      A6,#0

      MOVEQ     #TM_TICK,D0          ;LOAD FUNCTION CODE
      MOVE.L    D0,HPOS_CLOCK_TICK  ;HP-RTOS-Level-1

      MOVE.L    D0,HPOS_TM_TICK_Entry ;HP-RTOS-Level-1
      MOVE.W    #1,Start_OS_Ovrhd    ;HP-RTOS-Level-2
      TRAP      #SVCTRAP
      MOVE.W    #2,Stop_OS_Ovrhd     ;HP-RTOS-Level-2
      ;This line may be left out since it always returns
      MOVE.L    #TM_TICK,HPOS_TM_TICK_Exit ;HP-RTOS-Level-1
      MOVE.W    D0,HPOS_CHECK_ERRORS  ;HP-RTOS-Level-2

      UNLK     A6
      RTS

      NOP      ; Spacer to prevent prefetches
      NOP      ; from confusing trace
```

To track OS overhead

- Modify OS code to write to a "start overhead" location when service calls enter and a "stop overhead" location when service calls exit.

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 write instructions to the service call routines. The first write instruction, executed just after the service call entry instrumentation, writes to a location that represents the start of the OS interval. The second write instruction, executed just before the service call exit instrumentation, 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.

Examples

The 'allocate_memory' service call in the demo OS can be instrumented for OS overhead tracking as shown below.

First, declare the data table locations as externals:

```
extern short int HPOS_Start_Ovrhd;  
extern short int HPOS_Stop_Ovrhd;
```

Chapter 3: Modifying a Custom OS for Real-Time Measurements

Instrumenting Code for Real-Time OS Tracking

Then, instrument the service call so that parameters and return values are written on entry and exit:

```
/* _____ *
**
** PROCEDURE NAME:  allocate_memory()
**
** DESCRIPTION:
**
** _____ *
*/
long allocate_memory(int region_id, int size, char **mem_area)
{
    long ret_value = NO_ERROR_RET;

    /* INSTRUMENTATION */
    HPOS_alloc_mem_Entry[0] = region_id;
    HPOS_alloc_mem_Entry[1] = size;

    HPOS_Start_Ovrhd = 1;    /* INSTRUMENTATION for SPA */

    /* Initialize the return pointer */
    *mem_area = NULL;

    if ((size + memory_index) > MEM_AREA)
    {
        ret_value = REQ_TOO_LARGE;
    }

    if (memory_index == MEM_AREA)
    {
        ret_value = NO_MORE_MEMORY;
    }

    if (region_id != 1)
    {
        ret_value = BAD_REGION_ID;
    }

    if (ret_value == NO_ERROR_RET)
    {
        *mem_area = &memory_block[memory_index];
        memory_index += size;
    }

    HPOS_Stop_Ovrhd = 2;    /* INSTRUMENTATION for SPA */

    /* INSTRUMENTATION */
    HPOS_alloc_mem_Exit[0] = ret_value;
    HPOS_alloc_mem_Exit[1] = (long) *mem_area;

    return(ret_value);
}
```

Finally, add the locations to the data table:

```
short int HPOS_Start_Ovrhd;    /* Start of OS interval for SPA */
short int HPOS_Stop_Ovrhd;    /* End of OS interval for SPA */
```

Chapter 3: Modifying a Custom OS for Real-Time Measurements Instrumenting Code for Real-Time OS Tracking

A service call in an interface library for an assembly language based real-time OS can be instrumented for OS overhead tracking as shown below.

```
*****
*
* POST A MESSAGE TO A QUEUE
*
* rc = q_send(qid, msg);
*
* (msg is INT32 msg[4]);
*
*****
XDEF      _q_send
_q_send:

LINK      A6,#0
MOVEM.L  D2-D5,-(SP)          ;SAVE REGISTERS
MOVE.L   8(A6),D1             ;D1 EQU QID
MOVEA.L  12(A6),A0           ;A0 EQU ADDRESS OF MSG
MOVEM.L  (A0),D2-D5          ;GET THE MESSAGE

MOVEQ    #Q_SEND,D0

MOVEM.L  D1-D5,HPOS_Q_SEND_Entry ;HP-RTOS-Level-1
MOVE.W   #1,Start_OS_Ovrhd      ;HP-RTOS-Level-2
TRAP     #SVCTRAP
MOVE.W   #2,Stop_OS_Ovrhd       ;HP-RTOS-Level-2
MOVE.L   D0,HPOS_Q_SEND_Exit    ;HP-RTOS-Level-1

MOVEM.L  (SP)+,D2-D5          ; RESTORE REGISTERS
UNLK     A6
RTS

NOP      ; Spacer to prevent prefetches
NOP      ; from confusing trace
```



To track stack and memory

- Add data bucket locations for each task to save the task's stack values.
- Modify OS code to write stack pointer information when tasks switch.

Stack information such as pointers and bytes used can be tracked dynamically as an application runs.

The necessary data is mostly written out when tasks switch (whether it be instrumented code or a 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 instrumentation or 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 instrumentation or callout routine uses the previously saved data to keep track of stacks. In this code, 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.

Examples

See the example task switch callout routine in "To track task switches".

Organizing the Data Table

The previous section on "Instrumenting Code for Real-Time OS Tracking" showed you how to insert locations in the data table for service call and task switch tracking. This section shows you how to organize those locations in the data table.

Generally, the data table should be organized as shown below.

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

When setting up the data table, it's important to:

- Group locations for easy tracking.
- Create user-defined areas.

Group Locations for Easy Tracking

It's important to group data table locations for similar types of service calls to make selective tracking easy. For example, in order to provide a measurement that tracks only message service calls, it's easier to set up a trace command that captures a range of data table locations than one that captures locations scattered about the data table.

Create User-Defined Areas

If you're creating a general purpose RTOS measurement tool that will be used by a number of different people for different applications, it's important to set up user-defined locations in the data table. These locations are intended to allow a user 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 an application's interrupt service routines. By doing this, you could get a histogram in SPA of the time spent in any interrupt service routine.

The inverse assembler should be set up so that 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 (because, for example, "Loop" is easier to read than "0x4c6f6f70").

Note

If you are capturing a range in the data table that includes any of the user-defined locations, all of these locations must be written to with longword writes in order for the demo inverse assembly to work correctly. Your own inverse assembler may be written to understand any combination of memory write sizes.

Organizing the Task Data Buckets

Task data buckets are locations defined for each task in which information is saved by task start or task switch instrumentation (for example, callout routines). It's also common to place unique task entry and exit locations in the task data buckets; these locations are used by the Software Performance Analyzer to make profile measurements on task intervals.

The task buckets need not be organized in any order. What is important is the how easily a user can add or remove specific task buckets. You want to be able to quickly and simply add a new task to the bucket array if a new task is needed in an application. For assembly code, the best way is to create a macro that defines the buckets; then, use a simple list of macros calls to create the task buckets. For C code, the best way is to define a structure, then declare an occurrence of the structure for each task. For example:

```
struct t_bucket {
    long t_entry;
    long t_exit;
    long M_stack_ptr;
    long U_stack_ptr;
};

struct t_bucket task1_bucket;
struct t_bucket task2_bucket;
struct t_bucket task3_bucket;
struct t_bucket task4_bucket;
```



Chapter 3: Modifying a Custom OS for Real-Time Measurements

Organizing the Task Data Buckets

An example of assembly code follows:

```

XDEF  HPOS_Task_Offset
XDEF  HPOS_Task_Count
XDEF  HPOS_Start_Task_List,HPOS_End_Task_List
*
* Local macro to use when defining task entries
*
TASK_ENTRY    MACRO name
                XDEF  HPOS_Task_&&name&&
                XDEF  HPOS_Tenter_&&name&&
                XDEF  HPOS_Texit_&&name&&
                XDEF  HPOS_MStk_Ptr_&&name&&
                XDEF  HPOS_UStk_Ptr_&&name&&
HPOS_Task_&&name&&    DC.L  name           ;task name
HPOS_Tenter_&&name&&    DS.W  1           ;SPA interval start address
HPOS_Texit_&&name&&    DS.W  1           ;SPA interval end address
HPOS_MStk_Ptr_&&name&&    DC.L  1           ;Master stack ptr base value
HPOS_UStk_Ptr_&&name&&    DC.L  1           ;User stack ptr base value
                ENDM
*
* Local macro to define task count
*
HPOS_TSK_UNKNOWN    DC.L  '----'

USER_TASK_COUNT    MACRO  count
HPOS_Task_Offset    DCB.L  count+15,UNKNOWN_ID_BUCKET ; holds addresses of buckets
HPOS_Task_Count     EQU   count+15                ; symbol for count
                ENDM

*----- BEGIN TASK MODIFICATIONS -----
        USER_TASK_COUNT    4                ;number of tasks

HPOS_Start_Task_List
        TASK_ENTRY    0001
        TASK_ENTRY    0002
        TASK_ENTRY    0003
        TASK_ENTRY    0100
*----- END TASK MODIFICATIONS -----

```

A user only has to edit the "TASK_ENTRY" list and the "USER_TASK_COUNT" to add or delete a task from the bucket array.

The task data buckets are also referred to as extra memory locations.

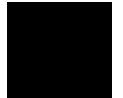
Re-Building the OS and the Application

After you have modified the custom OS for real-time measurements, you will have to re-build it.





4



Writing the RTOS Inverse Assembler Code

Writing the RTOS Inverse Assembler Code

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).

The RTOS inverse assembler lets you make analyzer data very easy to interpret. When written to do so, the inverse assembler can display all OS service calls just as they appear in the source code or in the OS vendor's manual. The inverse assembler can decode input parameters and return values into English language equivalents when there are a finite number of possibilities.

After writing the RTOS inverse assembler code, you must build the inverse assembler and install it in the proper Emulator/Analyzer interface directory.

When an HP 64700 Emulator/Analyzer interface starts up, it looks for a relocatable RTOS inverse assembler file in the appropriate directory. If this relocatable file is found, it is loaded by the Emulator/Analyzer interface.

When the **display trace real_time_os** command is entered in the Emulator/Analyzer interface, the interface calls the RTOS inverse assembler for each captured state in the trace. The inverse assembler decodes the captured state, and any other related trace states, and outputs information that is displayed by the Emulator/Analyzer interface.

Because the RTOS inverse assembler is dynamically loaded into the Emulator/Analyzer interface (when new interface windows are started by choosing **File→Emul700→Custom RTOS Emul** or by running the "rtos_emul" startup script), it can start out in a simple form and be enhanced over a period of time.

When creating your own inverse assembler, it will probably be easiest to copy and modify the demo inverse assembler files from the \$HP64000/rtos/B3082A/interpreter directory.

Before you write RTOS inverse assembler code, you should generally know how to write IAL code. Then, you need to understand what an inverse assembler for an RTOS trace display must do. Finally, you need to know how to build and install the RTOS inverse assembler. This chapter describes these topics in the following sections:

- Writing IAL Code
- What the RTOS Inverse Assembler Must Do
- Building and Installing the RTOS Inverse Assembler

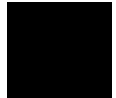
The Demo RTOS Inverse Assembler

The inverse assembler for the demo RTOS measurement tool was designed to be used with both 16- and 32-bit processors.

Because the data table is made up of long integer (32-bit) locations, 32-bit processors take 1 cycle to write a value to the data table whereas 16-bit processors take 2 cycles.

For a single value written to the data table, the inverse assembler must decode 1 analyzer state for 32-bit processors and 2 analyzer states for 16-bit processors. Consequently, the inverse assembler has special routines for 16- and 32-bit processors. These special routines are found in the "ial16.S" and "ial32.S" files.

Your own inverse assembler may be written to understand any combination of memory write sizes.



Writing IAL Code

Inverse assembler source code is written in Inverse Assembly Language (IAL). The inverse assembler source code is written in the same manner as an assembly program and then assembled by the IAL assembler. This special assembler understands the instruction set of the IAL and generates a relocatable file with the information needed by the Emulator/Analyzer interface.

It is not necessary to link the inverse assembler relocatable file into an absolute file. The relocatable inverse assembler file is read and loaded by the Emulator/Analyzer interface at run time.

The entire inverse assembler must be contained in one file because only one relocatable file will be loaded by the Emulator/Analyzer interface.

A sample of a simple, but complete inverse assembler follows.

```
1  " IAL"  
2  
3      OUTPUT "Inverse assembler not present"  
4      RETURN
```

This inverse assembler will display the message "Inverse assembler not present" and then return to the Emulator/Analyzer interface each time it is called.

IAL Instructions

There are two types of instructions in the Inverse Assembly Language:

- Executable instructions, which cause the inverse assembler to take some action. The first executable instruction is the entry point to the inverse assembler.
- Pseudo instructions, which define storage, text, and inverse assembler attributes. Pseudo instructions can appear anywhere in the inverse assembler source code.

The executable instructions and pseudo instructions are listed in the "Inverse Assembler Language (IAL) Instructions" chapter.

IAL Operands

The basic structure of the IAL pseudo-processor can be viewed as a single accumulator machine with special functions provided for the task of inverse assembly. Functions have been included to enable arithmetical and logical operations to be performed on memory variables as well as conditional execution based on the results of these operations. All of these operations are performed through the "ACCUMULATOR".

The accumulator is a register provided by the IAL interpreter that can be loaded from variables, operated on by arithmetic and logical operations, tested by IF or CASE instructions, and the results stored back into variables.

These are two types of variables:

- Local variables defined by the inverse assembler source code using the "VARIABLE" pseudo.
- Communication variables provided by the IAL interpreter.

All variables as well as the ACCUMULATOR are 32-bit unsigned integers.

Communication Variables

Trace states and other related information are passed to the inverse assembler through communication variables. The RTOS inverse assembler uses the following communication variables.

INPUT_ADDRESS Contains the address value of the particular trace state. You can load address values of additional trace states into this variable using the INPUT instruction.

INPUT_DATA Contains the data value of the particular trace state. You can load data values of additional trace states into this variable using the INPUT instruction.

INPUT_TAG Contains software tag information about whether the particular trace state has already been used. When you use the INPUT instruction to load additional trace states into the communication variables, this variable is updated to contain the tag information that corresponds to the additional trace state.

INITIAL_OPTIONS Contains the address of the symbol `HP_RTOS_TRACK_START`; this is the base address of the data table.

INITIAL_FLAGS When bit 1 is set, the base value of the data table has been passed in to the inverse assembler through the `INITIAL_OPTIONS` variable. Other bits in the `INITIAL_FLAGS` communication variable are not used.

These communication variables are all inputs. There are no variables used by the RTOS inverse assembler to return information to the Emulator/Analyzer interface.

Local Variables

Variables defined by the inverse assembler source code are local variables and can be used for holding temporary values or setting flags for internal use. These variables are never accessed or changed by the interpreter code.

Unsigned Comparisons

All compares done with the "IF" instruction are unsigned. Two's complement arithmetic is performed on all variables, which means values can be added and subtracted to get a correct positive or negative result. The highest order bit (31) of the variable can be tested to see if the value is positive or negative. The highest order bit will be 0 for positive and 1 for negative results. Since variables are unsigned, they cannot be tested for a value less than zero, since all variables have the unsigned value of greater than or equal to zero.

Bit Ranges

Another type of operand allows bit ranges to be specified on the conditional instructions (IF and CASE_OF). These only work on the ACCUMULATOR and specify a bit range to be used for the condition. It is identified by the syntax "MSB,LSB", which signifies the most significant bit to the least significant bit.

In the following example, bits 5, 4, and 3 of the ACCUMULATOR will be used in the test. They will be compared to the binary value 101 and if the

condition is satisfied, then control will be transferred to LABEL, which must be defined in the inverse assembler source code.

```
IF 5,3 = 101B THEN GOTO LABEL
```

Immediate Values

Some of the IAL instructions have "immediate values" as operands. This means that a numeric constant is expected. Constants have a range from 0 to 0FFFFFFFFH (32 bits) and can be assigned a symbolic name by using the "CONST" pseudo.

IAL Program Control

Execution will start with the first executable statement in the inverse assembler source code and continue in a linear order until a "RETURN" at the appropriate nesting level or an "ABORT" instruction is encountered. These can be used anywhere in the code and as often as necessary. The linear program flow can be altered by using "GOTO" or "CALL" instructions.

The GOTO Instruction

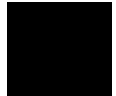
The GOTO instruction expects a label as an operand. This label must be defined somewhere in the inverse assembler source code. Execution will transfer to the label and continue from there.

The CALL Instruction

The CALL instruction performs in the same way except the address of the next instruction is saved so that when a RETURN instruction is executed in the called subroutine, program control will be transferred to the instruction following the CALL.

The RETURN Instruction

The RETURN instruction has two functions. If the inverse assembler source code is currently in a subroutine, RETURN will transfer control to the statement following the CALL. If program execution is not in a subroutine (subroutine level zero), then control will be passed back to the Emulator/Analyzer interface.



The ABORT Instruction

The ABORT instruction will pass control back to the Emulator/Analyzer interface regardless of the subroutine level. It is intended to be an error escape from the inverse assembler.

Conditional Instructions (IF and CASE_OF)

The IF and CASE_OF instructions, when used in conjunction with the CALL and GOTO instructions, let you alter the program flow conditionally.

Writing RTOS Inverse Assembler Code

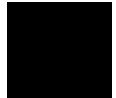
Once you know how to write IAL code, you must have an idea of what, specifically, an RTOS inverse assembler must do.

First the RTOS inverse assembler must calculate the address values of the various locations in the data table so that it will be able to recognize (in the trace data) when a write occurred to a particular location.

Then, once writes to particular locations are recognized, the RTOS inverse assembler must be able to decode the data values that are written to those locations and output formatted information to the trace display.

This section shows you how to:

- Define strings, variables, and number formats.
- Assign data table locations to variables.
- Decode address information from the trace.
- Decode data information and output to the trace display.



To define strings, variables, and number formats

- 1 Use the ASCII instruction to define strings.

Generally, you want to define strings for the information that will appear in the RTOS trace display, for example, service call names and the strings used to denote calls and returns.

- 2 Use the VARIABLE instruction to define variables.

You want to define variables for the address values of the various data table locations: starting and ending addresses of the data table, task entry and exit locations, service call entry and exit locations, and user-defined locations. The contents of these variables are compared with address values in the trace to identify service call or task entries or exits.

You may also want to define variables for clock ticks and for other information needed by the inverse assembler code.

- 3 Use the FORMAT instruction to define number formats.

You want to define formats for the numbers that will appear in the RTOS trace display. You can specify these formats with the OUTPUT instruction.

Examples

The ASCII instruction associates a name with a string. Inverse assembler language instructions that follow can use the name instead of a string operand. To define strings:

```
*-----  
* SERVICE CALL STRING DEFINITIONS  
*-----  
TASK_CREATE_STR      ASCII    "task_create"  
SEND_MESSAGE_STR     ASCII    "send_message"  
GET_MESSAGE_STR      ASCII    "get_message"  
  
ALLOC_MESSAGE_STR    ASCII    "alloc_message"  
FREE_MESSAGE_STR     ASCII    "free_message"  
  
ALLOC_MEM_STR        ASCII    "alloc_memory"  
  
CALL_STRING          ASCII    "-> "  
RETURN_STRING        ASCII    "<- "
```


Chapter 4: Writing the RTOS Inverse Assembler Code Writing RTOS Inverse Assembler Code

The VARIABLE instruction defines and initializes an internal 32-bit storage location that can be used by the inverse assembler. To define variables that will hold address values of data table locations:

```

*-----
* SERVICE CALL & GENERAL VARIABLES
*-----
HPOS_TRACK_START    VARIABLE  0
HPOS_TRACK_END      VARIABLE  0

TASK_ENTRY          VARIABLE  0
TASK_ENTRY_PLUS2    VARIABLE  0
TASK_EXIT           VARIABLE  0

TASK_CREATE_ENTRY   VARIABLE  0
TASK_CREATE_EXIT    VARIABLE  0
SEND_MESSAGE_ENTRY  VARIABLE  0
SEND_MESSAGE_EXIT   VARIABLE  0
GET_MESSAGE_ENTRY   VARIABLE  0
GET_MESSAGE_EXIT    VARIABLE  0

ALLOC_MSG_ENTRY     VARIABLE  0
ALLOC_MSG_EXIT      VARIABLE  0
FREE_MESSAGE_ENTRY  VARIABLE  0
FREE_MESSAGE_EXIT   VARIABLE  0

ALLOC_MEM_ENTRY     VARIABLE  0
ALLOC_MEM_EXIT      VARIABLE  0

USER_NUMERIC1       VARIABLE  0
USER_NUMERIC2       VARIABLE  0
USER_NUMERIC3       VARIABLE  0
USER_NUMERIC4       VARIABLE  0
USER_NUMERIC5       VARIABLE  0
USER_NUMERIC6       VARIABLE  0
USER_ASCII1         VARIABLE  0
USER_ASCII2         VARIABLE  0
USER_ASCII3         VARIABLE  0
USER_ASCII4         VARIABLE  0
USER_ASCII5         VARIABLE  0
USER_ASCII6         VARIABLE  0

END_OF_DATA_TABLE   VARIABLE  0

CLOCK_TICKS         VARIABLE  0

STRING_ARRAY        VARIABLE  0

```

To define other variables used by the inverse assembler:

```

*-----
DATE                VARIABLE  0
TIME                VARIABLE  0
TICKS               VARIABLE  0
REL_POSITION        VARIABLE  0
SAVE_REL_POS        VARIABLE  0
UPPER_BITS          VARIABLE  0
STACK_LIMIT         VARIABLE  0

```

Chapter 4: Writing the RTOS Inverse Assembler Code

Writing RTOS Inverse Assembler Code

The REL_POSITION variable is used to hold the current relative state at which the data resides. This variable is used as a parameter to many routines. Its value for 16-bit processors must be interpreted as twice (2x) the value as for 32-bit processors in identical situations because it takes twice as many states to store trace data when it is stored 16 bits at a time as compared to storing 32 bits at a time.

The FORMAT instruction defines how the value stored in the accumulator should be displayed when used in conjunction with the OUTPUT command which is used to display information in the Emulator/Analyzer interface. To define number formats:

```
*----- Formats -----  
HEX_FMT          FORMAT          32,HEX,8  
HEX_FMT_LEFT     FORMAT          32,HEX,LEFT_JUSTIFIED  
HEX_FMT16        FORMAT          16,HEX,4,DISPLAY_BASE  
  
DEC_FMT          FORMAT          32,DEC,LEFT_JUSTIFIED  
DEC_FMT_TIME     FORMAT          8,DEC,2  
DEC_FMT_YEAR     FORMAT          16,DEC,4  
  
BIN_FMT_LEFT     FORMAT          32,BIN,LEFT_JUSTIFIED  
BIN_FMT_4        FORMAT          4,BIN,4
```

To assign data table locations to variables

- Use the data table base address to calculate the addresses of the other data table locations.

When bit 1 of the INITIAL_FLAGS communications variable is set, the base address of the data table, as defined by the HP_RTOS_TRACK_START symbol, has been passed to the inverse assembler through the INITIAL_OPTIONS communications variable.

Examples

Below is the demo IAL code that assigns data table locations to variables.

First, to check if the base value of the data table, HP_RTOS_TRACK_START, has been passed in:

```
*-----  
*           Start actual Inverse Assembly code  
*-----  
  
      LOAD   INITIAL_FLAGS  
      IF     1,1 = 0 THEN GOTO HAVE_ADDRESSES
```

To check for new base value of the data table, the base address just passed in is compared to the current value of the HPOS_TRACK_START variable. If they're equal, current values of other variables are correct. If they're not equal, new values must be calculated and assigned to variables.

```
      LOAD   INITIAL_OPTIONS  
      SUBTRACT HPOS_TRACK_START  
      IF     31,1 = 0 THEN GOTO HAVE_ADDRESSES
```

To calculate address values of locations in the data table, the base address is loaded into the accumulator. Because the first location is a "long" type, the next location is 4 bytes ahead; therefore, 4 is added to the accumulator and the result, which equals the address of the next location in the data table, is saved in the TASK_ENTRY variable. This process is repeated until values have been assigned to all the variables.

```
      LOAD   INITIAL_OPTIONS  
      STORE  HPOS_TRACK_START  
  
      ADD    4  
      STORE  TASK_ENTRY  
      ADD    2  
      STORE  TASK_ENTRY_PLUS2  
      ADD    2  
      STORE  TASK_EXIT  
      ADD    4  
  
      STORE  TASK_CREATE_ENTRY  
      ADD    28  
      STORE  TASK_CREATE_EXIT  
      ADD    4  
  
      STORE  SEND_MESSAGE_ENTRY  
      ADD    12  
      STORE  SEND_MESSAGE_EXIT  
      ADD    4  
      STORE  GET_MESSAGE_ENTRY  
      ADD    8  
      STORE  GET_MESSAGE_EXIT  
      ADD    8
```

Chapter 4: Writing the RTOS Inverse Assembler Code

Writing RTOS Inverse Assembler Code

```
STORE  ALLOC_MSG_ENTRY
ADD     4
STORE  ALLOC_MSG_EXIT
ADD     8
STORE  FREE_MESSAGE_ENTRY
ADD     4
STORE  FREE_MESSAGE_EXIT
ADD     4

STORE  ALLOC_MEM_ENTRY
ADD     8
STORE  ALLOC_MEM_EXIT
ADD     8

STORE  STRING_ARRAY
ADD     12

STORE  USER_NUMERIC1
ADD     4
STORE  USER_NUMERIC2
ADD     4
STORE  USER_NUMERIC3
ADD     4
STORE  USER_NUMERIC4
ADD     4
STORE  USER_NUMERIC5
ADD     4
STORE  USER_NUMERIC6
ADD     4
STORE  USER_ASCII1
ADD     4
STORE  USER_ASCII2
ADD     4
STORE  USER_ASCII3
ADD     4
STORE  USER_ASCII4
ADD     4
STORE  USER_ASCII5
ADD     4
STORE  USER_ASCII6

ADD     4
STORE  CLOCK_TICKS

ADD     4
STORE  END_OF_DATA_TABLE

; *****
```

To decode address information from the trace

- 1 Check if a state has been used. If so, mark the state as suppressed and return.
- 2 If a state has not been used, compare the address value to known data table locations. If the address is in the data table, call a routine to output information about the associated data value (and perhaps the data values of related states). If the state is not in the data table, mark it as a non-RTOS state and output address and data values.

Examples

The demo IAL code that decodes address information from the trace is shown below.

When the inverse assembler reaches the HAVE_ADDRESSES label, all of the data table locations have been assigned to variables. The first thing to do is check the INPUT_TAG communication variable to see if the state has already been used.

```
HAVE_ADDRESSES  
  
    LOAD INPUT_TAG  
    IF 15,0 = 1 THEN GOTO ALREADY_USED
```

If the state has not been used, check if the address is within data table range and go to the OUTSIDE_RANGE label if it isn't.

```
    LOAD INPUT_ADDRESS  
    IF 31,0 >= END_OF_DATA_TABLE THEN GOTO OUTSIDE_RANGE  
    IF 31,0 < HPOS_TRACK_START THEN GOTO OUTSIDE_RANGE
```

The address is within the range of the data table, so compare it to the variables that contain addresses of the data table locations. If there is a match, go to the appropriate routine that decodes data from this state and the following states associated with the OS event.

```
    IF 31,0 = TASK_EXIT THEN GOTO TASK_EXIT_CALL  
    IF 31,0 = TASK_ENTRY THEN GOTO TASK_ENTRY_CALL  
    IF 31,0 = TASK_ENTRY_PLUS2 THEN GOTO TASK_ENTRY_WORD  
  
    IF 31,0 = TASK_CREATE_ENTRY THEN GOTO TASK_CREATE_FNC  
    IF 31,0 = TASK_CREATE_EXIT THEN GOTO TASK_CREATE_RTN  
  
    IF 31,0 = SEND_MESSAGE_ENTRY THEN GOTO SEND_MESSAGE_FNC  
    IF 31,0 = SEND_MESSAGE_EXIT THEN GOTO SEND_MESSAGE_RTN  
    IF 31,0 = GET_MESSAGE_ENTRY THEN GOTO GET_MESSAGE_FNC
```

Chapter 4: Writing the RTOS Inverse Assembler Code

Writing RTOS Inverse Assembler Code

```
IF 31,0 = GET_MESSAGE_EXIT THEN GOTO GET_MESSAGE_RTN

IF 31,0 = ALLOC_MSG_ENTRY THEN GOTO ALLOC_MESSAGE_FNC
IF 31,0 = ALLOC_MSG_EXIT THEN GOTO ALLOC_MESSAGE_RTN
IF 31,0 = FREE_MESSAGE_ENTRY THEN GOTO FREE_MESSAGE_FNC
IF 31,0 = FREE_MESSAGE_EXIT THEN GOTO FREE_MESSAGE_RTN

IF 31,0 = ALLOC_MEM_ENTRY THEN GOTO ALLOC_MEM_FNC
IF 31,0 = ALLOC_MEM_EXIT THEN GOTO ALLOC_MEM_RTN

IF 31,0 = USER_NUMERIC1 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_NUMERIC2 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_NUMERIC3 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_NUMERIC4 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_NUMERIC5 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_NUMERIC6 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_ASCII1 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_ASCII2 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_ASCII3 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_ASCII4 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_ASCII5 THEN GOTO USER_DEFINED_FNC
IF 31,0 = USER_ASCII6 THEN GOTO USER_DEFINED_FNC
```

If the inverse assembler reaches this point, the input address is within the RTOS data range but does not match a known starting offset. In this case, the inverse assembler ignores the address and assumes it was or will be covered by a known offset.

```
GOTO ALREADY_USED
```

If the inverse assembler reaches the `OUTSIDE_RANGE` label, the address is outside of the data table range. In this case, all data given with the state is displayed as a non-RTOS state. The `IF_NOT_MAPPED` instruction outputs a symbol (and offset) if possible. If there is no symbol defined for that address, the absolute address is output in a left-justified hexadecimal format.

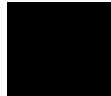
```
OUTSIDE_RANGE
OUTPUT " NON-RTOS: "
LOAD INPUT_ADDRESS
OUTPUT " addr="
IF_NOT_MAPPED THEN OUTPUT ACCUMULATOR,HEX_FMT_LEFT
LOAD INPUT_DATA
OUTPUT " data="
OUTPUT ACCUMULATOR,HEX_FMT
GOTO END_RTOS_IAL
```

If the inverse assembler reaches the `ALREADY_USED` label, the state has already been used in the trace display. In this case, the `MARK_STATE SUPPRESSED` instruction is used to tell the analyzer not to display the current analyzer state.

```
ALREADY_USED  
        MARK_STATE    SUPPRESSED
```

When the inverse assembler reaches the `END_RTOS_IAL` label, the inverse assembler exits.

```
END_RTOS_IAL  
        ;CALL DEBUG_PRINT  
        RETURN  
*----- Final Return -----  
*-----
```



To decode data information and output to the trace display

- Write an IAL routine to decode data values and output the information to be displayed.

These IAL routines are called when the address of a trace state matches a location in the data table.

You can use the `OUTPUT "string"` instruction to output defined strings, for example, the call (`->`) or return (`<-`) strings or service call names.

You can also use the `OUTPUT "string"` instruction to label parameters and return values in the trace display. You can use the same mnemonics that are used in the source code or described in the OS vendor's manual.

You can `CALL` the provided IAL routines to output data in different formats. The provided IAL routines do not contain all of the permutations of state and output format and therefore should not be considered a library; however, it should be relatively easy to create additional routines you might need. The provided IAL routines are listed in the following table.

Chapter 4: Writing the RTOS Inverse Assembler Code
Writing RTOS Inverse Assembler Code

Provided Inverse Assembler Routines	
Routine	Description
PRINT_BINARY	Output a number in the accumulator as binary (for example, 00011010001010110011110001001101).
DISPLAY_RET_CODE	Displays the return code from a service routine and a descriptive string.
WRITE_OUT_STRING	Outputs a null terminated string starting at HPOS_String_Array (for example, "abcdefghijkl").
PRINT_NAME_IN_ACCUM	Prints a 32-bit name stored in the accumulator (for example, 'abcd', # 0, or # 1a2b3c4d).
PRINT_NAME	Prints a 4 ASCII character name stored in the accumulator (for example, 'abcd').
PRINT_ASCII_CHAR	Prints data in accumulator bits 7-0 as ASCII character (for example, a).
DEBUG_PRINT	Outputs address and data for the current trace state.
OUT_RELO_HEX	Output current long word (32 bits) of trace data in hexadecimal format (for example, 1a2b3c4d). ¹
OUT_RELO_HEXLEFT	Output current long word (32 bits) of trace data in left-justified hexadecimal format (leading zeros are not printed). ¹
OUT_RELO_BINLEFT	Output current long word (32 bits) of trace data in left-justified binary format (leading zeros are not printed). ¹
OUT_REL1_HEX	Output current+ 1 long word (32 bits) of trace data in hexadecimal format. ¹
OUT_REL1_HEX16	Output current+ 1 long word (32 bits) of trace data in 16-bit hexadecimal format (for example, 1234H). ¹
OUT_REL1_HEXLEFT	Output current+ 1 long word (32 bits) of trace data in left-justified hexadecimal format. ¹
OUT_REL1_DECLEFT	Output current+ 1 long word (32 bits) of trace data in left-justified decimal format (leading zeros are not printed). ¹
OUT_REL1_BINLEFT	Output current+ 1 long word (32 bits) of trace data in left-justified binary format. ¹

Provided Inverse Assembler Routines	
Routine	Description
OUT_REL2_HEX	Output current+ 2 long word (32 bits) of trace data in hexadecimal format. ¹
OUT_REL2_HEXLEFT	Output current+ 2 long word (32 bits) of trace data in left-justified hexadecimal format. ¹
OUT_REL2_BINLEFT	Output current+ 2 long word (32 bits) of trace data in left-justified binary format. ¹
OUT_REL3_HEX	Output current+ 3 long word (32 bits) of trace data in hexadecimal format. ¹
OUT_REL3_HEXLEFT	Output current+ 3 long word (32 bits) of trace data in left-justified hexadecimal format. ¹
OUT_REL3_BINLEFT	Output current+ 3 long word (32 bits) of trace data in left-justified binary format. ¹
OUT_REL4_HEX	Output current+ 4 long word (32 bits) of trace data in hexadecimal format. ¹
OUT_REL4_HEXLEFT	Output current+ 4 long word (32 bits) of trace data in left-justified hexadecimal format. ¹
OUT_REL5_HEX	Output current+ 5 long word (32 bits) of trace data in hexadecimal format. ¹
OUT_REL6_HEX	Output current+ 6 long word (32 bits) of trace data in hexadecimal format. ¹
LD_ADDR_REL	Load the INPUT_ADDRESS relative to 'REL_POSITION' when 'REL_POSITION' is set relative to long words. ¹
LD_REL_32_BITS	For 16-bit processors only, loads the long word (32 bits) of trace data at 'REL_POSITION' into the accumulator. 'REL_POSITION' is set relative to 16-bit words.
CK_REL_POS_TAG	Check if relative position can be tagged. ¹
PRINT_NAME_AS_ASCII	Name as ascii hex is in INPUT_DATA at relative position 'REL_POSITION'. ¹

Chapter 4: Writing the RTOS Inverse Assembler Code
Writing RTOS Inverse Assembler Code

Provided Inverse Assembler Routines	
Routine	Description
LD_REL_TO_LONG	Loads the long word (32 bits) of trace data at 'REL_POSITION' into the accumulator. 'REL_POSITION' is set relative to long words. ¹
¹ These data output routines are written differently depending on whether the inverse assembler is for 16-bit or 32-bit microprocessors. This is because 16-bit processors take two bus cycles to write a 32-bit value to a data table location while 32-bit processors only take one bus cycle. Therefore, inverse assembler routines for 16-bit processors get data from two analyzer states while inverse assembler routines for 32-bit processors get data from one analyzer state.	

Examples

The demo IAL code to decode data for task switches is shown below.

```
TASK_EXIT_CALL
    OUTPUT    " ---EXITED TASK : index="
    CALL      OUT_RELO_HEXLEFT
    OUTPUT    "-----"
    GOTO      END_RTOS_IAL

TASK_ENTRY_CALL
    OUTPUT    " ---NEXT TASK   : index="
    CALL      OUT_RELO_HEXLEFT
    OUTPUT    "-----"
    GOTO      END_RTOS_IAL
```

The demo IAL code to decode data for the demo OS alloc_memory() service call entry location is shown below.

```
ALLOC_MEM_FNC
    OUTPUT    CALL_STRING
    OUTPUT    ALLOC_MEM_STR
    OUTPUT    "(region_id="
    CALL      OUT_RELO_HEX
    OUTPUT    ", size="
    CALL      OUT_REL1_HEX
    GOTO      END_CALL
```

Chapter 4: Writing the RTOS Inverse Assembler Code

Writing RTOS Inverse Assembler Code

The END_CALL routine is different depending on whether a 16-bit processor (which takes two bus cycles to write a 32-bit value) or 32-bit processor (which takes one bus cycle to write a 32-bit value) is being used. If a 16-bit processor is being used, the inverse assembler uses two states at a time, and the second state should be marked so that it isn't reused:

```
END_CALL
    OUTPUT      " )"
    INPUT       REL,1 ; always write at least one long word
    TAG_WITH    1     ; Set mark so know not to reuse 2nd state
    GOTO END_RTOS_IAL
```

If a 32-bit processor is being used, states are used one at a time, and additional states don't have to be marked:

```
END_CALL
    OUTPUT      " )"
    GOTO END_RTOS_IAL
```

The demo IAL code to decode data for the demo OS alloc_memory() service call exit location is shown below.

```
ALLOC_MEM_RTN
    OUTPUT      RETURN_STRING
    OUTPUT      ALLOC_MEM_STR
    OUTPUT      "(memarea_ptr="
    CALL        OUT_REL1_HEX
    GOTO        ERROR_CODE_END
```

The ERROR_CODE_END routine is different depending on whether a 16-bit processor (which takes two bus cycles to write a 32-bit value) or 32-bit processor (which takes one bus cycle to write a 32-bit value) is being used. If a 16-bit processor is being used, a special routine is called to place the current 16-bits of trace data in the high-order word of the accumulator and the 16-bits of trace data from the next state into the low-order word of the accumulator:

```
ERROR_CODE_END
    OUTPUT      " )" ; display any error on return
    SET        REL_POSITION,0
    CALL        LD_REL_32_BITS
    ;no error, no display
    IF        31,0=20001h THEN GOTO NO_ERROR_DISPLAY
    CALL        DISPLAY_RET_CODE
NO_ERROR_DISPLAY
    GOTO        END_RTOS_IAL
```

Chapter 4: Writing the RTOS Inverse Assembler Code

Writing RTOS Inverse Assembler Code

If a 32-bit processor is being used, a special routine is called to place the current 32-bits of trace data into the accumulator:

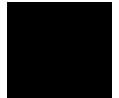
```
ERROR_CODE_END          ; display any error on return
    OUTPUT              " )"
    SET                 REL_POSITION,0
    CALL                LD_REL_TO_LONG
                       ;no error, no display
    IF                 31,0=20001h THEN GOTO NO_ERROR_DISPLAY
    CALL                DISPLAY_RET_CODE
NO_ERROR_DISPLAY
    GOTO                END_RTOS_IAL
```

Building and Installing the RTOS Inverse Assembler

Create your inverse assembler by copying and modifying the demo inverse assembler files from the \$HP64000/rtos/B3082A/interpreter directory; that way, you can use the same make file to build and install your own inverse assembler.

This section shows you how to:

- Build the RTOS inverse assembler.
- Install the RTOS inverse assembler.



To build the RTOS inverse assembler

The "Makefile" included with the demo inverse assembler code concatenates the "ial.S" source file with either "ial16.S" or "ial32.S" (common routines for 16- or 32-bit processors) before assembling the combined file to the inverse assembler.

To make the inverse assembler for 16-bit processors:

```
$ make ial16 <RETURN>
```

To make the inverse assembler for 32-bit processors:

```
$ make ial32 <RETURN>
```

To install the RTOS inverse assembler

- 1 Become the root user.
- 2 Enter the appropriate make command.

The "Makefile" included with the demo inverse assembler code also installs the inverse assembler by copying the inverse assembler to the appropriate directory for the emulator you're using. The demo "Makefile" includes the targets "install302", "install340", and "install020" for the 68302, 68340, and 68020 emulators, respectively.

If, when you start up the Emulator/Analyzer interface, the **display trace real_time_os** command is available, the inverse assembler was installed correctly.

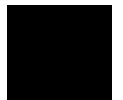
Examples

To install the RTOS inverse assembler for the 68302 emulator:

```
$ make install302 <RETURN>
```

(This step is also accomplished as part of running the \$HP64000/bin/install_rtos script.)

5



Making RTOS Measurements with the Emulator/Analyzer

Making RTOS Measurements with the Emulator/Analyzer

Real-time OS measurements in the emulator/analyzer interface are made using the HP 64700 series emulation bus analyzers. You can set up the analyzer to trace real-time OS activity such as service calls, task switches, and dynamic memory usage by capturing and storing writes made to data table locations.

If you store writes made to all data table locations, you see all OS activity. If you wish to see only certain types of OS activity, use the analyzer to store only writes made to particular data table locations; this is called *selective tracking*.

This chapter shows you how to set up trace commands for tracking particular types of real-time OS activity. Keep in mind that these trace commands can be automated by placing them in command files which can be run by clicking on action keys (refer to the "Automating RTOS Measurements" chapter).

These tasks are described in the following sections:

- Tracing Writes to the Data Table
- Displaying Traces

Analyzer Resource Limitations

The Emulation Bus Analyzer lets you trigger on a sequence of up to 8 captured states. (If you're tracing a window of code execution, you can trigger on a sequence of up to 4 captured states because sequence terms are paired in order to enable and disable the window.) You can qualify which states are stored in trace memory, and you can prestore states that occur prior to qualified store states.

The Emulation Bus Analyzers has 8 state qualifier resources and 1 range qualifier resource.

If there were more than 8 service calls in a real-time OS, it would be impossible to track them all using the analyzer to capture activity at discrete locations. That's why service call functions in the OS code are instrumented to

write to a common data area (the data table); this way, you can capture activity in 1 address range and track all OS activity.

Analyzer resource limitations are also the reason similar types of service calls are grouped together in the data table. If you want to capture only certain types of activity, you can use the single range resource to capture writes to a smaller section of the data table.

You can use the 8 state qualifier resources to capture activity in successive locations, thereby giving you the ability to capture activity in another, limited size address range.

Trace Command Overview

Before you start setting up trace commands to capture data table writes, you should have a general understanding about what the different trace command options allow you to do.

Refer to your Emulator/Analyzer interface *User's Guide* for complete details about the trace command.

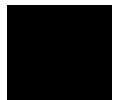
Storage Qualifiers

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.

To specify the storage qualifier, use the "only" trace command option.

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



Trigger Qualifiers

The trigger qualifier tells the analyzer when to store captured states. To trigger the analyzer on a certain event or occurrence in your program, use either the "after", "about", or "before" trace command option. The option you choose specifies the position of the trigger point in trace memory.

Sequence Triggering

To trigger the analyzer on a certain sequence of events or occurrences in your program, use the "find_sequence" trace command option.

Capturing Windows of Execution

To capture only certain sections (in other words, windows) of program execution, use the "enable" and "disable" trace command options.

Data Bus Width Differences

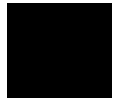
When using data qualifiers to identify the entry or exit of a particular task (or any other data value written to the data table), remember the emulation bus analyzer captures 16 bits of data per state when used with 16-bit processors and 32 bits of data per state when used with 32-bit processors. Because 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.

Tracing Writes to the Data Table

Once the real-time OS code has been instrumented to write values to the data table when tasks switch or when service calls are entered or exited, you can view OS activity in real-time by setting up the analyzer to capture writes to the data table.

When you display the real-time OS trace, the RTOS inverse assembler decodes the captured data table writes into OS task and service call mnemonics.

Remember that the amount of data captured for each state (16- or 32-bits depending on the processor) affects the trace commands used to capture RTOS information if, as in the demo RTOS, the data table is made up of 32-bit long integer locations.



This section shows you how to set up trace commands to:

- Track everything.
- Track task switches and service calls.
- Track groups of service calls.
- Track a single service call.
- Include task switches when selectively tracking.
- Track two service calls.
- 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.
- Track activity after a function is reached.
- Track activity about the access of a variable by a specific task.
- Track which tasks access a specific function or variable.
- Track all memory calls (including task switches).

Chapter 5: Making RTOS Measurements with the Emulator/Analyzer Tracing Writes to the Data Table

Example Data Table

The examples in this section assume the following data table (which is used with the demo RTOS measurement tool):

```
/*
*****
/*      --- THIS DATA TABLE MUST NOT BE CHANGED IN ANY WAY ---
/*      --- The interpretation of 'traced' data is dependent ---
/*      --- on the relative offsets of symbols ---
*****
long HP_RTOS_TRACK_START;    /* The name of this symbol MUST NOT CHANGE!!! */
                           /* 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. */

long HPOS_TASK_ENTRY;
long HPOS_TASK_EXIT;

long HPOS_task_create_Entry[7];
long HPOS_task_create_Exit;

long HPOS_send_message_Entry[2];
long HPOS_send_message_Exit;
long HPOS_get_message_Entry[2];
long HPOS_get_message_Exit[2];

long HPOS_alloc_message_Entry;
long HPOS_alloc_message_Exit[2];
long HPOS_free_message_Entry;
long HPOS_free_message_Exit;

long HPOS_alloc_mem_Entry;
long HPOS_alloc_mem_Exit;

char HPOS_String_array[12];    /* area to write string */

long HPOS_USER_DEFENTRY[12];    /* data entries to be used for */
                               /* either SPA intervals or */
                               /* for general program tracking */

char HP_RTOS_TRACK_END;

short int HPOS_CLOCK_TICK;
```

To track everything

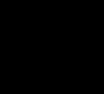
- Capture all writes to the data table.

Examples

```
trace only address range HP_RTOS_TRACK_START thru  
HP_RTOS_TRACK_END <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_trkall" command file which is run by clicking on the "Track Everything" action key.



To track task switches and service calls

- Capture all writes to the task and service call entry and exit locations in the data table.

Examples

```
trace only address range HP_RTOS_TRACK_START thru  
HPOS_USER_DEFENTRY-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_trkcalls" command file which is run by clicking on the "Track OS calls" action key.

To track groups of service calls

- Capture writes to the range of data table locations that contain the entry and exit locations of the group of service calls.

When organizing the data table, its important to group the entry and exit locations for certain types of related service calls so they are easy to trace (using the 1 range resource of the analyzer).

Examples

To track the service calls associated with message queues:

```
trace only address range HPOS_send_message_Entry thru  
HPOS_alloc_message_Entry-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_onlyqs" command file which is run by clicking on the "Only Queues" action key.

To track a single service call

- Capture writes to the range of data table locations that contain the entry and exit locations of the particular service call.

Examples

To track the send_message() service call:

```
trace only address range HPOS_send_message_Entry thru  
HPOS_get_message_Entry-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are created by the "e_onecall" command file which is run by clicking on the "Only Call X" action key.

To include task switches when selectively tracking

- Capture writes to the HPOS_TASK_ENTRY and HPOS_TASK_EXIT locations in addition to the range of data table locations that contain the entry and exit locations of the particular service calls.

Examples

To track the service calls associated with message queues including task switches:

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

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_trackqs_16" command file which is run by clicking on the "Tasks & Queues" action key. Note that this example is for a processor with a 16-bit data bus. There is a "e_trackqs_32" command file for processors with a 32-bit data bus which is also run by clicking on the "Tasks & Queues" action key.

To track the send_message() service call including task switches:

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

```
display trace real_time_os <RETURN>
```

To track two service calls

- Capture writes to each of the service call's entry and exit locations in data table.

Because two service calls represent two, perhaps non-contiguous, ranges in the data table (and the analyzer resources only let you specify one range), a different method must be used than when tracking a single service call.

Because you know values are written to an entry (or exit) location one after the other, you can consider the writes to the entry and exit locations of two service calls as 4 separate sections, or windows, of code execution. With this in mind, you can set up a trace command to store writes during one of 4 windows of code execution.

Examples

To track the `get_message()` and `alloc_message()` service calls:

```
trace enable HPOS_get_message_Entry or  
HPOS_get_message_Exit or HPOS_free_message_Entry or  
HPOS_free_message_Exit disable  
HPOS_get_message_Exit-2 or HPOS_alloc_message_Entry-2  
or HPOS_free_message_Exit-2 or HPOS_String_array-2  
only address range HP_RTOS_TRACK_START thru  
HP_RTOS_TRACK_END <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are created by the "e_twocalls_16" command file which is run by clicking on the "Only Calls X & Y" action key. Note that this example is for a processor with a 16-bit data bus. There is a "e_twocalls_32" command file for processors with a 32-bit data bus which is also run by clicking on the "Only Calls X & Y" action key.

To track a single task and all OS activity within it

- Capture the window of code execution between when the task is switched into and when it is switched out of and store all data table writes during that window of code execution.

Because task IDs are written to the HPOS_TASK_ENTRY and HPOS_TASK_EXIT locations when tasks start or switch, you can identify when a particular task is switched into by using address and data qualifiers in the trace command.

Remember:

With 16-bit processors, two bus cycles are used to write the 32-bit task ID to the HPOS_TASK_ENTRY and HPOS_TASK_EXIT locations.

With 32-bit processors, one bus cycle is used to write the 32-bit task ID to the HPOS_TASK_ENTRY and HPOS_TASK_EXIT locations.

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

Examples

To track task 3 and all the OS activity within it (16-bit processor):

```
trace enable address HPOS_TASK_ENTRY+2 data 3 disable  
address HPOS_TASK_EXIT+2 data 3 only address range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1  
prestore address HPOS_TASK_ENTRY <RETURN>  
  
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_trk1task_16" command file which is run by clicking on the "Only Task X" action key.

Chapter 5: Making RTOS Measurements with the Emulator/Analyzer

Tracing Writes to the Data Table

To track task 3 and all the OS activity within it (32-bit processor):

```
trace enable address HPOS_TASK_ENTRY data 3 disable  
address HPOS_TASK_EXIT data 3 only address range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1 <RETURN>  
  
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_trk1task_32" command file which is run by clicking on the "Only Task X" action key.

To track four tasks and all OS activity within them

- Capture one of four windows of code execution between when tasks are switched into and when they are switched out of and store all data table writes during those windows of code execution.

Because the four windows of code execution never overlap, you can use the "or" trace command option along with the "enable" and "disable" options to trace up to four tasks.

Chapter 5: Making RTOS Measurements with the Emulator/Analyzer Tracing Writes to the Data Table

Examples

To track tasks 1, 2, 3, or 4 and all the OS activity within them (16-bit processor):

```
trace enable HPOS_TASK_ENTRY+2 data 1 or  
HPOS_TASK_ENTRY+2 data 2 or HPOS_TASK_ENTRY+2 data 3  
or HPOS_TASK_ENTRY+2 data 4 disable address  
HPOS_TASK_EXIT+2 data 1 or HPOS_TASK_EXIT+2 data 2 or  
HPOS_TASK_EXIT+2 data 3 or HPOS_TASK_EXIT+2 data 4  
only address range HP_RTOS_TRACK_START thru  
HPOS_USER_DEFENTRY prestore address HPOS_TASK_ENTRY  
<RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_trk4task_16" command file which is run by clicking on the "Only Tsk W,X,Y,Z" action key.

To track tasks 1, 2, 3, or 4 and all the OS activity within them (32-bit processor):

```
trace enable HPOS_TASK_ENTRY data 1 or  
HPOS_TASK_ENTRY data 2 or HPOS_TASK_ENTRY data 3 or  
HPOS_TASK_ENTRY data 4 disable address HPOS_TASK_EXIT  
data 1 or HPOS_TASK_EXIT data 2 or HPOS_TASK_EXIT  
data 3 or HPOS_TASK_EXIT data 4 only address range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_trk4task_32" command file which is run by clicking on the "Only Tsk W,X,Y,Z" action key.

To track about a specific task switch

- Capture the sequence of the first task exiting followed immediately by the second task entering. (If any other task is entered after the first task exits, restart the sequence search.)

Examples

To track about the switch from task 1 to task 2 (16-bit processor):

```
trace find_sequence HPOS_TASK_EXIT+2 data 1 then  
HPOS_TASK_ENTRY+2 data 2 restart HPOS_TASK_ENTRY+2  
data not 2 trigger about 0xxxxxxxxxh only range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_AthenB_16" command file which is run by clicking on the "Task switch A-> B" action key.

To track about the switch from task 1 to task 2 (32-bit processor):

```
trace find_sequence HPOS_TASK_EXIT data 1 then  
HPOS_TASK_ENTRY data 2 restart HPOS_TASK_ENTRY data  
not 2 trigger about 0xxxxxxxxxh only range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_AthenB_32" command file which is run by clicking on the "Task switch A-> B" action key.

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

- Capture the sequence of the task entering followed by a message being sent to the queue. If the task exits before the message is sent, restart the sequence search.

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.

Examples

To track about task 5 sending a message to queue 6 (16-bit processor):

```
trace find_sequence HPOS_TASK_ENTRY+2 data 5 restart  
HPOS_TASK_EXIT+2 data 4 trigger about  
HPOS_send_message_Entry+10 data 6 only address range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_tsk2queue_16" command file which is run by clicking on the "Task A msg-> Que X" action key.

To track about task 5 sending a message to queue 6 (32-bit processor):

```
trace find_sequence HPOS_TASK_ENTRY data 5 restart  
HPOS_TASK_EXIT data 4 trigger about  
HPOS_send_message_Entry+8 data 6 only address range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1 <RETURN>
```

```
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_tsk2queue_32" command file which is run by clicking on the "Task A msg-> Que X" action key.

To track activity after a function is reached

- Capture the window of code execution between when the task is switched into and when it is switched out of, trigger after the function is reached, and store all states.

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.

Examples

To track activity after the function "send_message" is reached in task 5 (16-bit processor):

```
trace enable address HPOS_TASK_ENTRY+2 data 5 disable  
address HPOS_TASK_EXIT+2 data 5 after send_message  
<RETURN>
```

```
set source only <RETURN>
```

```
display trace mnemonic <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_afterfunc_16" command file which is run by clicking on the "Task A: FuncX" action key.

To track activity after the function "send_message" is reached in task 5 (32-bit processor):

```
trace enable address HPOS_TASK_ENTRY data 5 disable  
address HPOS_TASK_EXIT data 5 after send_message  
<RETURN>
```

```
set source only <RETURN>
```

```
display trace mnemonic <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_afterfunc_32" command file which is run by clicking on the "Task A: FuncX" action key.

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

- Capture the window of code execution between when the task is switched into and when it is switched out of, trigger about the access of the variable, and store all states.

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.

Note that in order to see source lines in the trace, the source files must either be in the current directory or in a directory defined by the HP64_DEBUG_PATH environment variable.

Examples

To track activity about the access of variable "task6_queue" by task 5 (16-bit processor):

```
trace enable address HPOS_TASK_ENTRY+2 data 5 disable  
address HPOS_TASK_EXIT+2 data 5 about task6_queue  
<RETURN>
```

```
set source only <RETURN>
```

```
display trace mnemonic <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_aftervar_16" command file which is run by clicking on the "Task A: VarX" action key.

Chapter 5: Making RTOS Measurements with the Emulator/Analyzer

Tracing Writes to the Data Table

To track activity about the access of variable "task6_queue" by task 5 (32-bit processor):

```
trace enable address HPOS_TASK_ENTRY data 5 disable  
address HPOS_TASK_EXIT data 5 about task6_queue  
<RETURN>
```

```
set source only <RETURN>
```

```
display trace mnemonic <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_aftervar_32" command file which is run by clicking on the "Task A: VarX" action key.

To track which tasks access a specific function or variable

- Capture the address of the function or variable and prestore writes to the HPOS_TASK_EXIT and HPOS_TASK_ENTRY data table locations.

(An easy way to specify these data table locations is to use the range HP_RTOS_TRACK_START through HP_RTOS_TRACK_START+ 7.)

Examples

To track which tasks access the variable "task6_queue":

```
trace prestore address range HP_RTOS_TRACK_START thru  
+7 only address task6_queue <RETURN>
```

```
display trace real_time_os <RETURN>
```


To track all memory calls (including task switches)

- Capture writes to the range of data table locations that contain the entry and exit locations of the memory allocation service call in addition to the HPOS_TASK_ENTRY and HPOS_TASK_EXIT locations.

Examples

(16-bit processors):

```
trace only address range HPOS_alloc_mem_Entry thru +7  
or HPOS_TASK_ENTRY or HPOS_TASK_ENTRY+2 or  
HPOS_TASK_EXIT or HPOS_TASK_EXIT+2 <RETURN>  
  
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_memory_16" command file which is run by clicking on the "Memory Usage" action key.

(32-bit processors):

```
trace only address range HPOS_alloc_mem_Entry thru +7  
or HPOS_TASK_ENTRY or HPOS_TASK_EXIT <RETURN>  
  
display trace real_time_os <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "e_memory_32" command file which is run by clicking on the "Memory Usage" action key.

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.

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.

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

- Enter the **display trace mnemonic** command.

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	tst_task_+000034	BLT.B	prog tst_task_6+000030	-----	
+001	_HPOS_send_messa	0004	sdata wr word	1.38	mS
+002	_HPOS_sen+000002	5C41	sdata wr word	240	nS
+003	_HPOS_sen+000004	0000	sdata wr word	1.5	uS
+004	_HPOS_sen+000006	62B8	sdata wr word	240	nS
+005	_HPOS_sen+000008	0000	sdata wr word	2.6	uS
+006	_HPOS_sen+00000A	0007	sdata wr word	280	nS
+007	[_HPOS_TASK_EXIT	0000	sdata wr word	186.	uS
+008	_HPOS_TAS+000002	0006	sdata wr word	240	nS
+009	_HPOS_TASK_ENTRY	0000	sdata wr word	12.9	uS
+010	_HPOS_TAS+000002	0007	sdata wr word	240	nS
+011	_HPOS_get_messag	0002	sdata wr word	40.76	uS
+012	_HPOS_get+000002	0001	sdata wr word	240	nS
+013	_HPOS_get+000004	0004	sdata wr word	3.1	uS
+014	_HPOS_get+000006	5C41	sdata wr word	280	nS
+015	_HPOS_free_messa	0004	sdata wr word	13.0	uS

The inverse assembler included with the Emulator/Analyzer interface is used to disassemble states captured by the analyzer.

To switch to the RTOS trace display

- Enter the **display trace real_time_os** command on the command line.

The screenshot shows a trace window with the following content:

Trace List	Offset=0	More data off screen
Label:	Real Time Operating System	time count
Base:	with symbols	relative
after	NON-RTOS: addr=prog tst_task_6+34 data=000060FA	-----
+001	-> send_message(msg_value=00045C41 queue=dat _task7_queue, q_index=00000007)	1.38 mS
+007	---EXITED TASK : index=6-----	191. uS
+009	---NEXT TASK : index=7-----	13.1 uS
+011	<- get_message(msg_value=00045C41)	41.00 uS
+015	-> free_message(msg_value=00045C41)	16.6 uS
+017	<- free_message()	11.1 uS
+019	-> get_message(msg_ptr=00005170 ,queue=dat _task7_queue)	19.0 uS
+023	---EXITED TASK : index=7-----	97.2 uS
+025	---NEXT TASK : index=6-----	13.1 uS
+027	<- send_message()	37.9 uS
+029	-> get_message(msg_ptr=00004970 ,queue=dat _task6_queue)	19.8 uS
+033	---EXITED TASK : index=6-----	97.2 uS
+035	---NEXT TASK : index=5-----	13.1 uS
+037	<- send_message()	37.9 uS

Annotations on the left side of the screenshot:

- Service call entry. (points to line +001)
- Service call exit. (points to line +017)
- Task switch. (points to line +025)
- Parameters (decoded if possible). (points to the parameters in line +001)

Annotation on the right side of the screenshot:

- Time stamp., (points to the time count column)

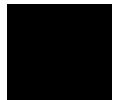
When you enter the **display trace real_time_os** command, the real-time OS inverse assembler is used to decode states captured by the analyzer.

The base address of the data table (that is, the address of the `HP_RTOS_TRACK_START` symbol) is passed to the RTOS inverse assembler where it is used to calculate the addresses of other data table locations.

Because the RTOS inverse assembler knows what writes to the different data table locations mean, it is able to output formatted information about real-time OS operation.

Notice 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.

6



Making RTOS Measurements with the SPA

Making RTOS Measurements with the SPA

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.

These tasks are grouped into the following sections:

- Making time profile measurements.
- Coordinating measurements with the emulator.

Keep in mind that the SPA commands shown in this chapter can be automated by placing them in command files which can be run by clicking on action keys (refer to the "Automating RTOS Measurements" chapter).

SPA Data Table Requirements

In order to be able to provide time profile measurements, the Software Performance Analyzer measures the time between accesses of different locations.

The data table entry and exit locations set up for service call tracking by the Emulation Bus Analyzer can be used by the Software Performance Analyzer for measuring service call time intervals.

Additional entry and exit locations must be set up for task intervals, OS overhead, and measurement intrusion. There are no Software Performance Analyzer limitations that require these locations to be grouped together in the data table or to even be placed in the data table in the first place. However, grouping these locations together in the data table makes for simpler OS code instrumentation.

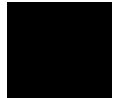
SPA Command Overview

Before you start setting up SPA commands to perform time profile measurements, you should have a general understanding about what the different SPA commands.

Refer to the *Software Performance Analyzer User's Guide* for complete details about the SPA commands.

Sorting Events and Re-scaling Histograms

You can sort the events displayed in a SPA profile measurement with the **display histogram sort_events** command. You can re-scale the histograms displayed in a SPA profile measurement with the **display histogram rescale** command.



Demo Data Table Entries for SPA

The sections of the demo RTOS measurement tool's data table that are for RTOS measurements with the software performance analyzer are:

```
short int HPOS_Start_Ovrhd;          /* Start of OS interval for SPA */
short int HPOS_Stop_Ovrhd;          /* End of OS interval for SPA */

short int HPOS_Start_Intrusion;     /* Start interval for measuring intrusion*/
short int HPOS_Stop_Intrusion;     /* End interval for measuring intrusion*/

/*-----*/
/*-----          BEGIN TASK MODIFICATIONS          -----*/

short int HPOS_TaskTable;
short int HPOS_Tenter_0000,HPOS_Texit_0000;
short int HPOS_Tenter_0001,HPOS_Texit_0001;
short int HPOS_Tenter_0002,HPOS_Texit_0002;
short int HPOS_Tenter_0003,HPOS_Texit_0003;
short int HPOS_Tenter_0004,HPOS_Texit_0004;
short int HPOS_Tenter_0005,HPOS_Texit_0005;
short int HPOS_Tenter_0006,HPOS_Texit_0006;
short int HPOS_Tenter_0007,HPOS_Texit_0007;
short int HPOS_Tenter_0008,HPOS_Texit_0008;
short int HPOS_Tenter_0009,HPOS_Texit_0009;

/*-----*/
/*-----          END TASK MODIFICATIONS          -----*/
/*-----*/
```

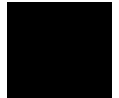

Making Time Profile Measurements

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

- 1 Define event intervals for tasks.
- 2 Define event intervals for measurement intrusion and overhead.
- 3 Define event intervals for service calls.
- 4 Define event intervals for user-defined events.

Examples

To define event intervals for tasks:

```
define single_event named Task_0001 interval  
HPOS_Tenter_0001 thru HPOS_Texit_0001 <RETURN>
```

```
define single_event named Task_0002 interval  
HPOS_Tenter_0002 thru HPOS_Texit_0002 <RETURN>
```

```
define single_event named Task_0003 interval  
HPOS_Tenter_0003 thru HPOS_Texit_0003 <RETURN>
```

```
define single_event named Task_0004 interval  
HPOS_Tenter_0004 thru HPOS_Texit_0004 <RETURN>
```

```
define single_event named Task_0005 interval  
HPOS_Tenter_0005 thru HPOS_Texit_0005 <RETURN>
```

```
define single_event named Task_0006 interval  
HPOS_Tenter_0006 thru HPOS_Texit_0006 <RETURN>
```

```
define single_event named Task_0007 interval  
HPOS_Tenter_0007 thru HPOS_Texit_0007 <RETURN>
```

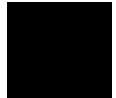
Chapter 6: Making RTOS Measurements with the SPA Making Time Profile Measurements

To define event intervals for measurement intrusion and overhead:

```
define single_event named OS_Time interval  
HPOS_Start_Ovrhd thru HPOS_Stop_Ovrhd <RETURN>  
  
define single_event named Measure_Ovrhd interval  
HPOS_Start_Intrusion thru HPOS_Stop_Intrusion <RETURN>
```

To define event intervals for service calls.

```
define single_event named Srvccall_task_create  
interval HPOS_task_create_Entry thru  
HPOS_task_create_Exit <RETURN>  
  
define single_event named Srvccall_send_message  
interval HPOS_send_message_Entry thru  
HPOS_send_message_Exit <RETURN>  
  
define single_event named Srvccall_get_message  
interval HPOS_get_message_Entry thru  
HPOS_get_message_Exit <RETURN>  
  
define single_event named Srvccall_alloc_message  
interval HPOS_alloc_message_Entry thru  
HPOS_alloc_message_Exit <RETURN>  
  
define single_event named Srvccall_free_message  
interval HPOS_free_message_Entry thru  
HPOS_free_message_Exit <RETURN>  
  
define single_event named Srvccall_alloc_memory  
interval HPOS_alloc_memory_Entry thru  
HPOS_alloc_memory_Exit <RETURN>
```



Chapter 6: Making RTOS Measurements with the SPA Making Time Profile Measurements

To define event intervals for user-defined events:

```
define single_event named UserIntr_1 interval  
HPOS_USER_DEFENTRY thru HPOS_USER_DEFENTRY+3h <RETURN>
```

```
define single_event named UserIntr_2 interval  
HPOS_USER_DEFENTRY+4 thru HPOS_USER_DEFENTRY+7h  
<RETURN>
```

```
define single_event named UserIntr_3 interval  
HPOS_USER_DEFENTRY+8 thru HPOS_USER_DEFENTRY+0bh  
<RETURN>
```

```
define single_event named UserIntr_4 interval  
HPOS_USER_DEFENTRY+0ch thru HPOS_USER_DEFENTRY+0fh  
<RETURN>
```

```
define single_event named UserIntr_5 interval  
HPOS_USER_DEFENTRY+10h thru HPOS_USER_DEFENTRY+13h  
<RETURN>
```

```
define single_event named UserIntr_6 interval  
HPOS_USER_DEFENTRY+14h thru HPOS_USER_DEFENTRY+17h  
<RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "s_init" command file which is created by running the \$HP64000/bin/rtos_edit script and run by clicking on the "Initialize" action key.

To display a time histogram of task events

- 1 Stop the current profile measurement.
- 2 Select the task events.
- 3 Start the measurement.
- 4 Display a time histogram.

Examples

To stop the current profile measurement:

```
stop_profile <RETURN>
```

To select the events:

```
select_events matching "Task_*" <RETURN>
```

```
select_events matching "OS_Time" <RETURN>
```

```
select_events matching "Measure_Ovrhd" <RETURN>
```

To start the measurement:

```
setup_measurement enable off <RETURN>
```

```
setup_measurement disable off <RETURN>
```

```
profile interval_duration <RETURN>
```

To display a time histogram:

```
display histogram data time <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "s_timetasks" command file which is run by clicking on the "Time Tasks" action key.

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.

To display a count histogram of task events

- 1 Stop the current profile measurement.
- 2 Select the task events.
- 3 Start the measurement.
- 4 Display a call histogram.

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.

Examples

To stop the current profile measurement:

```
stop_profile <RETURN>
```

To select the events:

```
select_events matching "Task_*" <RETURN>
```

To start the measurement:

```
setup_measurement enable off <RETURN>
```

```
setup_measurement disable off <RETURN>
```

```
profile interval_duration <RETURN>
```

To display a call histogram:

```
display histogram data calls <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "s_counttasks" command file which is run by clicking on the "Count Tasks" action key.

To measure only data from a specific task

- 1 Stop the current profile measurement.
- 2 Select the service call events.
- 3 Start the measurement.
- 4 Display a call histogram.

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

Examples

To measure the number of times each service call is invoked from Task_0003, perform the steps below.

To stop the current profile measurement:

```
stop_profile <RETURN>
```

To select the events:

```
select_events matching "Srvccall_*" <RETURN>
```

Chapter 6: Making RTOS Measurements with the SPA Making Time Profile Measurements

To start the measurement:

```
setup_measurement enable start_address Task_0003  
<RETURN>
```

```
setup_measurement disable end_address Task_0003  
<RETURN>
```

```
profile interval_duration <RETURN>
```

To display a call histogram:

```
display_histogram data call <RETURN>
```

In the demo RTOS measurement tool, these commands are created by, and found in, the "s_taskwindow" command file which is run by clicking on the "TaskX: Servcalls" action key.

To show a table of service call invocations

- 1 Stop the current profile measurement.
- 2 Select the service call events.
- 3 Start the measurement.
- 4 Display a call histogram.

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

Examples

To stop the current profile measurement:

```
stop_profile <RETURN>
```

To select the events:

```
select_events matching "Srvccall_*" <RETURN>
```

To start the measurement:

```
setup_measurement enable off <RETURN>
```

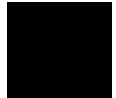
```
setup_measurement disable off <RETURN>
```

```
profile interval_duration <RETURN>
```

To display a call histogram:

```
display_histogram data calls <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "s_countsrvcls" command file which is run by clicking on the "Count Srv Calls" action key.



To show a normal function duration histogram

- 1 Stop the current profile measurement.
- 2 Select function events.
- 3 Start the measurement.
- 4 Display a time histogram.

This performs a normal function duration profile measurement.

Examples

To stop the current profile measurement:

```
stop_profile <RETURN>
```

To select the events:

```
define_multiple_events_functions <RETURN>
```

```
select_events <RETURN>
```

```
unselect_events_matching "Task_*" <RETURN>
```

```
unselect_events_matching "Srvccall_*" <RETURN>
```

```
unselect_events_matching "UserIntr*" <RETURN>
```

```
unselect_events OS_Time <RETURN>
```

```
unselect_events Measure_Ovrhd <RETURN>
```

To start the measurement:

```
setup_measurement enable off <RETURN>
```

```
setup_measurement disable off <RETURN>
```

```
profile function_duration exclude_calls <RETURN>
```

To display a time histogram:

```
display_histogram data time <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "s_funcdur" command file which is run by clicking on the "FunctionDuration" action key.

To show a histogram of task and user events

- Create a command file that selects function events, starts the measurement, and displays a time histogram.

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

Examples

To stop the current profile measurement:

```
stop_profile <RETURN>
```

To select the events:

```
select_events matching "UserIntr*" <RETURN>
```

```
select_events append matching "Task_*" <RETURN>
```

To start the measurement:

```
setup_measurement enable off <RETURN>
```

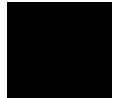
```
setup_measurement disable off <RETURN>
```

```
profile interval_duration <RETURN>
```

To display a time histogram:

```
display histogram data time <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "s_taskuser" command file which is run by clicking on the "Tsk & User Evnts" action key.



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:

- Configure the emulation analyzer to receive trig2.
- Break on task time overflow.
- Disable the SPA trig2.

To configure the emulation analyzer to receive trig2

- If you wish to make cross-trigger measurements between SPA and the emulation system, make sure the emulation configuration has the following question and answer:

Should Analyzer drive or receive Trig2? receive

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

To break on task time overflow

- 1 In the emulation window, set up the analyzer to be armed on trig2 while specifying the trace command.
- 2 In the SPA window, setup the measurement to drive trig2 after a task exceeds a certain amount of time.

You can 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.

Examples

To set up emulation analyzer to measure all OS activity before Task_0003 exceeds 5 milliseconds, follow the instructions below.

First, set up the emulation bus analyzer to trace all OS activity before the signal on trig2 by entering the following commands in the Emulator/Analyzer window:

```
trace arm_trig2 before 0xxxxxxxxxh only address range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1 <RETURN>  
  
display trace real_time_os <RETURN>
```

If you wish to stop program execution at that point, you could enter the command:

```
trace arm_trig2 before 0xxxxxxxxxh only address range  
HP_RTOS_TRACK_START thru HPOS_USER_DEFENTRY-1  
break_on_trigger <RETURN>
```

Note that the trace has started but has not completed because it is waiting for the trig2 signal as its trigger point.

In the demo RTOS measurement tool, these commands are found in the "e_spatrig" command file which is run by clicking on the "Before SPA trig2" action key.

Chapter 6: Making RTOS Measurements with the SPA Coordinating Measurements with the Emulator

Next, set up the software performance analyzer measurement to drive trig2 after a task exceeds a certain amount of time by entering the following commands in the SPA interface:

```
stop_profile <RETURN>

select_events_matching "Task_*" <RETURN>

setup_measurement drive trig2_after 5 msec Task_0003
<RETURN>

setup_measurement enable off <RETURN>

setup_measurement disable off <RETURN>

profile interval_duration <RETURN>

display_histogram data time <RETURN>

expand Task_0003 <RETURN>
```

When the trace has completed, it is displayed in the Emulator/Analyzer interface window. The resulting trace 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.

In the demo RTOS measurement tool, these commands are found in the "s_break_ovrflw" command file which is run by clicking on the "Trig2 on Overflow" action key.

Note

If the "TaskX: Servcalls" action key (or the "s_taskwindow" command file) is used before the "Trig2 on Overflow" action key (or the "s_break_ovrflw" command file), the "enable" and "disable" measurement setups must be removed by the command **setup_measurement default** before the first action.

To disable the SPA trig2

- In the SPA window, set up a measurement to turn the "driving" of trig2 OFF.

You must disable the driving of trig2 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.

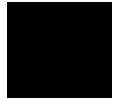
Examples

To turn OFF the "driving" of trig2:

```
stop_profile <RETURN>
```

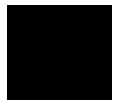
```
setup_measurement drive off <RETURN>
```

In the demo RTOS measurement tool, these commands are found in the "s_disabletrg2" command file which is run by clicking on the "Disable Trig2" action key.





7



Automating RTOS Measurements

Automating RTOS Measurements

Automating RTOS measurements, in either the Emulator/Analyzer or SPA interfaces, is simply a matter of placing commands in command files and setting up action keys that, when clicked, run the command files.

The action keys for the demo are set up by the two startup scripts: `rtos_emul` and `rtos_spa`. The action keys run command files in the `$HPP64000/rtos/B3082A/action_keys` directory. It's best to copy these files to a directory of your own before editing.

Setting up command files and placing them on action keys are described in the following sections:

- Using Command Files
- Using Action Keys

Refer to your Emulator/Analyzer interface *User's Guide* for complete details about using command files.

Demo Action Keys and Related Command Files

The following tables list the action key labels that appear in the Emulator/Analyzer and Software Performance Analyzer interfaces, the command files that are run by clicking on the action key, and a short description of what the command files do.

Command files are installed in the `$HPP64000/rtos/B3082A/action_keys` directory.

The contents of the Emulator/Analyzer command files appear as examples in the "Making RTOS Measurements with the Emulator/Analyzer" chapter.

The contents of the Software Performance Analyzer command files appear as examples in the "Making RTOS Measurements with the SPA" chapter.

Emulator/Analyzer Action Keys and Related Command Files		
Action Key	Command File/Script	Description
Track OS calls	e_trkcalls	Tracks OS service calls and task switches.
Track Everything	e_trkall	Tracks service calls, task switches, string writes, and user-defined activity.
Task switch A-> B	e_AthenB_16 e_AthenB_32	Tracks OS activity about a specific task switch.
Help RTOS	!tellrtosHP !in_browser	The "tellrtosHP" script places help information in the browser window.
Memory Usage	e_memory_16 e_memory_32	Tracks memory allocation service calls as well as task switches.
Only Call X	e_onecall	Tracks a single service call.
Only Calls X & Y	e_twocalls_16 e_twocalls_32	Tracks two service calls.
Tsk A msg-> Que X	e_tsk2queue_16 e_tsk2queue_32	Tracks OS activity about a specific task sending a message to a specific queue.
Custom OS Trace	display trace real_time_os	This command displays the real-time OS trace.
< UserDefinable1>		You must edit the "rtos_emul" script to define this key.
Only Task X	e_trk1task_16 e_trk1task_32	Tracks a single task and all OS activity within it.
Only Tsk W,X,Y,Z	e_trk4task_16 e_trk4task_32	Tracks four tasks and all OS activity within them.
Task A: VarX	e_aftervar_16 e_aftervar_32	Tracks OS activity about the access of a variable by a specific task.
NonCustom Trace	display trace mnemonic	This command displays the normal emulator/analyzer trace.
< UserDefinable2>		You must edit the "rtos_emul" script to define this key.

Emulator/Analyzer Action Keys and Related Command Files		
Action Key	Command File/Script	Description
Tasks & Queues	e_trackqs_16 e_trackqs_32	Tracks the service calls associated with message queues as well as task switches.
Only Queues	e_onlyqs	Tracks the service calls associated with message queues.
Task A: FuncX	e_afterfunc_16 e_afterfunc_32	Tracks C source code execution after a function is called from a certain task.
Before SPA trig2	e_spatrig	Sets up the emulation analyzer to trace all activity before the signal on trig2.

Software Performance Analyzer Action Keys and Related Command Files		
Action Key	Command File/Script	Description
Initialize	s_init\$RTOS_UNIQUE	Defines SPA events for tasks, service calls, and user events.
Time Tasks	s_timetasks	Displays a time histogram of task events.
Count Srvc Calls	s_countsrvcls	Displays a call histogram of service call invocations.
Trig2 on Overflow	s_break_ovrflw	Sets up the SPA to drive trig2 after a task exceeds a certain amount of time.
FunctionDuration	s_funcdur	Displays a function duration time histogram.
TaskX: Servcalls	s_taskwindow	Measures only data from a specific task.
Count Tasks	s_counttasks	Displays a count histogram of task events.
Tsk & User Evnts	s_taskuser	Displays a time histogram of task and user events.
Disable Trig2	s_disabletrg2	Turns OFF the driving of trig2.

Using Command Files

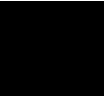
Command files are ASCII text files that contain interface commands.

Command files can prompt the user for input. This lets you set up general purpose command files. For example, you can set up a command file that tracks one service call and prompts the user for the name of the service call.

You can also run shell scripts from the interfaces. This gives you even greater flexibility when automating measurements.

Command files for the demo RTOS measurement tool are included with this product in the \$HWP64000/rtos/B3082A/action_keys directory.

This section shows you how to:

- Place measurements in command files.
 - Use command file parameters to pass in variables.
 - Use shell scripts from within command files.
 - Define command file search directories.
- 

To place your measurements in command files

- Enter the commands into an ASCII text file.

Examples

The Emulator/Analyzer command file to track only message queues looks like:

```
#
# HP RTOS Command File: e_onlyqs
#
# DESCRIPTION: Track only queue calls
#
trace only address range HPOS_send_message_Entry thru
HPOS_alloc_message_Entry-1
#
display trace real_time_os
```

The SPA command file to display a time histogram of the task events looks like:

```
#
# HP RTOS Command File: s_timetasks
#
# DESCRIPTION: Time histogram only of the tasks
#
stop_profile
select_events matching "Task_*"
setup_measurement enable off
setup_measurement disable off
profile interval_duration
display histogram data time
wait 2
display histogram sort_events time
display histogram rescale current_max
```

To use command file parameters to pass in variables

- Declare the formal parameters in the command file, and use them in the commands that follow.

When the command file is run from an emulation session, the user will be prompted for all parameters listed on the first line after the keyword PARMs. All parameters must be prefixed by the ampersand character "&", both on the first line and any place used within the command file.

Examples

The Emulator/Analyzer command file to track a single task and all OS activity within it looks like:

```
PARMS &TASK
#
# HP RTOS Command File: e_trkltask_32
#
# DESCRIPTION: Track a single task and all the OS activity within it
#
trace enable address HPOS_TASK_ENTRY data &TASK disable address
HPOS_TASK_EXIT data &TASK only address range HP_RTOS_TRACK_START thru
HPOS_USER_DEFENTRY-1
#
display trace real_time_os
```

To use shell scripts from within command files

- 1 Create the shell script.
- 2 Create command files that use the shell script.

Shell scripts can be used from within command files to increase the power and flexibility of command file. For example, you can use shell scripts to:

- Generate a task interval name given the task number.
- Generate a command to track 1 or 2 service calls given the names of the service calls.

Often, shell scripts perform common operations that can be used by several different command files.

Examples

The SPA command file to display a call histogram of service calls made within a particular task looks like:

```
PARMS &TASK
#
# HP RTOS Command File: s_taskwindow
#
# DESCRIPTION: Histogram of functions windowed on a specific task
#
stop_profile
select_events matching "Srvccall_*"
#
# Make task into form 'xxxx'
!get_task_number &TASK >/dev/null 2>&1!in_browser
wait 2
.s_settaskvar
setup_measurement enable start_address $TASK_EVENT
setup_measurement disable end_address $TASK_EVENT
#
profile interval_duration
display histogram data calls
wait 2
display histogram sort_events calls
display histogram rescale current_max
```

The "get_task_number" script creates the ".s_settaskvar" command file that sets the TASK_EVENT environment variable to "Task_<4digitID> ". For a detailed description of the "get_task_number" script, refer to the "Demo RTOS Measurement Tool Details" chapter.

Chapter 7: Automating RTOS Measurements Using Command Files

The Emulator/Analyzer command file to track two service calls looks like:

```
PARMS &SERV_CALL1 &SERV_CALL2
#
# HP RTOS Command File: e_twocalls_16
#
# DESCRIPTION: Track only two specific service calls
#
!create_12_call &SERV_CALL1 &SERV_CALL2 >/dev/null 2>&1!in_browser
wait 2
.e_onesrvcall
#
display trace real_time_os
```

The "create_12_call" script creates a command file to track 1 or 2 service calls. The names of the service calls to be tracked are specified as parameters. This script creates the ".e_onesrvcall" command file which contains a trace command to track the specified service call. For a detailed description of the "create_12_call" and "create_12_call32" scripts, refer to the "Demo RTOS Measurement Tool Details" chapter.

To define command file search directories

- Add the directories that contain command files to the HP64KPATH environment variable.

When you run a command file, the Emulator/Analyzer and SPA interfaces first search for the command file in the current directory. If the command file is not found, the interfaces then search in the directories defined by the HP64KPATH environment variable.

Examples

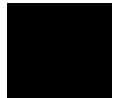
If your command file is placed in the `$HOME/rtoscmdf` directory, you should set the `HP64KPATH` environment variable as follows:

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>
```



Using Action Keys

You can make common RTOS measurements easy to set up and use by placing them on action keys. This allows a user to set up a measurement by simply pointing and clicking on an action key.

Action keys can run command files which can prompt for required parameters. (In the graphical interface, these prompts appear as dialog boxes in which a user can either type or cut-and-paste the required parameters.)

This section shows you how to:

- Place your measurements on action keys.
- Modify interface startup scripts.

To place your measurements on action keys

- Modify the "emul.< processor_type> *actionKeysSub.keyDefs" X resource to set up action keys for the Emulator/Analyzer interface.

Or:

- Modify the "spa.< processor_type> *actionKeysSub.keyDefs" X resource to set up action keys for the SPA interface.

Actions keys are only available when using the Graphical User Interface. When you are using the Softkey Interface from a terminal or terminal emulation window, you can only run command files by typing them in on the command line.

The easiest way to set X resources is to use the **-xrm** option in the **emul700** command that starts the interface. Because there may be many action keys to be defined, it's best to place the **emul700** command and its options in a startup script.

The startup scripts for the Emulator/Analyzer interface (when used with the demo RTOS measurement tool) is named "rtos_emul", and the startup script

for the Software Performance Analyzer is named "rtos_spa". These startup scripts are located in the \$HP64000/bin directory.

Examples

To place your measurements on action keys in the Emulator/Analyzer interface, modify the "emul.< processor_type> *actionKeysSub.keyDefs" X resource in the Emulator/Analyzer startup script:

```
#
# Bring up an emulator window
#
emul700 \
  $COMMAND_INPUT \
  "-xrm emul.$PROC_RESOURCE*actionKeys.packing:    PACK_COLUMN" \
  "-xrm emul.$PROC_RESOURCE*actionKeys.numColumns: 4" \
  "-xrm emul.$PROC_RESOURCE*actionKeysSub.keyDefs: \
    \"Track OS calls\"          \"action_keys/e_trkcalls\" \
    \"Track Everything\"        \"action_keys/e_trkall\" \
    \"Task switch A->B\"        \"action_keys/e_AthenB$ACTION_KEY_SFX\" \
    \"Help RTOS\"              \"!tellrtosHP !in_browser\" \
    \"Memory Usage\"            \"action_keys/e_memory$ACTION_KEY_SFX\" \
    \"Only Call X\"             \"action_keys/e_onecall\" \
    \"Only Calls X & Y\"        \"action_keys/e_twocalls$ACTION_KEY_SFX\" \
    \"Tsk A msg->Que X\"        \"action_keys/e_tsk2queue$ACTION_KEY_SFX\" \
    \"Custom OS Trace\"        \"display trace real_time_os\" \
    \"<UserDefinable1>\"        \"#Edit 'rtos_emul' to define this key\" \
    \"Only Task X\"             \"action_keys/e_trk1task$ACTION_KEY_SFX\" \
    \"Only Tsk W,X,Y,Z\"        \"action_keys/e_trk4task$ACTION_KEY_SFX\" \
    \"Task A: VarX\"           \"action_keys/e_aftervar$ACTION_KEY_SFX\" \
    \"NonCustom Trace\"        \"display trace mnemonic\" \
    \"<UserDefinable2>\"        \"#Edit 'rtos_emul' to define this key\" \
    \"Tasks & Queues\"          \"action_keys/e_trackqs$ACTION_KEY_SFX\" \
    \"Only Queues\"            \"action_keys/e_onlyqs\" \
    \"Task A: FuncX\"          \"action_keys/e_afterfunc$ACTION_KEY_SFX\" \
    \"Before SPA trig2\"       \"action_keys/e_spatrig\" \
  \" \
  $EMULATOR &
```

The PROC_RESOURCE environment variable is defined as a processor name such as "m68302", "i80960", etc.

Chapter 7: Automating RTOS Measurements Using Action Keys

To place your measurements on action keys in the Software Performance Analyzer interface, modify the "spa.< processor_type> *actionKeysSub.keyDefs" X resource in the Software Performance Analyzer startup script:

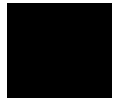
```
#
# Bring up window for SPA if requested
#
emul700 \
-u xperf \
$COMMAND_INPUT \
-xrm 'HP64_Softkey.geometry: +415+230' \
-xrm 'HP64_Softkey*enableCmdline: False' \
-xrm 'HP64_Softkey*actionKeys.packing: PACK_COLUMN' \
-xrm 'HP64_Softkey*actionKeys.numColumns: 2' \
-xrm "perf*actionKeysSub.keyDefs: \
    \"Initialize\"          \"action_keys/s_init$RTOS_UNIQUE\" \
    \"Time Tasks\"          \"action_keys/s_timetasks\" \
    \"Count Srvc Calls\"    \"action_keys/s_countsrvcls\" \
    \"Trig2 on Overflow\"   \"action_keys/s_break_ovrflw\" \
    \"FunctionDuration\"    \"action_keys/s_funcdur\" \
    \"TaskX: Servcalls\"    \"action_keys/s_taskwindow\" \
    \"Count Tasks\"        \"action_keys/s_counttasks\" \
    \"Tsk & User Evnts\"    \"action_keys/s_taskuser\" \
    \"Disable Trig2\"      \"action_keys/s_disabletrg2\" \
\" \
$EMULATOR &
```

To modify interface startup scripts

- 1 Copy the demo RTOS measurement tool startup scripts from the \$HP64000/bin directory.
- 2 Make your modifications to the startup scripts.
- 3 Run the "install_rtos" script.

The startup scripts for the Emulator/Analyzer interface (when used with the demo RTOS measurement tool) is named "rtos_emul", and the startup script for the Software Performance Analyzer is named "rtos_spa". These startup scripts are located in the \$HP64000/bin directory. The startup scripts are described in more detail in the "How the Demo RTOS Measurement Tool Works" chapter.

After you copy and modify these startup scripts, run the "install_rtos" script, and answer its questions as described in the "Installing New Custom OS Product Files" chapter.





8



**Installing New Custom OS Product
Files**

Installing New Custom OS Product Files

After you have modified or created the files needed for your own custom RTOS product, some of the files need to be installed in certain directories.

A script "install_rtos" has been provided to help you with this procedure. The actions accomplished by the script will be to customize the pulldowns in an emulation session so it will run your version of "rtos_emul" or "rtos_spa" and install your versions of the inverse assembly files into the emulation directories.

The script will ask you for an acronym, to be used to describe your OS, which will then be seen in the pull-down menu when you want to bring up an RTOS emulation session. You may also choose the emulators with which your custom product will be available. This script should be run as "root".

This chapter shows you how to:

- Answer install_rtos questions.
- Reinstall the original HP Custom RTOS product.

To answer install_rtos questions

- 1 Enter a 1-8 letter acronym to be used to identify your RTOS:

Answer this question with a word or acronym which will be seen in the emulation session's pulldown menu to describe the choice of bringing up an RTOS emulation session. The context will be either "RTOS tool for <acronym> " or "RTOS SPA for <acronym> ".

The script will then create and install the X resource file which will change the original pull-down choice.

- 2 You must have built your own inverse assembler relocatables and customized your own "rtos_emul" and "rtos_spa" files (which also means creating your own command files) to continue with this script. Continue? (y/n) [y]:

If you have not edited all of the files listed and "assembled" the inverse assembler files, stop here and read the chapters that explain how to customize the files for your own OS kernel.

The script will then copy two interface scripts that need to be modified to call your "rtos_emul" and "rtos_spa" scripts.

- 3 Enter the complete path name of your modified "rtos_emul" script. Name (default = "/usr/hp64000/bin/rtos_emul"):

Enter the full path name of where your modified "rtos_emul" script resides.

- 4 Enter the complete path name of your modified "rtos_spa" script. Name (default = "/usr/hp64000/bin/rtos_spa"):

Enter the full path name of where your modified "rtos_spa" script resides.

The script will then edit the interface scripts and keep them in the current directory.

Chapter 8: Installing New Custom OS Product Files
To reinstall the original HP Custom RTOS product

- 5 Enter the directory where your inverse assembler (IA) files (*.R) reside. (default = "/usr/hp64000/rtos/B3082A/interpreter") Directory:
- 6 Enter name of inverse assembler relocatable file you created for 16-bit processors. Name (default = "rtos_16.R")
- 7 Enter name of inverse assembler relocatable file you created for 32-bit processors. Name (default = "rtos_16.R")

The previous three questions request the full path name of the two inverse assembler files that you have created for use by 16 and 32 bit processors. If you have developed only one version, enter the same name for both of the latter two questions.

- 8 Question 8 is a lengthy statement that basically asks under which emulator you want your custom RTOS version to be installed. You may install it under all installed emulators, or under any subset. The list of emulators given with the question are the ones which have interfaces which support the RTOS tool.

The script will then attempt to install your RTOS version for each chosen emulator. It is required to install both an interface script and an inverse assembler under each emulator's install directory to get full functionality. If any failures occur during installation, it is most likely due to bad permissions or the script not being run with full system privileges.

If the script fails to complete any of its tasks, it will halt at the point of failure. The script must be rerun to complete the installation.

To reinstall the original HP Custom RTOS product

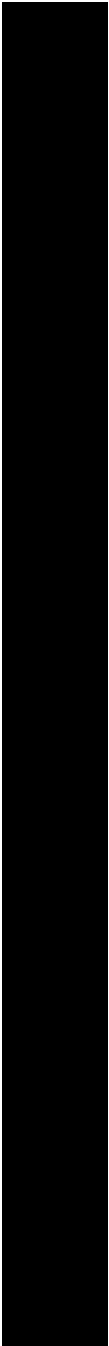
- To reinstall the original product so the "demo" RTOS product is available, either reinstall the product tape or run the script "reinstall_rtos".

Part 3

Reference

Descriptions of the product in a dictionary or encyclopedia format.

Part 3





**Inverse Assembler Language (IAL)
Instructions**

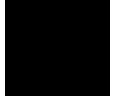
Inverse Assembler Language (IAL) Instructions

Instruction Set Summary

Executable Instructions

Arithmetic	Operand	Description
ADD	variable/immediate	accumulator= accumulator+ operand.
AND	variable/immediate	accumulator= accumulator AND operand.
DECREMENT	variable	variable= variable - 1.
EXCLUSIVE_OR	variable/immediate	accumulator= accumulator XOR operand.
INCLUSIVE_OR	variable/immediate	accumulator= accumulator OR operand.
INCREMENT	variable	variable= variable + 1.
LOAD	variable/immediate	accumulator= operand.
SET	variable/immediate	variable= immediate (-8,+ 7).
STORE	variable	variable= accumulator.
SUBTRACT	variable/immediate	accumulator= accumulator - operand.
Accumulator	Operand	Description
COMPLEMENT		One's complement accumulator.
EXTRACT_BIT	immediate	Bit number to extract (0= LSB, 31= MSB).
ROTATE	RIGHT/LEFT,immediate	Rotate accumulator right/left,bitcount.
TWOS_COMPLE- MENT		Two's complement accumulator.

Conditional	Operand	Description
CASE_END CASE_OF IF	variable / MSB,LSB variable rel variable/immediate THEN result MSB,LSB rel variable/immediate THEN result Where "rel" is = equal < > not equal < = less or equal > = greater or equal < less than > greater than	End of case statement. Variable or bit range in accumulator.
Program Control	Operand	Description
ABORT CALL GOTO RETURN	 label label	Leave inverse assembler. Branch to label stack return address. Branch to label. Return from CALL, or leave inverse assembler.
Output Buffer	Operand	Description
OUTPUT POSITION NEW_LINE FETCH_POSITION	variable/"string"/ACCU- MULATOR,FORMAT ABS/REL,immediate	Write string or numeric data. Absolute or relative column number. Begin new output line. Get column number.



Miscellaneous	Operand	Description
IF_NOT_MAPPED	THEN result	Symbolic reference.
INPUT	REL/ABS,variable [,QUALIFIED]	Read analysis data.
NOP		No operation.
TAG_WITH	variable/immediate	Tags state for future reference.
MARK_STATE	SUPPRESSED/DISPLAYED	Suppress/display analysis state.

Pseudo Instructions

Symbolic Operand Definitions	Operand	Description
ASCII/ASC	"string"	Defines string for OUTPUT instruction.
CONSTANT/CONST	immediate	Defines a commonly used constant.
FORMAT	a,b,c[,DISPLAY_BASE]	Defines format for OUTPUT instruction.
VARIABLE/VAR	immediate	Defines and initializes a variable.
Display Width Initialization	Operand	Description
DEFAULT_WIDTH	immediate	Defines max display width for RTOS column in Emul./Analyzer trace display.
MAPPED_WIDTH	immediate	Display width for symbols in RTOS column of Emul./Analyzer trace display.
Inverse Assembler Titles	Operand	Description
LABEL_TITLE	"string"	Label title line.
BASE_TITLE	"string"	Base title line.

Defining Initial Variables for INPUT Instruction	Operand	Description
QUALIFY_MASK	immediate	Mask and value used for qualified INPUT instructions.
QUALIFY_VALUE	immediate	Mask and value used for qualified INPUT instructions.
SEARCH_LIMIT	immediate	Defines maximum search limit.
Debug and Performance Aid	Operand	Description
MAX_INSTRUCTION	immediate	Limits number of instructions executed.

Predefined Communication Variables

Trace states and other related information are passed to the inverse assembler through communication variables. The RTOS inverse assembler predefines the following communication variables.

INPUT_ADDRESS Contains the address value of the particular trace state.

INPUT_DATA Contains the data value of the particular trace state.

INPUT_TAG Contains software tag information about whether the particular trace state has already been used.

INITIAL_OPTIONS Contains the address of the symbol **HP_RTOS_TRACK_START**; this is the base address of the data table.

INITIAL_FLAGS When bit 1 is set, the base value of the data table has been passed in to the inverse assembler through the **INITIAL_OPTIONS** variable. Other bits in the **INITIAL_FLAGS** communication variable are not used.

Instruction Descriptions

This chapter defines the syntax and explains the function of each executable and pseudo instruction. It also illustrates how the instructions are used with one or more examples. For quick reference, the instructions are arranged alphabetically.

Note

The following symbols are predefined (reserved) and will generate a "duplicate error":

VMS, HP64000, HPUX

Note

Labels appearing on lines without instructions are assumed to be code labels. Labels for ASCII, CONSTANT, FORMAT, and VARIABLE must be on the same line as the pseudo to be accepted as a label for the pseudo instruction.

ABORT - Leave inverse assembler

Syntax

Label	Operation	Operand	Comment
	ABORT		

Abort will pass control back to the system software even if the program is currently in a subroutine. This instruction is intended to be an error escape from the inverse assembler.

Examples

```
ABORT ;Return to Interpreter.
```



ADD - Add to accumulator

Syntax

Label	Operation	Operand	Comment
	ADD	variable\immediate	

accumulator = accumulator + operand

The contents of the operand field are added to the value in the accumulator. The operand can be either the name of a variable or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFH (32 bit value).

Examples

ADD	1	;Increment accumulator.
ADD	NAME	;Add contents of variable ;to accumulator.

AND - Logical AND with accumulator

Syntax

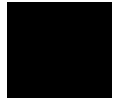
Label	Operation	Operand	Comment
	AND	variable/immediate	

accumulator = accumulator AND operand

Performs a logical "and" of the value of the operand and the value in the accumulator. The operand can be either the name of a variable or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32 bit value).

Examples

AND	1	;AND lower bit.
AND	MASK	;AND with MASK variable.



ASCII/ASC (Pseudo) - Define ASCII string

Syntax

Label	Operation	Operand	Comment
Label	ASCII/ASC	"string"	

The ASCII pseudo instruction is used to define an ASCII string to be used with an OUTPUT instruction. This is recommended for strings that are used more than once to minimize the size of the inverse assembler source code.

Examples

CALL_STRING	ASCII	"-> "	;Define text for ;OUTPUT instruction.
RETURN_STRING	ASC	"<- "	

CALL - Transfer program control to label w/RETURN

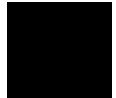
Syntax

Label	Operation	Operand	Comment
	CALL	LABEL	

Program control will be transferred to the label specified. The value of the next pc is pushed on a stack. A RETURN instruction will transfer control to the statement following the CALL. The maximum subroutine nest level is 16.

Examples

CALL	SUBROUTIN	;Control is ;transferred to ;SUBROUTIN.
------	-----------	---



CASE_OF - Conditional testing of variable or accumulator

Syntax

Label	Operation	Operand	Comment
	CASE_OF	variable\MSB,LSB	

The case statement allows conditional testing of either a user variable or the accumulator. Program control will branch to one of the instructions following the case depending on the value of the variable or accumulator.

If the operand is the name of a variable, the value of the variable becomes an index added to the current program counter to fetch the next instruction. Otherwise, the bit range specified is the index.

The case statement must be followed by a "CASE_END" statement that will determine the length of the CASE. If the value of the operand falls into the range of the CASE, the corresponding instruction will be executed. Otherwise, the statement following the case will be executed.

Any instruction may appear in the CASE except an IF, IF_NOT_MAPPED, or another CASE. If a CALL is executed, then the return from subroutine will execute the instruction following the CASE_END. Remaining instructions will be executed in sequence with normal exceptions (for example, GOTO or RETURN).

Examples

```

that          CASE_OF          15,0          ;"Maximum message length
                                           ;fit display"
Never reached NOP                ;0
              OUTPUT           "No queue members left to allocate." ;1
              OUTPUT           "Region request too large."           ;2
              OUTPUT           "No more memory available"            ;3
              OUTPUT           "Invalid region ID"                   ;4
              CASE_END          ;Execute next instruction
                                           ;for all other values
                                           ;of bits 15 thru 0.
                                           ;Execution
              <next instruction> ;will also take place
                                           ;after either statement
                                           ;in the body of CASE
                                           ;is executed.

```


Chapter 9: Inverse Assembler Language (IAL) Instructions
CASE_OF - Conditional testing of variable or accumulator

```
CASE_OF          NAME          ;Test contents of
                  ;variable NAME.
OUTPUT           AREG          ;Execute this if NAME=0.
CALL             SUB           ;Execute this if NAME=1.
CASE_END        ;Execute next instruction
<next instruction> ;if NAME<>0,1. Execution
<next instruction> ;will also take place
                  ;after either statement
                  ;in the body of CASE
                  ;is executed.
```



COMPLEMENT - One's complement on accumulator

Syntax

Label	Operation	Operand	Comment
	COMPLEMENT		

A one's complement is performed on the contents of the accumulator. Bits that are 1 change to 0 and those that are 0 change to 1.

Examples

COMPLEMENT	;One's complement ;on accumulator.
------------	---------------------------------------

CONSTANT/CONST (Pseudo) - Define constant

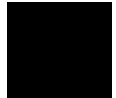
Syntax

Label	Operation	Operand	Comment
NAME	CONST	constant	

Allows commonly used constants to be defined and referenced by the label specified. Normally, immediate constants are handled automatically by the assembler, but the assembler will not optimize the use of constants that are identical. To avoid wasting data space, commonly used constants can be defined and referenced by symbolic names. All constants are 32-bit quantities.

Examples

ML16	CONST	0FFFFH	;Mask lower 16-bit ;constant.
------	-------	--------	----------------------------------



DECREMENT - Decrement variable

Syntax

Label	Operation	Operand	Comment
	DECREMENT	variable	

variable = variable - 1

Decrements the variable specified by the operand. The operand must be defined using the VARIABLE pseudo instruction.

To decrement the accumulator, use the "SUBTRACT 1" instruction.

Examples

DECREMENT	SAM	;Decrement the ;variable SAM.
-----------	-----	----------------------------------

DEFAULT_WIDTH (Pseudo) - Default width of display field

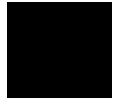
Syntax

Label	Operation	Operand	Comment
	DEFAULT_WIDTH	immediate	

Defines the maximum default width of the display field. Operands can range from 1 to 64. If none is specified, the default width is 32.

Examples

```
DEFAULT_WIDTH 40 ;The default  
                ;width is 40.
```



EXCLUSIVE_OR - Exclusive OR with accumulator

Syntax

Label	Operation	Operand	Comment
	EXCLUSIVE_OR	variable/immediate	

accumulator = accumulator XOR operand

Performs a logical "exclusive or" of the operand value and the accumulator value. The operand can be either the name of a variable or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32-bit value).

Examples

EXCLUSIVE_OR	1	;Toggle lower bit
EXCLUSIVE_OR	MASK	;Or variable MASK.

EXTRACT_BIT - Extract from accumulator

Syntax

Label	Operation	Operand	Comment
	EXTRACT_BIT	immediate	

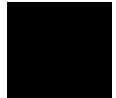
The operand is a bit number in the accumulator, with a range from 0 to 31. This bit will be extracted and the value of the accumulator set to its value (either 0 or 1).

Examples

```
EXTRACT_BIT    10           ;Set accumulator
                    ;to value of bit
                    ;10 in accumulator.
```

See Also

The CASE_OF and IF conditional instructions allow you to test bit ranges in the accumulator.



FETCH_POSITION - Get column number

Syntax

Label	Operation	Operand	Comment
	FETCH_POSITION		

The FETCH_POSITION instruction sets the accumulator to the column number the display output buffer where the next OUTPUT instruction will write characters. The instruction is particularly useful in determining if an output string will fit on the current output line.

Examples

```
                OUTPUT      STRING1      ; output a string
                FETCH_POSITION          ; determine next
                                        ; column # to write
;
; The accumulator now contains the next column number to
; to. If the second string to be output is 25 characters
; adding 25-1 to the accumulator will result in the
; containing the column number of the last character in the
; string if STRING2 is output immediately after STRING1.
;
                ADD          24           ; calculate column
                                        ; number of last
;
; Now see if the last column number to be used is greater
; maximum line length. If it is, issue a NEW_LINE command
; to outputting the second string.
;
                IF          31,0 > 64 THEN NEW_LINE ; start a new
                                                ; line if
                                                ; necessary
                OUTPUT      STRING2      ; output the
                                                ; second string
```

FORMAT (Pseudo) - Format accumulator

Syntax

Label	Operation	Operand	Comment
NAME	FORMAT	a,b,c[,DISPLAY_BASE]	

Format is used to define how the accumulator should be converted when used in conjunction with the OUTPUT instruction.

Operand "a" defines how many bits of the accumulator will be converted. It can range from 1 to 32; if a subrange is specified, then the most significant bits will be "ANDed" with 0.

Operand "b" specifies the base of the conversion. It can be BIN, OCT, DEC or HEX.

Operand "c" specifies the number of characters to be displayed by the OUTPUT instruction. Leading zeros will be supplied if the number is smaller than the converted field. A left justified, zero-suppressed number can be generated by specifying c = LEFT_JUSTIFIED, which will display as many digits as necessary.

The optional operand DISPLAY_BASE can be used to append a B, O, D, or H on the end of the constant converted, depending on the numeric base of the number.

The accumulator can be displayed as one ASCII character by using the keyword ASCII instead of the number-of-bits operand.

Examples

HEX_FMT	FORMAT	32,HEX,8	;Eight digit ;hex format.
HEX_FMT_LEFT	FORMAT	32,HEX,LEFT_JUSTIFIED	;Left-justified ;hex.
HEX_FMT16	FORMAT	16,HEX,4,DISPLAY_BASE	
DEC_FMT	FORMAT	32,DEC,LEFT_JUSTIFIED	
DEC_FMT_TIME	FORMAT	8,DEC,2	
DEC_FMT_YEAR	FORMAT	16,DEC,4	
BIN_FMT_LEFT	FORMAT	32,BIN,LEFT_JUSTIFIED	
BIN_FMT_4	FORMAT	4,BIN,4	
ASC_FMT	FORMAT	ASCII	;ASCII format.

GOTO - Transfer program control, no RETURN

Syntax

Label	Operation	Operand	Comment
	GOTO	LABEL	

Program control will be transferred to the label specified.

Examples

GOTO	END_RTOS_IAL	;Branch to ;end of routine.
------	--------------	--------------------------------

IF - Compare operands

Syntax

Label	Operation	Operand	Comment
	IF	variable rel_op variable/ immediate THEN result	
	or		
	IF	MSB,LSB rel_op variable/ immediate THEN result	
		Where "rel_op" is	
		= equal	
		<> not equal	
		<= less or equal	
		>= greater or equal	
		< less than	
		> greater than	

Allows operands to be compared and decisions made based on the results of the comparison. The first form of the IF instruction allows the contents of a variable to be compared to the contents of other variables or to immediate data. Immediate data can range from 0 to 0FFFFFFFH (32 bits). The operand to the THEN part can be any instruction except another IF, a CASE, or IF_NOT_MAPPED.

The second form of the IF instruction allows a bit range of the accumulator to be tested. Here the first operand specifies the most significant bit (MSB) and the second operand specifies the least significant bit (LSB). This allows all or part of the accumulator to be tested against an immediate value or the contents of a variable.

Examples

```

IF      NAME < 101 THEN      ;Test value of
      GOTO LABEL            ;NAME.

IF      6,3 = 1001B THEN    ;Test accumulator
      CALL SUBROUTIN        ;bit range 6-3
                              ;inclusive.

```

IF_NOT_MAPPED - Check for symbol in default map

Syntax

Label	Operation	Operand	Comment
	IF_NOT_MAPPED THEN	result	

Symbols are loaded into the Emulator/Analyzer interface as part of the OS program absolute file (provided the appropriate compiler/linker options to place symbolic "debug" information in the absolute file are used).

Symbols can be displayed in the real-time OS trace instead of absolute addresses. The IF_NOT_MAPPED instruction is used to display symbolic information if it is available.

```
IF_NOT_MAPPED THEN OUTPUT ACCUMULATOR,FORMAT
```

When this function is executed, the following three conditions are possible.

- 1 If the address in the ACCUMULATOR matches a symbol defined as a single valued address, that is, not included in a range, the symbol associated with the address is displayed. The "result" part of the instruction, in this case the "OUTPUT" instruction, is not executed. This means the value passed corresponds exactly with a particular symbol. For example:

```
SUBROUTIN
```

- 2 If the address is not found as a single valued address, but is part of a range, the symbol associated with the range will be displayed. The ACCUMULATOR will be set to the offset from the beginning or end of the range, depending on how the range was specified. The "result" part of the instruction will be executed to display this offset. For example:

```
SUBROUTIN+023H
```

Chapter 9: Inverse Assembler Language (IAL) Instructions
IF_NOT_MAPPED - Check for symbol in default map

- 3 If the address in the ACCUMULATOR is not found as a single valued address or as part of a range, that is, not in the symbol map, no symbolic information will be displayed. Here, the value in the accumulator will contain the absolute address and will be displayed since the "result" part of the function will be executed. For example:

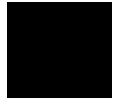
FFFFH

The RETURN_FLAGS communication variable has two flags in the upper 16 bits. These flags indicate the result of the IF_NOT_MAPPED instruction. They are interpreted as follows:

Bit 16: 0 = no mapping was done
 1 = mapping was successful

*Bit 17: 0 = mapped to a range
 1 = mapped to a single value symbol

* Bit 17 is valid only if bit 16 = 1.



INCLUSIVE_OR - Logical OR with accumulator

Syntax

Label	Operation	Operand	Comment
	INCLUSIVE_OR	variable/immediate	

accumulator = accumulator OR operand

Performs a "logical or" of the operand value and the accumulator value. The operand can be either be the name of a variable or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32-bit value).

Examples

INCLUSIVE_OR	1	;Set bit 1.
INCLUSIVE_OR	MASK	;Or with MASK.

INCREMENT - Increment variable

Syntax

Label	Operation	Operand	Comment
	INCREMENT	variable	

variable = variable + 1

This instruction increments the variable specified by the operand. The operand must be defined by the VARIABLE pseudo.

To increment the accumulator, use the "ADD 1" instruction.

Examples

INCREMENT	NAME	;Increment ;variable NAME.
-----------	------	-------------------------------



INPUT - Input data

Syntax

Label	Operation	Operand	Comment
	INPUT	ABS/REL, operand[, QUALIFIED]	

INPUT allows data to be read in order to complete a multibyte instruction or display the results of an executed instruction. Data can be read relative to the initial state in the analysis trace, or read from an absolute location.

If the QUALIFIED option is specified, data reads will be qualified depending on the values in QUALIFY_VALUE and QUALIFY_MASK, which must be initialized before the INPUT instruction is executed. (These pseudo instructions are explained separately in this chapter). Only one byte or one state is read each time the INPUT instruction is executed.

REL,operand

This option reads data relative to the initial state in the analysis trace. The operand can be either an immediate value or a user defined variable. The value is the number of states in analysis to skip over before reading the data. For example, INPUT REL,2 would mean to skip over the state following the initial position and read the data at the second state past this initial position. The value can be either positive (forward) or negative (backward).

The SEARCH_LIMIT variable limits the number of states to be searched by the INPUT routine.

If QUALIFIED is specified, the search count applies to the number of states that are satisfied by the status qualification check.

ABS,operand

This option will search the trace buffer for an absolute address specified by the operand and return the data at that address. The operand must be a user defined variable or communications variable. The status can be qualified if desired. The search is limited by the value in SEARCH_LIMIT.

Chapter 9: Inverse Assembler Language (IAL) Instructions
INPUT - Input data

Examples

```
INPUT      ABS,DATA_ADDRESS,QUALIFIED ;Qualified on
           ;status.
INPUT      ABS,DATA_ADDRESS           ;Not qualified.
INPUT      REL,2                       ;Get two states
           ;ahead.
INPUT      REL,COUNT                   ;Relative with
           ;count.
```



LOAD - Load accumulator

Syntax

Label	Operation	Operand	Comment
	LOAD	variable/immediate	

accumulator = operand

Loads the accumulator with the value specified by the operand field. The operand can be either the name of a variable or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32-bit value).

Examples

	LOAD	1	;Set accumulator ;to 1.
	LOAD	NAME	;Load value of SAM.

MAPPED_WIDTH (Pseudo) - Define maximum width of display

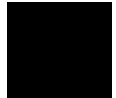
Syntax

Label	Operation	Operand	Comment
	MAPPED_WIDTH	immediate	

Defines the maximum width of the display in the mapped mode display. Operands can range from 1 to 64. If none is specified, the default width is 32.

Examples

```
MAPPED_WIDTH 40 ;Max display  
;width = 40.
```



MARK_STATE - Analysis state display control

Syntax

Label	Operation	Operand	Comment
	MARK_STATE	SUPPRESSED or DISPLAYED	

The MARK_STATE instruction is used to indicate to the state analyzer whether or not to display the current analysis state. If a MARK_STATE SUPPRESSED instruction is executed, the analysis state corresponding to the current inverse assembler call will be omitted from the trace list. MARK_STATE DISPLAYED causes the analysis state to be included in the trace list.

The last MARK_STATE to occur prior to exiting the inverse assembler is the only one in force for a particular inverse assembler call. If no MARK_STATE instruction is executed, MARK_STATE DISPLAYED is assumed.

MAX_INSTRUCTION (Pseudo) - Limit instruction execution

Syntax

Label	Operation	Operand	Comment
	MAX_INSTRUCTION	immediate	

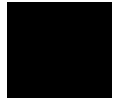
Since the programmer has the ability to control program flow, it is possible to program an infinite loop that will never return to the calling program. To avoid this problem, a maximum number of instructions variable is used to limit the number of instructions that can be executed each time the inverse assembler is called. This number is initialized to a large number and should never interfere with the inverse assembler; however, this number can be used to set a low limit on the instruction limit to see which calls to the inverse assembler take the most time. This can be used to optimize sections or stop near-infinite loops. The value of MAX_INSTRUCTION, initialized by this pseudo, cannot be changed during the inverse assembly. If the instruction count exceeds this value, the inverse assembler is aborted. In addition, an instruction overflow message is placed in the output buffer and displayed.

Default = 10000

Examples

```
MAX_INSTRUCTION 50          ;Check for execution
                             ;of more than 50
                             ;instructions.

MAX_INSTRUCTION 10000      ;This sets a large
                             ;limit so only infinite
                             ;loops will abort
                             ;the inverse assembler.
```



NEW_LINE - Begin generating a new output line

Syntax

Label	Operation	Operand	Comment
	NEW_LINE		

The NEW_LINE instruction is used when more than one line of information to be output by the inverse assembler for a single captured analysis. Following execution of the NEW_LINE instruction, subsequent OUTPUT and instructions refer to the new line of inverse assembler output. The instruction can be used to generate up to four inverse assembler output lines. Exceeding this limit will cause the inverse assembler to abort.

Examples

```
OUTPUT      "This is line 1"  
NEW_LINE  
OUTPUT      "**** This is line 2"
```

This series of instructions will produce output as shown below:

```
This is line 1  
**** This is line 2
```

NOP - No operation

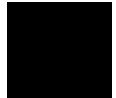
Syntax

Label	Operation	Operand	Comment
	NOP		

Has no effect on the execution of the inverse assembler. The instruction following NOP will be executed next.

Examples

NOP



OUTPUT - Output to output buffer

Syntax

Label	Operation	Operand	Comment
	OUTPUT	variable/"string"/ACCUMULATOR	FORMAT

The OUTPUT instruction expects an operand defined by the STRING pseudo, an immediate string, or the key work ACCUMULATOR followed by a conversion format defined by the FORMAT pseudo. The first two operands will copy ASCII text to the output buffer. The third operand will convert the accumulator using the specified format to a numeric display in the output buffer.

Examples

LOAD_STG	ASCII	"LOAD"	
	OUTPUT	LOAD_STG	;Output LOAD text.
	OUTPUT	"LOAD"	;Output immediate
			;text.
	OUTPUT	ACCUMULATOR,HEX_FMT	;Convert accumulator
			;to hex.

POSITION - Position column pointer

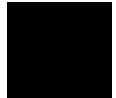
Syntax

Label	Operation	Operand	Comment
	POSITION	ABS/REL, column number	

Position allows the current column pointer to be moved to an absolute or relative position in the output buffer. The column number can range from 1 to 64 for absolute positioning or -32 to 31 for relative positioning. In the relative mode, negative numbers move the column position to the left of the current location and positive numbers move it to the right.

Examples

POSITION	ABS,10	;Move to column ;10.
POSITION	REL,-2	;Move to the left ;2 columns.



QUALIFY_MASK/_VALUE (Pseudos) - Set qualify specifications

Syntax

Label	Operation	Operand	Comment
	QUALIFY_MASK	immediate	
	or		
	QUALIFY_VALUE	immediate	

QUALIFY_MASK and QUALIFY_VALUE are used to set qualify specifications for the INPUT instruction. When INPUT ABS,operand,QUALIFIED or INPUT REL,operand,QUALIFIED is executed, both the address and status must be satisfied before data is returned. The mask operand is considered to be a 32-bit mask where a 0 represents a "don't care" state and a 1 represents a "care" state. The status in the analysis buffer is first masked (ANDed) with the value of QUALIFIED_MASK to obtain the value of "care" bits. Then this value is compared to QUALIFY_VALUE to see if the status is satisfied.

Examples

QUALIFY_MASK	00101B	;Only care about ;bits 0 and 2
QUALIFY_VALUE	00001B	;And the value must ;be 001B.

RETURN - Return

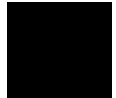
Syntax

Label	Operation	Operand	Comment
	RETURN		

Return can be used to return to the instruction following a `CALL` or to leave the inverse assembler if a `RETURN` is executed without any subroutine nesting.

Examples

```
RETURN                                ;Return to calling  
                                       ;routine or leave  
                                       ;inverse assembler if  
                                       ;not in a subroutine.
```



ROTATE - Rotate accumulator contents

Syntax

Label	Operation	Operand	Comment
	ROTATE	RIGHT,immediate or LEFT,immediate	

This instruction rotates the accumulator contents either right or left the number of bits specified. The operand can range from 1 to 32. Bits that are shifted off the left side (on left shifts) are rotated back on on the right side, and vice versa (circular shift).

Examples

ROTATE	LEFT,10	;Shift left 10 ;bits and rotate ;in on right.
ROTATE	RIGHT,20	;Shift right 20 ;bits and rotate ;in on left.

SEARCH_LIMIT (Pseudo) - Limit analysis search

Syntax

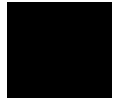
Label	Operation	Operand	Comment
	SEARCH_LIMIT	immediate	

The search limit applies to the INPUT instruction and reading data from the analysis buffer after a trace. The operand specifies how many analysis states should be searched in order to find the required data. The limit optimizes processing by not allowing the entire buffer to be searched each time. The search limit should be set to the maximum number of memory or I/O references made between opcode fetches. For example, if the inverse assembler was searching for a memory read state and that state was not captured by the analysis hardware, SEARCH_LIMIT would be used to limit the number of states scanned. The variables QUALIFY_MASK and QUALIFY_VALUE may be used to qualify the search. Every time the condition is satisfied, the search count is incremented. QUALIFY_MASK and QUALIFY_VALUE can be changed to search for other conditions and SEARCH_LIMIT can be defined to reflect the number of states that are expected to be found.

Default = 1

Examples

```
SEARCH_LIMIT      7           ;Search limited to 7
                        ;analysis states.
```



SET - Set variable

Syntax

Label	Operation	Operand	Comment
	SET	variable,immediate	

variable = immediate (-8, + 7)

The variable specified is set to the value in the immediate operand. The operand can range from -8 to + 7.

Examples

	SET	NAME,2	;Set value of ;NAME to 2.
--	-----	--------	------------------------------

STORE - Store value in accumulator

Syntax

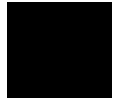
Label	Operation	Operand	Comment
	STORE	variable	

variable = accumulator

Stores the value in the accumulator in the location defined by the operand. The operand label must be defined by the VAR pseudo, or be one of the communication variables.

Examples

```
STORE NAME ;Store accumulator value  
;in variable NAME.
```



SUBTRACT - Subtract from accumulator

Syntax

Label	Operation	Operand	Comment
	SUBTRACT	variable/immediate	

accumulator = accumulator - operand

The value specified by the operand is subtracted from the value in the accumulator. The operand can be either the name of a variable or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32-bit value). Negative numbers are expressed in two's complement form.

Examples

SUBTRACT	1	;Decrement ;accumulator.
SUBTRACT	NAME	;Subtract value of ;NAME from ;accumulator.

TAG_WITH - Flag analysis states

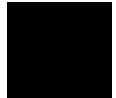
Syntax

Label	Operation	Operand	Comment
	TAG_WITH	operand	

The tag instruction provides a convenient method to "mark" states. This mark is then used during subsequent inverse assembly calls. Currently, the inverse assembler is called for each analysis state and if the current state is part of the previous instruction, a tag can be set to indicate that only status should be displayed. The tag operand is the value of the tag to be associated with the current state. For all systems, the tag has a 2-bit value and the nontagged value is 0. When the INPUT instruction is executed, the variable INPUT_TAG will be initialized with the tag value of the analysis state read.

Examples

```
TAG_WITH      TAG_VALUE      ;Tag current  
              ;analysis state.
```



TWOS_COMPLEMENT - Two's complement on accumulator

Syntax

Label	Operation	Operand	Comment
	TWOS_COMPLEMENT		

A two's complement is performed on the accumulator, changing all 1 bits to 0 and 0 bits to 1, then adding 1 to the result.

Examples

```
TWOS_COMPLEMENT ;Negate accumulator.
```

VARIABLE/VAR (Pseudo) - Define and initialize a variable

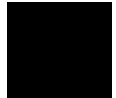
Syntax

Label	Operation	Operand	Comment
NAME	VARIABLE	immediate	

VARIABLE defines a storage location that can be used on arithmetic and conditional statements. It can be initialized to a specific value with the optional operand field.

Examples

NAME	VARIABLE	0FFH	;Define variable ;storage and assign ;an initial value ;of 0FFH.
------	----------	------	---





10



IAL Builder Error Messages

IAL Builder Error Messages

If errors occur when building an inverse assembler, a two-letter code is output along with a caret (^) character that indicates the location of the error. Also, if there was a previous error, the line number of that error is displayed. This chapter describes the possible error messages.

AS

ASCII STRING

Cause: The ASCII string was terminated improperly.

Action: Check that matching string delimiters (quotation marks, single quote marks, or carets) are used, and that the string is not too long.

DE

DEFINITION ERROR

Cause: Indicated symbol must be defined prior to it being referenced. Symbol may be defined later in the program sequence.

Action: Make sure the symbol is defined either before or after it is referenced.

DS

DUPLICATE SYMBOL

Cause: Indicates that the defined symbol noted has been previously defined in the program assembly sequence.

Action: Make sure symbols are defined once.

ET

EXPRESSION TYPE

Cause: The resulting type of expression is invalid. Absolute expression was expected and not found.

Action: Make sure the expression yields an absolute value; that is, the value must be known at assembly time.

IC

ILLEGAL CONSTANT

Cause: Indicates that the assembler encountered a constant that is not valid. For example: 109B (9 is invalid)

Action: Make sure that constants contain characters appropriate to the number base used.

IE ILLEGAL EXPRESSION

Cause: Specified expression is either incomplete or an invalid term was found within the expression.

Action: Make the expression complete or remove the invalid term.

IO INVALID OPERAND

Cause: Specified operand is either incomplete and inaccurately used for this operation. This occurs when an unexpected operand is encountered or the operand is missing. If the required operand is an expression, the error indicates that the first item in the operand field is illegal.

Action: Refer to the "Inverse Assembler Language Instructions" chapter for instruction syntax.

IS ILLEGAL SYMBOL

Cause: Syntax expected an identifier and encountered an illegal character or token.

Action: The first character of a label must be an upper case alphabetic character. The remaining characters may be either alphabetic or numeric. The alphanumeric character set includes the letters of the alphabet (upper or lower case), the underline symbol (), and numeric digits 0 through 9.

MO MISSING OPERATOR

Cause: An arithmetic operator was expected, but was not found.

Action: Refer to the "Inverse Assembler Language Instructions" chapter for instruction syntax.

MP MISMATCHED PARENTHESIS

Cause: Missing right or left parenthesis.

Action: Make sure that parentheses are matched.



SE

STACK ERROR

Cause: Indicates that a statement or expression does not conform to the required syntax.

Action: Refer to the "Inverse Assembler Language Instructions" chapter for instruction syntax.

TR

TEXT REPLACEMENT

Cause: Indicates that the specified text replacement string is invalid.

Action: The ampersand (&) character is not a valid character for labels. The first character of a label must be an upper case alphabetic character. The remaining characters may be either alphabetic or numeric. The alphanumeric character set includes the letters of the alphabet (upper or lower case), the underline symbol (_), and numeric digits 0 through 9.

UC

UNDEFINED CONDITIONAL

Cause: Conditional operation code is invalid.

Action: Refer to the "Inverse Assembler Language Instructions" chapter for instruction syntax.

UO

UNDEFINED OPERATION CODE

Cause: Operation code encountered is not defined or the assembler does not allow the operation to be processed in its current context.

Action: Make sure the operation code is spelled correctly, and that a valid delimiter (:) follows the label field.

US

UNDEFINED SYMBOL

Cause: The indicated symbol is not defined as a label.

Action: Make sure the symbol is defined as a label.

Part 4

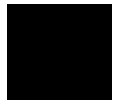
Concept Guide

Topics that explain concepts and apply them to advanced tasks.

Part 4



**Demo RTOS Measurement Tool
Details**



Demo RTOS Measurement Tool Details

The "Quick Start Guide" part of this manual describes how to use the demo RTOS measurement tool and how to perform one simple modification. The chapters in the "User's Guide" part of the manual describe parts of the demo RTOS measurement tool as examples of how to perform different types of tasks.

This chapter describes the following aspects of the demo RTOS measurement tool in more detail:

- The "rtos_edit" Script
This script is used when adding tasks to the demo application. It enters task names into the data table and the command file that sets up SPA intervals.
- The "rtos_emul" Startup Script
This script starts the Emulator/Analyzer interface, setting up action keys and command files for RTOS measurements.
- The "rtos_spa" Startup Script
This script starts the Software Performance Analyzer interface, setting up action keys and command files for RTOS measurements.
- Scripts Run by the Action Key Command Files
These scripts provide greater capability to several of the action key command files.

The "rtos_edit" Script

The "rtos_edit" script:

- Sets environment variables.
- Defines variables.
- Checks location of files.
- Displays install information.
- Prompts user for information and task IDs.
- Enters the task names into the "table.c" file (if requested).
- Enters the task names into the "s_init" file (if requested).

Setting Environment Variables

The paths needed by the "rtos_edit" script are added to the PATH environment variable. If the HP64000 environment variable is not already set, it is set to /usr/hp64000.

```
#!/bin/sh
#####
#
# FILE:          rtos_edit
#
# DESCRIPTION:   This is a shell script to edit the 'table.c' & 's_init'
#               files. New files will be created and will be placed in the
#               current directory.
#
# AUTHOR:       Hewlett Packard - Colorado Springs Division
#
#####

# Confirm setting of HP64000
if [ -z "$HP64000" ]
then
    HP64000=/usr/hp64000
fi

PATH=/usr/5bin:/bin:/usr/bin:/usr/ucb:/etc:$PATH
export PATH
```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_edit" Script

Defining Variables

The "rtos_edit" script defines variables that it uses later.

```
# Define variables
tables_orig=tables1
tables_stripped=tables2
tables_final=tables3
inittasks=s_init
edit_file=table.c

TASK_IDS=""
task_inputs="start_loop"
```

Displaying Install Information

```
echo "\n"
echo "          &dB-- RTOS Measurement Tool Edit Script --"
echo "          &dB-----"
ver_str="@(#)REV:          "
echo_str='echo $ver_str | sed 's/...../'`
echo "          $echo_str"
```

Checking Location of Files

This section of the "rtos_edit" script is used for testing the Custom RTOS Measurement Tool product. The files from the install directory should always be used.

```
#
# Check if files should be used from the test directory or install directory.
#
if [ "$TEST_INST_PATH" != "" ]
then
    set_path=/hp/rtos/custom/prdct_files
else
    set_path="$HWP64000/rtos/B3082A"
fi
inst_dir=$set_path
user_n_func_intervals=$inst_dir/action_keys/spabasecmd
```

Prompting the User for Information and Task IDs

As real-time OS applications are developed, the number of tasks may increase or decrease, so the "rtos_edit" script lets users define tasks by their ID numbers.

```
#
# Ask which files need to be created
#
echo "\n\n Do you want to create a new 'table.c' file? [y/n] (y): \c"
read create_table
create_table=${create_table:=y}

echo "\n\n Do you want to create a new 's_init' file? [y/n] (y): \c"
read create_inittasks
create_inittasks=${create_inittasks:=y}

#
# Check if either file wanted
if [ "$create_table" = "n" -a "$create_inittasks" = "n" ]
then
    exit
else
    #
    # Get the user's task id list
    #
    four_digit_tasks=""
    echo "\n Enter all the tasks IDs in your application."
    echo " (No more than 15 IDs per line!)\n"
    while [ ! "$task_inputs" = "" ]
    do
        #
        # Get a current list of inputs
        #
        index=0
        echo " Task ID(s) (return when done): \c"
        read task_inputs
        bad_task_id=FALSE
        for taskid in $task_inputs
        do
            #
            # Check that single input is a number and but not 9999
            #
            case $taskid in
                9999)
                    echo "\n Task id 9999 overlaps with error check id."
                    echo " (9999 removed from user list!)"
                    bad_task_id=TRUE;;
                [0-9])
                    tid=000$taskid;;
                [0-9][0-9])
                    tid=00$taskid;;
                [0-9][0-9][0-9])
                    tid=0$taskid;;
                [0-9][0-9][0-9][0-9])
                    tid=$taskid;;
                *)
                    echo " Task id '$taskid' was not a 1 to 4 digit number"
                    echo " ('$taskid' removed from user list!)"
                    tid=9999
                    bad_task_id=TRUE;;
            esac
        done
        #
        # Check if single input is a duplicate
        #
    done
fi
```

```

for dup_id in $TASK_IDS $four_digit_tasks
do
    if [ "$tid" = "$dup_id" ]
    then
        echo "      Duplicate task id '$tid' removed from list."
        bad_task_id=TRUE
    fi
done
#
# If input is valid, add it to current list of inputs
#
if [ "$bad_task_id" = "FALSE" ]
then
    four_digit_tasks="$four_digit_tasks $tid"
fi
bad_task_id=FALSE
done
#
# Add current list of inputs onto cumulative list
#
TASK_IDS="$TASK_IDS $four_digit_tasks"
four_digit_tasks=""
done

#
# List complete cumulative list of inputs
#
echo "\n  The following task IDs have been entered:"
echo $TASK_IDS |
awk '
{ for (i = 1; i <= NF; i++)
  {
    printf("\t%-12s", $i);
    if ((i % 4) == 0)
      printf("\n");
  }
,
}'
fi

```


Entering the Task Names into the "table.c" File

The "rtos_edit" script edits the data table file for the tasks that are being used.

```

if [ "$create_table" = "y" ]
then
#####
#
# Edit the task names into table.c
#
echo "\n\n"
echo "    Creating the new 'table.c' file..."

cp $inst_dir/./mikos/$edit_file $tables_orig
chmod 666 $tables_orig

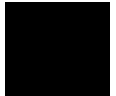
#
# Remove all defined macro entries
#
awk '
    BEGIN                                { delete_lines = 0 }
    /BEGIN TASK MODIFICATIONS/          { delete_lines = 1; print $0; next }
    /END TASK MODIFICATIONS/            { delete_lines = 0; print $0; next }
    /*/                                  { if (delete_lines == 0)
                                        {
                                            print $0
                                        }
                                        next
                                    }
' < $tables_orig > $tables_stripped
rm $tables_orig

#
# Create the task list of entry & exit variables
#
task_count_string=""
for task in $TASK_IDS
do
    task_entry_list="$task_entry_list\nshort int HPOS_Tenter_$task,HPOS_Texit_$task;"
done
task_entry_list="$task_entry_list\n\n"
task_entry_header="\nshort int HPOS_TaskTable;"

#
# Add the task variable list into the file
#
# Get the 'header' part of the file
awk '
    /BEGIN TASK MODIFICATIONS/ {
        print $0;
        printf("%s", taskheader);
        exit
    }
    /*/                          {print $0; next }
' \
    taskheader="$task_entry_header" \
    < $tables_stripped > $tables_final

# Add the task variables into the file
echo "$task_entry_list" >> $tables_final

```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_edit" Script

```
# Add the rest of the file on the end
awk '
BEGIN { start_lines = 0 }
/END TASK MODIFICATIONS/ { start_lines = 1; print $0; next }
./ */ { if (start_lines == 1)
        { print $0
        }
        }
next
}' < $tables_stripped >> $tables_final
rm $tables_stripped

#
# Move the file to it's final destination
#
chmod 666 $edit_file 2>/dev/null
if [ $? != 0 ]
then
echo "          (Could not change permissions on '$edit_file'.)"
fi
mv $tables_final $edit_file
if [ $? != 0 ]
then
echo "          Could not overwrite '$edit_file'."
echo "    Exiting script."
exit
else
chmod 444 $edit_file 2>/dev/null
fi

echo "    $edit_file - contains newly edited file."
fi
```

Entering the Task Names into the "s_init" File

The "rtos_edit" script edits the "s_init" command file for the tasks that are being used.

The "s_init" command file is run from the Software Performance Analyzer interface to define SPA event intervals for tasks, service calls, and user events.

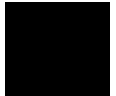
```
#
# Check if 's_init' file is to be created
#
if [ "$create_inittasks" = "y" ]
then
#####
#
# Edit the task names into s_init
#
echo "\n\n      *** Creating SPA command file for initializing intervals ***"

#
# Put header on file
#
echo "#" > $inittasks
echo "# HP RTOS Command File: spa_inittasks" >> $inittasks
echo "#" >> $inittasks
echo "# DESCRIPTION: Define all SPA intervals: tasks, service calls, user\"
>> $inittasks
echo "#" >> $inittasks
echo "stop_profile" >> $inittasks
echo "set byte_alignment word" >> $inittasks
echo "#" >> $inittasks

#
# Put task interval defines into file
#
pre_str="define single_event named"
for task in $TASK_IDS
do
    int_name=Task_$task
    int_start=HPOS_Tenter_$task
    int_end=HPOS_Texit_$task
    echo $pre_str $int_name interval $int_start thru $int_end >> $inittasks
done

#
# Put OS time & measurement overhead interval defines into file
#
echo $pre_str OS_Time interval HPOS_Start_Ovrhd thru HPOS_Stop_Ovrhd >>$inittasks
ovrhd_str="Measure_Ovrhd interval HPOS_Start_Intrusion thru HPOS_Stop_Intrusion"
echo $pre_str $ovrhd_str >> $inittasks

#
# Put standard OS function and user-defined intervals into file
#
cat $user_n_func_intervals >> $inittasks
```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_edit" Script

Checking if a Unique Environment is Required

If more than one project is using the same RTOS measurement tool on the same file system, the "s_init" file created must be unique to the currently loaded application.

In order to let users set up unique SPA windows for multiple projects, the "rtos_edit" script will set the RTOS_UNIQUE

```
#####
#
# Check if a unique environment is required for SPA use
#
echo "\n  The SPA environment can be customized by creating command files"
echo "  for making measurements specific to an application. You can set"
echo "  up environments so that each user can have his own environment"
echo "  or alternatively can use the global customizations."
echo "  (See the RTOS B3081 manual for more explanation of this question.)"

echo "\n  Do you want to set up a unique user environment? [y/n] (n): \c"
read mult_tool
mult_tool=${mult_tool:=n}

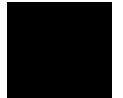
unique_suffix=
if [ "$mult_tool" = "y" ]
then
  # Get a user unique suffix
  RTOS_UNIQUE=${RTOS_UNIQUE:=NoT_SeT}
  if [ "$RTOS_UNIQUE" = "NoT_SeT" ]
  then
    echo "\n  Enter a unique suffix to differentiate your files."
    echo "  (currently used suffixes: \c"
    uniq_files=`ls $inst_dir/action_keys/s_init_* 2>/dev/null`
    for file in $uniq_files
    do
      suffix=`echo $file | sed 's/.*\s_init//`
      echo " $suffix\c"
    done
    echo ")"
    echo "  Your initials are suggested (and start with underscore '_'): \c"
    read unique_suffix
    unique_suffix=${unique_suffix:=_rtos}
    RTOS_UNIQUE=$unique_suffix
    export unique_suffix

    echo "\n"
    echo "  Remember to PERMANENTLY set 'RTOS_UNIQUE' to '$unique_suffix'"
    (preferably"
    echo "  by setting it in your startup shell script) and make it global. All"
    echo "  other users who want to use the same command files must also set"
    echo "  'RTOS_UNIQUE' in their execution environment."
    echo "  (ksh: export RTOS_UNIQUE=$unique_suffix)"
    echo "  (csh: setenv RTOS_UNIQUE $unique_suffix)"
    echo "  ( sh: RTOS_UNIQUE=$unique_suffix"
    echo "  export RTOS_UNIQUE)"
  fi
fi
final_init=$inittasks$unique_suffix
```

Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_edit" Script

```
echo "\n\n  Command file for initializing SPA intervals:"  
echo "      '$inst_dir/action_keys/$final_init'"  
mv $inittasks $inst_dir/action_keys/$final_init  
chmod 666 $inst_dir/action_keys/$final_init  
fi
```



The "rtos_emul" Startup Script

The "rtos_emul" script sets up environment variables and defines action keys before starting the emulator/analyzer interface. The "rtos_emul" script:

- Defines the supported processors.
- Sets environment variables.
- Checks parameters.
- Displays startup information.
- Prompts for the processor type (if it hasn't been included as a parameter).
- Sets variables based on the processor type.
- Checks if the RTOS inverse assembler has been installed.
- Sets the HP64KPATH environment variable.
- Adds the RTOS directory to the PATH environment variable.
- Starts the Emulator/Analyzer interface with "emul700".

Defining the Supported Processors

The demo RTOS measurement tool supports several 68000 family processors. These processors are listed in the "rtos_emul" script to give users a list from which to choose their processor.

```
#!/bin/sh
#####
#
#          CUSTOM REAL TIME OPERATING SYSTEM MEASUREMENT TOOL          #
#          EMULATION START COMMAND FILE                                #
#          HEWLETT PACKARD - COLORADO SPRINGS DIVISION                  #
#
#####

proc_list="68000 68010 68302 68030 68020 68331 68332 68040 68340"
#
# Assign this variable to one of the values of 'proc_list'.
#
PROCESSOR=""
```

Setting Environment Variables

The paths needed by the "rtos_emul" script are added to the PATH environment variable. If the HP64000 environment variable is not already set, it is set to /usr/hp64000.

```
PATH=/usr/5bin:/bin:/usr/bin:/usr/ucb:/etc:$PATH
export PATH

#####
# Confirm setting of HP64000
if [ -z "$HP64000" ]
then
    HP64000=/usr/hp64000
fi
```

Checking Parameters

The "rtos_emul" script lets you specify a startup command file and the processor type in addition to the emulator name.

The "emulator_name" is the logical emulator name given in the HP 64700 emulator device table file (\$HP64000/etc/64700tab.net).

```
show_usage()
{
    echo "usage: $0 [-c <cmd_file>] [PROCESSOR] <emulator_name> &"
    show_proc_choices
    exit
}

#
# Check for input
# Possible inputs:
# 1) emulrtos box-name
# 2) emulrtos processor box-name
# 3) emulrtos -c command_file box-name
# 4) emulrtos -c command_file processor box-name
#
COMMAND_INPUT=""

if [ $# -gt 4 ]          # Check if have too many parameters
then
    show_usage
fi

if [ $# -eq 1 ]         # Check if parameter option 1
then
    EMULATOR=$1
elif [ $# -eq 2 ]      # Check if parameter option 2
then
    PROCESSOR=$1
    EMULATOR=$2
else
    # Check for valid options 3 or 4
    if [ "$1" != "-c" ] # Check that '-c cmd_file' is in parms
    then
```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_emul" Startup Script

```
    show_usage
else
    COMMAND_INPUT="-c $2"      # Set command file string
fi
if [ $# -eq 3 ]              # Check if parm option 3 used
then
    EMULATOR=$3              # Must be parm option 3, 'PROCESSOR' not given
else
    PROCESSOR=$3              # Must be parm option 4
    EMULATOR=$4
fi
fi
```

Displaying Startup Information

```
echo "\n\n          &dB-- Real Time Operating System Custom Measurement Tool --&d@"
echo "\n          &dB-- Emulation Start Script --&d@"
echo "          Hewlett Packard\n"
ver_str="@(#)REV:          "
echo_str='echo $ver_str | sed 's/.....//''
echo "          $echo_str"
```

Prompting for the Processor Type

The processor type tells the "rtos_emul" script which emulator is being used and whether the processor has a 16- or 32-bit data bus.

```
show_proc_choices()
{
    echo "\n          Processor choices:"
    echo "          \c"
    for possible_proc in $proc_list
    do
        echo " $possible_proc\c"
    done
    echo ""
}

#
# Let the user choose the processor that RTOS will be run with if not entered
# on command line or not set within script.
#
proc_choice=$PROCESSOR
while [ "$proc_choice" = "" ]
do
    echo "\n          Which emulator will you use with HP's Real Time OS Custom
Measurement Tool?"
    show_proc_choices
    echo "\n          ENTER THE EMULATOR'S PROCESSOR TYPE: \c"
    read PROCESSOR
    for possible_proc in $proc_list
    do
        if [ "$PROCESSOR" = "$possible_proc" ]
        then
            proc_choice=$PROCESSOR
            break
        fi
    done
done
```



```
    fi
done
if [ "$proc_choice" = "" ]
then
    echo "          **Error: Invalid processor choice! Try again."
fi
done
```

Setting Variables Based on the Processor Type

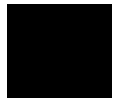
The processor type tells the "rtos_emul" script where to look for the inverse assembler, what the specific X resource application name is, and whether the processor has a 16- or 32-bit data bus.

The "rtos_emul" script makes sure the inverse assembler has been installed before starting the emulator/analyzer interface. If the inverse assembler has not been installed, the script exits.

The X resource application name is used when defining the X resources that set up action keys in the emulator/analyzer interface.

The processor data bus width identifies the command files that are assigned to action keys when there are different versions for 16- and 32-bit processors.

```
#
# Set all variables related to the processor choice
#
case $PROCESSOR in
*68000)
    rtos_install=$HP64000/inst/emul/64742A/rtos;
    PROC_RESOURCE=m68000;
    ACTION_KEY_SFX=_16;
    ;;
*68010)
    rtos_install=$HP64000/inst/emul/64745A/rtos;
    PROC_RESOURCE=m68010;
    ACTION_KEY_SFX=_16;
    ;;
*68302)
    rtos_install=$HP64000/inst/emul/64746A/rtos;
    PROC_RESOURCE=m68302;
    ACTION_KEY_SFX=_16;
    ;;
*68030)
    rtos_install=$HP64000/inst/emul/64747A/rtos;
    PROC_RESOURCE=m68030;
    ACTION_KEY_SFX=_32;
    ;;
*68020)
    rtos_install=$HP64000/inst/emul/64748A/rtos;
    PROC_RESOURCE=m68020;
    ACTION_KEY_SFX=_32;
    ;;
*68331)
    rtos_install=$HP64000/inst/emul/64749A/rtos;
    PROC_RESOURCE=m68331;
```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_emul" Startup Script

```
        ACTION_KEY_SFX=_16;
        ;;
*68332)
    rtos_install=$HP64000/inst/emul/64749A/rtos;
    PROC_RESOURCE=m68332;
    ACTION_KEY_SFX=_16;
    ;;
*68040)
    rtos_install=$HP64000/inst/emul/64750A/rtos;
    PROC_RESOURCE=m68040;
    ACTION_KEY_SFX=_32;
    ;;
*68340)
    rtos_install=$HP64000/inst/emul/64751A/rtos;
    PROC_RESOURCE=m68340;
    ACTION_KEY_SFX=_16;
    ;;
*)
    show_usage;
    ;;
esac
```

Checking if the RTOS Inverse Assembler Has Been Installed

If the RTOS inverse assembler has not been installed, the capability to display real-time OS traces will not be present in the emulator/analyzer interface.

```
#
# Check that the RTOS IAL has been installed.
#
if [ ! -f $rtos_install/rtos.R ]
then
    echo "\n\n"
    echo "    *** NO RTOS SPECIFIC IAL HAS BEEN INSTALLED. ***"
    echo "    (Couldn't find '$rtos_install/rtos.R')"
```

Setting the HP64KPATH Environment Variable

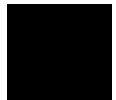
The HP64KPATH environment variable tells the Emulator/Analyzer interface where to search for command files if they are not found in the current directory. The "rtos_emul" script adds the \$HP64000/rtos/B3082A directory to the HP64KPATH environment variable.

```
#
# Check setting of environment variable for command files
#
cur_pwd=`pwd`
if [ "$cur_pwd" = "/hp/rtos/custom/demo" ]
then
    set_path="$HP64000/rtos/B3082A"
else
    set_path=/hp/rtos/custom/demo
fi
echo "        Adding '$set_path' to HP64KPATH"
if [ "$HP64KPATH" = "" ]
then
    HP64KPATH=$set_path
    export HP64KPATH
else
    var_set=`echo $HP64KPATH | egrep '$set_path'`
    if [ "$var_set" = "" ]
    then
        HP64KPATH=$HP64KPATH:$set_path
        export HP64KPATH
    fi
fi
```

Adding the RTOS Directory to the PATH Environment Variable

The PATH environment variable tells the UNIX command shell where to find commands and scripts. Because some of the command files run scripts in the \$HP64000/rtos/B3082A directory, this directory is added to the PATH environment variable.

```
#
# Check setting of environment variable for RTOS scripts
#
var_set=`echo $PATH | egrep '$set_path'`
if [ "$var_set" = "" ]
then
    echo "        Adding '$set_path' to PATH"
    PATH=$PATH:$set_path
    export PATH
fi
```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_emul" Startup Script

Starting the Emulator/Analyzer Interface with "emul700"

The **emul700** command is used to start the Emulator/Analyzer interface, and the **-xrm** command line option is used to define the X resources that set up action key.

```
#
# Write message
#
echo "      Emulation Window is now being created."
echo "      Please wait..."

#
# Bring up an emulator window
#
emul700 \
$COMMAND_INPUT \
"-xrm emul.$PROC_RESOURCE*actionKeys.packing:    PACK_COLUMN" \
"-xrm emul.$PROC_RESOURCE*actionKeys.numColumns: 4" \
"-xrm emul.$PROC_RESOURCE*actionKeysSub.keyDefs: \
  \"Track OS calls\"          \"action_keys/e_trkcalls\" \
  \"Track Everything\"        \"action_keys/e_trkall\" \
  \"Task switch A->B\"        \"action_keys/e_AthenB$ACTION_KEY_SFX\" \
  \"Help RTOS\"              \"!tellrtosHP !in_browser\" \
  \"Memory Usage\"           \"action_keys/e_memory$ACTION_KEY_SFX\" \
  \"Only Call X\"            \"action_keys/e_onecall\" \
  \"Only Calls X & Y\"       \"action_keys/e_twocalls$ACTION_KEY_SFX\" \
  \"Tsk A msg->Que X\"       \"action_keys/e_tsk2queue$ACTION_KEY_SFX\" \
  \"Custom OS Trace\"       \"display trace real_time_os\" \
  \"<UserDefinable1>\"      \"#Edit 'rtos_emul' to define this key\" \
  \"Only Task X\"           \"action_keys/e_trk1task$ACTION_KEY_SFX\" \
  \"Only Tsk W,X,Y,Z\"     \"action_keys/e_trk4task$ACTION_KEY_SFX\" \
  \"Task A: VarX\"         \"action_keys/e_aftervar$ACTION_KEY_SFX\" \
  \"NonCustom Trace\"     \"display trace mnemonic\" \
  \"<UserDefinable2>\"     \"#Edit 'rtos_emul' to define this key\" \
  \"Tasks & Queues\"       \"action_keys/e_trackqs$ACTION_KEY_SFX\" \
  \"Only Queues\"         \"action_keys/e_onlyqs\" \
  \"Task A: FuncX\"       \"action_keys/e_afterfunc$ACTION_KEY_SFX\" \
  \"Before SPA trig2\"    \"action_keys/e_spatrig\" \
  \" \
$EMULATOR &
```

The "rtos_spa" Startup Script

The "rtos_spa" startup script:

- Sets environment variables.
- Checks parameters.
- Displays startup information.
- Sets the HP64KPATH environment variable.
- Adds the RTOS directory to the PATH environment variable.
- Starts the SPA interface with "emul700".

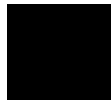
Setting Environment Variables

The paths needed by the "rtos_spa" script are added to the PATH environment variable. If the HP64000 environment variable is not already set, it is set to /usr/hp64000.

```
#!/bin/sh
#####
#
#          CUSTOM REAL TIME OPERATING SYSTEM MEASUREMENT TOOL          #
#          SOFTWARE PERFORMANCE ANALYZER START SHELL SCRIPT            #
#          HEWLETT PACKARD - COLORADO SPRINGS DIVISION                 #
#
#####
ver_str="@(#)REV:          "
#####

#####
PATH=/usr/5bin:/bin:/usr/bin:/usr/ucb:/etc:$PATH
export PATH

# Confirm setting of HP64000
if [ -z "$HP64000" ]
then
    HP64000=$HP64000
fi
```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_spa" Startup Script

Checking Parameters

The "rtos_spa" script lets you specify a startup command file and the processor type in addition to the emulator name.

The "emulator_name" is the logical emulator name given in the HP 64700 emulator device table file (\$HP64000/etc/64700tab.net).

```
show_usage()
{
    echo "usage: $0 [-c command_file] <emulator_name> &"
    exit
}

#
# Check for input
# Possible inputs:
#   emulrtos box-name
#   emulrtos -c command_file box-name
#
COMMAND_INPUT=""

if [ $# -eq 1 ]
then
    EMULATOR=$1
elif [ $# -eq 3 ]
then
    if [ "$1" != "-c" ]
    then
        show_usage
    else
        COMMAND_INPUT="-c $2"
    fi
    EMULATOR=$3
else
    show_usage
fi
```

Displaying Startup Information

```
# echo "HJ"
echo "\n\n          &dB-- Real Time Operating System Custom Measurement Tool --&d@"
echo "          &dB-- SPA Start Script --&d@"
echo "          Hewlett Packard\n"
echo_str='echo $ver_str | sed 's/.....//''
echo "          $echo_str"
```

Setting the HP64KPATH Environment Variable

The HP64KPATH environment variable tells the Software Performance Analyzer interface where to search for command files if they are not found in the current directory. The "rtos_spa" script adds the \$HP64000/rtos/B3082A directory to the HP64KPATH environment variable.

```
#
# Check setting of environment variable for command files & scripts
#

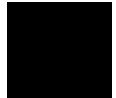
# Get test path if set
if [ "$TEST_INST_PATH" = "" ]
then
    set_hp64_path="$HP64000/rtos/B3082A"
else
    set_hp64_path=/hp/rtos/custom/prdct_files
fi

# Set HP64KPATH
echo "    Adding '$set_hp64_path' to HP64KPATH"
if [ "$HP64KPATH" = "" ]
then
    HP64KPATH=$set_hp64_path
    export HP64KPATH
else
    var_set=`echo $HP64KPATH | egrep '$set_hp64_path'`
    if [ "$var_set" = "" ]
    then
        HP64KPATH=$HP64KPATH:$set_hp64_path
    fi
    export HP64KPATH
fi
```

Adding the RTOS Directory to the PATH Environment Variable

The PATH environment variable tells the UNIX command shell where to find commands and scripts. Because some of the command files run scripts in the \$HP64000/rtos/B3082A directory, this directory is added to the PATH environment variable.

```
# Set environment variable for RTOS scripts
echo "    Adding '$set_hp64_path' to PATH"
var_set=`echo $PATH | egrep '$set_hp64_path'`
if [ "$var_set" = "" ]
then
    PATH=$PATH:$HP64000/rtos/B3082A
    export PATH
fi
```



Chapter 11: Demo RTOS Measurement Tool Details

The "rtos_spa" Startup Script

Starting the SPA Interface with "emul700"

The **emul700 -u xperf** command is used to start the Software Performance Analyzer interface, and the **-xrm** command line option is used to define the X resources that set up action keys.

```
#
# Write message
#
echo "          SPA Window is now being created."
echo "          Please wait..."

#
# Bring up window for SPA if requested
#
emul700 \
-u xperf \
$COMMAND_INPUT \
-xrm 'HP64_Softkey.geometry: +415+230' \
-xrm 'HP64_Softkey*enableCmdline: False' \
-xrm 'HP64_Softkey*actionKeys.packing:   PACK_COLUMN' \
-xrm 'HP64_Softkey*actionKeys.numColumns: 2' \
-xrm "perf*actionKeysSub.keyDefs: \
  \"Initialize\"          \"action_keys/s_init$RTOS_UNIQUE\" \
  \"Time Tasks\"          \"action_keys/s_timetasks\" \
  \"Count Srvc Calls\"    \"action_keys/s_countsrvcls\" \
  \"Trig2 on Overflow\"   \"action_keys/s_break_ovrflw\" \
  \"FunctionDuration\"    \"action_keys/s_funcdur\" \
  \"TaskX: Servcalls\"    \"action_keys/s_taskwindow\" \
  \"Count Tasks\"         \"action_keys/s_counttasks\" \
  \"Tsk & User Evnts\"    \"action_keys/s_taskuser\" \
  \"Disable Trig2\"       \"action_keys/s_disabletrg2\" \
\" \
$EMULATOR &
```


Scripts Run by the Action Key Command Files

Several of the demo RTOS measurement tool action key command files call scripts to set up trace commands or task names. The ability to run scripts from command files gives the interfaces greater power and flexibility when making real-time OS measurements.

create_12_call

The "create_12_call" script is used by some of the Emulator/Analyzer interface action key command files for 16-bit processors.

The "create_12_call" script creates a command file to track 1 or 2 service calls when using a 16-bit processor. The names of the service calls to be tracked are specified as parameters. This script creates the ".e_onesrvcall" command file which contains a trace command to track the specified service calls.

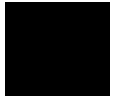
This script identifies the next service call locations that appear in the data table.

If the script is creating a command file that tracks one service call, it creates a trace command that stores writes in the range from the current service call location to the start of the next service call location.

If the script is creating a command file that tracks two service calls, it creates a trace command that captures two windows of code execution (one for each service call) and stores writes to the data table during these windows of execution.

```
#!/bin/sh
#####
#
#          CUSTOM REAL TIME OPERATING SYSTEM MEASUREMENT TOOL
#          HEWLETT PACKARD - COLORADO SPRINGS DIVISION
#          Product #B3082A
#
# This is a shell script to create the command to trace a single service
# call OR trace two service calls.
#
PATH=/usr/5bin:/bin:/usr/bin:/usr/ucb:/etc:$PATH
export PATH

find_adjacent_labels()
{
    case $parm in
        task_create)    funcname=HPOS_task_create
                        nextfunc=HPOS_send_message_Entry
                        ;;
    )
    )
}
```



Chapter 11: Demo RTOS Measurement Tool Details

Scripts Run by the Action Key Command Files

```
send_message) funcname=HPOS_send_message
               nextfunc=HPOS_get_message_Entry
               ;;
get_message)  funcname=HPOS_get_message
               nextfunc=HPOS_alloc_message_Entry
               ;;
alloc_message) funcname=HPOS_alloc_message
               nextfunc=HPOS_free_message_Entry
               ;;
free_message) funcname=HPOS_free_message
               nextfunc=HPOS_alloc_mem_Entry
               ;;
alloc_memory) funcname=HPOS_alloc_mem
               nextfunc=HPOS_String_array
               ;;
*)           funcname=error
               nextfunc=error
               ;;
esac
}

#
# Initialize names
#
funcnameI=""
nextfuncI=""
funcnameII=""
nextfuncII=""

#
# Find corresponding symbols for each input
#
suffix="I"
for parm in $*
do
    find_adjacent_labels $parm

    eval funcname$suffix=$funcname
    eval nextfunc$suffix=$nextfunc

    suffix="II"
done

#
# Check if either function name was unknown
#
if [ "$funcnameI" = "error" -o "$funcnameII" = "error" ]
then
    string="# One of entered service call function names was unknown"
else
    #
    # Determine whether 1 or 2 input parms were entered
    #
    if [ "$funcnameII" = "" ]
    then
        string="trace only address range "$funcnameI"_Entry thru "$nextfuncI"-2"
    else
        #
        # Enable and disable on the first and last (respectively) writes for
        # each service call
        #
        string="trace enable "$funcnameI"_Entry or "$funcnameII"_Entry or
```

Chapter 11: Demo RTOS Measurement Tool Details Scripts Run by the Action Key Command Files

```
"$funcnameI"_Exit or "$funcnameII"_Exit disable "$funcnameI"_Exit-2 or  
"$funcnameII"_Exit-2 or "$nextfuncI"-2 or "$nextfuncII"-2 only address range  
HP_RTOS_TRACK_START thru HPOS_String_array-1"  
    fi  
fi  
  
#  
# Create the command file  
#  
echo "# Command file to trace specific service calls" > .e_onesrvcall  
echo $string >> .e_onesrvcall
```

create_12_call32

The "create_12_call32" script is used by some of the Emulator/Analyzer interface action key command files for 32-bit processors.

The "create_12_call32" script creates a command file to track 1 or 2 service calls when using a 32-bit processor. The names of the service calls to be tracked are specified as parameters. This script creates the ".e_onesrvcall" command file which contains a trace command to track the specified service calls.

This script identifies the next service call locations that appear in the data table.

If the script is creating a command file that tracks one service call, it creates a trace command that stores writes in the range from the current service call location to the start of the next service call location.

If the script is creating a command file that tracks two service calls, it creates a trace command that stores writes in the range associated with the first service call and uses the remaining 8 state qualifiers to trace successive locations associated with the second service call.

```
#!/bin/sh  
#####  
#  
#           CUSTOM REAL TIME OPERATING SYSTEM MEASUREMENT TOOL  
#           HEWLETT PACKARD - COLORADO SPRINGS DIVISION  
#           Product #B3082  
#  
# This is a shell script to create the command to trace a single service  
# call OR trace two service calls for 32 bit processors.  
#  
PATH=/usr/5bin:/bin:/usr/bin:/usr/ucb:/etc:$PATH  
export PATH  
  
find_adjacent_labels()  
{  
    case $parm in  
        task_create)    funcname=HPOS_task_create
```

Chapter 11: Demo RTOS Measurement Tool Details

Scripts Run by the Action Key Command Files

```
        dataspace=8
        nextfunc=HPOS_send_message_Entry
        ;;
send_message) funcname=HPOS_send_message
              dataspace=4
              nextfunc=HPOS_get_message_Entry
              ;;
get_message)  funcname=HPOS_get_message
              dataspace=4
              nextfunc=HPOS_alloc_message_Entry
              ;;
alloc_message) funcname=HPOS_alloc_message
              dataspace=3
              nextfunc=HPOS_free_message_Entry
              ;;
free_message) funcname=HPOS_free_message
              dataspace=2
              nextfunc=HPOS_String_array
              ;;
*)          funcname=error
           nextfunc=error
           dataspace=0
           ;;
    esac
}

#
# Initialize names
#
funcnameI=""
nextfuncI=""
dataspaceI=0
funcnameII=""
nextfuncII=""
dataspaceII=0

#
# Find corresponding symbols for each input
#
suffix="I"
for parm in $*
do
    find_adjacent_labels $parm

    eval funcname$suffix=$funcname
    eval nextfunc$suffix=$nextfunc
    eval dataspace$suffix=$dataspace

    suffix="II"
done

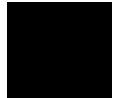
echo $funcnameI
echo $nextfuncI
echo $dataspaceI
echo $funcnameII
echo $nextfuncII
echo $dataspaceII

#
# Check if either function name was unknown
#
if [ "$funcnameI" = "error" -o "$funcnameII" = "error" ]
```

Chapter 11: Demo RTOS Measurement Tool Details Scripts Run by the Action Key Command Files

```
then
  string="# One of entered service call function names was unknown"
else
  #
  # Determine whether 1 or 2 input parms were entered
  #
  if [ "$funcnameII" = "" ]
  then
    string="trace only address range "$funcnameI"_Entry thru "$nextfuncI"-2"
  else
    #
    # Do trace only x or +4 or +8 or ... thru full range
    #
    increment_string=" or "$funcnameII"_Entry
    increment=4
    while [ "$dataspaceII" -gt 1 ]
    do
      # Add another 'or' address on to string
      increment_string="$increment_string" or "$funcnameII"_Entry+"$increment
      dataspaceII=`expr "$dataspaceII" - 1`
      increment=`expr "$increment" + 4`
    done
    string="trace only address range "$funcnameI"_Entry thru
"$nextfuncI"-2"$increment_string
  fi
fi

#
# Create the command file
#
echo "# Command file to trace specific service calls" > .e_onesrvcall
echo "$string" >> .e_onesrvcall
```



Chapter 11: Demo RTOS Measurement Tool Details

Scripts Run by the Action Key Command Files

get_task_number

The "get_task_number" script is used by some of the Emulator/Analyzer interface action key command files for 16-bit processors.

The "get_task_number" script creates a command file that sets the TASK_EVENT environment variable to "Task_< 4digitID>".

```
#!/bin/sh
#
# A script to convert the input number into a form with leading zeroes and
# only four digits and then create a command file to 'window' on the task
# with the id of the given number.
#
PATH=/usr/5bin:/bin:/usr/bin:/usr/ucb:/etc:$PATH
export PATH
#
chmod +w .s_settaskvar
#
case $1 in
  [0-9])          tid=000$1;;
  [0-9][0-9])     tid=00$1;;
  [0-9][0-9][0-9]) tid=0$1;;
  [0-9][0-9][0-9][0-9]) tid=$1;;
  *)              echo "ERROR: Task id '$1' was not a 1 to 4 digit number" > .s_settaskvar;
                  exit
                  ;;
esac
#
# Create command file
#
echo set TASK_EVENT = Task_$tid > .s_settaskvar
chmod -w .s_settaskvar
```

Part 5

Installation Guide

Instructions for installing and configuring the product.

Part 5



12

Installation



Installation

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

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

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

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

This chapter shows you how to:

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

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

To install HP 9000 software

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

1 Check the HP-UX operating system version

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

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

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

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

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

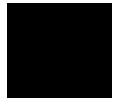
3 Make sure the tape's write-protect screw points to SAFE.

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

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

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

```
/etc/update
```



Chapter 12: Installation
To install HP 9000 software

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

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

`/` (for the destination directory)

- 8 If you do not use the defaults, change the "source device" and/or "destination directory" as appropriate.
- 9 Select `Load Everything from Source Media` when your source and destination directories are correct.
- 10 To begin the update, press the softkey `<Select Item>`. At the next menu, press the softkey `<Select Item>` again. Answer the last prompt with

`y`

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

To install Sun SPARCsystem software

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

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

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





Glossary

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

clock tick a unit of time used by the OS for the purpose of scheduling tasks or processes. The length of time is determined by a periodic interrupt which is handled by a special interrupt service routine that lets the OS know a clock tick should occur. The OS may switch tasks that have specified "time slices" to "blocked" after a certain number of clock ticks.

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.

communication variables the locations through which the Emulator/Analyzer interface passes information to the inverse assembler.

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.

Glossary

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.

selective tracking the ability to track a single service call or a subset of service calls.

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

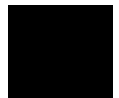
- A** ABORT instruction (IAL), **86, 171**
 - about, trace command option, **106**
 - action keys, **5, 19, 28, 32, 154-157**
 - action keys, Emulator/Analyzer
 - Before SPA trig2, **141**
 - Memory Usage, **121**
 - Only Call X, **111**
 - Only Calls X & Y, **112**
 - Only Queues, **110**
 - Only Task X, **113-114**
 - Only Tsk W,X,Y,Z, **115**
 - Task A msg-Que X, **117**
 - Task A: FuncX, **118-119**
 - Task A: VarX, **119-120**
 - Task switch A-B, **116**
 - Tasks & Queues, **111**
 - Track OS calls, **109**
 - action keys, SPA
 - Count Srv Calls, **137**
 - Count Tasks, **135**
 - Disable Trig2, **143**
 - FunctionDuration, **138**
 - Initialize, **132**
 - TaskX: Servcalls, **136, 142**
 - Time Tasks, **133**
 - Trig2 on Overflow, **142**
 - Tsk & User Evnts, **139**
 - ADD instruction (IAL), **172**
 - address info. in trace, decoding, **93**
 - after, trace command option, **106**
 - AND instruction (IAL), **173**
 - ASCII instruction (IAL), **88, 174**
 - ASCII STRING error, **214**



- B** background emulation monitor, **26**
 - Before SPA trig2, Emulator/Analyzer action key, **141**
 - before, trace command option, **106**
 - bit ranges (IAL), **84**
 - buckets, **63, 72, 75-76, 255**
 - bytes used on stack, **72**

- C** CALL instruction (IAL), **85, 95, 175**
 - callout routines, **18, 75, 255**
 - task switch, **63, 72**
 - CASE_OF instruction (IAL), **86, 176-177**
 - CK_REL_POS_TAG inverse assembler routine, **97**
 - clock ticks, **57, 59, 67, 69, 255**
 - command files, **5, 47-49, 149, 151-153**
 - e_afterfunc_16, **118**
 - e_afterfunc_32, **119**
 - e_aftervar_16, **119**
 - e_aftervar_32, **120**
 - e_AthenB_16, **116**
 - e_AthenB_32, **116**
 - e_memory_16, **121**
 - e_memory_32, **121**
 - e_onecall, **111**
 - e_onlyqs, **110**
 - e_spatrig, **141**
 - e_trackqs_16, **111**
 - e_trackqs_32, **111**
 - e_trk1task_16, **113**
 - e_trk1task_32, **114**
 - e_trk4task_16, **115**
 - e_trk4task_32, **115**
 - e_trkall, **109**
 - e_trkcalls, **109**
 - e_tsk2queue_16, **117**
 - e_tsk2queue_32, **117**
 - e_twocalls_16, **112**
 - e_twocalls_32, **112**
 - parameters, **150**
 - s_break_ovrflw, **142**
 - s_countsrvcis, **137**
 - s_counttasks, **135**
 - s_disabletrg2, **143**

- command files (continued)
 - s_funcdur, **138**
 - s_init, **132**
 - s_taskuser, **139**
 - s_taskwindow, **136, 142**
 - s_timetasks, **133**
 - scripts, using, **151**
 - search directories, **152**
 - communication variables (IAL), **83, 189, 255**
 - COMPLEMENT instruction (IAL), **178**
 - configuration, emulator, **26**
 - CONSTANT instruction (IAL), **179**
 - coordinated measurements, **140-143**
 - count histogram display of task events, **134**
 - Count Srv Calls, SPA action key, **137**
 - Count Tasks, SPA action key, **135**
 - create_12_call, trace command creation script, **47, 241**
 - create_12_call32, trace command creation script, **47, 243**
 - custom product files, installing, **160**
 - customize script, **25**
- D**
- data bucket, **55**
 - data buckets, **63, 75-76**
 - data bus width, **106**
 - data info. in trace, decoding, **95**
 - data table, **5, 18, 38, 55, 61, 73-74, 90, 104, 108, 255**
 - DEBUG_PRINT inverse assembler routine, **96**
 - DECREMENT instruction (IAL), **180**
 - DEFAULT_WIDTH instruction (IAL), **181**
 - DEFINITION ERROR error, **214**
 - demo inverse assembler, **41**
 - demo OS, **20, 24, 26, 37**
 - demo RTOS measurement tool, **20, 36**
 - device call, **255**
 - device table file, **25, 231, 238**
 - Disable Trig2, SPA action key, **143**
 - disable, trace command option, **106**
 - DISPLAY_RET_CODE inverse assembler routine, **96**
 - DUPLICATE SYMBOL error, **214**
 - duration (function), show histogram, **137**



- E**
 - e_afterfunc_16 command file, **118**
 - e_afterfunc_32 command file, **119**
 - e_aftervar_16 command file, **119**
 - e_aftervar_32 command file, **120**
 - e_AthenB_16 command file, **116**
 - e_AthenB_32 command file, **116**
 - e_memory_16 command file, **121**
 - e_memory_32 command file, **121**
 - e_onecall command file, **111**
 - e_onlyqs command file, **110**
 - e_spatrig command file, **141**
 - e_trackqs_16 command file, **111**
 - e_trackqs_32 command file, **111**
 - e_trk1task_16 command file, **113**
 - e_trk1task_32 command file, **114**
 - e_trk4task_16 command file, **115**
 - e_trk4task_32 command file, **115**
 - e_trkall command file, **109**
 - e_trkcalls command file, **109**
 - e_tsk2queue_16 command file, **117**
 - e_tsk2queue_32 command file, **117**
 - e_twocalls_16 command file, **112**
 - e_twocalls_32 command file, **112**
 - emul700 command, **25, 29, 154**
 - emulation bus analyzer, **18-19, 24, 104, 141, 255**
 - measurements, **54**
 - resource limitations, **54, 104**
 - emulation monitor, **26**
 - emulator
 - configuration, **26**
 - device table file, **25, 231, 238**
 - name, **25, 231, 238**
 - enable, trace command option, **106**
 - end command, **33**
 - environment variables, **25**
 - HP64000, **24, 221, 231, 237**
 - HP64_DEBUG_PATH, **119**
 - HP64KPATH, **152, 235, 239**
 - PATH, **25, 221, 231, 235, 237, 239**
 - PROC_RESOURCE, **155**

- environment variables (continued)
 - PROCESSOR, **25**
 - RTOS_UNIQUE, **228**
- error messages (IAL builder), **214**
- error returns (from service calls), **57**
- events (SPA)
 - defining for tasks, **130**
 - table display, **134**
- events (task)
 - count histogram display, **134**
 - time histogram display, **133**
- EXCLUSIVE_OR instruction (IAL), **182**
- EXPRESSION TYPE error, **214**
- EXTRACT_BIT instruction (IAL), **183**
- F**
 - FETCH_POSITION instruction (IAL), **184**
 - find_sequence, trace command option, **106**
 - foreground emulation monitor, **26**
 - FORMAT instruction (IAL), **88, 185**
 - function
 - any task using a, **120**
 - specific task using a, **118**
 - function duration histogram, show normal, **137**
 - FunctionDuration, SPA action key, **138**
- G**
 - get_task_number script, **246**
 - glossary, **255-256**
 - GOTO instruction (IAL), **85, 186**
- H**
 - histogram
 - normal function duration, **137**
 - task events, **139**
 - user events, **139**
 - histogram display of task events
 - count, **134**
 - time, **133**
 - HP 64700 interfaces, exiting and releasing, **33**
 - HP64000 environment variable, **24, 221, 231, 237**
 - HP64000, reserved IAL symbol, **170**
 - HP64_DEBUG_PATH environment variable, **119**
 - HP64KPATH environment variable, **152, 235, 239**
 - HP_RTOS_TRACK_START symbol, **90**
 - HPUX, reserved IAL symbol, **170**

- I
 - IAL (Inverse Assembly Language), **80**
 - IAL code, writing, **82-86**
 - IAL definitions, **42**
 - IAL instructions, **82**
 - executable, **82, 166**
 - pseudo, **82, 168**
 - IAL operands, **83**
 - IAL program control, **85**
 - IAL routines, **44, 95**
 - common routines, **44**
 - IAL symbols, predefined, **170**
 - IAL variables, **42**
 - ial.S, inverse assembler source file, **42, 45, 101**
 - ial16.S, common 16-bit inverse assembler routines, **45, 101**
 - ial32.S, common 32-bit inverse assembler routines, **45, 101**
 - IF instruction (IAL), **86, 187**
 - IF_NOT_MAPPED instruction (IAL), **188-189**
 - ILLEGAL CONSTANT error, **214**
 - ILLEGAL EXPRESSION error, **215**
 - ILLEGAL SYMBOL error, **215**
 - immediate values (IAL), **85**
 - INCLUSIVE_OR instruction (IAL), **190**
 - INCREMENT instruction (IAL), **191**
 - INITIAL_FLAGS communication variable (IAL), **84, 90, 169**
 - INITIAL_OPTIONS communication variable (IAL), **84, 90, 169**
 - Initialize, SPA action key, **132**
 - INPUT instruction (IAL), **192-193**
 - INPUT_ADDRESS communication variable, **83, 169**
 - INPUT_DATA communication variable, **83, 169**
 - INPUT_TAG communication variable, **83, 169**
 - install_rtos script, **102, 157, 160**
 - questions, answering, **161**
 - installation, **250**
 - HP 9000 software, **251-252**
 - Sun SPARCsystem, **253**
 - instrumented code, **5, 39, 54, 59-72**
 - commenting, **57**
 - guidelines, **57-58**
 - instrumented service call library, **255**
 - interface library, **60, 255**
 - intrusion, **57, 59**

- INVALID OPERAND error, **215**
- inverse assembler, **5, 18, 41-46, 72, 80, 87-102, 162, 256**
 - building, **45, 101**
 - installing, **45**
- inverse assembler code, **87-100**
- Inverse Assembly Language (IAL), **80**
- invocations (service call), show table, **136**

- K** keyDefs, X resource, **154**

- L** labels in IAL code, **170**
 - LD_ADDR_REL inverse assembler routine, **97**
 - LD_REL_32_BITS inverse assembler routine, **97**
 - LD_REL_TO_LONG inverse assembler routine, **98**
 - levels of RTOS measurements, **57**
 - LOAD instruction (IAL), **194**
 - local variables (IAL), **84**

- M** MAPPED_WIDTH instruction (IAL), **195**
- MARK_STATE instruction (IAL), **196**
- MAX_INSTRUCTION instruction (IAL), **197**
- memory calls, **121**
- memory usage, **72**
- Memory Usage, Emulator/Analyzer action key, **121**
- memory, extra locations, **76**
- message, from specific task to specific queue, **117**
- MISMATCHED PARENTHESIS error, **215**
- MISSING OPERATOR error, **215**
- monitor, emulation, **26**

- N** NEW_LINE instruction (IAL), **198**
- non-RTOS states, **105**
- NOP instruction (IAL), **199**
- number formats (IAL), **88**

- O** Only Call X, Emulator/Analyzer action key, **111**
- Only Calls X & Y, Emulator/Analyzer action key, **112**
- Only Queues, Emulator/Analyzer action key, **110**
- Only Task X, Emulator/Analyzer action key, **113-114**
- Only Tsk W,X,Y,Z, Emulator/Analyzer action key, **115**
- only, trace command option, **105**
- operating system versions supported, **251**
- OS overhead tracking, **57, 69**

- OUT_REL0_BINLEFT inverse assembler routine, **96**
- OUT_REL0_HEX inverse assembler routine, **96**
- OUT_REL0_HEXLEFT inverse assembler routine, **96**
- OUT_REL1_BINLEFT inverse assembler routine, **96**
- OUT_REL1_DECLEFT inverse assembler routine, **96**
- OUT_REL1_HEX inverse assembler routine, **96**
- OUT_REL1_HEX16 inverse assembler routine, **96**
- OUT_REL1_HEXLEFT inverse assembler routine, **96**
- OUT_REL2_BINLEFT inverse assembler routine, **97**
- OUT_REL2_HEX inverse assembler routine, **97**
- OUT_REL2_HEXLEFT inverse assembler routine, **97**
- OUT_REL3_BINLEFT inverse assembler routine, **97**
- OUT_REL3_HEX inverse assembler routine, **97**
- OUT_REL3_HEXLEFT inverse assembler routine, **97**
- OUT_REL4_HEX inverse assembler routine, **97**
- OUT_REL4_HEXLEFT inverse assembler routine, **97**
- OUT_REL5_HEX inverse assembler routine, **97**
- OUT_REL6_HEX inverse assembler routine, **97**
- OUTPUT instruction (IAL), **88, 95, 200**
- overflow, task time, **141**
- overhead (OS) tracking, **69**

- P** PATH environment variable, **25, 221, 231, 235, 237, 239**
- POSITION instruction (IAL), **201**
- predefined communication variables (IAL), **169**
- predefined IAL symbols, **170**
- PRINT_ASCII_CHAR inverse assembler routine, **96**
- PRINT_BINARY inverse assembler routine, **96**
- PRINT_NAME inverse assembler routine, **96**
- PRINT_NAME_AS_ASCII inverse assembler routine, **97**
- PRINT_NAME_IN_ACCUM inverse assembler routine, **96**
- PROC_RESOURCE environment variable, **155**
- PROCESSOR environment variable, **25**
- processor type, **25**
- product files (customized), installing, **160**

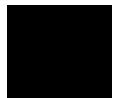
- Q** QUALIFY_MASK instruction (IAL), **202**
- QUALIFY_VALUE instruction (IAL), **202**

- R** re-scale histograms (SPA), **127**
- real-time runs, emulator restriction, **26**
- reinstall_rtos script, **162**
- REL_POSITION, inverse assembler variable, **90**

- release_system, end command option, **33**
 - requirements, **24**
 - reserved IAL symbols, **170**
 - RETURN instruction (IAL), **85, 203**
 - RETURN_FLAGS communication variable, **189**
 - ROTATE instruction (IAL), **204**
 - RTOS information, trace commands to capture, **107-121**
 - RTOS inverse assembler, **87-100**
 - building, **101-102**
 - installing, **101-102**
 - RTOS measurement tool
 - custom product, reinstalling, **162**
 - installing customized product files, **160**
 - overview, **4**
 - RTOS measurements, **104, 126**
 - automating, **146**
 - RTOS symbol names, **58**
 - rtos_edit script, **49, 221-223, 225-229**
 - rtos_emul, emulator startup script, **25, 80, 160-161, 230-236**
 - rtos_spa, SPA startup script, **29, 160-161, 237-240**
 - RTOS_UNIQUE environment variable. Users should permanently set the RTOS_UNIQUE environment variable in their startup sc, **228**
- S**
- s_break_ovrflw command file, **142**
 - s_countsrvcfs command file, **137**
 - s_counttasks command file, **135**
 - s_disabletrg2 command file, **143**
 - s_funcdur command file, **138**
 - s_init command file, **132**
 - s_init, SPA command file, **49**
 - s_taskuser command file, **139**
 - s_taskwindow command file, **136, 142**
 - s_timetasks command file, **133**
 - scripts, **47-49**
 - command files that use, **151**
 - create_12_call, **47, 241**
 - create_12_call32, **47, 243**
 - customize, **25**
 - get_task_number, **246**
 - install_rtos, **102, 157, 160**
 - reinstall_rtos, **162**
 - rtos_edit, **49, 221-223, 225-229**

- scripts (continued)
 - rtos_emul, startup, **157, 160-161, 230-236**
 - rtos_spa, startup, **157, 160-161, 237-240**
 - SEARCH_LIMIT instruction (IAL), **205**
 - selective tracking, **104, 110, 256**
 - service calls, **5, 55, 57, 60, 256**
 - error returns, **57**
 - show table of invocations, **136**
 - single call tracking, **110**
 - two call tracking, **112**
 - SET instruction (IAL), **206**
 - software performance analyzer, **5, 18-20, 29, 57, 59, 63, 69, 126, 256**
 - measurements, **55**
 - software versions, **251**
 - sort events (SPA), **127**
 - SPA command overview, **127**
 - SPA events
 - See* events (SPA)
 - spabasecmd, SPA command file, **48**
 - STACK ERROR error, **216**
 - stack pointers, **72**
 - stack usage, **57, 59, 72**
 - storage qualifiers in trace commands, **105**
 - STORE instruction (IAL), **207**
 - strings (IAL), **88**
 - SUBTRACT instruction (IAL), **208**
 - supported system versions, **251**
 - symbol map, **188-189**
 - symbol names, **58**
- T**
- table display of SPA events, **134**
 - table of service call invocations, **136**
 - table.c, demo OS data table, **38**
 - TAG_WITH instruction (IAL), **209**
 - Task A msg-Que X, Emulator/Analyzer action key, **117**
 - Task A: FuncX, Emulator/Analyzer action key, **118-119**
 - Task A: VarX, Emulator/Analyzer action key, **119-120**
 - task buckets, **18, 75-76**
 - Task Control Block, **72**
 - task events histogram, **139**
 - Task switch A-B, Emulator/Analyzer action key, **116**
 - task switch callout routine, **63, 72**

- task switches, **55, 57, 63, 69**
 - in memory call tracking, **121**
 - specific task switch tracking, **116**
- task time overflow, **141**
- tasks, **256**
 - four task tracking, **114**
 - single task tracking, **113**
 - SPA data for specific task, **135**
 - SPA event definition, **130**
 - time interval measurements, **129-137, 139**
- Tasks & Queues, Emulator/Analyzer action key, **111**
- TaskX: Servcalls, SPA action key, **136, 142**
- TEXT REPLACEMENT error, **216**
- time histogram display of task events, **133**
- time interval measurements, **129-137, 139**
- time overflow, task, **141**
- time profile measurements, **55**
- time slice, **69**
- time stamp, **122**
- Time Tasks, SPA action key, **133**
- trace commands, **107-121**
 - about option, **106**
 - after option, **106**
 - before option, **106**
 - disable option, **106**
 - display, **122-124**
 - enable option, **106**
 - find_sequence option, **106**
 - normal display, **123**
 - only option, **105**
 - overview, **105**
 - RTOS display, **124**
 - storage qualifier, **105**
- Track OS calls, Emulator/Analyzer action key, **109**
- tracking
 - memory, **72**
 - OS overhead, **69**
 - selective, **104, 110**
 - stack, **72**
- Trig2 on Overflow, SPA action key, **142**



Index

- trig2 signal, **140-141**
 - disabling, **143**
- Tsk & User Evnts, SPA action key, **139**
- TWOS_COMPLEMENT instruction (IAL), **210**
- type of processor, **25**
- U** UNDEFINED CONDITIONAL error, **216**
- UNDEFINED OPERATION CODE error, **216**
- UNDEFINED SYMBOL error, **216**
- unsigned comparisons (IAL), **84**
- user events histogram, **139**
- user-defined areas in data table, **74**
- V** variable
 - any task accessing a, **120**
 - specific task accessing a, **119**
- VARIABLE instruction (IAL), **88, 211**
- variables (IAL), **88, 90**
 - See also* communication variables (IAL)
- VMS, reserved IAL symbol, **170**
- W** WRITE_OUT_STRING inverse assembler routine, **96**

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.