IBM Parallel System Support Programs for AIX Event Management Programming Guide and Reference Version 2 Release 2

Document Number SC23-3996-00

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

First Edition (November 1996)

This edition applies to Version 2 Release 2 of the IBM Parallel System Support Programs for AIX (PSSP) Licensed Program, program number 5765-529, and to all subsequent releases and modifications until otherwise indicated in new editions. This is a new manual.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+914+432-9405
FAX (Other Countries):
Your International Access Code +1+914+432-9405

IBMLink (United States customers only): KGNVMC(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet e-mail: mhvrcfs@vnet.ibm.com
World Wide Web: http://www.rs6000.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Copyright International Business Machines Corporation 1996. All rights reserved. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices vii
Trademarks vii
Publicly Available Software viii

About This Book ix
Who Should Use This Book ix
How This Book Is Organized x
Typographic Conventions x
Obtaining Documentation x
Related Publications xii
SP Hardware Publications xii
SP Software Publications xii
IBM Virtual Shared Disk Publications xii
LoadLeveler Publications xii
Performance Toolbox (PTX) Publication xiii
Performance Monitor Publication xiii
Parallel Environment for AIX Publications xiii
PVMe for AIX Publications xiii
Client Input Output/Sockets (CLIO/S) Publications xiii
Parallel I/O File System Publications xiii
Network Tape Access and Control System for AIX (NetTAPE) Publications xiv
Other IBM Publications xiv
International Technical Support Organization Publications (Red Books) . . . xiv
AIX and RS/6000 Publications xiv
Service xv
Network Queueing System/MVS (NQS/MVS) xv
Network Connectivity xv
Adapters xv
Non-IBM Publications xvi
Parallel Computing xvi
Tcl xvi
Manual Pages for Public Code xvii

Chapter 1. Understanding Event Management 1
Introducing Event Management 1
The Event Management Components 1
Communicating across Nodes 3
Event Management and System Partitioning 3
How Can You Use Event Management Services? 4
Writing Resource Monitors 5
Defining the Resource Data 5
Choosing the Resource Monitor Type 12
Configuring the Resource Data and Resource Monitor 14
Coding and Testing the Resource Monitor 16
Writing Event Management Clients 20
Finding Out What Resource Data Is Available 20
Defining Predicates to Create Events 25
Coding and Testing the EM Client 27
Event Management Performance Considerations 30
Resource Monitors and PTX Shared Memory 30

© Copyright IBM Corp. 1996

DDS Shared Memory . . . 31
Releasing Shared Memory Manually . . . 31

Chapter 2. Event Management Configuration Data Reference . . . 33
Event Management Configuration Data (emcdb) . . . 34

Chapter 3. Event Management Subroutine Reference . . . 49
EMAPI Subroutine Summary . . . 49
RMAPI Subroutine Summary . . . 49
ha_em_end_session Subroutine . . . 51
ha_em_get_ecgid Subroutine . . . 53
ha_em_receive_response Subroutine . . . 55
ha_em_restart_session Subroutine . . . 66
ha_em_send_command Subroutine . . . 68
ha_em_start_session Subroutine . . . 77
ha_rr_add_var Subroutine . . . 80
ha_rr_del_var Subroutine . . . 83
ha_rr_end_session Subroutine . . . 86
ha_rr_get_ctrlmsg Subroutine . . . 88
ha_rr_get_interval Subroutine . . . 93
ha_rr_init Subroutine . . . 95
ha_rr_makserv Subroutine . . . 97
ha_rr_reg_var Subroutine . . . 99
ha_rr_send_val Subroutine . . . 102
ha_rr_start_session Subroutine . . . 104
ha_rr_terminate Subroutine . . . 107
ha_rr_touch Subroutine . . . 109
ha_rr_unreg_var Subroutine . . . 111

Chapter 4. Event Management Files Reference . . . 115
EMAPI Errors (err_emapi) . . . 116
ha_emapi.h File . . . 122
Predicates (haempred) . . . 138
Resource Variables (haemrviv) . . . 141
RMAPI Errors (err_rmapi) . . . 148
ha_rmapi.h File . . . 151

Chapter 5. Using the RMAPI: Some Resource Monitor Examples . . . 155
The rmapi_smpcmd.c Sample Program . . . 156
The rmapi_smpdae.c Sample Program . . . 167
The rmapi_smpsig.c Sample Program . . . 186
The rmapi_smp.msg Message File . . . 206
The rmapi_smp.loadsdr Shell Script . . . 208
The rmapi_smp.unloadsdr Shell Script . . . 214

Chapter 6. Using the EMAPI: Some Event Management Client Examples 217
The emapi_v01_ex01.c Sample Program . . . 218
The emapi_v01_ex02.c Sample Program . . . 238
The emapi_v01_ex03.c Sample Program . . . 258
The emapi_v01_ex04.c Sample Program . . . 278

Glossary of Terms and Abbreviations . . . 289

Index . . . 299

Figures

1. An Overview of the Event Management Components 3
2. Examples of Structured Byte String Definitions 8
3. Examples of Resource Variable Names and Instance Vectors 10
4. Examples of Resource Variable Classes 12
5. An Pseudocode Outline of a Command-Based Resource Monitor 17
6. A Pseudocode Outline of a Daemon-Based Resource Monitor 19
7. Sample Output from PSSP SDRGetObjects commands 22
8. Using the Event Perspective to Get Resource Variable Information . . . 24

© Copyright IBM Corp. 1996

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	NQS/MVS
AIX/600	POWERparallel
DATABASE 2	POWERserver
DB2	RS/6000
ES/9000	RS/6000 Scalable POWERparallel Systems
ESCON	Scalable POWERparallel Systems
HACMP/6000	SP
IBM	System/370
IBMLink	System/390
LoadLeveler	TURBOWAYS

© Copyright IBM Corp. 1996

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

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Publicly Available Software

This product includes software that is publicly available:

Amd Berkeley Software Distribution automount daemon

expect Programmed dialogue with interactive programs

Kerberos Provides authentication of the execution of remote commands

NTP Network Time Protocol

Perl Practical Extraction and Report Language

SUP Software Update Protocol

Tcl Tool Command Language

TclX Tool Command Language Extended

Tk Tcl-based Tool Kit for X-windows

This book discusses the use of these products only as they apply specifically to the SP system. The distribution for these products includes the source code and associated documentation. (Kerberos does not ship source code.) **/usr/lpp/ssp/public** contains the compressed **tar** files of the publicly available software. (IBM has made minor modifications to the versions of Tcl and Tk used in the SP system to improve their security characteristics. Therefore, the IBM-supplied versions do not match exactly the versions you may build from the compressed **tar** files.) All copyright notices in the documentation must be respected. You can find version and distribution information for each of these products that are part of your selected install options in the **/usr/lpp/ssp/README/ssp.public.README** file.

About This Book

This book contains conceptual, guidance, and reference information that will help you write programs that use the Event Management APIs provided by PSSP. Specifically, this book contains information to help you write the following types of programs:

- Resource monitors, using the Resource Monitor Application Programming Interface (RMAPI)
- Event Management client programs, using the Event Management Application Programming Interface (EMAPI).

In addition, this book contains information about how to configure the Event Management subsystem.

This book applies to PSSP Version 2 Release 2. To find out what version of PSSP is running on your system, enter the following:

```
SDRGetObjects Syspar syspar_name code_version
```

In response, the system displays something similar to:

```
syspar_name  code_version
Part_1       PSSP-2.2
```

If the response indicates **PSSP-2.2**, this book applies to the version of PSSP that is running on your system.

Who Should Use This Book

This book is intended primarily for programmers of applications that manage system resources (which may or may not be subsystems) who want to use Event Management services to make their applications highly available. This book contains information for programmers who want to write new resource monitors, write new clients that use the EMAPI, or add the use of Event Management services to existing programs.

It assumes that you are an experienced C programmer and have a thorough understanding of RS/6000 Scalable POWERparallel Systems (SP) hardware, IBM Parallel System Support Programs for AIX (PSSP) software, and AIX operating system fundamentals. In particular, you should be familiar with the operation of the SP control workstation, PSSP system partitions, and the PSSP System Data Repository (SDR). It also assumes that you are familiar with the management of system resources and recovery concepts.

The Event Management subsystem requires the Performance Aide component of the Performance Toolbox for AIX (PTX), Version 2 Release 2 (5765-654) licensed product to be installed on every SP node. The Performance Aide component is included with the SP hardware at no charge. This book assumes a thorough understanding of PTX programming concepts, which are discussed in the *Performance Toolbox for AIX Guide and Reference*, SC23-2625.

System administrators will need to use the information about configuring the Event Management subsystem.

© Copyright IBM Corp. 1996

The commands and interfaces described in this book require that you have the appropriate privileges and authorizations. For example, you may need to be running with an effective user ID of **root**. The specific security requirements for each subroutine and command are listed in the reference material.

How This Book Is Organized

This book contains chapters that:

- Introduce you to Event Management concepts
- Provide reference material for configuring the Event Management subsystem
- Provide reference material for the EMAPI and RMAPI subroutines, header files, and errors
- Contain example programs to help you get started in writing your own programs.

The back of the book includes a glossary and an index.

Typographic Conventions

This book uses the following typographic conventions:

Typographic	Usage
Bold	• **Bold** words or characters represent system elements that you must use literally, such as commands, flags, and path names. • **Bold** words also indicate the first use of a term included in the glossary.
Italic	• *Italic* words or characters represent variable values that you must supply. • *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices. (In other words, it means “or.”)
< >	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <**Enter**> refers to the key on your terminal or workstation that is labeled with the word Enter.
...	An ellipsis indicates that you can repeat the preceding item one or more times.
<**Ctrl-***x*>	The notation <**Ctrl-***x*> indicates a control character sequence. For example, <**Ctrl-c**> means that you hold down the control key while pressing <**c**>.

Obtaining Documentation

This book is available in PostScript format. You can obtain PostScript files for the product documentation by:

- Downloading them from the **lscftp.pok.ibm.com** anonymous FTP server. The files for PSSP can be found in the directory called /**pub**/**pps**/**pssp**.

- Printing the PostScript documentation files that are shipped on the product media.

You can also view or download this book from the IBM RS/6000 web site at **http://www.rs6000.ibm.com**.

Related Publications

SP Hardware Publications

- *IBM RS/6000 Scalable POWERparallel Systems: Site Planning*, GC23-3905
- *IBM RS/6000 Scalable POWERparallel Systems: Maintenance Information, Volume 1, Installation and CE Operations*, GC23-3903
- *IBM RS/6000 Scalable POWERparallel Systems: Maintenance Information, Volume 2, Maintenance Analysis Procedures and Parts Catalog*, GC23-3904
- *IBM RS/6000 Scalable POWERparallel Systems: Library Guide*, GC23-3868
- *IBM RS/6000 Scalable POWERparallel Systems: Installation Aid Licensed Program Specification*, GC23-3838

SP Software Publications

- *IBM Parallel System Support Programs for AIX: System Planning*, GC23-3902
- *IBM Parallel System Support Programs for AIX: Administration Guide*, GC23-3897
- *IBM Parallel System Support Programs for AIX: Installation and Migration Guide*, GC23-3898
- *IBM Parallel System Support Programs for AIX: Diagnosis and Messages Guide*, GC23-3899
- *IBM Parallel System Support Programs for AIX: Command and Technical Reference*, GC23-3900
- *IBM Parallel System Support Programs for AIX: Event Management Programming Guide and Reference*, SC23-3996
- *IBM Parallel System Support Programs for AIX: Group Services Programming Guide and Reference*, SC28-1675
- *IBM Parallel System Support Programs for AIX: Licensed Program Specification*, GC23-3901

As an alternative to ordering the individual books, you can use SBOF-8587 to order the entire SP software library.

IBM Virtual Shared Disk Publications

- *IBM Parallel System Support Programs for AIX: Managing Shared Disks: IBM Virtual Shared Disks, Data Striping Devices, and the IBM Recoverable Virtual Shared Disk Licensed Program*, SC23-3849
- *IBM Recoverable Virtual Shared Disk: Licensed Program Specification*, GC23-3848

LoadLeveler Publications

- *Using and Administering LoadLeveler*, SC23-3989
- *IBM LoadLeveler: Licensed Program Specification*, GH23-0040

Performance Toolbox (PTX) Publication

- *IBM Performance Toolbox Version 1.2 and 2 for AIX: Guide and Reference*, SC23-2625

Performance Monitor Publication

- *IBM Performance Toolbox Parallel Extensions for AIX: Guide and Reference*, SC23-3997

Parallel Environment for AIX Publications

- *IBM Parallel Environment for AIX: Hitchhiker's Guide*, GC23-3895
- *IBM Parallel Environment for AIX: Operation and Use, Volume 1*, SC28-1979
- *IBM Parallel Environment for AIX: Operation and Use, Volume 2*, SC28-1980
- *IBM Parallel Environment for AIX: Installation Guide*, GC28-1981
- *IBM Parallel Environment for AIX: Messages*, GC28-1982
- *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*, GC23-3893
- *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, GC23-3894
- *IBM Parallel Environment for AIX: Licensed Program Specification*, GC23-3896

As an alternative to ordering the individual books, you can use SBOF-8588 to order the entire Parallel Environment for AIX library.

PVMe for AIX Publications

- *IBM PVMe for AIX: Fact Sheet*, GC23-3907
- *IBM PVMe for AIX: User's Guide and Reference*, GC23-3884
- *IBM PVMe for AIX: Licensed Program Specification*, GC23-3885

Client Input Output/Sockets (CLIO/S) Publications

- *IBM CLIO/S General Information*, GC23-3879
- *IBM CLIO/S Licensed Program Specification*, GC23-3789
- *IBM CLIO/S User's Guide and Reference*, GC28-1676

Parallel I/O File System Publications

- *IBM AIX Parallel I/O File System: Fact Sheet*, G325-0649
- *IBM AIX Parallel I/O File System: Installation, Administration, and Use*, SH34-6065
- *IBM AIX Parallel I/O File System: Licensed Program Specification*, GH34-6066

Network Tape Access and Control System for AIX (NetTAPE) Publications

- *IBM NetTAPE General Information*, GC23-3990
- *IBM NetTAPE User's Guide and Reference*, available from your IBM representative

---

Other IBM Publications

International Technical Support Organization Publications (Red Books)

- *IBM International Technical Support Centers Implementing High Availability on RS/6000 SP*, SG24-4742
- *IBM International Technical Support Centers RS/6000 SP High Availability Infrastructure*, SG24-4838
- *IBM International Technical Support Centers RS/6000 SP PSSP 2.2 Technical Presentation*, SG24-4868

AIX and RS/6000 Publications

- *IBM AIX Version 4 Getting Started*, SC23-2527
- *IBM General Concepts and Procedures for RS/6000*, GC23-2202
- *IBM AIX Version 4 Files Reference*, SC23-2512
- *IBM AIX Version 4 System Management Guide: Communications and Networks*, SC23-2526
- *IBM AIX Version 4.1 Installation Guide*, SC23-2550
- *IBM AIX Version 4.2 Installation Guide*, SC23-1924
- *IBM AIX Version 4 Commands Reference*, SBOF-1851
- *IBM AIX Versions 3.2 and 4 Performance Tuning Guide*, SC23-2365
- *IBM AIX Version 4 Problem Solving Guide and Reference*, SC23-2606
- *IBM AIX Version 4 Messages Guide and Reference*, SC23-2641
- *IBM AIX Version 4.1 Network Installation Management Guide and Reference*, SC23-2627
- *IBM AIX Version 4.2 Network Installation Management Guide and Reference*, SC23-1926
- *IBM AIX Version 4 System Management Guide: Operating System and Devices*, SC23-2525
- *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*, SC23-2533
- *IBM AIX Version 4 Communications Programming Concepts*, SC23-2610
- *Diskless Workstation Management Guide*, SC23-2433
- *C++ for AIX/6000: Language Reference*, SC09-1606
- *C++ for AIX/6000: Standard Class Library Reference*, SC09-1604
- *C++ for AIX/6000: User's Guide*, SC09-1605

- *IBM Performance Toolbox Version 1.2 and 2 for AIX: Guide and Reference*, SC23-2625

Service

- *RS/6000 Problem Solving Guide*, SC23-2204
- *Common Diagnostics Micro Channel*, SA23-2765
- *7012 300 Series Operator Guide*, SA23-2623
- *7012 300 Series Installation and Service Guide*, SA23-2624
- *7013 500 Series Operator Guide*, SA23-2621
- *7013 500 Series Installation and Service Guide*, SA23-2622
- *Electrical Safety for IBM Service Representatives*, S229-8124
- *Service Information Binder*, SX33-6060
- *7015 Model R30 CPU Operator Guide*, SA23-2742
- *7015 Model R30 CPU Service Guide*, SA23-2743
- *Supplemental Information for 7012 G Series Models, 7013 J Series Models, 7015 Models R30 and R40* (No order number — this book comes as part of the ship group with the SP hardware)

Network Queueing System/MVS (NQS/MVS)

- *IBM NQS/MVS Installing and Administering*, SC23-0232
- *IBM NQS/MVS Client User's Guide*, SC23-0231
- *IBM NQS/MVS General Information*, GC23-3684
- *IBM NQS/MVS Licensed Program Specifications*, GC23-0233

Network Connectivity

- *IBM LAN Cabling System Planning and Installation Guide*, GA27-3361
- *IBM Cabling System Optical Fiber Planning and Installation Guide*, GA27-3943
- *IBM 8250/8260/8285 Planning and Site Preparation Guide*, GA33-0285
- *IBM 6611 Network Processor: Introduction and Planning Guide*, GK2T-0334

Adapters

- *FDDI Introduction and Planning Guide*, GA27-3892
- *FDDI User's Guide and Programming Reference*, SC28-2823
- *Block Multiplexer Channel Adapter User's Guide and Service Information*, SC23-2427
- *Planning for Fiber Optic Channel Links*, GA23-0367
- *Token-Ring Network Introduction and Planning Guide*, GA27-3677
- *RS/6000 Token Ring Adapter Card*, G511-1681
- *HiPPI User's Guide and Programmer's Reference*, SA23-0369 and SA23-2488
- *AIX Parallel and ESCON Channel Tape Attachment/6000 Installation and User's Guide*, GA32-0311

- *9334 SCSI Expansion Units Operator Guide*, GA33-3232
- *9334 SCSI Model 010 Expansion Unit: Installation and Service Guide*, SY33-0165
- *9334 SCSI Model 500 Expansion Unit: Installation and Service Guide*, SY33-0167
- *SCSI-2 Fast/Wide Adapter*, SC23-2646
- *IBM SCSI-2 Fast/Wide Adapter/A Technical Reference*, S83G-7545
- *Turboways 100 User's Guide ATM*, GA27-4057
- *9333 Model 010 and 011 High-Performance Disk-Drive Subsystem Operator Guide*, GA33-3208
- *9333 Model 010 and 011 High-Performance Disk-Drive Subsystem Installation and Service Guide*, SY33-0161
- *9333 Model 010 and 011 High-Performance Disk-Drive Subsystem Hardware Technical Information*, SA33-3209
- *9333 Model 500 and 501 High-Performance Disk-Drive Subsystem Operator Guide*, GA33-3234
- *9333 Model 500 and 501 High-Performance Disk-Drive Subsystem Installation and Service Guide*, SY33-0168
- *9333 Model 500 and 501 High-Performance Disk-Drive Subsystem Hardware Technical Information*, SA33-3235
- *IBM SCSI Tape Drive, Medium Changer, and Library Device Drivers Installation and User's Guide*, GC35-0154

Non-IBM Publications

Parallel Computing

- Almasi, G., Gottlieb, A., *Highly Parallel Computing*, Benjamin-Cummings Publishing Company, Inc., 1989.
- Gropp, W., Lusk, E., Skjellum, A., *Using MPI*, The MIT Press, 1994.
- Message Passing Interface Forum, MPI: *A Message-Passing Interface Standard*. Version 1.1, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Foster, I., *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- Bergmark, D., Pottle, M. *Optimization and Parallelization of a Commodity Trade Model for the SP1*. Cornell Theory Center, Cornell University, June, 1994.
- Phister, Gregory, F., *In Search of Clusters*. Prentice Hall, 1995.

Tcl

- Ousterhout, John K., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-63337-X.

Manual Pages for Public Code

The following manual pages for public code are available in this product:

SUP	/usr/lpp/ssp/man/man1/sup.1
Amd	/usr/lpp/ssp/man/man8/amd.8
	/usr/lpp/ssp/man/man8/amq.8
NTP	/usr/lpp/ssp/man/man8/xntpd.8
	/usr/lpp/ssp/man/man8/xntpdc.8
Perl (Version 4.036)	/usr/lpp/ssp/perl/man/perl.man
	/usr/lpp/ssp/perl/man/h2ph.man
	/usr/lpp/ssp/perl/man/s2p.man
	/usr/lpp/ssp/perl/man/a2p.man
Perl (Version 5.002)	Man pages are in the /usr/lpp/ssp/perl5/man/man1 directory

Manual pages and other documentation for Tcl, TclX, Tk, and expect can be found in the compressed **tar** files located in **/usr/lpp/ssp/public**.

Chapter 1. Understanding Event Management

This chapter introduces you to the concepts and operation of the Event Management subsystem, and the use of the Event Management application programming interfaces (APIs) to write resource monitors and Event Management client programs. It briefly discusses some performance considerations to keep in mind as you write resource monitors and EM clients. Finally, it provides some information about resource monitors and shared memory.

Introducing Event Management

The Event Management subsystem is a distributed subsystem of PSSP that provides a set of high availability services for the RS/6000 Scalable POWERparallel Systems (SP). By matching information about the state of system resources with information about resource conditions that are of interest to client programs, it creates events. Client programs can use events to detect and recover from system failures, thus enhancing the availability of the SP system.

The Event Management Components

Event Management operates through the cooperation of three software components:

- Resource monitors, which supply information about the current state of system resources.
- Event Management clients (EM clients), which supply information about which of those resource states are of interest.
- The Event Management subsystem, which puts the two together. It compares a condition of interest to a current state of a resource. If the state meets the test, the Event Management subsystem generates an **event** and notifies the EM client that the event has occurred.

Let's take a closer look at the roles of each of these components.

Resource Monitors

The first Event Management component is the resource monitor. A **resource monitor** supplies information about the resources in the system. A **system resource** is an object in the system that provides services to system users. Examples of system resources are nodes, memory, and disks, application programs, data bases, and file systems.

Resource monitors keep track of system resources and supply information about the characteristics or attributes of system resources to the Event Management subsystem. For example, a resource monitor might report whether a node is up or down, the percentage of free memory for a particular node, the percentage of space on a particular disk on a particular node that is free, whether a particular program or process is running, and so on.

For each attribute of a resource, the resource monitor defines a **resource variable**.

Of course, there may be more than one copy of a resource in the system. For example, an SP system may have multiple nodes, a node may have multiple disks,

© Copyright IBM Corp. 1996

a program may have multiple processes, and a database may have multiple partitions. Accordingly, the resource monitor reports the value of each copy of the resource variable in the system. Each copy is called an **instance** of the resource variable. So, if a node has three disks, for example, the resource monitor that monitors the percentage of free disk space would report the values of three instances of the percentage-of-free-disk-space resource variable for that node.

The resource monitor reports the value of each resource variable instance to the Event Management subsystem at defined times. The reporting occurs either periodically at configurable intervals or every time the attribute changes, depending on the type of resource variable.

A resource monitor can be a command, a daemon, or part of an application or subsystem that manages any type of system resource. It uses the Resource Monitor Application Programming Interface (RMAPI) to report its data to and receive control information from the Event Management subsystem.

Event Management Clients

The second Event Management component is the Event Management client. An **Event Management client (EM client)** acts on information about the condition of system resources. Typical actions include recovery from system failures and reporting system changes.

Different conditions are of interest to different EM clients. For example, one EM client may want to know when the percentage-of-free-disk-space for any disk on any node in the system falls below 10%. Another EM client may want to know when a node in the system fails. The disk space client might allocate more space or send a note to a data administrator about the condition. The node health client might initiate a program that causes a backup node to take over the work of the failed node.

A condition in which an EM client is interested is called a **predicate**. An EM client indicates that it wishes to be notified when the state of a resource instance matches a certain condition by registering a predicate for particular resource variable instances.

An EM client is an application program or a subsystem. It uses the Event Management Application Programming Interface (EMAPI) to register interest in particular events, to receive information when the events occur, and to query resource variable values and definitions.

The Event Management Subsystem

The last component, and the one that ties the other two together, is the Event Management subsystem. The **Event Management subsystem** communicates with the resource monitors and EM clients. It receives the values of resource variable instances as they are reported by the resource monitors. It keeps track of the resource variables and predicates for which EM clients have registered.

At configurable intervals (or every time that values are reported, depending on the type of resource variable), the Event Management subsystem **observes** the values of resource variable instances. This means that it applies a predicate to the value of the associated resource variable instances. When a predicate evaluates to true (that is, when the value of a resource variable instance matches the condition expressed by the predicate), the Event Management subsystem generates an event

and notifies the appropriate EM client. The EM client then takes action appropriate to the event.

Figure 1 shows the relationship between the three basic Event Management components.

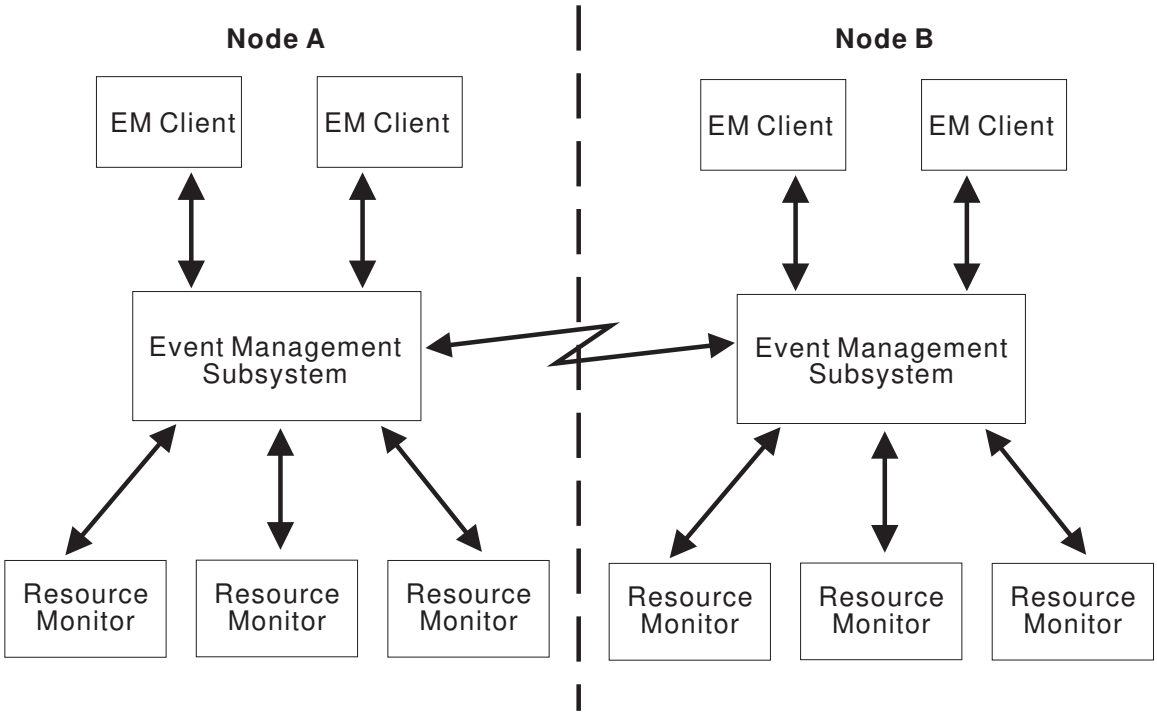

Figure 1. An Overview of the Event Management Components

Communicating across Nodes

As Figure 1 shows, a resource monitor provides data to an instance of the Event Management subsystem that is running on the same node as the resource monitor. An EM client communicates with the instance of the Event Management subsystem that is running on the same node as the client. An instance of the Event Management subsystem on one node communicates with instances of the Event Management subsystem on other nodes to provide events to its local clients. Therefore, an EM client can register for and receive events about any resource in the SP system. In addition, an EM client that is executing outside of the SP system can remotely communicate with the Event Management subsystem that is located on the control workstation.

Event Management and System Partitioning

The Event Management subsystem operates in a system-wide domain, where "system" means an SP system partition. If the SP consists of only one system partition, then "system" refers to the entire SP.

Instances of the Event Management subsystem communicate only with other instances of the Event Management subsystem that are running in the same system partition.

On the control workstation, there is one instance of the Event Management subsystem per system partition.

System partitioning is optionally visible to EM clients. By default, a client starts a session with the Event Management subsystem in the current system partition. The current system partition is the system partition to which the node on which the client is running belongs. If the EM client is executing on the control workstation or on a workstation outside of the SP, the current system partition is determined from the value of the **SP_NAME** environment variable. However, by specifying the system partition name, a client can start a session with the Event Management subsystem in any partition. To receive events from, and to issue queries to, other system partitions, the client must start a session with the Event Management subsystem in each other system partition.

Multiple sessions may also be started with any single system partition. This provides multiple communication paths to the Event Management subsystem in a single system partition. Multiple paths are useful if the client wishes to ignore certain responses for a time while accepting others. For example, a client can register events in one session and issue queries in another.

How Can You Use Event Management Services?

PSSP provides the following interfaces for using event management services:

- The RMAPI, an application programming interface that resource monitors use to communicate information about resource states to the Event Management subsystem
- The EMAPI, an application programming interface that EM clients use to get information about resource state changes from the Event Management subsystem
- The Scalable POWERparallel Perspectives for AIX (SP Perspectives), a graphical user interface (GUI) that you can use to perform SP system management tasks, one of which is managing system events.

Using the RMAPI

Programmers can use the RMAPI to write standalone C programs that monitor resources or to incorporate resource monitoring function into existing subsystems or applications. For example, a distributed database program can use the RMAPI to supply information about the state of its resources.

This book provides guidance and reference information to help you use the RMAPI.

Using the EMAPI

Programmers can use the EMAPI to write standalone C programs that take appropriate action when monitored resources change state. For example, an EM client that is notified when a resource becomes unavailable can initiate actions to recover the resource or circumvent the failure.

This book provides guidance and reference information to help you use the EMAPI.

Using SP Perspectives to Monitor Events

System administrators and operators can use the Event Management portion of SP Perspectives to receive notice when resources of interest change state and run shell scripts that take recovery action. Detailed panels help you use the SP Perspectives graphical user interface to perform event management and other system management tasks.

For information on using SP Perspectives, see the *PSSP Administration Guide*.

Writing Resource Monitors

To write a resource monitor, you must perform the following steps:

- Define the resource data you want to monitor
- Choose the type of resource monitor you are going to write
- Configure the resource data and the resource monitor to the system
- Code and test the resource monitor.

This section provides you with an overview of each of these steps. For complete details on how to specify any of the data or subroutines, see the reference information in the following chapters:

- Chapter 2, “Event Management Configuration Data Reference” on page 33
- Chapter 3, “Event Management Subroutine Reference” on page 49
- Chapter 4, “Event Management Files Reference” on page 115.

For details on any commands you need to issue, see *PSSP Command and Technical Reference*.

For listings of several sample resource monitor programs, see Chapter 5, “Using the RMAPI: Some Resource Monitor Examples” on page 155.

Defining the Resource Data

The first task in writing a resource monitor is to define the data that it is to collect and send to the Event Management subsystem. To do that, you must understand the following terms as they are used by the Event Management subsystem:

- Resources
- Resource variables
- Resource variable names and descriptions
- Resource variable value types and data types
- Resource variable instances and instance vectors
- Dynamically instantiable resource variables
- The location of a resource variable instance
- Resource variable classes.

Resources

A **resource** is an entity in the system that provides a set of services. Examples of resources include hardware entities such as processors, disk drives, memory, and adapters, and software entities such as database applications, processes, and file systems. Each resource in the system has one or more attributes that define the state of the resource. The number of attributes and the semantics of each attribute are defined by the resource.

The purpose of the Event Management subsystem is to inform interested programs whenever the state of a resource changes in some way. The resource state change is reflected by a change in a resource attribute. To perform this function without having to understand the semantics of resource attributes, each attribute is represented by a variable. This variable is called a resource variable.

Resource Variables

A **resource variable** is the representation of an attribute of a resource in the Event Management subsystem. Associated with each resource variable is a name, a description, a value type, a data type, an instance vector and, optionally, a location. Resource variables are grouped together into classes.

Resource Variable Names

A resource variable is identified by a **resource variable name**, which is a string that consists of a resource name followed by a period followed by the resource attribute.

By convention, the **resource name** consists of two or more components that include the name of the vendor and the name of the product that supplies the subsystem or application that manages the resource. The resource name may include additional components as needed to further specify the resource.

The **resource attribute** is a single component that follows the resource name.

The specification of components must follow certain rules, such as the type of character that is allowed in each component. For complete details on the specification of resource variable names, see “Event Management Configuration Data (emcdb)” on page 34. For examples of resource variable names, see Figure 3 on page 10.

Resource Variable Description

A **resource variable description** provides information about the resource variable, in particular, its semantics. It should contain enough information so that writers of EM clients will be able to define meaningful predicates.

Resource Variable Value Types

A resource variable can have one of the following value types: Counter, Quantity, or State.

A **Counter** is a resource variable that represents a rate. It can have a data type of either long or float. The long and float formats are identical to the C language types of the same names. Typically, a Counter represents throughput. Examples include paging rates, I/O rates, and transaction rates.

By default, the Event Management subsystem presents the value of an instance of a Counter as a rate. The rate represents the change in the actual contents of the

instance of the Counter from one observation to the next divided by the time between the two observations. However, the Event Management subsystem can also present the **raw value** of the instance of the Counter, which is the actual contents of the instance of the Counter from the latest observation. The raw value of a Counter always increases over time.

For example, the resource variable called **IBM.PSSP.aixos.Mem.Virt.pagein** is a Counter that represents the rate at which pages are paged into virtual memory. Normally, its value is presented as a rate (number of pages per second). However, its raw value indicates the total number of pages that have been paged in since the Counter was initialized.

A **Quantity** is a resource variable whose value fluctuates over time. It can have a data type of either long or float. The long and float formats are identical to the C language types of the same names. Typically, a Quantity represents a level. Examples include the size of real memory or paging space, the number of processes in a queue, or the percentage of free disk space.

The Counter and Quantity value types have the same definition as the value types for statistics objects that are defined by the Performance Toolbox for AIX (PTX/6000) System Performance Measurement Interface (SPMI). Counters and Quantities can be reported to the Performance Monitor subsystem at the same time as they are reported to the Event Management subsystem. Instances of these two types of resource variable can be updated frequently by the resource monitor for the purpose of monitoring performance and observed relatively infrequently by the Event Management subsystem without losing the ability to generate useful events.

A **State** is a variable whose value fluctuates over time. In addition, every fluctuation is significant and must be observed; otherwise, meaningful information would be lost. A State variable typically represents any attribute other than throughput or a level. A simple example of a State is an indicator of whether a node is up or down. For more complicated examples of State resource variables, see Figure 2 on page 8.

Values of a State variable can be reported only to the Event Management subsystem. They cannot be reported to the Performance Monitor subsystem. The Event Management subsystem observes instances of a State variable when they are reported. While a resource monitor can update a State variable instance rapidly, on average, instances of State variables are expected to change at a relatively slow rate.

Resource Variable Data Types

Each of the value types (Counter, Quantity, and State) can have a data type of long or float. The long and float formats are identical to the C language types of the same names.

In addition, a State can have a data type of **structured byte string (SBS)**. An SBS is a string of bytes that consists of an SBS length field followed by one or more structured fields. The SBS length specifies the total length of the structured fields that follow it.

A structured field consists of a header followed by a value. The structured field header contains the length and data type of the structured field value and a serial

number. A structured field may have a data type of long, float, null-terminated character string, or byte string.

The serial number is a unique value that identifies the structured field. Within an SBS, the serial numbers are numbered sequentially, starting with 0.

Associated with each structured field in an SBS, but not included in the actual SBS, is a structured field name. This name contains a short description of the structured byte field.

Figure 2 shows some examples of structured byte string definitions.

```
------------------------------------------------------------------
SBS associated with resource variable IBM.PSSP.Prog.pcount

SBS Length = variable
SBS Fields
    Field Name   Field Length   Field Type         Field Serial Number
    CurPIDCount    4               long              0
    PrevPIDCount   4               long              1
    CurPIDList     variable        cstring           2

------------------------------------------------------------------
SBS associated with resource variable IBM.PSSP.pm.Errlog

SBS Length = variable
SBS Fields
    Field Name          Field Length   Field Type    Field Serial Number
    sequenceNumber      variable       cstring          0
    errorID             variable       cstring          1
    errorClass          variable       cstring          2
    errorType           variable       cstring          3
    alertFlagsValue     variable       cstring          4
    resourceName        variable       cstring          5
    resourceType        variable       cstring          6
    resourceClass       variable       cstring          7
    errorLabel          variable       cstring          8
```

Figure 2. Examples of Structured Byte String Definitions

Resource Variable Instances and Instance Vectors

Most resources in the system have multiple copies. For example, there is more than one logical volume per node, more than one processor per node if the node is an SMP node, more than one pool of kernel memory buffers, more than one partition in a parallel database, more than one node in the system partition, and so on. Each of these copies is an **instance** of the resource.

The resource variables that represent the states of these resources also have multiple copies. Each of these copies is an **instance** of the resource variable.

To uniquely identify each copy of a resource and all of its variables, each resource in the system has one, and only one, associated instance vector.

An **instance vector** is a list of elements, where each element is a name/value pair. A name/value pair consists of an instance vector element name followed by an equal sign followed by the value of the vector element. In the EMAPI, the elements

are separated by semicolons. In the RMAPI, the elements are separated by commas.

An **instance vector element name** is a string that describes the element.

For the EMAPI, an **instance vector element value** is a single value, a range of values, a comma-separated list of single values, or a comma-separated list of ranges. A range takes the form *a-b* and is valid only for vector elements of type integer.

For the RMAPI, an **instance vector element value** is a single value.

The instance vector of a resource specifies a particular instance of a resource and all of its resource variables within the system. To identify an instance, an instance vector can contain up to 4 elements, as defined by the **HA_EM_VECTOR_SIZE** constant.

For example, a variable that represents the number of used blocks in the /**tmp** file system requires an instance vector with three elements: the node ID, the volume group name, and the logical volume name. A variable that represents the power state of a node requires a single element: the node ID.

An instance vector element may be wildcarded in a manner that varies by subroutine.

The set of values in the instance vector uniquely identifies the copy of the resource in the system. By extension, they also uniquely identify the copy of the resource variable in the system. If there is only one copy of the resource in the system (for example, the control workstation), its instance vector is null.

The semantics of an instance vector are defined by the semantics of the resource variable with which it is associated. The names of the vector elements must be unique within a given resource variable's instance vector.

Figure 3 on page 10 shows some examples of resource variable names and instance vectors.

```
Resource Variable Name              Instance Vector

IBM.PSSP.aixos.CPU.gluser           NodeNum=5
IBM.PSSP.aixos.CPU.kern             NodeNum=5;Cpu=cpu0
IBM.PSSP.aixos.Mem.Real.size        NodeNum=5
IBM.PSSP.aixos.Mem.Virt.pagein      NodeNum=5
IBM.PSSP.aixos.Mem.Virt.pageout     NodeNum=5
IBM.PSSP.aixos.Mem.Kmem.inuse       NodeNum=5;Type=mbuf
IBM.PSSP.aixos.PagSp.%totalfree     NodeNum=5
IBM.PSSP.aixos.Disk.busy            NodeNum=5;Name=hdisk0
IBM.PSSP.aixos.Disk.busy            NodeNum=5;Name=hdisk1
IBM.PSSP.aixos.VG.free              NodeNum=5;VG=rootvg
IBM.PSSP.aixos.VG.free              NodeNum=5;VG=spdata
IBM.PSSP.aixos.FS.%totused          NodeNum=5;VG=rootvg;LV=hd4
IBM.PSSP.aixos.FS.%totused          NodeNum=5;VG=spdata;LV=lv00
IBM.PSSP.SP_HW.Node.powerLED        NodeNum=5
IBM.PSSP.SP_HW.Frame.frACLED        FrameNum=1
IBM.PSSP.SP_HW.Switch.powerLED      SwitchNum=1
IBM.PSSP.Membership.Node.state      NodeNum=5
IBM.DB2.Part.size                   DBname=foo;Part=3
```

Figure 3. Examples of Resource Variable Names and Instance Vectors

In Figure 3, although most instance vectors contain a node number, some resource variables are not associated with a particular node.

Although components of a resource name can be used in more than one name, resource names must uniquely identify a resource.

Dynamically Instantiable Resource Variables

In most cases, the instances of a resource that a resource monitor is responsible for tracking and reporting on are static or relatively so. Thus, a resource monitor knows from the time it starts running, after any necessary initialization occurs, all of the instances of the resources it monitors. For example, after any necessary initialization, a hardware monitor would know what frames there are in the system, a node health monitor would know what nodes there are in the system, and a disk monitor would know what disks there are in the system.

However, in some cases, a resource monitor knows how to track and report on a type of resource, but the resource monitor does not know all of the possible instances of the resource that may occur. For example, the program resource monitor knows how to track and report on a program that is running in a system partition, but it does not know in advance the names of all possible programs that can run. For a resource like this, you can define the resource variable as **dynamically instantiable**.

If a resource variable is dynamically instantiable, the resource monitor does not register all of the instances of its resource variables. Instead, it waits for the Event Management subsystem to tell it what instances to register. As the Event Management subsystem receives requests from EM clients for events based on instances of dynamically instantiable variables, the Event Management subsystem passes the instances to the resource monitor in a control message with an "instantiate variables" command.

Location of a Resource Variable Instance

The **location** of a resource variable instance is the node on which the *resource monitor* that supplies the instance resides. Note that the resource monitor may or may not reside on the same node as the instance of the resource.

For example, in Figure 3 on page 10, the resource variables that begin with **IBM.PSSP.SP_HW** represent hardware resources that are monitored by the hardware monitor subsystem. A resource monitor that connects to the **hardmon** daemon can reside on any node of the SP. It can even reside on an RS/6000 outside of the SP, such as the control workstation. Thus, a resource monitor on a single node supplies all instances of **IBM.PSSP.SP_HW** resource variables. Accordingly, the location fields in all of the instances specify that node.

As another example, in Figure 3 on page 10, the resource variable named **IBM.DB2.Part.size** represents the size of a partition in a DB2 parallel database. In this case, the resource variable instance originates on the node that contains the database partition. The database partition itself may move from one node to another if recovery of the database is required.

If an instance vector implies the location of a resource variable, the description of the resource variable should so specify.

Resource Variable Classes

The **class** of a resource variable indicates the subsystem or application that manages the associated resource. By convention, class names follow the same naming conventions that are used for resource names: the first component is a vendor name, the second component is a product name, and subsequent components, if any, further specify the resource class. Class names must be unique among all of the class names defined for a system partition.

Grouping resource variables by class can also be used to set the observation interval for resource variables of value type Counter and Quantity. All instances of all variables of these value types in the same class are observed at the same intervals.

For examples of class names, see Figure 4 on page 12.

```
Resource Variable Class    Resource Variables in the Class

IBM.PSSP.aixos.CPU         IBM.PSSP.aixos.CPU.gluser
                           IBM.PSSP.aixos.CPU.glkern
                           IBM.PSSP.aixos.CPU.glwait
                           IBM.PSSP.aixos.CPU.glidle
                           IBM.PSSP.aixos.CPU.user
                           IBM.PSSP.aixos.CPU.kern
                           IBM.PSSP.aixos.CPU.wait
                           IBM.PSSP.aixos.CPU.idle

IBM.PSSP.aixos.Mem         IBM.PSSP.aixos.Mem.Real.size
                           IBM.PSSP.aixos.Mem.Real.numfrb
                           IBM.PSSP.aixos.Mem.Real.%free
                           IBM.PSSP.aixos.Mem.Real.%pinned
                           IBM.PSSP.aixos.Mem.Virt.pagein
                           IBM.PSSP.aixos.Mem.Virt.pageout
                           IBM.PSSP.aixos.Mem.Virt.pgspgin
                           IBM.PSSP.aixos.Mem.Virt.pgspgout
                           IBM.PSSP.aixos.Mem.Virt.pagexct
                           IBM.PSSP.aixos.Mem.Kmem.inuse
                           IBM.PSSP.aixos.Mem.Kmem.calls
                           IBM.PSSP.aixos.Mem.Kmem.failures
                           IBM.PSSP.aixos.Mem.Kmem.memuse

IBM.PSSP.aixos.Disk        IBM.PSSP.aixos.Disk.busy
                           IBM.PSSP.aixos.Disk.xfer
                           IBM.PSSP.aixos.Disk.rblk
                           IBM.PSSP.aixos.Disk.wblk
```

Figure 4. Examples of Resource Variable Classes

Name Spaces and System Partitions

With respect to system partitioning, the name space for node numbers, whether contained in an instance vector element or in a location field, are global across all system partitions. The name spaces for other element values of instance vectors are local to a system partition. The name space for resource variable names and classes are also local to a system partition.

Choosing the Resource Monitor Type

After defining the data, the next step is to choose the type of resource monitor you are going to write.

The way in which a resource monitor operates depends upon the way in which it is implemented. A resource monitor can be implemented as one of the following:

- A daemon
- A command
- Routines you add to an existing program.

Daemon-Based Resource Monitors

A resource monitor that is implemented as a daemon has the following characteristics:

- It is long-lived.
- It is started automatically by the Event Management subsystem whenever an EM client queries or registers for events that are generated from resource variables it supplies.
- It can supply values for Counter, Quantity, and State resource variable instances.

  Once the resource monitor daemon initializes itself and starts a server session, the Event Management subsystem connects to it and begins to request values for the Counter and Quantity resource variables that the resource monitor has defined. A daemon-based resource monitor can also supply values for State resource variables to the Event Management subsystem.
- All values of Counter and Quantity variable instances are supplied to both the Event Management subsystem and the Performance Monitor subsystem.

To configure a daemon-based resource monitor, specify a connection type of server and the fully-qualified path and file name of the daemon. The server connection type indicates that the Event Management subsystem and the Performance Monitor subsystem initiate the connection and that the program is to act as a daemon. The path and file name tell the Event Management subsystem which daemon to start.

Command-Based Resource Monitors

A resource monitor that is implemented as a command is more limited but also less complex. It can be used to supply resource variable values from a shell script. A command-based resource monitor has the following characteristics:

- It is relatively short-lived.
- It is not started automatically by the Event Management subsystem; instead, it initiates the connection to the Event Management subsystem. It supplies values whether or not any Event Management client has registered interest in events that are generated from resource variables it is monitoring.
- It can supply values only for State resource variables and only to the Event Management subsystem. It sends the values as the states of the monitored resources change. A command-based resource monitor cannot supply values for Counter or Quantity variables and cannot respond to requests for variables.
- It cannot supply resource variable values to the Performance Monitor subsystem.

If your resource monitor is command-based, specify a connection type of client when you configure it. The client connection type indicates that the resource monitor initiates the connection and that the program is to act as a command.

Resource Monitors that Are Incorporated in Other Programs

A resource monitor that is incorporated in another program can act as either a daemon-based resource monitor (in most respects) or a command-based resource monitor.

If it is to act as a daemon-based resource monitor, specify a connection type of server and a null path name when you configure it. The server connection type

indicates that the Event Management subsystem and the Performance Monitor subsystem initiate the connection and that the resource monitoring function will behave in most respects as if it were a daemon. However, the null path name indicates that there is no daemon to be started.

If the resource monitoring function incorporated in another program is to act as a command-based resource monitor, specify a connection type of client. The client connection type indicates that the resource monitor initiates the connection and that the program is to act as a command.

Resource Monitor Type Considerations

In making your choice, consider the following questions:

- Do you already have an existing program that manages a resource?

  If so, you may want to incorporate resource monitoring function into it rather than write a stand-alone program.

- What kind of data do you need to handle?

  If your resource monitor deals only with State resource variables, your resource monitor can act as either a daemon or a command. However, if you have Counter or Quantity resource variables, your resource monitor must act as a daemon, although you can incorporate the function in another program.

- Do you want to supply data to the Performance Monitor subsystem as well as the Event Management subsystem?

  If you do, you must write a resource monitor (or function) that acts as a daemon. Keep in mind, however, that only the values of Counter and Quantity resource variables are supplied to the Performance Monitor subsystem.

- How complicated does your resource monitor need to be?

  A command-based resource monitor is simpler to code than a daemon-based resource monitor.

Configuring the Resource Data and Resource Monitor

Once you have determined the data your resource monitor is going to supply and the type of resource monitor you are going to write, you must make the information available to the Event Management subsystem as a set of formal definitions. This is called **Event Management configuration**.

As part of Event Management configuration, you must supply information about:

- The resource variables that your program will monitor
- Any structured byte string definitions that are required
- The instance vectors for each resource variable
- The classes in which the resource variables are grouped
- Your resource monitor.

Event Management configuration updates the SDR and "production" files for each system partition. Accordingly, it is advisable to create a separate system partition for testing purposes before you alter your production environment.

Event Management configuration requires several steps. First, from the control workstation, you must store the resource definitions for a given system partition as

a set of objects in the System Data Repository (SDR). There are several ways to do this.

The preferred way to load your data for a new resource monitor into the SDR is to create a load list file and run the **haemloadcfg** command. For details on how to do this, see the Event Management subsystem chapter in *PSSP Administration Guide*.

Although creating a separate load list file is preferred, you could also add your definitions to the load list file that contains the PSSP resource definitions. Again, you would run the **haemloadcfg** command to load your data into the SDR.

A final way of loading your data into the SDR is to use such commands as the **SDRCreateObjects** command. You can do this most easily by creating a shell script with the appropriate commands. For an example, see "The rmapi_smp.loadsdr Shell Script" on page 208.

You place the source information in the following partitioned classes in the SDR:

- Class = EM_Resource_Variable
- Class = EM_Structured_Byte_String
- Class = EM_Instance_Vector
- Class = EM_Resource_Class
- Class = EM_Resource_Monitor

Once you have stored the information in the SDR, you must issue the **haemcfg** command. This command compiles the Event Management data in the SDR for a system partition and creates a staging version of the system partition's Event Management Configuration Data Base (EMCDB) file. The data in the EMCDB file, not the SDR, is the data that is used by the Event Management subsystem.

In order to make the new EMCDB file active, that is, to change it from the staging version to the production or real-time version, you must shut down and restart the Event Management subsystem in the system partition. First, shut down any resource monitors that are defined with a connection type of client. Next, shut down all of the Event Management daemons for the system partition (on the control workstation and all of the nodes) and then restart them all. Finally, restart the client resource monitors. For details, see *PSSP Administration Guide*.

Finally, you must repeat this process for each of the system partitions in which Event Management services are required.

Each time you update the EMCDB for a system partition, the **haemcfg** command modifies a version number. The version number allows each instance of the Event Management subsystem and each resource monitor in a system partition to verify that they have access to the correct version of configuration information. The **haemcfg** command maintains the version number in both the EMCDB and the SDR.

Each database may be updated at any time without interfering with the operation of any Event Management subsystem, client, or resource monitor. However, once a database has been updated in a system partition, the Event Management subsystem in the system partition must be restarted to refresh its configuration information from the newly updated database.

Coding and Testing the Resource Monitor

Having defined and configured the resource monitor and its data, the next task is to code the resource monitor.

In general, all resource monitors perform certain functions. They initialize themselves with the RMAPI, manage RMAPI sessions, update resource variables, and terminate. However, the structure and functions of the resource monitor (or resource monitoring function) depend on the implementation you have chosen: daemon-based or command-based. Because the coding of a command-based resource monitor is simpler, we'll describe that type first.

Also, Chapter 5, “Using the RMAPI: Some Resource Monitor Examples” on page 155 contains several listings of sample resource monitor programs.

To test your new resource monitor with the test version of the EMCDB, you must run it in the separate system partition that you created for that purpose.

Coding a Command-Based Resource Monitor

A command-based resource monitor includes the following functions:

- Initializing

  The resource monitor calls the **ha_rr_init** subroutine to inform the RMAPI of the identity of the resource monitor.

- Starting a session

  After the **ha_rr_init** subroutine completes successfully, the command-based resource monitor calls the **ha_rr_start_session** subroutine to start a session with the Event Management subsystem.

- Registering resource variables and their instances

  After starting a session, the resource monitor calls the **ha_rr_reg_var** subroutine to register one or more resource variable instances with the RMAPI.

- Adding resource variables

  After registering resource variables and their instances, the resource monitor must call the **ha_rr_add_var** subroutine to add the resource variables to the list of those known to the RMAPI and supply their current values.

- Sending updated values of resource variable instances

  Whenever it is time to send updated values of resource variable instances to the RMAPI, the resource monitor calls the **ha_rr_send_val** subroutine to send them to the Event Management subsystem.

- Deleting resource variables

  If a resource monitor is no longer sending one or more resource variables to the RMAPI, the resource monitor must call the **ha_rr_del_var** subroutine to delete them from the list of those known to the RMAPI.

- Ending a session

  When the command-based resource monitor is to ready to end, it calls the **ha_rr_end_session** subroutine to end the session.

- Terminating

Once the session has ended, the command-based resource monitor calls the **ha_rr_terminate** subroutine and then exits.

Figure 5 shows a pseudocode outline of a command-based resource monitor that uses RMAPI subroutines.

```
Call ha_rr_init

Call ha_rr_start_session

Call ha_rr_reg_var with all known variable instances.

Call ha_rr_add_var with the variables that were specified to
                   the ha_rr_reg_var routine and supply current
                   values.

Call ha_rr_send_val with the State variable values.  Do this
                    as frequently as required by the operation
                    of the resource monitor.

Call ha_rr_del_var to delete variables added.

Call ha_rr_end_session

Call ha_rr_terminate

Exit
```

Figure 5. An Pseudocode Outline of a Command-Based Resource Monitor

Coding a Daemon-Based Resource Monitor

A daemon-based resource monitor includes the following functions:

- Initializing

  The resource monitor calls the **ha_rr_init** subroutine to inform the RMAPI of the identity of the resource monitor.

- Making a server session

  After the **ha_rr_init** subroutine completes successfully, the daemon-based resource monitor calls the **ha_rr_makserv** subroutine to start a server session so that one or both of the resource monitor managers can connect to it. The resource monitor managers are the Event Management subsystem and the Performance Monitor subsystem.

- Registering resource variables and their instances

  After starting a server session, the resource monitor calls the **ha_rr_reg_var** subroutine to register one or more resource variable instances with the RMAPI.

- Starting a session

  Whenever the file descriptor returned by the **ha_rr_makserv** subroutine is ready for reading, the daemon-based resource monitor calls the **ha_rr_start_session** subroutine to start a session with the resource monitor manager.

- Getting a control message from the Event Management subsystem

  When the resource monitor determines that there is data to be read for the session, it calls the **ha_rr_get_ctrlmsg** subroutine to read the control message, which contains a command from the resource monitor manager.

- Processing the control message

  Once the resource monitor has received the control message, it processes the command it contains. For information on the commands a control message can contain and the processing that the resource monitor performs, see "Processing Control Messages."

- Ending a session

  When a resource monitor manager has disconnected, the daemon-based resource monitor calls the **ha_rr_end_session** subroutine to end the session.

- Terminating.

  When the daemon-based resource monitor no longer has a session with either resource monitor manager, it should call the **ha_rr_terminate** subroutine and then exit.

Processing Control Messages: A control message contains one of the following commands:

HA_RR_CMD_INSTV
: The "instantiate variables" command asks the resource monitor to create instances of specified resource variables. This command is sent only for variables that have been configured as dynamically instantiable.

  To process the command, the resource monitor creates actual instances for each specified instance of a resource variable. Once the instances have been created, the resource monitor calls the **ha_rr_reg_var** subroutine to register the instances.

HA_RR_CMD_ADDV
: The "add variables" command tells the resource monitor to add the specified resource variable instances to the RMAPI.

  To process the command, the resource monitor calls the **ha_rr_add_var** subroutine to add the variables and supply their current values. Then, it calls the **ha_rr_send_val** subroutine to start sending updated values for the resource variable instances that have been added.

  To find out how often to send values for each class of resource variables of type Counter or Quantity, it calls the **ha_rr_get_interval** subroutine to get the reporting interval that has been configured.

HA_RR_CMD_ADDALL
: The "add all variables" command tells the resource monitor to add all of its Counter and Quantity resource variable instances to the RMAPI.

  To process the command, the resource monitor calls the **ha_rr_add_var** subroutine to add the variables and supply their current values. Then, it calls the **ha_rr_send_val** subroutine to start sending updated values for the resource variable instances that have been added.

  To find out how often to send values for each class of resource variables of type Counter or Quantity, it calls the **ha_rr_get_interval** subroutine to get the reporting interval that has been configured.

HA_RR_CMD_DELV

The "delete variables" command tells the resource monitor to delete the specified resource variable instances from the RMAPI.

To process the command, the resource monitor calls the **ha_rr_del_var** subroutine to delete the resource variables. Then, it stops sending values for the resource variable instances that have been deleted.

HA_RR_CMD_DELALL

The "delete all variables" command tells the resource monitor to delete all of its Counter and Quantity resource variable instances from the RMAPI.

To process the command, the resource monitor calls the **ha_rr_del_var** subroutine to delete the variables. Then, it stops sending values for the resource variable instances that have been deleted.

HA_RR_CMD_REFRESH

The "refresh values" command tells the resource monitor to send immediately the latest possible values for the specified variable instances. This command is sent only for Counter and Quantity resource variables that have configurable reporting intervals.

To process the command, the resource monitor calls the **ha_rr_send_val** subroutine to send the latest values for the specified resource variable instances.

Figure 6 shows a pseudocode outline of a daemon-based resource monitor that uses RMAPI subroutines.

```
Call ha_rr_init

Call ha_rr_makserv
    Add the file descriptor (fdc) to the select mask.

Call ha_rr_reg_var with all known variable instances.

Loop on select
     If fdc is ready
           Call ha_rr_start_session
              Add the file descriptor (fdN) to the select mask.

     If fdN is ready
           Call ha_rr_get_ctrlmsg with fdN

           Process the control message

           If the session is disconnected
                   Call ha_rr_end_session

end Loop

Call ha_rr_terminate.
```

Figure 6. A Pseudocode Outline of a Daemon-Based Resource Monitor

Writing Event Management Clients

To write an Event Management client (EM client), you must perform the following steps:

- Find out what resource data is available
- Define predicates to create events
- Code and test the EM client.

This section provides you with an overview of each of these steps. For complete details on how to specify any of the data or subroutines, see the reference information in the following chapters:

- Chapter 2, “Event Management Configuration Data Reference” on page 33
- Chapter 3, “Event Management Subroutine Reference” on page 49
- Chapter 4, “Event Management Files Reference” on page 115.

For details on any commands you need to issue, see *PSSP Command and Technical Reference*.

For listings of several sample Event Management clients, see Chapter 6, “Using the EMAPI: Some Event Management Client Examples” on page 217.

Finding Out What Resource Data Is Available

As described in “Defining the Resource Data” on page 5, resource data is collected and sent to the Event Management subsystem by resource monitors.

The first task in writing an EM client is to determine what resource data is being collected in the system.

There are two ways to do this:

- You can use commands.

  You can use the PSSP **SDRGetObjects** command to list Event Management objects that have been stored in the SDR. You can use the AIX **dspmsg** and **dspcat** commands to list individual messages and the contents of message catalogs that contain descriptions of resource variables.

- You can use the SP Perspectives graphical user interface.

  You can use the Event Management portion of SP Perspectives to list information about the resource variables and instance vectors that have been defined in the system.

Getting Information about Event Management Objects in the SDR

To get information about all of the event management objects in the SDR, you can issue the **SDRGetObjects** command for each class of objects. You can also direct the output to a file, which you can then view or print. To do this, enter the following commands:

```
SDRGetObjects -G EM_Resource_Variable > rv_data
SDRGetObjects -G EM_Resource_Class > class_data
SDRGetObjects -G EM_Structured_Byte_String > sbs_data
SDRGetObjects -G EM_Instance_Vector > iv_data
SDRGetObjects -G EM_Resource_Monitor > rm_data
```

Specifying the **-G** flag ensures that you receive information about all of the resource variables in all of the system partitions in the SP system. If you are only interested in the resource variables in the current system partition, omit the **-G** flag.

Figure 7 on page 22 contains an example of the output of each **SDRGetObjects** command for the **IBM.PSSP.SP_HW.Node.lcd1** resource variable and its associated resource class, structured byte string, instance vector, and resource monitor definitions. For ease of presentation, the output has been reformatted vertically. The actual output of the command is horizontally formatted.

For a complete description of the attributes of Event Management SDR classes, see "Event Management Configuration Data (emcdb)" on page 34.

You can also restrict the information you receive. For example, to get only the **rvName** and **rvClass** information for the **IBM.PSSP.SP_HW.Node.lcd1** resource variable, enter:

```
SDRGetObjects -G EM_Resource_Variable \
    rvName==IBM.PSSP.SP_HW.Node.lcd1 rvName rvClass
```

In response, the system displays:

```
rvName       rvClass
IBM.PSSP.SP_HW.Node.lcd1          IBM.PSSP.SP_HW
```

For more information about the **SDRGetObjects** command, see *PSSP Command and Technical Reference*.

```
Information from EM_Resource_Variable class for Resource Variable IBM.PSSP.SP_HW.Node.lcd1

    rvName                  IBM.PSSP.SP_HW.Node.lcd1
    rvDescription           244
    rvValue_type            State
    rvData_type             SBS
    rvInitial_value         0
    rvClass                 IBM.PSSP.SP_HW
    rvPTX_name              ""
    rvPTX_description       ""
    rvPTX_min               ""
    rvPTX_max               ""
    rvPredicate             ""
    rvEvent_description     ""
    rvLocator               ""
    rvOrder_group           ""
    rvDynamic_instance      0
    rvIndex_vector          NodeNum

Information from EM_Resource_Class class for Resource Class IBM.PSSP.SP_HW

    rcClass                 IBM.PSSP.SP_HW
    rcResource_monitor      IBM.PSSP.hmrmd
    rcObservation_interval  0
    rcReporting_interval    0

Information from EM_Structured_Byte_String class for Resource Variable IBM.PSSP.SP_HW.Node.lcd1

    sbsVariable_name        IBM.PSSP.SP_HW.Node.lcd1
    sbsField_name           STRING
    sbsField_type           cstring
    sbsField_SN             0
    sbsField_init_val       ""

Information from EM_Instance_Vector class for Resource Name IBM.PSSP.SP_HW.Node

    ivResource_name         IBM.PSSP.SP_HW.Node
    ivElement_name          NodeNum
    ivElement_description   256

Information from EM_Resource_Monitor class for Resource Monitor IBM.PSSP.hmrmd

    rmName                  IBM.PSSP.hmrmd
    rmPath                  /usr/lpp/ssp/bin/haemRM/hmrmd
    rmArguments             IBM.PSSP.hmrmd
    rmMessage_file          hmrmd_des.cat
    rmMessage_set           1
    rmConnect_type          server
    rmPTX_prefix            0
    rmPTX_description       0
    rmPTX_asnno             0
```

Figure 7. Sample Output from PSSP SDRGetObjects commands

Displaying Event Management Descriptions

As shown in Figure 7 on page 22, the description of resource variable **IBM.PSSP.SP_HW.Node.lcd1** is message number 244 of message set 1 of the **hmrmd_des.cat** message catalog.

To display the description, enter:

```
dspmsg -s 1 hmrmd_des.cat 244
```

In response, the system displays:

```
"(string) lcd1: 'LCD line 1'."
```

To list all of the messages in the catalog and direct the output to a file, enter:

```
dspcat hmrmd_des.cat > hmrmdcat
```

For more information on the AIX **dspmsg** and **dspcat** commands, see *AIX Version 4.1 Commands Reference*.

Getting Resource Variable Information from SP Perspectives

You can use the Event Management portion of SP Perspectives to get information about the resource variables and instance vectors that have been defined in the system. To do this:

- Start the Event Management portion of SP Perspectives.

  Enter:

  ```
  spevent
  ```

  If you have more than one system partition defined, click on the icon that represents the system partition in which you are interested.

- Open the Create Condition dialog box.

  From the Menu bar of the Event Perspective, click on the Actions button and, from the cascading menus, select Event Definitions and then Create. In response, the system displays a notebook that is open to the Create Event Definition page.

  On the Create Event Definition page, click on the Create Condition button. In response, the system displays the Create Condition dialog box.

- Click the down arrow next to the Resource Variable Name field. In response, the system displays a list of all of the resource variables that are defined in the system partition you selected.

  To get information about any particular resource variable, select it. In response, the system displays the resource variable's name in the Resource Variable Name field, its description in the Resource Variable Description field, and the format of the instance vector in the Resource Identifier Format field.

  For an example of the Create Condition dialog box after the **IBM.PSSP.Prog.pcount** resource variable has been selected, see Figure 8 on page 24.

Note that SP Perspectives uses slightly different terminology than the Event Management APIs, as follows:

- Where the Event Management APIs use the term "predicate," SP Perspectives uses "expression."

- Where the Event Management APIs use the term "instance vector," SP Perspectives uses "resource identifier."

For complete information on using SP Perspectives, see the *PSSP Administration Guide* and the SP Perspectives help panels.

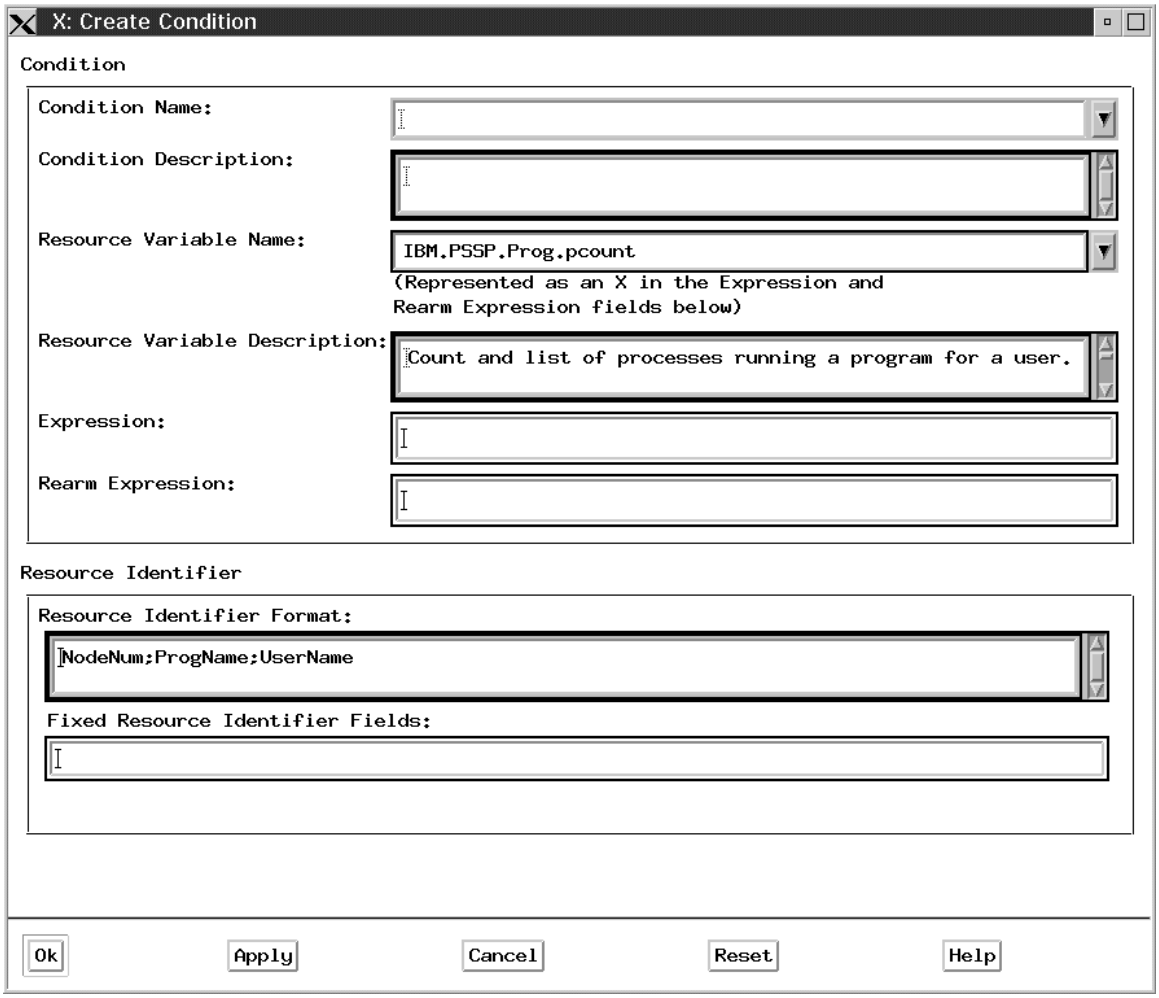

Figure 8. Using the Event Perspective to Get Resource Variable Information

Defining Predicates to Create Events

Once you know what resource data is being collected in the system, you can start defining predicates that will cause your program to be notified when conditions of interest occur. This notification is called an event. Before we get into defining predicates, though, let's take a look at how the Event Management subsystem generates an event.

How the Event Management Subsystem Generates Events

As discussed in “Defining the Resource Data” on page 5, a resource is represented by one or more variables that are defined and updated by a resource monitor. The Event Management subsystem observes each resource variable instance at an interval that is configurable (for Counter and Quantity resource variables), or as the resource variable instance changes (for State resource variables). At each observation of a resource variable instance, the Event Management subsystem applies one or more predicates that were previously supplied by EM clients.

A **predicate** is the relational expression between a resource variable and other elements, such as a constant or the value of a variable from the previous observation. The relationship can be an arithmetic one, a logical one, or a combination.

The predicate is applied to each instance of the resource variable as it is observed. If the predicate is true, an event is generated. More than one predicate may be applied to an instance of the same resource variable at the same observation.

Here's an example of an event that could be generated. Suppose a resource monitor is keeping track of whether a particular node is up or down. The resource monitor represents the node by the **IBM.PSSP.Membership.Node.state** integer variable. As the state of the node changes, it sets the variable equal to a value of 1 (when the node is up) or a value of 0 (when the node is down).

An EM client is interested in knowing when the node goes down. Accordingly, it asks the Event Management subsystem to observe instances of the variable and apply the following predicate to it:

```
X == 0
```

where **X** represents the **IBM.PSSP.Membership.Node.state** variable.

At each observation, the Event Management subsystem applies the predicate to the resource variable instance. If the Event Management subsystem observes that the value of an instance of the **IBM.PSSP.Membership.Node.state** variable is equal to a value of 0, the predicate is true and the Event Management subsystem generates an event, thereby notifying the client.

Another example of a simple predicate is **X < 10**, where **X** is a resource variable that represents the percentage of free space in a file system. This predicate would generate an event whenever the file system's free space was observed to be less than 10%.

To summarize, then, a **predicate** is the relational expression between a resource variable and other elements, such as a constant or the value of a variable from the previous observation. An **event** is the notification that a predicate that contains a resource variable is true.

Defining Predicates

To define a predicate, you must specify the resource variable and its relationship to at least one other element.

To specify the resource variable, you use the letter "X." Unmodified, "X" represents the value of the latest observation of an instance of the resource variable.

You can also modify the resource variable name to indicate any, or a combination, of the following:

- The value of the previous observation of the instance (**P**)
- The raw value of the Counter rather than the default rate (**R**)
- The value of a specific structured byte field (the serial number of the field).

The relationship you specify can be arithmetic, logical, or a combination.

For the complete definition and syntax of a predicate, see “Predicates (haempred)” on page 138. Here are some examples of valid predicate definitions:

`X == 0`	The value of the resource variable instance is equal to zero.
`X < 20 || X > 80`	The value of the resource variable instance is less than 20 or greater than 80.
`!(X < 20 || X > 80)`	The value of the resource variable instance is neither less than 20 nor greater than 80.
`X@R > X@PR`	The current raw value of the variable instance is greater than the raw value of the variable instance from its previous observation.
`X >= X@P + 5`	The current value of the variable instance is greater than or equal to the value of the variable instance from its previous observation plus 5.
`X@2 != 100`	For a resource variable instance that is defined as an SBS, the value of structured field number 2 is not equal to 100.
`X@0 != X@P0 || X@1 != X@P1`	For a resource variable instance that is defined as an SBS, the value of either structured field number 0 or structured field number 1 has changed since its previous observation.

Defining Rearm Predicates

When an EM client registers for an event, it can specify either one or two predicates.

By default, it specifies a single predicate, the **event predicate**. When the event predicate is true, the EM client receives an event.

However, an EM client can also specify a second predicate called the **rearm predicate** in addition to the original event predicate. When an EM client specifies a rearm predicate, the two predicates are used alternately, as follows: the event predicate is used until it is true, then the rearm predicate is used until it is true, then the event predicate is used, and so on.

Rearm predicates can be used in a couple of different ways. One common way is to define the rearm predicate as the inverse of the event predicate. For example, if

the event predicate tests whether a resource variable value is "on," the rearm predicate tests whether it is "off." If it registers for events from both predicates, the EM client is notified when the resource variable changes rather than each time it is observed.

A rearm predicate can also be used with the event predicate to define an upper and lower boundary for a condition of interest. For example, suppose an EM client is interested in how much disk space is used in the system. It defines a normal disk space condition to be in the range between 80% and 90% of the total space. However, if the amount of disk space used rises above 90% or falls below 80%, the EM client wants to be notified.

In this case, it would define the event predicate to be "the amount of disk space used is greater than 90%" and the rearm predicate to be "the amount of disk space used is less than 80%." The EM client would also register for events from both predicates.

The first time disk space usage rises above 90%, the Event Management subsystem notifies the EM client and switches to the rearm predicate. The EM client is not notified again until disk space usage decreases to an amount less than 80%, and then the Event Management subsystem switches to the event predicate.

Using a Default Predicate

When the writer of a resource monitor defines a resource variable, he or she may also optionally define for it a single **default predicate**. As the writer of an EM client, you can use the default predicate (if it is defined) in addition to, or instead of, predicates you define.

Coding and Testing the EM Client

Having determined what resource data is available and the conditions your program will be interested in, the next task is to code the EM client.

In general, your EM client will start one or more EMAPI sessions, register for events with the Event Management subsystem, process events from the Event Management subsystem, and end its EMAPI sessions. It can also query resource variable information and unregister for events. The following sections describe each of these activities in more detail.

Also, Chapter 6, “Using the EMAPI: Some Event Management Client Examples” on page 217 contains several listings of sample EM client programs.

From an Event Management point of view, you do not need to set up a separate system partition just for testing an EM client. However, if your EM client is taking actions that might adversely affect a production environment, you may wish to set up a separate test system partition or other testing environment.

Starting a Session

To start an event management session, the EM client calls the **ha_em_start_session** subroutine.

By default, an EM client starts a session with the Event Management subsystem in the current system partition. The current system partition is the system partition to which the node on which the client is running belongs. However, by specifying the system partition name, a client can start a session with the Event Management

subsystem in any partition. By starting sessions for multiple system partitions, the EM client can receive events from, and information about, any resource variable instance in the system. To receive events from and to issue queries to other system partitions, the client must start at least one session with the Event Management subsystem in each other system partition.

An EM client can also start multiple sessions with any single system partition, which allows multiple communication paths to the Event Management subsystem in a single system partition. Multiple paths are useful if the client wishes to ignore certain responses for a time while accepting others. For example, a client can register events in one session and issue queries in another.

If the communication path for a session is lost, the EM client calls the **ha_em_restart_session** subroutine to restart an existing session.

Registering for Events

Once it has started a session, the EM client calls the **ha_em_send_command** subroutine to register for events.

The EM client may specify either of two register commands:

- **HA_EM_CMD_REG**, which the EM client uses to register for an event from a single predicate, the event predicate. If a rearm predicate is defined, it does not generate an event.
- **HA_EM_CMD_REG2**, which the EM client uses to register for an event from both an event predicate and a rearm predicate.

For each event, the EM client supplies the name of a resource variable, an instance vector, and the predicate or predicates that are to be used to generate the event. For each successfully registered event, the EMAPI returns an event ID to the EM client.

The instance vector identifies the particular instance or instances of the resource variable that the Event Management subsystem should use to generate the event. To receive events from more than one instance of the resource variable, the EM client can use wildcard characters in the instance vector to specify multiple instances.

The EM client can supply its own predicate or predicates or use the default predicate (if any) that was defined when the resource variable was configured.

If the EM client registers for more than one event with a single command, the events belong to an **event command group**. The EM client can call the **ha_em_get_ecgid** subroutine to get the identifier of the event command group to which an event belongs.

By default, the Event Management subsystem does not generate an event until the predicate is true. However, if the EM client specifies the **HA_EM_SCMD_REVAL** subcommand on either register command, the Event Management subsystem generates an event at the first observation of an instance of the resource variable, whether or not the predicate is true. In this way, the EM client can obtain the initial state of the resource variable. Then the Event Management subsystem generates events as usual.

Unregistering for Events

When the EM client no longer wishes to receive an event, it calls the **ha_em_send_command** subroutine to unregister for the event, specifying the **HA_EM_CMD_UNREG** command. For each event to be unregistered, the EM client supplies the event ID that was returned when the event was registered. The EM client can unregister one or more events with a single command.

Sending Queries to the Event Management Subsystem

The EM client can also get certain kinds of information by calling the **ha_em_send_command** subroutine, specifying the **HA_EM_CMD_QUERY** command. Using query subcommands, the EM client can get the following information:

- The current value of one or more resource variables, using the **HA_EM_SCMD_QCUR** subcommand
- A list of the resource variables and default predicates that are defined in the Event Management Configuration Database (EMCDB), using the **HA_EM_SCMD_QDEF** subcommand.

The scope of each query is defined by the specification of a resource variable name and an instance vector. These specifications may be wildcarded.

Receiving Responses from the Event Management Subsystem

Once the EM client has sent a command, it must be ready to receive a response.

First, the EM client checks the session's file descriptor, using either the **poll** or the **select** system call. When the file descriptor is ready for reading, the EM client calls the **ha_em_receive_response** routine.

By default, the Event Management subsystem returns the response in a response block. The response block contains one of the following:

- One or more events
- A notification that one or more events have been unregistered
- A response to a previously issued query command
- An indicator that there was an error in a previously issued register or query command.

If you prefer to program using callback routines, your EM client may specify one or more callback routines instead. A callback routine can handle one or more events, responses to unregister commands, and responses to query commands.

You specify callback routines on input to the **ha_em_send_command** subroutine. You can specify a different callback routine for each event or the same callback routine for multiple events on a single register command. For queries, you can specify only one callback routine on a single query command.

When there is no error in the event or response, the Event Management subsystem delivers it directly to the callback routine. The Event Management subsystem passes information about the event or response as input to the callback routine.

For some error conditions, the Event Management subsystem always returns from the **ha_em_receive_response** subroutine with a response block that contains the

error, even if the EM client specified a callback routine. For details on error handling in response to commands, see “ha_em_receive_response Subroutine” on page 55.

Ending a Session

When the EM client no longer needs Event Management services, it ends the session by calling the **ha_em_end_session** subroutine.

Event Management Performance Considerations

The amount of overhead that the Event Management subsystem incurs is directly proportional to the number of resource variable instances it is observing and the number of predicates that it is evaluating at each observation. Accordingly, you can improve performance by specifying longer reporting and observation intervals.

Also, the Event Management subsystem does not start certain daemon-based resource monitors until at least one EM client has registered interest in the resources they are monitoring.

Resource Monitors and PTX Shared Memory

A resource monitor that uses Quantity and Counter variables can send those variables to the Performance Monitor as well as the Event Management subsystem. Such a resource monitor uses shared memory to pass the variables and, in PTX parlance, is a "dynamic data-supplier program." The Event Management subsystem and the Performance Monitor play the role of PTX "local data-consumer programs."

If your resource monitor uses shared memory in this way, it is important to make sure that the shared memory is released when the resource monitor exits, either normally or abnormally. The best way to make sure this happens is by calling the **ha_rr_terminate** subroutine before exiting. The **ha_rr_terminate** subroutine causes the RMAPI to release the shared memory. In addition to a routine that handles a normal exit, you should establish a signal handler to catch abnormal termination signals. The signal handler should handle the following signals: **SIGTERM**, **SIGHUP**, **SIGINT**, and **SIGSEGV**.

If the resource monitor abnormally terminates without calling the **ha_rr_terminate** subroutine, you may need to release the shared memory manually. The following sections, adapted from PTX documentation, explain how to do this.

You may have to use this procedure when any program that uses PTX shared memory fails to release it, due to abnormal termination or any other cause. Such programs are not limited to resource monitors, but also include the Event Management subsystem, the Performance Monitor subsystem, and any other dynamic data-supplier programs that may be running in the system partition.

DDS Shared Memory

If a dynamic data-supplier program is terminated in a way that cannot be detected from the program itself, the DDS shared memory is not released and subsequent attempts to start the dynamic data-supplier program fail. If this happens, you must release the DDS shared memory manually, as described in “Releasing Shared Memory Manually.”

To avoid the situation, never kill a dynamic data-supplier program with the option **-9** That would terminate the program with a **SIGKILL** signal, which is not detectable.

Releasing Shared Memory Manually

In situations where one or more data-supplier or local data-consumer programs have terminated in such a way that their shared memory allocations have not been released, the shared memory segments should be released from the command line before attempting to restart the programs. It is recommended that all data-supplier and local data-consumer programs, including **xmservd**, are killed before you attempt to release shared memory. Clearing of all shared memory segments could be done through the following steps:

1. Identify all data-supplier and local data-consumer programs that are running. Use the **ps** command and your knowledge of the programs in use on your system to locate all of them. For each of the running data-supplier or local data-consumer programs, note their process IDs.
2. Kill all processes associated with data-supplier and local data-consumer programs without using a command line flag.
3. Verify that all data-supplier and local data-consumer processes have been killed. If not, use the **kill -9** command to kill them.
4. List the shared memory segments in use with the **ipcs -m** command. This produces a list like the following:

   ```
   IPC status from /dev/mem as of Fri Dec 31 07:54:44 CST 1993
   T  ID       KEY           MODE    OWNER GROUP
   Shared Memory:
   m      0  0x0d050296 --rw-------   root system
   m  20481  0x5806188b --rw-rw-rw- nchris system
   m  28674  0x780502ea --rw-rw-rw-   root system
   m  12292  0x780502e3 --rw-rw-rw-   root system
   m  20485  0x780502d1 --rw-rw-rw-   root system
   ```

5. Identify all shared memory segments with a KEY that begins with "0x78". All shared memory allocated by data-supplier and local data-consumer programs has this key. Now use the **ipcrm** command to remove the shared memory segments, specifying as command arguments the IDs of the segments you want to remove. To remove all three data-supplier and local data-consumer segments listed above, your command would be:

   ```
   ipcrm -m 28674 -m 12292 -m 20485
   ```

6. Restart the dynamic data-supplier and data-consumer programs as required. To start **xmservd** and any dynamic data-supplier programs started by it, enter the **xmpeek** command.

Chapter 2. Event Management Configuration Data Reference

This chapter contains the reference material for the Event Management configuration data that is stored as a set of objects in the System Data Repository (SDR).

These objects are created from a load list file (**haemloadlist**) by the **haemloadcfg** utility command. The SDR objects are compiled using the **haemcfg** utility command to produce a binary file called the Event Management Configuration Database (EMCDB) that is used by the Event Manager daemon.

The man page in this chapter describes the SDR classes and attributes for the Event Management objects.

For information about the **haemloadcfg** utility command, the **haemloadlist** load list file and its format, and the **haemcfg** utility command, see *PSSP Command and Technical Reference*.

© Copyright IBM Corp. 1996

Event Management Configuration Data (emcdb)

Purpose

Event Management Configuration Data (SDR) – System Data Repository (SDR) classes and attributes for Event Management resource variables, instance vectors, resource classes, and resource monitors

Description

Before you can use Event Management services through the Event Management Application Programming Interface (EMAPI), information about the system resources being monitored must be defined and configured to the Event Management subsystem. This information includes the resource variables from which events may be generated, the instance vectors for each resource variable, the classes in which the resource variables are grouped, and the resource monitors that supply the variables. For resource variables that are structured byte strings (SBSs), the SBS structured fields must also be defined. A distinct database is created for each system partition on the IBM RS/6000 Scalable POWERparallel Systems (SP).

The configuration process for each system partition consists of the following steps:

1. The programmer that creates the resource monitor defines the Event Management resource information and stores it in an Event Management load list file.
2. The programmer or system administrator issues the **haemloadcfg** command to load the resource definitions from the load list file into the System Data Repository (SDR).
3. The programmer or system administrator uses the **haemcfg** command to compile the SDR objects, producing a binary file called the Event Management Configuration Database (EMCDB). The new EMCDB is placed in a staging directory.
4. The programmer or system administrator activates the new EMCDB by using **haemctrl** commands to stop all of the system partition's Event Manager daemons and restart them. The commands must stop and restart the system partition's daemon on the control workstation and the daemon on each of the system partition's nodes. This has the effect of replacing the run-time version of the EMCDB with the new EMCDB from the staging directory.

For more details on the procedures for configuring Event Management, see the Event Management subsystem chapter in *PSSP Administration Guide*.

To place information in the SDR, you must have the appropriate authorization. If you do not have the appropriate authorization, consult with your system administrator. For more information about the requirements for access to the SDR, see *PSSP Administration Guide*.

You can use SDR commands to work with objects in the Event Management classes directly. However, IBM recommends that you use load list files and the **haemloadcfg** command to work with objects in these classes.

You cannot use System Management Interface Tool (SMIT) panels to work with objects in these classes.

EMCDB Source Information

Source information for the EMCDB is kept in the following partitioned classes in the SDR:

- Class = EM_Resource_Variable
- Class = EM_Structured_Byte_String
- Class = EM_Instance_Vector
- Class = EM_Resource_Class
- Class = EM_Resource_Monitor

The tables below summarize the SDR classes for Event Management and their attributes. The value in the **Type** column is the datatype of the attribute: S=string, I=integer, F=floating point.

Detailed explanations of each class and attribute, as well as examples showing their use, follow the tables.

Class = EM_Resource_Variable

Attribute Name	*Type*	*Description*
rvName	S	The name of a resource variable.
rvDescription	I	The ID of the message that contains the description of the variable, including its semantics.
rvValue_type	S	The value type of the variable.
rvData_type	S	The data type of the variable.
rvInitial_value	S	The initial value of the resource variable.
rvClass	S	The name of a resource variable class.
rvPTX_name	S	The name that is used to read and write the variable in the Performance Toolbox (PTX) shared memory.
rvPTX_description	S	A comma-separated list, with no blanks, of message IDs.
rvPTX_min	I	The lower range for plotting this variable by the Performance Monitor.
rvPTX_max	I	The upper range for plotting this variable by the Performance Monitor.
rvPredicate	S	The default predicate that is to be applied to the resource variable.
rvEvent_description	I	The ID of the message that contains a short description of the event
rvLocator	S	An instance vector element name or a null string.

Attribute Name	*Type*	*Description*
rvDynamic_instance	I	If nonzero (true), instances of this variable are created dynamically by its resource monitor whenever the instance is referenced in an EMAPI command.
rvIndex_vector	S	An instance vector element name or a null string.

Class = EM_Structured_Byte_String

Attribute Name	*Type*	*Description*
sbsVariable_name	S	The name of the SBS resource variable for which this object defines one of its structured fields.
sbsField_name	S	A structured field name, which provides a short description of the structured byte field.
sbsField_type	S	The data type of this field.
sbsField_SN	I	The serial number of the structured field.
sbsField_init_val	S	The initial value of the structured field, with a format suitable to the field's type.

Class = EM_Instance_Vector

Attribute Name	*Type*	*Description*
ivResource_name	S	The name of the resource (and all of its resource variables) for which this instance vector element is defined.
ivElement_name	S	An instance vector element name, which describes the element.
ivElement_description	I	The ID of the message that describes the vector element, including its semantics.

Class = EM_Resource_Class

Attribute Name	*Type*	*Description*
rcClass	S	The name of a resource variable class.
rcResource_monitor	S	The name of the resource monitor definition for the resource monitor that supplies the resource variables in this class.
rcObservation_interval	I	The observation interval, which is the amount of time, in seconds, between each observation by the Event Management subsystem of instances of any variable of value type Counter or Quantity in this class.

Attribute Name	*Type*	*Description*
rcReporting_interval	I	The reporting interval, which is the amount of time, in seconds, between each update of any variable of value type Counter or Quantity in this class by the corresponding resource monitor.

Class = EM_Resource_Monitor

Attribute Name	*Type*	*Description*
rmName	S	A resource monitor definition name.
rmPath	S	The executable absolute path name of the resource monitor if it is a daemon that can be started by the Event Management subsystem.
rmArguments	S	A list of optional arguments that the Event Management subsystem should pass when it starts the resource monitor daemon.
rmMessage_file	S	The name of the message catalog that contains all of the descriptions for the resource variables that are supplied by this resource monitor.
rmMessage_set	I	The set ID of the descriptions for the resource variables that are supplied by this resource monitor.
rmConnect_type	S	The type of connection.
rmPTX_prefix	S	The PTX name prefix.
rmPTX_description	S	A comma-separated list, with no blanks, of message IDs.
rmPTX_asnno	I	The ASN.1 number equal to the SNMP Assigned Enterprise Number for the vendor that supplies this resource monitor.

Class = EM_Resource_Variable

The **EM_Resource_Variable** class contains one object for each resource variable defined in the database.

Some of the attributes in this class are used to meet PTX requirements. For more information on PTX, see the *Performance Toolbox for AIX Guide and Reference*.

The attributes of this class are defined as follows:

rvName A string that contains the name of a resource variable. It consists of a resource name followed by a period followed by the resource attribute. For details on specifying the components of a resource variable name, see “Resource Variables (haemrviv)” on page 141.

Resource variables that represent different attributes of the same resource must contain the same resource name.

rvDescription

An integer that identifies the message that contains the description of the variable, including its semantics. The first line of the message must be no more than 63 bytes, excluding the newline character at the end of the line, and should be a brief abstract of the description.

The description provides information about the resource variable, in particular its semantics. It must provide enough information for a programmer or administrator to understand what information is contained in the resource variable, what the motivation is for the default predicate, and under what circumstances additional predicates could be provided through the EMAPI.

If the variable is of value type Counter or Quantity, the first line of the message is used when viewing the variable through the Performance Monitor.

To find the message, use the message ID with the message set and message catalog attributes of the **EM_Resource_Monitor** class object for the resource monitor that defines the resource variable.

rvValue_type

A string that indicates the value type of the variable. It may be one of the following: Counter, Quantity, or State. Case is significant.

A **Counter** is a variable whose value increases monotonically. By default, a counter is interpreted as a rate, that is, the change in the value of an instance of the counter from one observation to the next divided by the time, in seconds, between the two observations. However, the actual contents of the instance of the counter from the latest observation, called the raw value, is also available.

A Counter is typically used to represent an attribute of a resource that is an indicator of throughput. Instances of a Counter are observed by the Event Management subsystem at an interval, in seconds, that is configurable for the class to which the Counter belongs. Instances of a Counter may be updated more often than they are observed without losing meaningful information for generating events.

A Counter is implemented as a PTX variable, using shared memory, and requires a PTX name.

A **Quantity** is a variable whose value fluctuates over time. It is typically used to represent an attribute of a resource that is a level, that is, an indicator of how many. Instances of a Quantity are observed by the Event Management subsystem at an interval, in seconds, that is configurable for the class to which the Quantity belongs. Instances of a Quantity may be updated more often than they are observed without losing meaningful information for generating events.

A Quantity is implemented as a PTX variable, using shared memory, and requires a PTX name.

A **State** is a variable whose value fluctuates over time, like a Quantity. However, every time an instance of a State resource variable is updated, it must be observed so that potential events are not missed.

A State variable can be used to represent an attribute of a resource that indicates anything other than throughput or a level. A resource monitor passes instances of a State variable to the Event Management

subsystem as messages. Thus, an instance of a State variable is observed whenever it is received by the Event Management subsystem, rather than at regular intervals.

A State is not implemented as a PTX variable, does not use shared memory, and does not require a PTX name.

rvData_type
A string that indicates the data type of the variable. It may be one of the following: long, float, or SBS. Case is significant.

The long and float formats are identical to the C language types of the same names.

The SBS data type specifies a structured byte string and is permitted only for a variable of value type State. If specified, the structured fields that make up the structured byte string must be defined as objects in the **EM_Structured_Byte_String** class.

rvInitial_value
A string that contains the initial value of the resource variable.

If the data type of the variable is long, the initial value is an integer constant in a format suitable for conversion by the **strtol** subroutine (with a base parameter of 0). If the initial value is the null string, the initial value is 0, by default.

If the data type of the variable is float, the initial value is a floating point constant in a format suitable for conversion by the **strtof** subroutine. If the initial value is the null string, the initial value is 0.0, by default.

If the data type of the variable is structured byte string, this attribute value must be the null string.

rvClass A string that contains the name of a resource variable class.

The **class** of a resource variable indicates the application or subsystem that manages the associated resource and can also be used to group variables by observation interval.

For details on specifying the name of a resource variable class, see “Resource Variables (haemrviv)” on page 141.

rvPTX_name
A string that contains the name that is used to read and write the variable in the Performance Toolbox (PTX) shared memory. This name is required for variables with a value type of Counter or Quantity. Otherwise, this attribute value is a null string.

The **PTX name** used to fetch a variable from shared memory or to place it into shared memory must match the definition of the matching PTX statistic, if it is already defined by a Dynamic Data Supplier or by AIX, or it must follow the PTX naming convention, if the variable is newly defined. If the PTX name contains components that are instances, these component values must be taken from one or more of the instance vector element values. This implies that for existing PTX variables, the instance vector is derived from the PTX name. For newly-created variables, this implies that the PTX name and the instance vector for the resource variable are designed in concert.

Where a PTX name contains a component that is an instance, that part of the name is specified by the name of the corresponding instance vector element, prepended with the dollar sign. For example,

```
Resource Variable Name             Instance Vector

IBM.PSSP.aixos.Mem.Real.size       NodeNum=5
IBM.PSSP.aixos.Mem.Virt.pagein     NodeNum=5
IBM.PSSP.aixos.Mem.Virt.pageout    NodeNum=5
IBM.PSSP.aixos.Mem.Kmem.inuse      NodeNum=5;Type=mbuf
IBM.PSSP.aixos.PagSp.%totalfree    NodeNum=5
IBM.PSSP.aixos.VG.free             NodeNum=5;VG=rootvg
IBM.PSSP.aixos.FS.%totused         NodeNum=5;VG=rootvg;LV=hd4
IBM.PSSP.aixos.Disk.busy           NodeNum=5;Name=hdisk0
```

would have the corresponding PTX names defined in the configuration file:

```
PTX Name

Mem/Real/size
Mem/Virt/pagein
Mem/Virt/pageout
Mem/Kmem/$Type/inuse
PagSp/%totalfree
FS/$VG/free
FS/$VG/$LV/%totused
Disk/$Name/busy
```

The Event Management subsystem replaces PTX name components that begin with a dollar sign by the value of the corresponding instance vector element. Conversely, the value of an instance vector element is taken from the positionally corresponding component name of the matching PTX name.

rvPTX_description

A string that contains a comma-separated list, with no blanks, of message IDs. Each ID specifies a message that contains a short description (no more than 63 bytes) of a PTX context.

The first ID in the list corresponds to the context specified by the first component of the name specified by the **rvPTX_name** attribute, the second ID in the list corresponds to the context specified by the first two components of the name, the third ID corresponds to the context specified by the first three components of the name, and so on. However, there is one fewer ID in this string than the number of components in **rvPTX_name**. (The last component in the PTX name is a statistic, not a context.)

This string is required for variables of value type Counter or Quantity. Otherwise, the attribute value is a null string.

To find the message, use the message ID with the message set and message catalog attributes of the **EM_Resource_Monitor** class object for the resource monitor that defines the resource variable.

rvPTX_min

An integer that is equal to the lower range for plotting this variable by the Performance Monitor. This value must be specified for variables of value type Counter or Quantity. Otherwise, the attribute value is 0.

rvPTX_max
An integer that is equal to the upper range for plotting this variable by the Performance Monitor. This value must be specified for variables of value type Counter or Quantity. Otherwise, the attribute value is 100.

rvPredicate
A string that contains the default predicate that is to be applied to the resource variable.

If you do not wish to specify a default predicate, set this attribute value to the null string.

For details on how to specify a predicate, see "Predicates (haempred)" on page 138.

rvEvent_description
An integer that equals the ID of the message that contains a short description of the event that is generated when the default predicate is applied to the variable. If a default predicate is not specified, this attribute must be set to 0.

rvLocator
A string that contains either an instance vector element name or a null string.

If the instance vector for this resource implies the resource's location, the value of this item is the name of the vector element whose value is the number of the node that contains the resource instance. Values of the vector element that is specified by the **rvLocator** attribute must be of type integer.

If the value of this item is a null string, the Event Management subsystem must determine the location of the resource.

rvDynamic_instance
An integer that, if nonzero (true), indicates that instances of this variable are created dynamically by its resource monitor whenever the instance is referenced in an EMAPI command.

rvIndex_vector
A string that contains either an instance vector element name or a null string.

If an instance vector element for this resource variable can be used as an index into an array that represents all of the variable's instances, specify its name. However, you cannot specify the same element for both the **rvLocator** and the **rvIndex_vector** fields.

Values of the vector element that is specified by the **rvIndex_vector** attribute must be of type integer.

Specifying an index vector can improve the performance of the Event Management subsystem.

If more than one of the vector elements can be used as an index, specify the one that can take on the larger number of possible values. For example, if one instance vector element represents a node number in a 128-node system and another represents a frame number in the same system, choose the element that represents the node number, because there are more nodes than frames.

Here is an example of creating an object in the **EM_Resource_Variable** class for a resource variable of type Quantity using the **SDRCreateObjects** command:

```
SDRCreateObjects EM_Resource_Variable \
    'rvName=IBM.PSSP.aixos.PagSp.totalsize' \
    'rvDescription=99' \
    'rvValue_type=Quantity' \
    'rvData_type=long' \
    'rvInitial_value=0' \
    'rvClass=IBM.PSSP.aixos.PagSp' \
    'rvPTX_name=PagSp/totalsize' \
    'rvLocator=NodeNum' \
    'rvDynamic_instance=0'
```

Here is an example of creating an object in the **EM_Resource_Variable** class for a resource variable of type State using the **SDRCreateObjects** command:

```
SDRCreateObjects EM_Resource_Variable \
    'rvName=IBM.PSSP.Prog.pcount' \
    'rvDescription=1009' \
    'rvValue_type=State' \
    'rvData_type=SBS' \
    'rvClass=IBM.PSSP.Prog' \
    'rvLocator=NodeNum' \
    'rvDynamic_instance=1'
```

Class = EM_Structured_Byte_String

The **EM_Structured_Byte_String** class contains one object for each structured field that is defined in a structured byte string.

The attributes of this class are defined as follows:

sbsVariable_name
: A string that contains the name of the SBS resource variable for which this object defines one of its structured fields.

sbsField_name
: A string that contains a structured field name, which provides a short description of the structured byte field. The name must consist of only alphanumeric characters and underscores; the first character must be alphabetic. This name is available through the Event Management API.

sbsField_type
: A string that indicates the data type of this field. It can be one of the following values: long, float, cstring, or bstring, indicating a structured field of data type long, float, character string, or byte string, respectively. The long and float data types are the same as in the C language. A character string consists of one or more nonzero bytes terminated by a null byte; the null byte is included in the structured field value length. A byte string consists of one or more bytes, where each byte may have any value from 0 through 255.

sbsField_SN
: An integer that is the serial number of the structured field. The serial number is a one-byte value that uniquely identifies the structured field within the structured byte string. This serial number is defined for each structured field by the resource monitor that supplies the SBS resource

variable. However, the set of serial numbers for the structured byte string starts with 0 and continues sequentially, incrementing by 1.

sbsField_init_val
A string that contains the initial value of the structured field, with a format suitable to the field's type.

If the type of the structured field is long, the initial value is an integer constant in a format suitable for conversion by the **strtol** subroutine (with a base parameter of 0).

If the type of the structured field is float, the initial value is a floating point constant in a format suitable for conversion by the **strtof** subroutine.

Otherwise, the initial value is a string constant. To indicate nondisplayable characters, a string constant may contain *ooo*, where *ooo* is an octal constant that represents the character. If the type of the structured field is **cstring**, a terminating null character is appended automatically.

If the value of the **sbsField_init_val** attribute is the null string, default initial values are used. The default values are 0 for long fields, 0.0 for float fields, the null string for character strings and, for byte strings, a string consisting of a single byte with a value of 0.

The **IBM.PSSP.Prog.pcount** resource variable is an SBS that has three structured fields. Here is an example of creating the objects in the **EM_Structured_Byte_String** class using the **SDRCreateObjects** command:

```
SDRCreateObjects EM_Structured_Byte_String \
    'sbsVariable_name=IBM.PSSP.Prog.pcount' \
    'sbsField_name=CurPIDCount' \
    'sbsField_type=long' \
    'sbsField_SN=0'

SDRCreateObjects EM_Structured_Byte_String \
    'sbsVariable_name=IBM.PSSP.Prog.pcount' \
    'sbsField_name=PrevPIDCount' \
    'sbsField_type=long' \
    'sbsField_SN=1'

SDRCreateObjects EM_Structured_Byte_String \
    'sbsVariable_name=IBM.PSSP.Prog.pcount' \
    'sbsField_name=CurPIDList' \
    'sbsField_type=cstring' \
    'sbsField_SN=2'
```

Class = EM_Instance_Vector

The **EM_Instance_Vector** class contains one object for each instance vector element that is defined for a resource and all of its resource variables.

The attributes of this class are defined as follows:

ivResource_name
A string that contains the name of the resource (and all of its resource variables) for which this instance vector element is defined.

ivElement_name
A string that contains an instance vector element name, which describes the element. The name must consist of only alphanumeric characters and underscores; the first character must be alphabetic. The set of values in the vector uniquely identify the copy of the resource variable in the system. Note that if a resource variable can only ever have one instance in the system, the instance vector is null.

The semantics of an instance vector are defined by the semantics of the resource variable with which it is associated. The names of the vector elements must be unique within a given resource variable's instance vector. These names may be used in other instance vectors and may or may not have the same semantics. In other words, the name space for instance vector element names is local to each defined resource variable.

ivElement_description
An integer that is the ID of the message that describes the vector element, including its semantics.

To find the message, use the message ID with the message set and message catalog attributes of the **EM_Resource_Monitor** class object for the resource monitor that defines the resource variable of which this is an instance vector.

Here are two examples of creating an object in the **EM_Instance_Vector** class using the **SDRCreateObjects** command:

```
SDRCreateObjects EM_Instance_Vector \
    'ivResource_name="IBM.PSSP.aixos.PagSp' \
    'ivElement_name=NodeNum' \
    'ivElement_description=60'

SDRCreateObjects EM_Instance_Vector \
    'ivResource_name=IBM.PSSP.Prog' \
    'ivElement_name=UserName' \
    'ivElement_description=4'

SDRCreateObjects EM_Instance_Vector \
    'ivResource_name=IBM.PSSP.Prog' \
    'ivElement_name=ProgName' \
    'ivElement_description=3'

SDRCreateObjects EM_Instance_Vector \
    'ivResource_name=IBM.PSSP.Prog' \
    'ivElement_name=NodeNum' \
    'ivElement_description=5'
```

Class = EM_Resource_Class

The **EM_Resource_Class** class contains one object for each resource variable class that is defined in the database.

The attributes of this class are defined as follows:

rcClass A string that contains the name of a resource variable class.

The **class** of a resource variable indicates the application or subsystem that manages the associated resource and can also be used to group variables by observation interval.

For details on specifying the name of a resource variable class, see "Resource Variables (haemrviv)" on page 141.

rcResource_monitor
A string that contains the name of the resource monitor definition for the resource monitor that supplies the resource variables in this class.

rcObservation_interval
An integer that is the observation interval, which is the amount of time, in seconds, between each observation by the Event Management subsystem of instances of any variable of value type Counter or Quantity in this class. This value must be greater than or equal to the reporting interval.

rcReporting_interval
An integer that is the reporting interval, which is the amount of time, in seconds, between each update of any variable of value type Counter or Quantity in this class by the corresponding resource monitor.

The value of the reporting interval may be 0 if the design of the resource monitor fixes the interval between variable updates. It may also be 0 if the resource monitor is incorporated into a subsystem and the subsystem updates the variable as part of its normal execution.

Here are two examples of creating an object in the **EM_Resource_Class** class using the **SDRCreateObjects** command:

```
SDRCreateObjects EM_Resource_Class \
    'rcClass=IBM.PSSP.aixos.PagSp' \
    'rcResource_monitor=aixos' \
    'rcObservation_interval=30' \
    'rcReporting_interval=0'

SDRCreateObjects EM_Resource_Class \
    'rcClass=IBM.PSSP.Prog' \
    'rcResource_monitor=IBM.PSSP.harmpd' \
    'rcObservation_interval=0' \
    'rcReporting_interval=0'
```

Class = EM_Resource_Monitor

The **EM_Resource_Monitor** class contains one object for each resource monitor that is defined in the database.

Some of the attributes in this class are used to meet PTX requirements. For more information on PTX, see the *Performance Toolbox for AIX Guide and Reference*.

The attributes of this class are defined as follows:

rmName A string that contains a resource monitor definition name. This name must be unique in the database.

By convention, resource monitor names follow the same hierarchical naming conventions that are used for resource names except that they may not contain percent signs. For more information, see "Resource Variables (haemrviv)" on page 141.

rmPath A string that contains the executable absolute path name of the resource monitor if it is a daemon that can be started by the Event Management subsystem. If the resource monitor cannot be started by

the Event Management subsystem, this attribute must be set to the null string.

rmArguments
A string that contains a list of optional arguments that the Event Management subsystem should pass when it starts the resource monitor daemon. If there are no arguments, this attribute must be set to the null string.

rmMessage_file
A string that contains the name of the message catalog that contains all of the descriptions for the resource variables that are supplied by this resource monitor.

For information about message catalogs, see the "Message Facility Overview for Programming" topic in *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs*.

rmMessage_set
An integer that is the set ID of the descriptions for the resource variables that are supplied by this resource monitor.

For information about message sets, see the "Message Facility Overview for Programming" topic in *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs*.

rmConnect_type
A string that indicates the type of connection. It may be one of the following: client or server.

If the resource monitor is to connect to the Event Management subsystem, specify client. The resource monitor is a command.

If the Event Management subsystem and the Performance Monitor subsystem are to connect to the resource monitor, specify server. In this case, the resource monitor is either a daemon or incorporated within a subsystem.

rmPTX_prefix
A string that contains the PTX name prefix. This prefix is prepended to each PTX name for resource variables of value type Counter or Quantity that are supplied by this resource monitor.

Resource variables of value type Counter or Quantity that are supplied by a resource monitor are implemented as PTX variables and must therefore conform to the PTX naming architecture. The PTX name for either of these types of variables is formed by concatenating the following strings:

- The string **DDS**/
- The string specified by the **rmPTX_prefix** attribute of the **EM_Resource_Monitor** object that defines the resource monitor.
- The string specified by the **rvPTX_name** attribute of the **EM_Resource_Variable** object.

The PTX name prefix is expected to consist of at least one component but probably not more than two or three. The prefix is used to group all of the resource variables of value type Counter or Quantity that are supplied by a resource monitor.

Each resource monitor must have a unique PTX name prefix. For PSSP resource monitors, the prefix is **IBM/PSSP.***res_mon_name*. For vendor-supplied resource monitors, IBM recommends specifying the PTX name prefix as *vendor_name***.***res_mon_name*.

rmPTX_description
A string that contains a comma-separated list, with no blanks, of message IDs. Each ID specifies a message that contains a short description (no more than 63 bytes) of a PTX context.

The first ID in the list corresponds to the context specified by the first component of the name specified by the **rmPTX_prefix** attribute, the second ID in the list corresponds to the context specified by the first two components of the name, the third ID corresponds to the context specified by the first three components of the name, and so on.

This string is required for variables of value type Counter or Quantity. Otherwise, the attribute value is a null string.

To find the message, use the message ID with the message set and message catalog attributes of the **EM_Resource_Monitor** class object for this resource monitor.

rmPTX_asnno
An integer that is the ASN.1 number equal to the SNMP Assigned Enterprise Number for the vendor that supplies this resource monitor. For IBM supplied resource monitors, this value is 2.

Here are some examples of creating an object in the **EM_Resource_Monitor** class using the **SDRCreateObjects** command:

```
SDRCreateObjects EM_Resource_Monitor \
    'rmName=aixos' \
    'rmMessage_file=harm_des.cat' \
    'rmMessage_set=3' \
    'rmConnect_type=internal'

SDRCreateObjects EM_Resource_Monitor \
    'rmName=IBM.PSSP.harmpd' \
    'rmPath=/usr/lpp/ssp/bin/haemRM/harmpd' \
    'rmMessage_file=harm_des.cat' \
    'rmMessage_set=2' \
    'rmConnect_type=server' \
    'rmPTX_prefix=0' \
    'rmPTX_description=0' \
    'rmPTX_asnno=0'

SDRCreateObjects EM_Resource_Monitor \
    'rmName=IBM.PSSP.harmld' \
    'rmPath=/usr/lpp/ssp/bin/haemRM/harmld' \
    'rmMessage_file=harm_des.cat' \
    'rmMessage_set=1' \
    'rmConnect_type=server' \
    'rmPTX_prefix=IBM/PSSP.harmld' \
    'rmPTX_description=1,2' \
    'rmPTX_asnno=2'
```

Related Information

See the following PSSP commands and files in *PSSP Command and Technical Reference*:

Commands: **haemloadcfg**, **haemcfg**, **haemctrl**, **SDRCreateObjects**, **SDRDeleteObjects**

Files: **haemloadlist**

For more information about configuring and managing the Event Management subsystem, see *PSSP Administration Guide*.

Chapter 3. Event Management Subroutine Reference

This chapter contains the reference material for each of the subroutines in the Event Management Application Programming Interface (EMAPI) and the Resource Monitor Application Programming Interface (RMAPI). Following the next two sections, which summarize each API's subroutines by function, the subroutines are listed alphabetically.

EMAPI Subroutine Summary

The EMAPI consists of the following subroutines:

Subroutine	Action
ha_em_start_session	Establish a session with the Event Management subsystem
ha_em_end_session	End a session with the Event Management subsystem
ha_em_restart_session	Reconnect a session to the Event Management subsystem
ha_em_send_command	Send a command to the Event Management subsystem Commands are provided to register for events, unregister events, and query the Event Management subsystem for information about resource variables.
ha_em_receive_response	Receive a response from the Event Management subsystem
ha_em_get_ecgid	Get an event command group ID

RMAPI Subroutine Summary

The RMAPI consists of the following subroutines:

Subroutine	Action
ha_rr_init	Register to use the RMAPI
ha_rr_makserv	Create a server session with a resource monitor manager
ha_rr_start_session	Start a session with a resource monitor manager
ha_rr_get_ctrlmsg	Get a control message from the RMAPI
ha_rr_reg_var	Register a resource variable instance with the RMAPI
ha_rr_add_var	Add resource variables to the RMAPI
ha_rr_get_interval	Get the reporting interval for a class of resource variables
ha_rr_send_val	Send variable values to the RMAPI
ha_rr_touch	Meet "send" frequency requirements when there are no resource variable values to send

© Copyright IBM Corp. 1996

ha_rr_del_var	Delete resource variables from the RMAPI
ha_rr_unreg_var	Unregister a variable instance
ha_rr_end_session	End an RMAPI session
ha_rr_terminate	Terminate an RMAPI session

ha_em_end_session Subroutine

Purpose

ha_em_end_session – End an EM client session with the Event Management subsystem

Library

EMAPI Thread-Safe Library (**libha_em_r.a**)

EMAPI Library (not thread-safe) (**libha_em.a**)

Syntax

```
#include <ha_emapi.h>

int
    ha_em_end_session(
        int                     session_fd,
        struct ha_em_err_blk    *em_errb)
```

Parameters

session_fd	The file descriptor of the session that is to be ended. This file descriptor was returned by a previous call to either the **ha_em_start_session** or **ha_em_restart_session** subroutine.
em_errb	A pointer to an error block structure.

Description

The **ha_em_end_session** subroutine is used by an EM client to end a session.

Restrictions

Using the **ha_em_end_session** subroutine to end a session causes all event registrations for the session to be lost. To continue to use an existing session to which the connection has been lost, use the **ha_em_restart_session** subroutine to reestablish the connection.

When you use the EMAPI in a threaded program, IBM recommends that an Event Management session be used by only one thread at a time. If multiple threads try to use the same Event Management session at the same time, EMAPI subroutines may return the **HA_EM_EBUSY** error code. The thread-safe version of the EMAPI library was designed assuming that each thread using the EMAPI would have its own Event Management session or sessions.

Return Values

If the **ha_em_end_session** subroutine is successful, it returns a value of 0 and the session is ended. The file descriptor that was specified on input must no longer be used as an argument to the **select** or **poll** system call.

Error Values

If the **ha_em_end_session** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The EMAPI error block and error numbers are defined in the **ha_emapi.h** header file. For more information on EMAPI errors, see "EMAPI Errors (err_emapi)" on page 116.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

The file descriptor that was provided on input in the *session_fd* argument is still valid and can be used in another attempt to end the session.

Examples

For examples of using EMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/emapi**.

Files

ha_emapi.h.

Suggested Reading and Prerequisite Information

Chapter 1, "Understanding Event Management" on page 1.

Related Information

Structures in the **ha_emapi.h** header file: **ha_em_err_blk**

Subroutines: **ha_em_start_session**, **ha_em_restart_session**

ha_em_get_ecgid Subroutine

Purpose

ha_em_get_ecgid – Get an event command group ID

Library

EMAPI Thread-Safe Library (**libha_em_r.a**)

EMAPI Library (not thread-safe) (**libha_em.a**)

Syntax

```
#include <ha_emapi.h>

ha_em_ecgid_t
    ha_em_get_ecgid(
        ha_em_eid_t event_id)
```

Parameters

event_id	An event ID. This event ID was returned when the caller registered for events by a previous call to the **ha_em_send_command** subroutine using the **HA_EM_CMD_REG** or **HA_EM_CMD_REG2** command.

Description

The **ha_em_get_ecgid** subroutine is used by an EM client to get the ID of the event command group to which the event specified on input belongs. All events for which the caller registered through a single **HA_EM_CMD_REG** or **HA_EM_CMD_REG2** command belong to the same event command group.

Return Values

The **ha_em_get_ecgid** subroutine returns the event command group ID. This event command group ID is valid until all of the events that were registered by the command are unregistered. Once these events are unregistered, the EMAPI may reuse the event command group ID.

Error Values

None.

Examples

For examples of using EMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/emapi**.

ha_em_get_ecgid

Files

ha_emapi.h.

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Subroutine:

ha_em_send_command

ha_em_receive_response Subroutine

Purpose

ha_em_receive_response – Receive a response from the Event Management subsystem

Library

EMAPI Thread-Safe Library (**libha_em_r.a**)

EMAPI Library (not thread-safe) (**libha_em.a**)

Syntax

```
#include <ha_emapi.h>

int
    ha_em_receive_response(
        int                     session_fd,
        struct ha_em_rsp_blk    **em_rsp_blk,
        struct ha_em_err_blk    *em_errb)
```

Parameters

session_fd	The file descriptor of the Event Management session for which the response is to be received.
em_rsp_blk	A pointer to the address of a buffer containing a response block structure. The response block contains information returned by the Event Management subsystem in response to a previously issued command.
em_errb	A pointer to an error block structure.

Description

The **ha_em_receive_command** subroutine is used by an EM client to receive information returned in response to a previously issued command.

When an EM client starts or restarts a session, it obtains a file descriptor. The EM client uses this file descriptor as an argument to the **select** or **poll** system call. When the system call that is used indicates that there is data to be read from the file descriptor, the EM client calls the **ha_em_receive_response** subroutine to read the data and thereby receive the response from the Event Management subsystem.

The *session_fd* argument contains the file descriptor of the Event Management session that has data to be read.

The **ha_em_rsp_blk** response block has the following definition:

```
struct ha_em_rsp_blk {
       int           em_rsp_blk_len;
       int           em_rsp_num_resp;
       short         em_cmd;
       short         em_subcmd;
       ha_em_qid_t   em_qid;
       int           em_qend;
       union ha_em_resp_blk {
           struct ha_em_rpb_event em_rpb_event[1];
           struct ha_em_rpb_qcur  em_rpb_qcur[1];
           struct ha_em_rpb_qdef  em_rpb_qdef[1];
           struct ha_em_rpb_rerr  em_rpb_rerr[1];
           struct ha_em_rpb_qerr  em_rpb_qerr[1];
       } em_resp_blk;
}
```

In an actual response block, the arrays in the **em_resp_blk** union may contain any number of elements; the **em_rsp_num_resp** field contains the actual number that are returned. The **em_cmd** and **em_subcmd** fields contain the values of the command and subcommands to which this is a response. If the response block contains information in response to a query and the **em_qend** field is nonzero, this response is the last piece of information to be returned for the query specified by the **em_qid** field. The remaining fields in the response block are used as specified in the response descriptions that follow.

Responses to Register Event Commands

When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is **HA_EM_CMD_REG** or **HA_EM_CMD_REG2**, the response block contains one or more events. The **em_rpb_event** array is used in the response block.

The **ha_em_rpb_event** structure has the following definition:

```
struct ha_em_rpb_event {
     union ha_em_errnum  em_error;
     ha_em_eid_t         em_event_id;
     unsigned long       em_event_flags;
     struct timeval      em_timestamp;
     int                 em_location;
     char                *em_name;
     char                *em_ivector;
     enum ha_emData_Type em_data_type;
     union ha_em_val {
          long      em_vall;
          float     em_valf;
          void      *em_valsbs;
     } em_val;
}
```

The **ha_em_errnum** union contains information about any error that may have occurred. It has the following definition:

```
union ha_em_errnum {
    unsigned int    em_error_number;
    unsigned short  em_error_codes[2];
};
```

```
#define em_errnum   em_error.em_error_number    /* for quick tests  */
#define em_generr   em_error.em_error_codes[0]  /* general error code */
#define em_specerr  em_error.em_error_codes[1]  /* specific error code */
```

The **em_error_number** field indicates whether there is an error. If there is no error, the **em_error_number** field contains a value of 0 and the remainder of **ha_em_rpb_event** information should be used.

If the **em_error_number** field is nonzero, there is an error and the validity of the remaining fields in the **ha_em_rpb_event** structure depend on the error. For more information, see “Error Responses to Register and Unregister Event Commands” on page 61.

The **em_event_id** field contains the ID of the event. The value can be used to match this event to the information that was used to register for the event.

The **em_event_flags** field contains one of the following bit flags:

HA_EM_EVENT_RE_ARM
: The event was generated from the rearm predicate.

HA_EM_EVENT_PRED_FALSE
: The predicate specified in the **HA_EM_SCMD_REVAL** subcommand evaluated to false.

HA_EM_EVENT_UNREG
: The event was unregistered.

The **em_timestamp** field is the time that the event was generated by the Event Management subsystem. The **em_location** field contains the number of the node where the event was generated.

The **em_name** field is a pointer to a string that contains the name of the resource variable that was used to register the predicate that generated the event. The **em_ivector** field is a pointer to the instance vector of the variable instance from which the event was generated. The **em_val** field is the current value of the variable instance at the time of the event. The **em_data_type** field indicates whether the value is a long, a float, or a structured byte string. For information about resource variable definitions, see “Resource Variables (haemrviv)” on page 141.

If, when registering for this event, the EM client supplied a pointer to a callback routine, then instead of returning the **ha_em_rpb_event** structure in a buffer, the **ha_em_receive_response** subroutine invokes the callback routine. The session file descriptor, a pointer to the **ha_em_rpb_event** structure, and the value of the **em_cb_arg** field, which was specified when registering for this event, are all passed to the callback routine. The memory addressed by the pointer to the **ha_em_rpb_event** structure must not be freed or modified by the callback routine. If all of the events received by the invocation of the **ha_em_receive_response** subroutine have callback routines, the **ha_em_receive_response** subroutine returns a value of 0. Otherwise, it returns a value greater than zero and also returns in the response block all of the events that were not processed by callback routines.

Responses to Unregister Event Commands

When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is **HA_EM_CMD_UNREG**, the response block contains one or more unregister event responses. The **em_rpb_event** array is used in the response block. For a description of the information that is contained in each **ha_em_rpb_event** structure, see "Responses to Register Event Commands" on page 56.

The **em_event_id** field contains the ID of the event that has been unregistered. The **em_event_flags** field contains the **HA_EM_EVENT_UNREG** flag. This flag is used so that the callback routine, if any is specified, can distinguish between an event and an unregister response. The **em_timestamp** field is the time that the response was generated by the Event Management subsystem. All other fields in the **em_rpb_event** array element are undefined.

If the EM client supplied a callback routine for an event that is specified in one of the responses, then instead of returning the **ha_em_rpb_event** structure in a buffer, the **ha_em_receive_response** subroutine invokes the callback routine. The session file descriptor, a pointer to the **ha_em_rpb_event** structure, and the value of the **em_cb_arg** field, which was specified when registering for the event, are passed to the callback routine. The memory addressed by the pointer to the **ha_em_rpb_event** structure must not be freed or modified by the callback routine. If all of the events specified in the responses have callback routines, the **ha_em_receive_response** subroutine returns a value of 0. Otherwise, it returns a value greater than zero and also returns in the response block all of the unregistered event responses that were not processed by callback routines.

Responses to Queries about Current Values

When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is **HA_EM_CMD_QUERY** and the value of the **em_subcmd** field is **HA_EM_SCMD_QCUR**, the response is a reply to a query for the current values of the resource variables that were targeted in the query. The **em_qid** field contains the unique value that was returned to the EM client when the query command was issued. The **em_rpb_qcur** array is used in the response block.

The **ha_em_rpb_qcur** structure has the following definition:

```
struct ha_em_rpb_qcur {
       union ha_em_errnum  em_error;
       int                 em_location;
       char                *em_name;
       char                *em_ivector;
       int                 em_data_type;
       union ha_em_val {
            long            em_vall;
            float           em_valf;
            void            *em_valsbs;
       } em_val;
}
```

The **ha_em_errnum** union contains information about any error that may have occurred. It has the following definition:

```
union ha_em_errnum {
    unsigned int    em_error_number;
    unsigned short  em_error_codes[2];
};

#define em_errnum   em_error.em_error_number    /* for quick tests  */
#define em_generr   em_error.em_error_codes[0]  /* general error code */
#define em_specerr  em_error.em_error_codes[1]  /* specific error code */
```

The **em_error_number** field indicates whether there is an error. If there is no error, the **em_error_number** field contains a value of 0 and the remainder of **ha_em_rpb_qcur** information should be used.

If the **em_error_number** field is nonzero, there is an error and the validity of the remaining fields in the the **ha_em_rpb_qcur** structure depend on the error. For more information, see “Error Responses to Query Commands” on page 63.

The **em_location** field contains the number of the node that contains the resource variable. The **em_name** field is a pointer to a string that contains the name of the resource variable. The **em_ivector** field is a pointer to a string that contains the instance vector of the variable instance. The **em_val** field is the current value of the variable. The **em_data_type** field indicates whether the value is a long, a float, or a structured byte string. For information about resource variable definitions, see “Resource Variables (haemrviv)” on page 141.

If, when issuing the query command that resulted in this response, the EM client supplied a pointer to a callback routine, then instead of returning the **ha_em_rsp_blk** structure in a buffer, the **ha_em_receive_response** subroutine invokes the callback routine. The session file descriptor, a pointer to the **ha_em_rsp_blk** structure, and the value of the **em_qcb_arg** field, which was specified when the query command was issued, are passed to the callback routine. The memory addressed by the pointer to the **ha_em_rsp_blk** structure or by any of the pointers contained within it must not be freed or modified by the callback routine. The **ha_em_receive_response** subroutine then returns a value of 0.

Responses to Queries about Defined Variables and Predicates

When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is **HA_EM_CMD_QUERY** and the value of the **em_subcmd** field is **HA_EM_SCMD_QDEF**, the response is a reply to a query for a list of resource variables, targeted in the query, that are defined in the Event Management Configuration Database (EMCDB). The **em_qid** field contains the unique value that was returned to the EM client when the query command was issued. The **em_rpb_qdef** array is used in the response block.

The **ha_em_rpb_qdef** structure has the following definition:

```
struct ha_em_rpb_qdef {
     union ha_em_errnum      em_error;
     char                    *em_name;
     char                    *em_descrp;
     enum ha_emValue_Type    em_value_type;
     enum ha_emData_Type     em_data_type;
     char                    *em_sbs_format;
     char                    *em_init_value;
     char                    *em_class;
     char                    *em_ivector;
     char                    *em_ivector_descrp;
     char                    *em_ptx_name;
     char                    *em_dflt_pred;
     char                    *em_event_descrp;
     char                    *em_locator;
     char                    *em_order_group;
}
```

The **ha_em_errnum** union contains information about any error that may have occurred. It has the following definition:

```
union ha_em_errnum {
    unsigned int    em_error_number;
    unsigned short  em_error_codes[2];
};

#define em_errnum   em_error.em_error_number    /* for quick tests  */
#define em_generr   em_error.em_error_codes[0]  /* general error code */
#define em_specerr  em_error.em_error_codes[1]  /* specific error code */
```

The **em_error_number** field indicates whether there is an error. If there is no error, the **em_error_number** field contains a value of 0 and the remainder of **ha_em_rpb_qdef** information should be used.

If the **em_error_number** field is nonzero, there is an error and the validity of the remaining fields the **ha_em_rpb_qdef** structure depend on the error. For more information, see “Error Responses to Query Commands” on page 63.

The **em_name** field is a pointer to a string that contains the name of the resource variable. The **em_descrp** field is a pointer to a string that contains the text of the resource variable description.

The **em_value_type** field is one of Counter, Quantity, or State. The **em_data_type** field indicates whether the variable value is a long, a float, or a structured byte string. If the variable is a structured byte string, the **em_sbs_format** field points to a string that contains a description of its format.

The **em_init_value** field points to a string that contains the initial value of the variable. For a structured byte string, the initial value is specified as a list of name/value pairs, separated by commas, in serial number order. The **em_class** field is a pointer to a string that contains the variable's resource class.

The **em_ivector** field points to a string that contains the variable's instance vector definition. This definition is a semicolon-separated list of name/value pairs, where the value is the data type of the vector element. The **em_ivector_descrp** field points to a string that contains the text of each vector element's description; the

descriptions are separated by the newline character. For information about resource variable definitions, see “Resource Variables (haemrviv)” on page 141.

The **em_ptx_name** field points to a string that contains the name that is used to read and write the resource variable in Performance Toolbox (PTX) shared memory. If the resource variable is of value type State, the **em_ptx_name** field points to a null string.

The **em_dflt_pred** field is a pointer to a string that contains the default predicate. If no default predicate is defined, the **em_dflt_pred** field points to a null string. For information about defining predicates, see “Predicates (haempred)” on page 138.

The **em_event_descrp** field is a pointer to a string that contains the text of the event description for the event specified by the default predicate.

The **em_locator** field is a pointer to a string that contains the Locator value specified for the variable. If no Locator value is defined, the **em_locator** field points to a null string.

The **em_order_group** field is reserved for IBM use.

If, when issuing the query command that resulted in this response, the EM client supplied a pointer to a callback routine, then instead of returning the **ha_em_rsp_blk** structure in a buffer, the **ha_em_receive_response** subroutine invokes the callback routine. The session file descriptor, a pointer to the **ha_em_rsp_blk** structure, and the value of the **em_qcb_arg** field, which was specified when the query command was issued, are passed to the callback routine. The memory addressed by the pointer to the **ha_em_rsp_blk** structure or by any of the pointers contained within it must not be freed or modified by the callback routine. The **ha_em_receive_response** subroutine then returns a value of 0.

Error Responses to Register and Unregister Event Commands

Errors can be returned in response to the registration commands **HA_EM_CMD_REG** and **HA_EM_CMD_REG2** in one of two ways, as follows:

- When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is either **HA_EM_CMD_REG** or **HA_EM_CMD_REG2**, then the response block contains one or more events that may have an error. The **em_rpb_event** array is used in the response block.

  If a callback routine has been specified for an event, it is invoked if the event has an error.

  If the event has an error, the **em_error_number** field of the **em_error** union is nonzero.

  An error returned in this way indicates that the corresponding event has been successfully registered but an error was encountered after registration. The event remains registered.

- When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is either **HA_EM_CMD_RERR** or **HA_EM_CMD_R2ERR**, then an error was encountered while processing the registration command **HA_EM_CMD_REG** or **HA_EM_CMD_REG2**, respectively. The **em_rpb_rerr** array is used in the response block.

Callback routines are never invoked for this type of response.

An error returned in this way indicates that the corresponding event has not been registered.

The **ha_em_rpb_rerr** structure has the following definition:

```
struct ha_em_rpb_rerr {
    union ha_em_errnum  em_error;
    char                *em_name;
    char                *em_ivector;
    char                *em_pred;
    char                *em_rapred;
    short               em_errinfo0;
    unsigned short      em_errinfo1;
    ha_em_eid_t         em_event_id;
};
```

The **ha_em_errnum** union, in either the **ha_em_rpb_event** structure or the **ha_em_rpb_rerr** structure, contains information about the error that occurred. It has the following definition:

```
union ha_em_errnum {
    unsigned int    em_error_number;
    unsigned short  em_error_codes[2];
};

#define em_errnum   em_error.em_error_number    /* for quick tests  */
#define em_generr   em_error.em_error_codes[0]  /* general error code */
#define em_specerr  em_error.em_error_codes[1]  /* specific error code */
```

The **em_error_number** field indicates whether there is an error. When the field is nonzero, there is an error.

The **em_error_codes** array contains further information about the error. The first element of the array is a general error code; the second element is a more specific error code. For information about error codes, see “EMAPI Errors (err_emapi)” on page 116.

The **em_name** field is a pointer to a string that contains the name of the resource variable specified in the original registration command. The **em_ivector** field is a pointer to the instance vector that was specified in the original registration command. The **em_pred** and **em_rapred** fields point to strings that contain the predicate and rearm predicate, respectively, specified in the original registration command. For information about defining predicates, see “Predicates (haempred)” on page 138.

The **em_errinfo0** and **em_errinfo1** fields may contain additional error information, depending on the specific error.

The **em_event_id** field contains the ID of the event related to the error. The value can be used to match this event to the information that was used to register for the event.

Errors can also be returned in response to the unregister event command **HA_EM_CMD_UNREG**. When the buffer returned by the

ha_em_receive_response subroutine contains a response block in which the value of the **em_cmd** field is **HA_EM_CMD_UNREG**, then the response block contains one or more unregistration events that may have an error. The setting of **HA_EM_EVENT_UNREG** bit in the **em_event_flags** field indicates an unregistration event. The **em_rpb_event** array is used in the response block.

If a callback routine has been specified for an event, it is invoked if the event has an error.

If the event has an error, the **em_error_number** field of the **em_error** union is nonzero.

An error returned in this way indicates that the corresponding event has not been unregistered.

Error Responses to Query Commands

Errors can be returned in response to the query command **HA_EM_CMD_QUERY** in one of two ways, as follows:

- When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is **HA_EM_CMD_QUERY** and the value of the **em_subcmd** field is **HA_EM_SCMD_QCUR**, then the response block contains one or more responses that may have an error. The **em_rpb_qcur** array is used in the response block.

  If a callback routine has been specified for the query, it is invoked if the query response has an error.

  If the query response has an error, the **em_error_number** field of the **em_error** union is nonzero.

- When the buffer returned by the **ha_em_receive_response** subroutine contains a response block in which the value of the **em_cmd** field is **HA_EM_CMD_QERR**, then an error was encountered while processing the query command. This can occur while processing either of the query subcommands **HA_EM_SCMD_QCUR** or **HA_EM_SCMD_QDEF**. The **em_subcmd** field in the response block contains the original subcommand value. The **em_rpb_qerr** array is used in the response block.

  Callback routines are never invoked for this type of response.

The **ha_em_rpb_qerr** structure has the following definition:

```
struct ha_em_rpb_qerr {
    union ha_em_errnum  em_error;
    char                *em_class;
    char                *em_name;
    char                *em_ivector;
    unsigned short      em_errinfo;
};
```

The **ha_em_errnum** union in either the **ha_em_rpb_qcur** structure or the **ha_em_rpb_qerr** structure contains information about the error that occurred. It has the following definition:

```
union ha_em_errnum {
    unsigned int    em_error_number;
    unsigned short  em_error_codes[2];
};

#define em_errnum   em_error.em_error_number    /* for quick tests  */
#define em_generr   em_error.em_error_codes[0]  /* general error code */
#define em_specerr  em_error.em_error_codes[1]  /* specific error code */
```

The **em_error_number** field indicates whether there is an error. When the field is nonzero, there is an error.

The **em_error_codes** array contains further information about the error. The first element of the array is a general error code; the second element is a more specific error code. For information about error codes, see “EMAPI Errors (err_emapi)” on page 116.

The **em_class** field is a pointer to a string that contains the name of the resource variable class as specified in the original query command.

The **em_name** field is a pointer to a string that contains the name of the resource variable as specified in the original query command.

The **em_ivector** field is a pointer to the instance vector of the variable instance as specified in the original query command.

The **em_errinfo** field may contain additional error information, depending on the specific error.

Restrictions

When you use the EMAPI in a threaded program, IBM recommends that an Event Management session be used by only one thread at a time. If multiple threads try to use the same Event Management session at the same time, EMAPI subroutines may return the **HA_EM_EBUSY** error code. The thread-safe version of the EMAPI library was designed assuming that each thread using the EMAPI would have its own Event Management session or sessions.

Return Values

If the **ha_em_receive_response** subroutine returns a value that is greater than 0, it also returns a pointer to a buffer that contains a response block. The *em_rsp_blk* argument specified on input points to an area where the buffer pointer is to be returned. It is the responsibility of the calling routine to free the buffer. When the buffer is freed, any memory areas to which there are pointers in the response block can no longer be accessed.

If the **ha_em_receive_response** subroutine returns a value that is equal to 0, the response either was intended for the EMAPI or it contained events or query responses, all of which were processed by the callback routines supplied by the EM client. No further action by the calling routine is necessary.

Error Values

If the **ha_em_receive_response** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

If the **HA_EM_ECONNLOST** error is received, it is recommended that the EM client no longer use the session file descriptor that was specified on the failing call on subsequent **select** system calls. To reconnect to the Event Management subsystem, use the **ha_em_restart_session** subroutine.

The EMAPI error block and error numbers are defined in the **ha_emapi.h** header file. For more information on EMAPI errors, see "EMAPI Errors (err_emapi)" on page 116.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using EMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/emapi**.

Files

ha_emapi.h.

Suggested Reading and Prerequisite Information

Chapter 1, "Understanding Event Management" on page 1.

Related Information

Structures in the **ha_emapi.h** header file: **ha_em_rsp_blk**, **ha_em_err_blk**, **ha_em_rb_reg**, **ha_em_rb_query**, **ha_em_rpb_event**, **ha_em_rpb_qcur**, **ha_em_rpb_qdef**

Subroutines: **ha_em_send_command**

ha_em_restart_session Subroutine

Purpose

ha_em_restart_session – Restart an EM client session to the Event Management subsystem

Library

EMAPI Thread-Safe Library (**libha_em_r.a**)

EMAPI Library (not thread-safe) (**libha_em.a**)

Syntax

```
#include <ha_emapi.h>

int
    ha_em_restart_session(
        int                     session_fd,
        struct ha_em_err_blk    *em_errb)
```

Parameters

session_fd The file descriptor of the session that is to be reconnected. This file descriptor was returned by a previous call to either the **ha_em_start_session** or **ha_em_restart_session** subroutine.

em_errb A pointer to an error block structure.

Description

The **ha_em_restart_session** subroutine is used by an EM client to reconnect to an existing session when the connection has been lost.

If an error returned by the **ha_em_send_command** or **ha_em_receive_response** subroutine indicates that the connection to the Event Management subsystem has been lost, this subroutine can be called to create a new connection. If the EM client ended the existing session and established a new session, all of the event registrations for the old session would be lost. The **ha_em_restart_session** subroutine lets the EM client reestablish a connection that has been lost without having to reregister all of the events.

Restrictions

When you use the EMAPI in a threaded program, IBM recommends that an Event Management session be used by only one thread at a time. If multiple threads try to use the same Event Management session at the same time, EMAPI subroutines may return the **HA_EM_EBUSY** error code. The thread-safe version of the EMAPI library was designed assuming that each thread using the EMAPI would have its own Event Management session or sessions.

Return Values

If the return value of the **ha_em_restart_session** subroutine is greater than or equal to 0, the return value is a file descriptor. The **ha_em_restart_session** subroutine closes the file descriptor that was provided on input in the *session_fd* argument and returns a new one. The newly returned file descriptor replaces the file descriptor that was provided on input, and the EM client can use the new file descriptor in the same way as it used the file descriptor that was originally returned by the **ha_em_start_session** subroutine.

Error Values

If the **ha_em_restart_session** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The EMAPI error block and error numbers are defined in the **ha_emapi.h** header file. For more information on EMAPI errors, see "EMAPI Errors (err_emapi)" on page 116.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

The file descriptor that was provided on input in the *session_fd* argument is still valid and can be used in another attempt to restart the session.

Examples

For examples of using EMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/emapi**.

Files

ha_emapi.h.

Suggested Reading and Prerequisite Information

Chapter 1, "Understanding Event Management" on page 1.

Related Information

Structures in the **ha_emapi.h** header file: **ha_em_err_blk**

Subroutines: **ha_em_start_session**, **ha_em_end_session**

ha_em_send_command Subroutine

Purpose

ha_em_send_command – Send a command to the Event Management subsystem

Library

EMAPI Thread-Safe Library (**libha_em_r.a**)

EMAPI Library (not thread-safe) (**libha_em.a**)

Syntax

```
#include <ha_emapi.h>

int
    ha_em_send_command(
        int                     session_fd,
        struct ha_em_cmd_blk    *em_cmdb,
        struct ha_em_err_blk    *em_errb)
```

Parameters

session_fd	The file descriptor of the Event Management session to which the command is to be sent. This file descriptor was returned by a previous call to either the **ha_em_start_session** or **ha_em_restart_session** subroutine.
em_cmdb	A pointer to a command block structure. The command block contains the command to be sent to the Event Management subsystem in the specified session and any additional parameters that are needed to execute the command.
em_errb	A pointer to an error block structure.

Description

The **ha_em_send_command** subroutine is used by the EM client to send a command to the instance of the Event Management subsystem that is associated with the session supplied on input. The command and any additional parameters that it requires are contained in the command block specified on input.

The **ha_em_cmd_blk** command block has the following definition:

```
struct ha_em_cmd_blk {
       int            em_cmd_num_elem;
       short          em_cmd;
       short          em_subcmd;
       ha_em_qid_t    em_qid;
       void           (*em_qcb)(int, struct ha_em_rsp_blk *, void *);
       void           *em_qcb_arg;
       union ha_em_res_blk {
             struct ha_em_rb_reg     em_rb_reg[1];
             ha_em_eid_t             em_rb_unreg[1];
             struct ha_em_rb_query   em_rb_query[1];
       } em_res_blk;
}
```

In an actual command block, the arrays in the **ha_em_res_blk** union may contain any number of elements; the **em_cmd_num_elem** field contains the actual number that are allocated. It is the responsibility of the caller to allocate a buffer to contain the command block with the desired number of array elements.

The values of the **em_cmd** fields are:

HA_EM_CMD_REG
: Register for an event from a single predicate, the event predicate. If a rearm predicate is defined, it does not generate an event.

HA_EM_CMD_REG2
: Register for an event from both an event predicate and a rearm predicate.

HA_EM_CMD_UNREG
: Unregister events

HA_EM_CMD_QUERY
: Query for information

For the **HA_EM_CMD_REG** or **HA_EM_CMD_REG2** command, the values of the **em_subcmd** field are:

0
: No subcommand is required.

HA_EM_SCMD_REVAL
: Generate an event at the first observation, even if the predicate evaluates to false.

For the **HA_EM_CMD_QUERY** command, the values of the **em_subcmd** fields are:

HA_EM_SCMD_QCUR
: Query the current values of resource variables

HA_EM_SCMD_QDEF
: Query defined resource variables and their default predicates

The remaining fields in the command block are used as specified in the command descriptions that follow.

HA_EM_CMD_REG and HA_EM_CMD_REG2 Commands (Register for Events)

An EM client uses the command value **HA_EM_CMD_REG** or **HA_EM_CMD_REG2** to register for events. The **em_rb_reg** array is used in the command block. One array element is defined for each event to be registered.

The **ha_em_rb_reg** structure has the following definition:

```
struct ha_em_rb_reg {
       char          *em_name;
       char          *em_ivector;
       char          *em_pred;
       char          *em_rapred;
       ha_em_eid_t   em_event_id;
       void         (*em_cb)(int, struct ha_em_rpb_event *, void *);
       void          *em_cb_arg;
}
```

The **em_name** field is a pointer to a string that contains the name of the resource variable from which the event is to be generated.

The **em_ivector** field is a pointer to a string that contains an instance vector. The event is registered for all instances of the specified resource variable that match the instance vector.

The instance vector is a semicolon-separated list of name/value pairs. A name/value pair consists of an instance vector element name followed by an equal sign followed by the value of the vector element. There are no blanks in the instance vector. A vector element is wildcarded by specifying its value as an asterisk. The instance vector string must contain each element that is defined for the variable's instance vector. An element value may consist of a single value, a range of values, a comma-separated list of single values, or a comma-separated list of ranges. A range takes the form *a-b* and is valid only for vector elements of type integer.

The instance vector must contain each element that is defined for the resource variable. The instance vector may be wildcarded to match more than one instance by specifying an asterisk for the value of one or more elements, as shown in the following examples:

```
NodeNum=5;VG=rootvg;LV=hd4
NodeNum=*;VG=rootvg;LV=hd4
NodeNum=*;VG=*;LV=*
```

In the first example, the Event Management subsystem registers an event for one instance, associated with node 5, volume group **rootvg**, and logical volume **hd4**. In the second example, the Event Management subsystem registers an event for all instances that are associated with all nodes, volume group **rootvg**, and logical volume **hd4**. In the last example, the Event Management subsystem registers an event for all instances that are associated with all nodes, all volume groups, and all logical volumes.

The **em_pred** field is a pointer to a string that contains the predicate that is to be used to generate the event. If the predicate string is null, the default predicate, which is defined for the specified variable in the Event Management Configuration

Database (EMCDB), is to be used to generate the event. For information about defining predicates, see “Predicates (haempred)” on page 138.

The **em_rapred** field is a pointer to a string that contains a rearm predicate, which has the same syntax as the predicate pointed to by the **em_pred** field. If the **em_rapred** field points to a null string, a rearm predicate is not specified.

If a rearm predicate is specified, whenever a "true" event is generated by applying the event predicate to the observed variable instance, the rearm predicate is applied to the instance of the variable as it is observed from then on. When the rearm predicate is true, the original event predicate is used again. In other words, whenever either predicate is true, the other predicate is used for that instance until it is true.

When the **HA_EM_CMD_REG** command is specified, an event is generated only from the event predicate, not from the rearm predicate. However, when the **HA_EM_CMD_REG2** command is specified, an event is also generated when the rearm predicate is TRUE. The event response indicates which predicate was used to generate the event.

Rearm predicates can be used in a couple of different ways. One common way is to define the rearm predicate as the inverse of the event predicate. For example, if the event predicate tests whether a resource variable value is "on," the rearm predicate tests whether it is "off." If it registers for events from both predicates, the EM client is notified when the resource variable changes rather than each time it is observed.

A rearm predicate can also be used with the event predicate to define an upper and lower boundary for a condition of interest. For example, suppose an EM client is interested in how much disk space is used in the system. It defines a normal disk space condition to be in the range between 80% and 90% of the total space. However, if the amount of disk space used rises above 90% or falls below 80%, the EM client wants to be notified.

In this case, it would define the event predicate to be "the amount of disk space used is greater than 90%" and the rearm predicate to be "the amount of disk space used is less than 80%." The EM client would also register for events from both predicates.

The first time disk space usage rises above 90%, the Event Management subsystem notifies the EM client and switches to the rearm predicate. The EM client is not notified again until disk space usage decreases to an amount less than 80%, and then the Event Management subsystem switches to the event predicate.

The predicate must contain the resource variable name, represented by the letter "X," in any one or more of its modified or unmodified forms. Events generated by the specified predicate(s) from all matching instances of the specified resource variable have the same event ID. These events are differentiated by inclusion of the instance vector value with the event.

If the **HA_EM_SCMD_REVAL** subcommand is specified with either the **HA_EM_CMD_REG** or **HA_EM_CMD_REG2** command, an event is generated at the first observation of an instance of the variable after registration, even if the predicate is false. The event response indicates whether the predicate was true or false. This allows the EM client to obtain an initial state of the resource variable.

When the **ha_em_send_command** subroutine returns successfully, the **em_event_id** field has been set by the EMAPI with an event ID. The event ID is returned with each generated event for reference back to the registration information. At this point, the EM client can now expect to receive events from the registered resource variables.

All events that are registered by a single call to the **ha_em_send_command** subroutine can be identified by an event command group ID. This ID can be used to relate individual event responses back to the command with which they were registered. The event command group ID can be obtained by calling the **ha_em_get_ecgid** subroutine.

If the caller specifies a function pointer in the **em_cb** field, the specified function is used as a callback routine by the EMAPI. Whenever an event that is associated with this registration is received by the EMAPI, the callback routine is called.

The first argument to the callback routine is the file descriptor of the session in which the event was registered. The second argument is a pointer to a structure that contains the event information. The third argument is the value of the **em_cb_arg** field. The **em_cb_arg** field specifies an argument, meaningful in the context of the specified event only to the EM client, that can be passed to the callback routine. The argument must be valid when the callback routine is invoked. For example, a pointer to a stack must be valid when the callback routine is invoked, or an argument must be valid in the context of the thread in which the callback routine executes. For more information about callback routines, see the man page for the **ha_em_receive_response** subroutine.

HA_EM_CMD_UNREG Command (Unregister Events)

An EM client uses the command value **HA_EM_CMD_UNREG** to stop receiving events for which it registered. The **em_rb_unreg** array is used in the command block. One array element is defined for each event that is no longer to be sent to the EM client. The array element contains an event ID that was returned in the **ha_em_rb_reg** structure when the EM client registered for events.

Because the command to unregister events is sent asynchronously to the schedule on which events are generated by the Event Management subsystem, the EM client may still receive the events for which it just unregistered until the Event Management subsystem can process the command. Once the command has been processed, the Event Management subsystem generates a response for each unregistered event. This response has a format similar to that of the event that has just been unregistered.

If the EM client specified a callback routine for any of the unregistered events, then, when the unregistered event response is received by the EMAPI, the callback routine is invoked with the same arguments as if the response were the event. Thus, the callback routine sees some number of events followed by an indication that no more events will be forthcoming.

HA_EM_CMD_QUERY Command (Query for Information)

An EM client uses the **HA_EM_CMD_QUERY** command to query for information. It is used with the following subcommands:

HA_EM_SCMD_QCUR
: Obtain the current value of one or more resource variables.

HA_EM_SCMD_QDEF
: Obtain a list of resource variables and default predicates that are defined in the Event Management Configuration Database (EMCDB).

Each query subcommand uses the **em_rb_query** array in the command block. One array element is defined for each query; a single query can specify multiple resource variables.

The **ha_em_rb_query** structure has the following definition:

```
struct ha_em_rb_query {
       char      *em_class;
       char      *em_name;
       char      *em_ivector;
}
```

For both query subcommands, the target of the operation is one or more resource variables. The **em_class**, **em_name**, and **em_ivector** fields are used to define the resource variables to be queried. Resource variables may be specified individually in separate **ha_em_rb_query** structures or one **ha_em_rb_query** structure may be used to specify many resource variables through the use of wildcards. Any number of **ha_em_rb_query** structures may be specified in either manner in one subcommand.

If the **em_class** field is a pointer to a string that contains the name of a resource variable class, then all variables in that class, as further limited by the **em_name** and **em_ivector** fields, are the targets of the query. If the **em_class** field points to a null string, then variables of all classes, as further limited by the **em_name** and **em_ivector** fields, are the targets of the query.

The **em_name** field points to a string that contains the resource variable name that is the target of the query. The resource variable name may be wildcarded in one of two ways:

- Specifying the name as a null string.
- Truncating the name after any component.

When the resource variable name is wildcarded in the first manner, then all resource variables, as further limited by the **em_class** and **em_ivector** fields, are targets of the query. When the resource variable name is wildcarded in the second manner, all resource variables whose high-order (leftmost) components match the specified resource variable name, as further limited by the **em_class** and **em_ivector** fields, are the targets of the query.

The **em_ivector** field is a pointer to a string that contains an instance vector. All instances, or definitions in the case of **HA_EM_SCMD_QDEF**, of the resource variable(s) specified by the **em_class** and **em_name** fields that match the instance vector are the targets of the query.

If the query is **HA_EM_SCMD_QCUR**, the instance vector is a semicolon-separated list of name/value pairs. A name/value pair consists of an instance vector element name followed by an equal sign followed by the value of the vector element. An element value may consist of a single value, a range of values, a comma-separated list of single values, or a comma-separated list of ranges. A range takes the form *a-b* and is valid only for vector elements of type integer. There are no blanks in the instance vector.

If the query is **HA_EM_SCMD_QDEF**, the instance vector is a semicolon-separated list of vector element names only.

A vector element is wildcarded by specifying its value as the asterisk character. Only variables that are defined to contain the elements, and only the elements, specified in the instance vector are candidates for a match of the query. If any element of the instance vector consists of the asterisk character, rather than a name/value pair (or name, in the case of the **HA_EM_SCMD_QDEF** subcommand), all variables that are defined to contain *at least* the remaining specified elements are candidates for a match of the query. The entire vector is wildcarded if the **em_ivector** field points to a string that contains a single asterisk.

Here are some examples of using wildcards for instance vectors specified on a query:

```
NodeNum=5;VG=rootvg;LV=hd4
NodeNum=*;VG=rootvg;LV=hd4
NodeNum=*;VG=*;LV=*

NodeNum=9
NodeNum=*

NodeNum=9;VG=*;*
NodeNum=*;*
```

For these examples, assume that the class and resource variable names are specified as null strings. If either the class or resource variable name or both are specified, matches for the query are restricted accordingly.

In the first three examples, all variables whose instance vectors are defined to contain the elements **NodeNum**, **VG**, and **LV**, and only those elements, are matched. The instances matched are the same for the event registration examples in “HA_EM_CMD_REG and HA_EM_CMD_REG2 Commands (Register for Events)” on page 70.

In the fourth example, all variables whose instance vectors are defined to contain only the element **NodeNum** are matched. The instances matched are associated with node 9. In the fifth example, the same set of variables are matched, but all instances of each variable are matched.

In the sixth example, all variables whose instance vectors are defined to contain the elements **NodeNum** and **VG**, as well as zero or more additional elements, are matched. The instances matched are associated with node 9. In the last example, all variables whose instance vectors are defined to contain the element **NodeNum**, as well as zero or more additional elements, are matched. All instances of the variables are matched.

Given the flexibility in specifying resource variables for a query, it is possible that no resource variable instance or resource variable definition will match the specification in the **ha_em_rb_query** structure. If there is no match, an appropriate error response is returned.

If the caller specifies a function pointer in the **em_qcb** field, the specified function is used as a callback routine by the EMAPI. When the response or responses for this query are received by the EMAPI, the callback routine is called.

The first argument to the callback routine is the file descriptor of the session in which the query was made. The second argument is a pointer to a structure that contains the query information. The third argument is the value of the **em_qcb_arg** field. The **em_qcb_arg** field specifies an argument, meaningful in the context of the query response only to the EM client, that can be passed to the callback routine. The argument must be valid when the callback routine is invoked. For example, a pointer to a stack must be valid when the callback routine is invoked, or an argument must be valid in the context of the thread in which the callback routine executes. For more information about callback routines, see the man page for the **ha_em_receive_response** subroutine.

If the **ha_em_send_command** subroutine returns successfully, the **em_qid** field in the command block contains a value unique to each query; the EM client can now expect one or more responses that contain the requested information. Each response contains the value returned in **em_qid**. Because responses are returned asynchronously, this value is used to match responses to the query command.

Restrictions

When you use the EMAPI in a threaded program, IBM recommends that an Event Management session be used by only one thread at a time. If multiple threads try to use the same Event Management session at the same time, EMAPI subroutines may return the **HA_EM_EBUSY** error code. The thread-safe version of the EMAPI library was designed assuming that each thread using the EMAPI would have its own Event Management session or sessions.

Return Values

If the **ha_em_send_command** subroutine is successful, it returns a value of 0. Other information that may be returned depends on the command that is sent. For details, see:

"HA_EM_CMD_REG and HA_EM_CMD_REG2 Commands (Register for Events)" on page 70
"HA_EM_CMD_UNREG Command (Unregister Events)" on page 72
"HA_EM_CMD_QUERY Command (Query for Information)" on page 73.

Error Values

If the **ha_em_send_command** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

If the **HA_EM_ECONNLOST** error is received, it is recommended that the EM client no longer use the session file descriptor that was specified on the failing call on subsequent **select** system calls. To reconnect to the Event Management subsystem, use the **ha_em_restart_session** subroutine.

The EMAPI error block and error numbers are defined in the **ha_emapi.h** header file. For more information on EMAPI errors, see “EMAPI Errors (err_emapi)” on page 116.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using EMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/emapi**.

Files

ha_emapi.h.

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_emapi.h** header file: **ha_em_cmd_blk**, **ha_em_err_blk**, **ha_em_rb_reg**, **ha_em_rb_query**

Subroutines: **ha_em_start_session**, **ha_em_restart_session**, **ha_em_receive_response**

ha_em_start_session Subroutine

Purpose

ha_em_start_session – Start an EM client session with the Event Management subsystem

Library

EMAPI Thread-Safe Library (**libha_em_r.a**)

EMAPI Library (not thread-safe) (**libha_em.a**)

Syntax

```
#include <ha_emapi.h>

int
    ha_em_start_session(
        char            *part_name,
        ha_em_err_blk   *em_errb)
```

Parameters

part_name A pointer to a string that indicates the system partition in which a session is to be established.

If the *part_name* argument points to a null string, the session is established with the Event Management subsystem in the current system partition. If the EM client is executing on the control workstation or on a workstation outside of the SP, the current system partition is determined from the value of the **SP_NAME** environment variable.

If the *part_name* argument points to a string that is not null, the string is the name of the system partition in which a session is to be established with the Event Management subsystem.

em_errb A pointer to an error block structure.

Description

The **ha_em_start_session** subroutine is used by an EM client to establish a session with the Event Management subsystem. The session validates that the EM client is permitted to use Event Management services and then provides a communication path to the specified Event Management subsystem. If this communication path is lost, it can be restored without starting another session by using the **ha_em_restart_session** subroutine. When the EM client no longer needs Event Management services, it ends the session by using the **ha_em_end_session** subroutine.

By default, a session is established with the Event Management subsystem in the current system partition. To receive events from, and to issue queries to, other system partitions, the **ha_em_start_session** subroutine must be called once for each other system partition.

Multiple sessions may also be started with any single system partition. This provides multiple communication paths to the Event Management subsystem in a single system partition. Multiple paths are useful if the EM client wishes to ignore certain responses for a time while accepting others. For example, an EM client can register events in one session and issue queries in another.

Restrictions

The EMAPI uses connection-oriented socket communication. An EM client may want to change the handling of the **SIGPIPE** signal so that it is ignored. If the EM client does not ignore the **SIGPIPE** signal, and the connection between the EM client and the Event Management subsystem is lost, delivery of a **SIGPIPE** signal may kill the EM client.

When you use the EMAPI in a threaded program, IBM recommends that an Event Management session be used by only one thread at a time. If multiple threads try to use the same Event Management session at the same time, EMAPI subroutines may return the **HA_EM_EBUSY** error code. The thread-safe version of the EMAPI library was designed assuming that each thread using the EMAPI would have its own Event Management session or sessions.

Return Values

If the return value of the **ha_em_start_session** subroutine is greater than or equal to 0, the return value is a file descriptor. The EM client uses the file descriptor to determine when the Event Management subsystem has sent a response. The file descriptor is used as an argument in the **select** or **poll** system call. This descriptor is also used as a session handle in all other EMAPI subroutine calls.

Error Values

If the **ha_em_start_session** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The EMAPI error block and error numbers are defined in the **ha_emapi.h** header file. For more information on EMAPI errors, see “EMAPI Errors (err_emapi)” on page 116.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using EMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/emapi**.

Files

ha_emapi.h.

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_emapi.h** header file: **ha_em_err_blk**

Subroutines: **ha_em_restart_session**, **ha_em_end_session**

ha_rr_add_var Subroutine

Purpose

ha_rr_add_var – Add resource variables to the list of those known to the Resource Monitor Application Programming Interface (RMAPI) and supply current values for resource variable instances

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_add_var(
        int                     session_fd,
        struct ha_rr_variable   *pv,
        int                     numv,
        int                     add_complete,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

session_fd The file descriptor of the session from which the **HA_RR_CMD_ADDV** or **HA_RR_CMD_ADDALL** command was received or, if the resource monitor is command-based, the file descriptor that was returned by the **ha_rr_start_session** subroutine.

pv A pointer to an array of **ha_rr_variable** structures, which contain the names, instance vectors, and values of the resource variables to be added.

numv The number of elements in the array pointed to by the *pv* parameter.

add_complete A flag that indicates when the resource monitor has completed adding all of its resource variables. A value of 0 indicates that there are more variables to be added. A nonzero value indicates that, with this call, the adding of variables is complete.

More than one call of the **ha_rr_add_var** subroutine may be used to add variables. When multiple calls are used, the *add_complete* parameter is set to 0 on each call but the last.

rr_errb A pointer to an error block structure.

Description

The **ha_rr_add_var** subroutine is used by the resource monitor to add resource variables to the list of those known to the RMAPI and supply current values for resource variable instances. It is called whenever the resource monitor receives the **HA_RR_CMD_ADDV** or **HA_RR_CMD_ADDALL** commands. The resource

monitor must add and supply values for the variables that were specified by the command, even if some or all of the variables were previously added.

A command-based resource monitor calls the **ha_rr_add_var** subroutine immediately after it registers its variable instances to add those instances to the list of those known to the RMAPI and supply current values for resource variable instances.

The **ha_rr_variable** structure has the following definition:

```
struct ha_rr_variable {
         char            *rr_var_name;
         char            *rr_var_ivector;
         union {
                 int             rr_var_inst_id;
                 void            **rr_var_hndl;
         } rr_varu;
#define rr_var_handle      rr_varu.rr_var_hndl
#define rr_var_iid         rr_varu.rr_var_inst_id
         void            *rr_value;
         int             rr_var_errno;
}
```

The **rr_var_name** field is a pointer to a string that contains the name of a resource variable.

The **rr_var_ivector** field is a pointer to a string that contains the instance vector of an instance of the specified variable. The instance vector must be fully qualified; no wildcarding is permitted.

The instance vector is specified as a comma-separated list of name/value pairs. A name/value pair consists of an instance vector element name followed by an equal sign followed by the value of the vector element. There are no blanks in the instance vector.

The **rr_var_hndl** field is a pointer to a buffer that will be used to store the handle of the specified resource variable. On input, it must point to a valid location.

The **rr_value** field is a pointer to a buffer that contains the current value of the variable.

The **rr_var_errno** field is used to hold the result of the operation for each variable.

Security

The effective user ID of the calling process must be **root**.

Restrictions

A resource monitor of connection type server must call the **ha_rr_add_var** subroutine only in response to receiving the **HA_RR_CMD_ADDV** or **HA_RR_CMD_ADDALL** command. Calling this subroutine at other times is a programming error for a server type of resource monitor, with undefined results.

If the name of one of the elements in any instance vector matches the vector element name specified by the Locator field in the associated resource variable definition, that vector element name/value pair is never supplied by the resource

monitor in the instance vector specified by the **rr_var_ivector** field. This vector element is added by the Event Management subsystem when the instance is given in a response to any Event Management client.

Return Values

If the **ha_rr_add_var** subroutine is successful, it returns the number of resource variables that were added.

For each variable in the array that was successfully added, a variable handle is placed by the RMAPI in the location that is pointed to by the associated **rr_var_hndl** field in the **ha_rr_variable** structure. This variable handle is used when sending the resource variable to the RMAPI. If a variable has already been added, its variable handle is returned unchanged. The **ha_rr_add_var** subroutine can be called multiple times for the same resource variable.

For each variable in the array that was successfully added, the associated **rr_var_errno** field is also set to 0.

If a variable could not be added, an error code is placed in its **rr_var_errno** field.

For all variables that are successfully added, the resource monitor should immediately start sending values at the intervals determined by the resource variable value type and the design of the resource monitor.

Error Values

If the **ha_rr_add_var** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_rr_variable**, **ha_em_err_blk**

Subroutines: **ha_rr_del_var**, **ha_rr_get_ctrlmsg**

ha_rr_del_var Subroutine

Purpose

ha_rr_del_var – Delete resource variables from the list of those known to the Resource Monitor Application Programming Interface (RMAPI)

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_del_var(
        int                     session_fd,
        struct ha_rr_variable   *pv,
        int                     numv,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

session_fd	The file descriptor of the session from which the **HA_RR_CMD_DELV** or **HA_RR_CMD_DELALL** command was received or, if the resource monitor is command-based, the file descriptor that was returned by the **ha_rr_start_session** subroutine.
pv	A pointer to an array of **ha_rr_variable** structures, which contain the handles of the resource variables to be deleted.
numv	The number of elements in the array pointed to by the *pv* parameter.
rr_errb	A pointer to an error block structure.

Description

The **ha_rr_del_var** subroutine is used by the resource monitor to delete resource variables from the list of those known to the RMAPI. It is called whenever the resource monitor receives the **HA_RR_CMD_DELV** or **HA_RR_CMD_DELALL** commands. The resource monitor must delete the variables that were specified by the command, even if some or all of the variables were previously specified on a call to this subroutine.

A command-based resource monitor calls the **ha_rr_del_var** subroutine when it has stopped sending variable values to the RMAPI.

The **ha_rr_variable** structure has the following definition:

```
struct ha_rr_variable {
         char            *rr_var_name;
         char            *rr_var_ivector;
         union {
                 int             rr_var_inst_id;
                 void            **rr_var_hndl;
         } rr_varu;
#define rr_var_handle      rr_varu.rr_var_hndl
#define rr_var_iid         rr_varu.rr_var_inst_id
         void            *rr_value;
         int             rr_var_errno;
}
```

The **rr_var_name** and **rr_var_ivector** fields are not used.

The **rr_var_hndl** field points to the variable handle of each variable that is to be deleted. The variable handle was previously assigned by the RMAPI when the variable was added by the **ha_rr_add_var** subroutine.

The **rr_value** field is not used.

The **rr_var_errno** field is used to hold the result of the operation for each variable.

Security

The effective user ID of the calling process must be **root**.

Restrictions

A resource monitor of connection type server must call the **ha_rr_del_var** subroutine only in response to receiving the **HA_RR_CMD_DELV** or **HA_RR_CMD_DELALL** command, or before calling the **ha_rr_end_session** subroutine. Calling this subroutine at other times is a programming error for a server type of resource monitor, with undefined results.

Return Values

If the **ha_rr_del_var** subroutine is successful, it returns the number of variables that must no longer be sent to the RMAPI. If a variable is no longer to be sent, its variable handle is cleared from the location specified by the **rr_var_hndl** field. If a variable should continue to be sent, its variable handle is unchanged.

For each variable in the array that was successfully processed, the associated **rr_var_errno** field is set to 0. If a variable could not be processed, an error code is placed in its **rr_var_errno** field.

A return value of 0 indicates that none of the variables have been deleted; they should all continue to be sent.

Error Values

If the **ha_rr_del_var** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_rr_variable**, **ha_em_err_blk**

Subroutines: **ha_rr_add_var**, **ha_rr_get_ctrlmsg**, **ha_rr_end_session**

ha_rr_end_session Subroutine

Purpose

ha_rr_end_session – End a resource monitor session

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_end_session(
        int                     session_fd,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

session_fd	The file descriptor of the session to be ended.
rr_errb	A pointer to an error block structure.

Description

The **ha_rr_end_session** subroutine ends the session between the resource monitor and the resource monitor manager. A resource monitor that is not a command calls this subroutine when the resource monitor manager associated with the session has disconnected. A command-based resource monitor calls this subroutine before it terminates.

Restrictions

Before it calls the **ha_rr_end_session** subroutine, the resource monitor must first call the **ha_rr_del_var** subroutine to delete all of the variables that were added to the RMAPI during the session.

If it was started by a resource monitor manager and if it no longer has an active session after it calls the **ha_rr_end_session** subroutine, the resource monitor should terminate.

Return Values

If the **ha_rr_end_session** subroutine is successful, it returns a value of 0. The file descriptor for the session just ended must no longer be used as an argument to the **select** or **poll** system call.

Error Values

If the **ha_rr_end_session** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_em_err_blk**

Subroutines: **ha_rr_start_session**

ha_rr_get_ctrlmsg Subroutine

Purpose

ha_rr_get_ctrlmsg – Get a control message

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_get_ctrlmsg(
        int                     session_fd,
        struct ha_rr_ctrl_msg   **rr_ctrl_msg,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

session_fd	A session file descriptor that is ready to be read.
rr_ctrl_msg	A pointer to the address of a control message buffer.
rr_errb	A pointer to an error block structure.

Description

When a resource monitor is notified that a file descriptor returned by the **ha_rr_start_session** subroutine is ready for reading, as indicated by the **select** or **poll** system call, or **SIGIO** is caught, the resource monitor calls the **ha_rr_get_ctrlmsg** subroutine to get a control message.

If **HA_RR_NOTIFY_SELECT** notification is being used, *session_fd* contains a session file descriptor that is ready to be read.

If **HA_RR_NOTIFY_SIGIO** notification is being used, *session_fd* contains a session file descriptor that may be ready for reading. The **ha_rr_get_ctrlmsg** subroutine must be called once for each file descriptor that is returned by the **ha_rr_start_session** subroutine.

Restrictions

It is the responsibility of the calling routine to free the control message buffer. When the buffer is freed, any memory areas to which there are pointers in the control message can no longer be accessed.

If the **SIGIO** notification method is being used, the returned buffer may contain more than one message. It is possible that more than one message had been received before the **SIGIO** signal could be processed.

Before calling the **ha_rr_get_ctrlmsg** subroutine, the resource monitor must block the **SIGIO** signal and must establish a signal handler for it.

Return Values

If the **ha_rr_get_ctrlmsg** subroutine returns a value of 0, the response is intended for the Resource Monitor Application Programming Interface (RMAPI) itself. No action by the calling routine is necessary.

If the **ha_rr_get_ctrlmsg** subroutine returns a value that is greater than 0, it also returns a pointer to a buffer that contains one or more control messages. The *rr_ctrl_msg* argument points to an area where the buffer pointer is returned. The return value indicates the length of the data in the control message buffer.

The control message has the following structure:

```
struct ha_rr_ctrl_msg {
         int             rr_ctrl_msg_len;
         int             rr_ctrl_cmd;
         int             rr_ctrl_cmdarg;
         int             rr_ctrl_num_vars;
         union ha_rr_ctrlv {
                 struct ha_rr_ctrl_var {
                         char            *rr_ctrl_name;
                         char            *rr_ctrl_ivector;
                 } rr_ctrl_varn[1];
                 int     rr_ctrl_vari[1];
         } rr_ctrlv;
}
```

The **rr_ctrl_msg_len** field contains the length of the message. If this value is less than the value that was returned by the **ha_rr_get_ctrlmsg** subroutine, one or more additional messages are present in the buffer. To get a pointer to the next message in the buffer, increment the buffer pointer by the length of the message.

An actual control message may contain any number of elements in the **rr_ctrl_varn** or **rr_ctrl_vari** arrays; the **rr_ctrl_num_vars** field contains the actual number.

The **rr_ctrl_cmd** field contains a command to the resource monitor. The **rr_ctrl_cmdarg** field contains an optional command argument.

The **rr_ctrl_cmd** field can contain one of the following values:

HA_RR_CMD_INSTV

This command tells the resource monitor to create instances of resource variables that match the variables listed in the **rr_ctrl_varn** array. Each element of the array contains two pointers: a pointer to a string that contains the name of a resource variable and a pointer to a string that contains an instance vector. If any vector element is omitted, the instance vector is wildcarded. If all vector elements are omitted, the instance vector string is null.

The resource monitor creates actual instances for each specified instance of a resource variable. If the instance vector is wildcarded, the resource monitor creates all actual instances that match the wildcarded vector for a specified resource variable.

This command is sent only for variables that are configured as being dynamically instantiable.

In response, the resource monitor should create the instances and then register them with the RMAPI using the **ha_rr_reg_var** subroutine.

Note: If one of the instance vector element names of a resource variable matches the vector element name specified by the Locator field of the variable's definition, then that vector element is never supplied in an instance vector that is specified by the **HA_RR_CMD_INSTV** command. Note that this may result in a null vector.

HA_RR_CMD_ADDV

This command tells the resource monitor to add to the RMAPI the variables listed in the **rr_ctrl_vari** array. Each element of the array is an identifier for a particular instance of a resource variable. The identifier is supplied to the RMAPI when the resource monitor registers its variable instances.

In response, the resource monitor should call the **ha_rr_add_var** subroutine to add the variables and supply current values for resource variable instances. Then, it should call the **ha_rr_send_val** subroutine to start sending updated resource variable values for those instances that are indicated by the **ha_rr_add_var** subroutine.

HA_RR_CMD_ADDALL

This command tells the resource monitor to add to the RMAPI all of the variables of value type Counter or Quantity of which it is aware. The **rr_ctrl_num_vars** field is set to a value of 0.

In response, the resource monitor should call the **ha_rr_add_var** subroutine to add the variables and supply current values for resource variable instances. Then, it should call the **ha_rr_send_val** subroutine to start sending updated resource variable values for those instances that are indicated by the **ha_rr_add_var** subroutine.

HA_RR_CMD_DELV

This command tells the resource monitor to delete from the RMAPI the variables that are listed in the **rr_ctrl_vari** array. Each element of the array is an identifier for a particular instance of a resource variable. The identifier is supplied to the RMAPI when the resource monitor registers its variable instances.

In response, the resource monitor should call the **ha_rr_del_var** subroutine to delete the variables. Then, it should stop sending resource variable values for those instances that are indicated by the **ha_rr_del_var** subroutine.

HA_RR_CMD_DELALL

This command tells the resource monitor to delete from the RMAPI, using the **ha_rr_del_var** subroutine, all of the variables of value type Counter or Quantity of which it is aware. The **rr_ctrl_num_vars** field is set to a value of 0.

In response, the resource monitor should call the **ha_rr_del_var** subroutine to delete the variables. Then, it should stop sending resource variable values for those instances that are indicated by the **ha_rr_del_var** subroutine.

HA_RR_CMD_REFRESH
This command tells the resource monitor to send immediately the latest possible values for the variables that are listed in the **rr_ctrl_vari** array. Each element of the array is an identifier for a particular instance of a resource variable. The identifier is supplied to the RMAPI when the resource monitor registers its variable instances.

This command is sent to the resource monitor only for variables of value type Counter or Quantity that have configurable reporting intervals.

In response, the resource monitor should call the **ha_rr_send_val** subroutine to send the latest values for the variables.

Error Values

If the **ha_rr_get_ctrlmsg** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see "RMAPI Errors (err_rmapi)" on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

The following error numbers are either normal or require specific action by the resource monitor:

HA_RR_EAGAIN
The resource monitor specified **HA_RR_NOTIFY_SIGIO** on the call to the **ha_rr_start_session** subroutine and no data was read by the **ha_rr_get_ctrlmsg** subroutine. Because the **ha_rr_get_ctrlmsg** subroutine must be called each time **SIGIO** is caught, this error is normal.

HA_RR_EDISCONNECT
The resource monitor manager that is associated with the session file descriptor specified on input to the **ha_rr_get_ctrlmsg** subroutine has dropped its connection. In response, the resource monitor should call the **ha_rr_del_var** subroutine using this file descriptor and specifying all variables that have been added by the resource monitor. Finally, the resource monitor should call the **ha_rr_end_session** subroutine for the specified session.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_rr_ctrl_msg**, **ha_em_err_blk**

Subroutines: **ha_rr_start_session**, **ha_rr_end_session**

ha_rr_get_interval Subroutine

Purpose

ha_rr_get_interval – Get the reporting interval for a class of resource variables

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_get_interval(
        char                    *cname,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

cname	A pointer to a string that contains the name of the class of resource variables for which the reporting interval is to be returned.
rr_errb	A pointer to an error block structure.

Description

The **ha_rr_get_interval** subroutine is used by the resource monitor to get the reporting interval for a class of resource variables whose reporting interval is configurable. The reporting interval specifies how often a resource variable of value type Counter or Quantity of a particular class is to be sent to the RMAPI.

Security

The effective user ID of the calling process must be **root**.

Return Values

If the **ha_rr_get_interval** subroutine is successful, it returns the reporting interval, in seconds.

Error Values

If the **ha_rr_get_interval** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_em_err_blk**

ha_rr_init Subroutine

Purpose

ha_rr_init – Initiate communications with the Resource Monitor Application Programming Interface (RMAPI)

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_init(
        char                    *name,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

name	A pointer to a string that contains the resource monitor name, as defined in the Event Management Configuration Database (EMCDB).
rr_errb	A pointer to an error block structure.

Description

The **ha_rr_init** subroutine initializes a resource monitor, informing the Resource Monitor Application Programming Interface (RMAPI) of its identity. The subroutine also verifies that this is the only copy of the resource monitor that is executing.

Return Values

If the **ha_rr_init** subroutine is successful, it returns a value of 0.

Error Values

If the **ha_rr_init** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

The following error number requires specific action by the resource monitor:

HA_RR_EEXIST
: Another copy of the resource monitor is running. Exit immediately.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_em_err_blk**

ha_rr_makserv Subroutine

Purpose

ha_rr_makserv – Create a server session that enables resource monitor managers to connect to the resource monitor

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_makserv(
        int                     rr_notify_proto,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

rr_notify_proto A value that indicates the method of communication the resource monitor expects to use with its manager. It may take one of the following values:

HA_RR_NOTIFY_SELECT
The resource monitor expects to use the **select** or **poll** system call to communicate with its manager.

HA_RR_NOTIFY_SIGIO
The resource monitor expects to use **SIGIO** notification to communicate with its manager.

rr_errb A pointer to an error block structure.

Description

The **ha_rr_makserv** subroutine creates a server session for a resource monitor so that resource monitor managers can connect to it. Resource monitor managers are the Event Management subsystem or the Performance Monitor subsystem.

If the resource monitor is not a command, the **ha_rr_makserv** subroutine should be called after the call to **ha_rr_init**.

If the resource monitor is a command, do not call the **ha_rr_makserv** subroutine.

Return Values

If the **ha_rr_makserv** subroutine is successful, it returns a value greater than or equal to 0. The value is a file descriptor that is used by the resource monitor to determine when a resource monitor manager is attempting to start a session with the resource monitor.

If the *rr_notify_proto* argument is the value **HA_RR_NOTIFY_SELECT**, the resource monitor is expected to use this descriptor in the argument to the **select** or **poll** system call (as a read event).

If the value of the **rr_notify_proto** argument is the value **HA_RR_NOTIFY_SIGIO**, the resource monitor is expected to catch the **SIGIO** signal. The **ha_rr_makserv** subroutine enables the descriptor for **SIGIO** notification. Notification that the descriptor is ready for read, or potentially ready for read if **SIGIO** is caught, indicates that a resource monitor manager is, or may be, attempting to connect to the resource monitor.

Note that a resource monitor must always select (or poll) on the file descriptor that is returned by the **ha_rr_makserv** subroutine or must always catch the **SIGIO** signal. The Event Management subsystem and the Performance Monitor subsystem may come and go while the resource monitor is otherwise executing normally.

Error Values

If the **ha_rr_makserv** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_em_err_blk**

ha_rr_reg_var Subroutine

Purpose

ha_rr_reg_var – Register a resource variable instance with the Resource Monitor Application Programming Interface (RMAPI)

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_reg_var(
        struct ha_rr_variable   *pv,
        int                     numv,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

pv	A pointer to an array of **ha_rr_variable** structures, which contain the names, instance vectors, and instance identifiers of the resource variable instances to be registered.
numv	The number of elements in the array pointed to by the *pv* parameter.
rr_errb	A pointer to an error block structure.

Description

The **ha_rr_reg_var** subroutine is used by the resource monitor to register resource variable instances with the RMAPI.

The resource monitor must register each instance of a resource variable known to the resource monitor at the following times:

- After it calls the **ha_rr_init** subroutine
- After it instantiates any resource variable upon receipt of the **HA_RR_CMD_INSTV** command
- After it creates any resource variable's instances during normal operation.

The **ha_rr_variable** structure has the following definition:

```
struct ha_rr_variable {
         char            *rr_var_name;
         char            *rr_var_ivector;
         union {
                 int             rr_var_inst_id;
                 void            **rr_var_hndl;
         } rr_varu;
#define rr_var_handle      rr_varu.rr_var_hndl
#define rr_var_iid         rr_varu.rr_var_inst_id
         void            *rr_value;
         int             rr_var_errno;
}
```

The **rr_var_name** field is a pointer to a string that contains the name of a resource variable.

The **rr_var_ivector** field is a pointer to a string that contains the instance vector of an instance of the specified variable. The instance vector must be fully qualified; no wildcarding is permitted.

The instance vector is specified as a comma-separated list of name/value pairs. A name/value pair consists of an instance vector element name followed by an equal sign followed by the value of the vector element. There are no blanks in the instance vector.

The **rr_var_inst_id** field is an instance identifier that the resource monitor assigns to the instance. The instance identifier is used to specify variable instances efficiently in commands that are sent to the resource monitor. Typically, the instance identifier is an index into a table that is maintained by the resource monitor.

The **rr_value** field is not used.

The **rr_var_errno** field is used to hold the result of the registration for each variable.

Security

The effective user ID of the calling process must be **root**.

Restrictions

If the name of one of the elements in any instance vector matches the vector element name specified by the Locator field in the associated resource variable definition, that vector element name/value pair is never supplied by the resource monitor in the instance vector specified by the **rr_var_ivector** field. This vector element is added by the Event Management subsystem when the instance is given in a response to any Event Management client.

Return Values

If the **ha_rr_reg_var** subroutine is successful, it returns the number of resource variables that were registered. For each variable that was successfully registered, the associated **rr_var_errno** field in the **ha_rr_variable** structure is set to 0. If a variable could not be registered, an error code is returned in its **rr_var_errno** field.

Error Values

If the **ha_rr_reg_var** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see "RMAPI Errors (err_rmapi)" on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, "Understanding Event Management" on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_rr_variable**, **ha_em_err_blk**

Subroutines: **ha_rr_unreg_var**

ha_rr_send_val Subroutine

Purpose

ha_rr_send_val – Send variable values to the Resource Monitor Application Programming Interface (RMAPI)

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_send_val(
        struct ha_rr_val        *pv,
        int                     numv,
        int                     refresh,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

pv	A pointer to an array of **ha_rr_val** structures, which contain pointers to the values and handles of the resource variables that the resource monitor is sending to the RMAPI.
numv	The number of elements in the array pointed to by the *pv* parameter.
refresh	A value that indicates whether the resource variable values are being sent in response to a **HA_RR_CMD_REFRESH** command. A value of 0 indicates that the values are not being refreshed. A nonzero value indicates that they are being refreshed.
rr_errb	A pointer to an error block structure.

Description

The **ha_rr_send_val** subroutine is used by the resource monitor whenever it is time to send values of resource variables to the RMAPI.

The **ha_rr_val** structure has the following definition:

```
struct ha_rr_val {
         void            *rr_value;
         void            *rr_var_hndl;
}
```

The **rr_value** field is a pointer to the value of a resource variable. The **rr_var_hndl** field contains the variable handle of the resource variable that was returned by the **ha_rr_add_var** subroutine.

Security

The effective user ID of the calling process must be **root**.

Restrictions

The resource monitor may send as many variable values as it desires in a single call of the **ha_rr_send_val** subroutine. The actual number that it sends depends on the semantics of the variables, the required frequency of update, and the design of the resource monitor.

However, if any of the variables that were added are of value type Counter or Quantity, the resource monitor must perform some "send" activity every 500 seconds. If the resource monitor has no values to send within 500 seconds from the last call to **ha_rr_send_val**, it must call the **ha_rr_touch** subroutine.

Return Values

If the **ha_rr_send_val** subroutine is successful, it returns a value of 0.

Error Values

If the **ha_rr_send_val** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_rr_val**, **ha_em_err_blk**

Subroutines: **ha_rr_get_ctrlmsg**, **ha_rr_touch**

ha_rr_start_session Subroutine

Purpose

ha_rr_start_session – Start a session with a resource monitor manager

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_start_session(
        int                     rr_notify_proto,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

rr_notify_proto A value that indicates the method of communication the resource monitor expects to use with its manager. It may take one of the following values:

HA_RR_NOTIFY_SELECT
The resource monitor expects to use the **select** or **poll** system call to communicate with its manager.

HA_RR_NOTIFY_SIGIO
The resource monitor expects to use **SIGIO** notification to communicate with its manager.

rr_errb A pointer to an error block structure.

Description

The **ha_rr_start_session** subroutine is called by a resource monitor to establish a communication path, or session, between the resource monitor and one of its managers. This communication path is used to send control messages between the resource monitor manager and the resource monitor. For a command-based resource monitor, a session establishes a communication path to the Event Management subsystem only.

If the resource monitor is command-based, it does not call the **ha_rr_makserv** subroutine. Instead, it calls the **ha_rr_start_session** subroutine immediately after the call to the **ha_rr_init** subroutine completes successfully.

Otherwise, whenever the file descriptor returned by the **ha_rr_makserv** subroutine is ready for reading, as indicated by the **select** or **poll** system call, or whenever **SIGIO** is caught, the resource monitor calls the **ha_rr_start_session** subroutine.

Restrictions

If the **ha_rr_makserv** subroutine is called before the **ha_rr_start_session** subroutine is called, the value of the *rr_notify_proto* argument must be the same for both calls.

Before calling the **ha_rr_start_session** subroutine, the resource monitor must block the **SIGIO** signal and must establish a signal handler for it.

Return Values

If the **ha_rr_start_session** subroutine is successful, it returns a value greater than or equal to 0. The value is a file descriptor that is used by the resource monitor to determine when a resource monitor manager has sent a control message to the resource monitor.

If it is not command-based, the resource monitor must be prepared to accept multiple sessions, up to a value of **HA_RR_MAXSESSIONS**.

If the *rr_notify_proto* argument is the value **HA_RR_NOTIFY_SELECT**, the resource monitor is expected to use this descriptor in the argument to the **select** or **poll** system call (as a read event).

If the value of the **rr_notify_proto** argument is the value **HA_RR_NOTIFY_SIGIO**, the resource monitor is expected to catch the **SIGIO** signal. The **ha_rr_start_session** subroutine enables the descriptor for **SIGIO** notification. Notification that the descriptor is ready for read, or potentially ready for read if **SIGIO** is caught, indicates that a control message has, or may have, arrived.

The file descriptor is also used as a session handle by other RMAPI subroutines. Because a command-based resource monitor does not receive control messages, it uses the file descriptor only as a session handle.

Error Values

If the **ha_rr_start_session** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

The following error number is normal:

HA_RR_EAGAIN
: The resource monitor specified **HA_RR_NOTIFY_SIGIO** on the call to the **ha_rr_makserv** subroutine and no connection request was found by the **ha_rr_start_session** routine. Because the **ha_rr_start_session** subroutine must be called each time **SIGIO** is caught, this error is normal.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_em_err_blk**

ha_rr_terminate Subroutine

Purpose

ha_rr_terminate – Terminate communications with the Resource Monitor Application Programming Interface (RMAPI)

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_terminate(
        struct ha_em_err_blk    *rr_errb)
```

Parameters

rr_errb A pointer to an error block structure.

Description

The **ha_rr_terminate** subroutine is used by the resource monitor to terminate its activity with the RMAPI.

If the resource monitor is a daemon, it calls the **ha_rr_terminate** subroutine whenever it no longer has any sessions with resource monitor managers. Upon return from this subroutine, it exits.

If the resource monitor logic is contained within a subsystem, the **ha_rr_terminate** subroutine is called before the exit of the subsystem.

If the resource monitor is a command, it calls the **ha_rr_terminate** subroutine before it exits.

Security

The effective user ID of the calling process must be **root**.

Return Values

If the **ha_rr_terminate** subroutine is successful, it returns a value of 0. The file descriptor that was returned by the **ha_rr_makserv** subroutine must no longer be used in an argument to the **select** or **poll** system call.

Error Values

If the **ha_rr_terminate** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see “RMAPI Errors (err_rmapi)” on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, “Understanding Event Management” on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_em_err_blk**

Subroutines: **ha_rr_makserv**

ha_rr_touch Subroutine

Purpose

ha_rr_touch – Meet "send" frequency requirements when there are no resource variable values to send to the Resource Monitor Application Programming Interface (RMAPI)

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_touch(
        struct ha_em_err_blk    *rr_errb)
```

Parameters

rr_errb A pointer to an error block structure.

Description

The **ha_rr_touch** subroutine is used by the resource monitor to meet "send" frequency requirements when it has no resource variable values to send to the RMAPI.

If any of the variables that were added to the list of those known to the RMAPI by a resource monitor are of value type Counter or Quantity, then every 500 seconds the resource monitor must call either the **ha_rr_send_val** subroutine (to send values) or the **ha_rr_touch** subroutine.

Security

The effective user ID of the calling process must be **root**.

Restrictions

When any of the variables that were added to the RMAPI are of value type Counter or Quantity, the resource monitor must perform some "send" activity every 500 seconds. If the resource monitor has no values to send within 500 seconds from the last call to **ha_rr_send_val**, it must call the **ha_rr_touch** subroutine. As long as it has no values to send, it must call **ha_rr_touch** every 500 seconds.

Return Values

If the **ha_rr_touch** subroutine is successful, it returns a value of 0.

Error Values

If the **ha_rr_touch** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see "RMAPI Errors (err_rmapi)" on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, "Understanding Event Management" on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_em_err_blk**

Subroutines: **ha_rr_send_val**

ha_rr_unreg_var Subroutine

Purpose

ha_rr_unreg_var – Unregister a variable instance with the Resource Monitor Application Programming Interface (RMAPI)

Library

RMAPI Library (not thread-safe) (**libha_rr.a**)

Syntax

```
#include <ha_rmapi.h>

int
    ha_rr_unreg_var(
        struct ha_rr_variable   *pv,
        int                     numv,
        struct ha_em_err_blk    *rr_errb)
```

Parameters

pv	A pointer to an array of **ha_rr_variable** structures, which contain the names and instance vectors of the resource variable instances to be unregistered.
numv	The number of elements in the array pointed to by the *pv* parameter.
rr_errb	A pointer to an error block structure.

Description

The **ha_rr_unreg_var** subroutine is used by the resource monitor to unregister resource variable instances with the RMAPI. It is called when the resource variable no longer exists.

If a variable instance has been added to the RMAPI by a call to the **ha_rr_add_var** subroutine, the call to the **ha_rr_unreg_var** subroutine automatically deletes it. If the variable instance is successfully unregistered, the handle returned by the **ha_rr_add_var** subroutine is no longer valid and must not be used in any further calls to the RMAPI.

The **ha_rr_variable** structure has the following definition:

```
struct ha_rr_variable {
         char            *rr_var_name;
         char            *rr_var_ivector;
         union {
                 int             rr_var_inst_id;
                 void            **rr_var_hndl;
         } rr_varu;
#define rr_var_handle      rr_varu.rr_var_hndl
#define rr_var_iid         rr_varu.rr_var_inst_id
         void            *rr_value;
         int             rr_var_errno;
}
```

The **rr_var_name** field is a pointer to a string that contains the name of a resource variable.

The **rr_var_ivector** field is a pointer to a string that contains the instance vector of an instance of the specified variable. The instance vector must be fully qualified; no wildcarding is permitted.

The instance vector is specified as a comma-separated list of name/value pairs. A name/value pair consists of an instance vector element name followed by an equal sign followed by the value of the vector element. There are no blanks in the instance vector.

The **rr_varu** field is not used.

The **rr_value** field is not used.

The **rr_var_errno** field is used to hold the result of the operation for each variable.

Security

The effective user ID of the calling process must be **root**.

Restrictions

If the name of one of the elements in any instance vector matches the vector element name specified by the Locator field in the associated resource variable definition, that vector element name/value pair is never supplied by the resource monitor in the instance vector specified by the **rr_var_ivector** field.

The Event Management subsystem starts to cache resource variable instances once they have been registered through the RMAPI. Until the first send of an instance value occurs, the instance assumes the default value from the configuration file. Variable instances are removed from this cache only when they are unregistered; they are not removed if the resource monitor terminates without calling the **ha_rr_unreg_var** subroutine. Therefore, even if a resource monitor terminates, its variable instances are still available for query through the EMAPI. Typically, a command-based resource monitor does not call the **ha_rr_unreg_var** subroutine.

Return Values

If the **ha_rr_unreg_var** subroutine is successful, it returns the number of resource variables that were unregistered. For each variable that was successfully unregistered, the associated **rr_var_errno** field in the **ha_rr_variable** structure is set to 0. If a variable could not be unregistered, an error code is placed in its **rr_var_errno** field.

Error Values

If the **ha_rr_unreg_var** subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI error block and error numbers are defined in the **ha_rmapi.h** header file. For more information on RMAPI errors, see "RMAPI Errors (err_rmapi)" on page 148.

For information about error messages, see *PSSP Diagnosis and Messages Guide*.

Examples

For examples of using RMAPI subroutines, see the programs in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

Files

ha_rmapi.h

Suggested Reading and Prerequisite Information

Chapter 1, "Understanding Event Management" on page 1.

Related Information

Structures in the **ha_rmapi.h** header file: **ha_rr_variable**

Subroutines: **ha_rr_reg_var**.

ha_rr_unreg_var

Chapter 4. Event Management Files Reference

This chapter contains the reference material for the error numbers and header files in the Event Management Application Programming Interface (EMAPI) and the Resource Monitor Application Programming Interface (RMAPI). The files are listed alphabetically.

© Copyright IBM Corp. 1996

EMAPI Errors (err_emapi)

Purpose

EMAPI Errors – Error numbers and the error block for the Event Management Application Programming Interface (EMAPI)

Description

The Event Management Application Programming Interface (EMAPI) provides several classes of errors. These consist of errors that are returned:

- Synchronously by EMAPI subroutines
- Asynchronously as the result of EMAPI commands.

Synchronous Errors Returned by EMAPI Subroutines

If an EMAPI subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The EMAPI uses an error block that is common to both the EMAPI and the RMAPI. The block is defined in the **ha_emcommon.h** header file that is included by the **ha_emapi.h** header file.

The EMAPI error numbers are defined in the **ha_emapi.h** header file.

The Event Management error block has the following definition:

```
struct ha_em_err_blk {
    int     em_errline;
    char    em_errlevel[HA_EM_MAXERRLVL];
    char    em_errfile[HA_EM_MAXERRFN];
    int     em_errno;
    char    em_errmsg[HA_EM_MAXERRMSG];
}
```

The **em_errline**, **em_errlevel**, and **em_errfile** fields are reserved for IBM use in providing information for problem determination.

If an error occurs, the **em_errno** field contains one of the values listed below. The **em_errmsg** field contains a message text that describes the error in further detail. For information about EMAPI messages, see *PSSP Diagnosis and Messages Guide*.

The EMAPI error numbers are:

HA_EM_EGETPARTINFO
: The Event Management subsystem could not get information about the system partition that is associated with a session.

HA_EM_EGETNODNUM
: The Event Management subsystem could not determine the number of the node on which the application is running.

HA_EM_ESDROPEN
: The Event Management subsystem could not establish a System Data Repository (SDR) session.

HA_EM_ESDRGET
The Event Management subsystem could not obtain required information from the System Data Repository (SDR).

HA_EM_ENOMEM
The Event Management subsystem could not allocate required memory.

HA_EM_ESYSCALL
The Event Management subsystem received an unexpected error from a system call.

HA_EM_EEXIST
A connection to the Event Management subsystem was being created using a file descriptor that was already being used for another connection to the Event Management subsystem. This can occur if the EM client closes a file descriptor that is an Event Management session handle. An EM client should not close such a descriptor. When an Event Management session is no longer needed, it should be ended through a call to the **ha_em_end_session** subroutine.

HA_EM_ECONNREFUSED
The Event Management subsystem refused a requested connection.

HA_EM_ENOCONNECT
An error occurred while the Event Management subsystem attempted a connection.

HA_EM_ECONNLOST
The connection with the Event Management subsystem was lost. To reconnect to the Event Management subsystem, use the **ha_em_restart_session** subroutine.

HA_EM_EAUTHENT
The Event Management subsystem could not authenticate the user.

HA_EM_EAUTHOR
The user is not authorized to use the Event Management subsystem.

HA_EM_ESECSERV
An error occurred while requesting a security service.

HA_EM_ENOENT
The specified session does not exist.

HA_EM_EBUSY
The specified session is busy. This can occur when multiple threads are attempting to use the same session at the same time. It is recommended that an Event Management session be used by only one thread.

HA_EM_EINVALID
An invalid parameter was specified.

HA_EM_ENOTOWNER
The process does not own the specified session.

HA_EM_EUNEXPECTED
The Event Management subsystem encountered an unexpected condition.

HA_EM_EUNUSABLE
The specified session has become unusable. The session may have become unusable because of an error previously reported for the session. In a multi-threaded program, a session can also become unusable if a thread that is executing an EMAPI subroutine is canceled with the **pthread_cancel** system call.

HA_EM_ECONNECTED
A restart of the specified session is not permitted, because the session's connection with the Event Management subsystem has not been lost.

Asynchronous Errors Resulting from EMAPI Commands

When an asynchronous error occurs as the result of a command, the Event Management subsystem indicates that there is an error by setting the **em_error** field of one of the structures specified in the **em_resp_blk** union of the **ha_em_rsp_blk** structure to a nonzero value. It also passes two error codes in the **em_error_codes** array. The first element of the array contains a general error code that indicates the nature of the operation that resulted in the error. The second element indicates in more detail the actual error.

The general error codes are:

HA_EM_RSP_EGEN_COMMAND
An error was detected in the specification of a command.

HA_EM_RSP_EGEN_RESOURCE
An error was caused because an internal Event Management subsystem resource was temporarily unavailable.

HA_EM_RSP_EGEN_RESMON
An error resulted from starting or communicating with a resource monitor.

HA_EM_RSP_EGEN_EVALERR
An error was detected during the evaluation of a predicate.

Each general error code can have a specific error code associated with it.

Unless otherwise stated, if any of these specific error codes are returned in the **em_error** field of the **ha_em_rpb_rerr** or **ha_em_rpb_qerr** structures, then if the value of the **em_errinfo1** field in the **ha_em_rpb_rerr** structure or the value of the **em_errinfo** field in the **ha_em_rpb_qerr** structure is nonzero, the nonzero value represents the position of the error within the input string that is in error, as specified by the specific error code.

The specific error codes associated with each general error code are listed below.

The HA_EM_RSP_EGEN_COMMAND General Error Code

Specific error codes associated with the **HA_EM_RSP_EGEN_COMMAND** general error code are:

HA_EM_RSP_ENOVARENT
The specified variable name does not exist in the Event Management Configuration Database (EMCDB).

HA_EM_RSP_EPRED
The specified predicate could not be parsed correctly. In the **ha_em_rpb_rerr** structure, the predicate parsing error is contained in

the **em_errinfo0** field and the position of the error within the predicate is contained in the **em_errinfo1** field.

For a list of parsing errors, see “Predicate Parsing Error Codes” on page 120.

HA_EM_RSP_ERAPRED
The specified rearm predicate could not be parsed correctly. In the **ha_em_rpb_rerr** structure, the predicate parsing error is contained in the **em_errinfo0** field and the position of the error within the predicate is contained in the **em_errinfo1** field.

For a list of parsing errors, see “Predicate Parsing Error Codes” on page 120.

HA_EM_RSP_ENODFLTPRED
No predicate was specified and no default predicate is configured.

HA_EM_RSP_ENOVECNAME
The specified instance vector was missing a vector element name.

HA_EM_RSP_ENOVECVALUE
The specified instance vector was missing a vector element value.

HA_EM_RSP_EVECSYNTAX
The specified instance vector contained a syntax error.

HA_EM_RSP_EVECVALLENGTH
A vector element value length, including the terminating null character, was longer than the maximum allowed (**HA_EM_VECTOR_ELEM_VALUE_LEN_MAX**).

HA_EM_RSP_EMISSINGVEC
The specified instance vector was missing a required element.

HA_EM_RSP_EDUPVECENT
The specified instance vector contained a duplicate element.

HA_EM_RSP_ENOVECENT
The specified instance vector contained an element that is not defined for the specified variable.

HA_EM_RSP_EINVALEID
An invalid event ID was specified.

HA_EM_RSP_ECLASSMISMATCH
The specified class does not match the class defined for the specified variable.

HA_EM_RSP_EVECMISMATCH
The specified instance vector does not match the vector defined for the specified variable(s).

HA_EM_RSP_EWCVMISMATCH
The class, if specified, and/or the instance vector, if specified, do not match the class and/or vector defined for any of the variables that matched the specified wildcarded variable name.

HA_EM_RSP_ENOCLASS
The specified class name does not exist in the Event Management Configuration Database (EMCDB).

HA_EM_RSP_EVECTOOLONG
The specified instance vector string is too long.

HA_EM_RSP_EINVALNODENUM
A node number implied in an instance vector is not within the range supported by the Event Management subsystem.

HA_EM_RSP_EINVALVECVAL
A vector element value of type integer is invalid. It is either less than 0, greater than 4096 or, if a range is specified, the second value is less than the first.

Predicate Parsing Error Codes

Predicate parsing error codes associated with the **HA_EM_RSP_EPRED**, **HA_EM_RSP_ERAPRED**, and **HA_EM_RSP_EGEN_EVALERR** error codes are:

HA_EM_PRED_ERVAR
There is an internal resource variable error. Contact the IBM Support Center.

HA_EM_PRED_EPREDLEN
The parsing table overflowed.

HA_EM_PRED_ESYNTAX
There is an error in the syntax of the predicate.

HA_EM_PRED_ESTACK
An internal stack overflowed. Contact the IBM Support Center.

HA_EM_PRED_EILLEGALOP
The specified operation is illegal between the specified data types.

HA_EM_PRED_ESTRING
There is an error in the syntax of a string constant.

HA_EM_PRED_ESBSSN
The serial number of a structured byte string field is either missing or out of range.

HA_EM_PRED_EFIELDTYPE
The data type of a structured byte string field is either illegal or does not match its value.

HA_EM_PRED_EDATATYPE
There is an internal data type error. Contact the IBM Support Center.

HA_EM_PRED_ESBSLEN
There is a mismatch between the data type and the length of structured byte string field.

HA_EM_PRED_EDIVIDE
An attempt was made to divide by zero.

The HA_EM_RSP_EGEN_RESOURCE General Error Code

Specific error codes associated with the **HA_EM_RSP_EGEN_RESOURCE** general error code are:

HA_EM_RSP_EQNONE
The Event Management subsystem could not generate a response for this query.

HA_EM_RSP_EQCOMPLETE
The query is complete. All information was returned in previous responses.

HA_EM_RSP_EINSTNOTAVAIL
The variable instance was not available.

HA_EM_RSP_EINSTSTALE
The variable instance is stale, that is, the resource monitor that supplies the instance has terminated unexpectedly.

HA_EM_RSP_EDAEMONGONE
The Event Management daemon that supplies instances that match the instance vector has terminated.

The HA_EM_RSP_EGEN_RESMON General Error Code

Specific error codes associated with the **HA_EM_RSP_EGEN_RESMON** general error code are:

HA_EM_RSP_ERMLOCKSTATUS
An error was encountered while trying to obtain the lock status of the resource monitor.

HA_EM_RSP_ERMNOTRUNNING
The resource monitor is not running and it cannot be started by the Event Management subsystem.

HA_EM_RSP_ENOSTARTRM
The resource monitor could not be started by the Event Management subsystem.

HA_EM_RSP_ENORMCONNECT
The Event Management subsystem could not connect to the resource monitor.

The HA_EM_RSP_EGEN_EVALERR General Error Code

The specific error codes associated with the **HA_EM_RSP_EGEN_EVALERR** general error code are the error codes listed in “Predicate Parsing Error Codes” on page 120.

Related Information

EMAPI subroutines: **ha_em_end_session**, **ha_em_receive_response**, **ha_em_restart_session**, **ha_em_send_command**, **ha_em_start_session**

EMAPI header file: **ha_emapi.h**

Messages: *PSSP Diagnosis and Messages Guide*

ha_emapi.h File

Purpose

ha_emapi.h – Header file for the Event Management Application Programming Interface (EMAPI)

Description

The **ha_emapi.h** header file provides data types and structures for use with the Event Management Application Programming Interface (EMAPI) subroutines. The EMAPI provides two libraries, one that is thread-safe (**libha_em_r.a**) and one that is not (**libha_em.a**). Any program that uses the EMAPI subroutines must include this file, which resides in the /**usr**/**include** directory.

The following listing shows the contents of the **ha_emapi.h** file. Nested header files, **ha_emapi_base.h** and **ha_emcommon.h**, follow the **ha_emapi.h** file listing.

ha_emapi.h File

```
/*                                                          vi:set tabstop=4: */
/*============================================================================*/
/* Licensed Materials - Property of IBM                                       */
/*                                                                            */
/* 5765-529                                                                   */
/*                                                                            */
/* (C) Copyright IBM Corp. 1996 All Rights Reserved.                          */
/*                                                                            */
/* US Government Users Restricted Rights - Use, duplication or disclosure    */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                     */
/*============================================================================*/
/*                                                                            */
/* Module Name:  ha_emapi.h                                                   */
/*                                                                            */
/* Description:                                                               */
/*                                                                            */
/*     Interface definitions of the Event Manager API needed by               */
/* Event Manager API clients and the Event Manager API library.               */
/*                                                                            */
/*     This file is formatted to be viewed with tab stops set to 4.          */
/*============================================================================*/

#ifndef _HA_EMAPI_H
#define _HA_EMAPI_H

#include <ha_emapi_base.h>


/*----------------------------------------------------------------------------*/
/*  Event Manager API constants.                                              */
/*----------------------------------------------------------------------------*/

/*
 *  HA_EM_VERSION identifies the version of the Event Manager Application
 *  Programming Interface desired by a client.  If the client has not
 *  specified a particular version, version 1 is used.
 */
```

```
#ifndef HA_EM_VERSION
#define HA_EM_VERSION           1
#endif  /* HA_EM_VERSION */

/*
 *  The following macros convert references to the generic names of EMAPI
 *  routines, like ha_em_start_session(), to references to version specific
 *  EMAPI routines, like ha_em_start_session_1().  The version used during
 *  conversion is determined by the value of HA_EM_VERSION.  Note that these
 *  macros can deal with the different versions of a routine having different
 *  arguments.
 */

#define ha_em_start_session                                             \
    HA_EM_ROUTINE_VERSION(ha_em_start_session, HA_EM_VERSION)

#define ha_em_restart_session                                           \
    HA_EM_ROUTINE_VERSION(ha_em_restart_session, HA_EM_VERSION)

#define ha_em_end_session                                               \
    HA_EM_ROUTINE_VERSION(ha_em_end_session, HA_EM_VERSION)

#define ha_em_send_command                                              \
    HA_EM_ROUTINE_VERSION(ha_em_send_command, HA_EM_VERSION)

#define ha_em_receive_response                                          \
    HA_EM_ROUTINE_VERSION(ha_em_receive_response, HA_EM_VERSION)

#define ha_em_get_ecgid                                                 \
    HA_EM_ROUTINE_VERSION(ha_em_get_ecgid, HA_EM_VERSION)

#define ha_em_get_partinfo                                              \
    HA_EM_ROUTINE_VERSION(ha_em_get_partinfo, HA_EM_VERSION)

#define HA_EM_ROUTINE_VERSION(routine, version)                         \
    HA_EM_ROUTINE_VERSION_GLUE(routine, version)

#define HA_EM_ROUTINE_VERSION_GLUE(routine, version)                    \
    routine ## _ ## version

/*
 *  Event Manager error values.
 */

#define HA_EM_EGETPARTINFO  1       /* Could not get partition information   */
#define HA_EM_EGETNODNUM    2       /* Could not determine host node number  */
#define HA_EM_ESDROPEN      3       /* Could not establish a SDR session     */
#define HA_EM_ESDRGET       4       /* Could not obtain needed info from SDR */
#define HA_EM_ENOMEM        5       /* Could not allocate needed memory      */
#define HA_EM_ESYSCALL      6       /* Unexpected system call error          */
#define HA_EM_EEXIST        7       /* Unexpected re-use of file descriptor  */
#define HA_EM_ECONNREFUSED  8       /* Connection to E.M. daemon refused     */
#define HA_EM_ENOCONNECT    9       /* Connection error to E.M. daemon       */
#define HA_EM_ECONNLOST     10      /* Connection lost with E.M. daemon      */
#define HA_EM_EAUTHENT      11      /* Could not authenticate user           */
#define HA_EM_EAUTHOR       12      /* User not authorized to use E.M. daemon*/
#define HA_EM_ESECSERV      13      /* Security service error encountered    */
```

ha_emapi.h

```
#define HA_EM_ENOENT        14      /* Specified session does not exist      */
#define HA_EM_EBUSY         15      /* Specified session is busy             */
#define HA_EM_EINVALID      16      /* Invalid parameter specified           */
#define HA_EM_ENOTOWNER     17      /* The process does not own the session  */
#define HA_EM_EUNEXPECTED   18      /* Unexpected condition occurred         */
#define HA_EM_EUNUSABLE     19      /* Specified session has become unusable */
#define HA_EM_ECONNECTED    20      /* Session restart not permitted         */


/*--------------------------------------------------------------------------*/
/*  Event Manager API type and structure definitions.                       */
/*--------------------------------------------------------------------------*/


/*
 *  One event registration request is made through a ha_em_rb_reg
 *  (Event Manager Request Block: REGister) structure.  This structure is
 *  included in the ha_em_res_blk union, defined later in this file.
 */

struct ha_em_rb_reg {
    char        *em_name;           /* input:  resource variable name        */
    char        *em_ivector;        /* input:  instantiation vector          */
    char        *em_pred;           /* input:  predicate                     */
    char        *em_rapred;         /* input:  re-arm predicate              */
    ha_em_eid_t em_event_id;        /* output: event identifier              */
    void        (*em_cb)(int, struct ha_em_rpb_event *, void *);
                                    /* input:  callback routine address      */
    void        *em_cb_arg;         /* input:  callback routine argument     */
};

/*
 *  One query request is made through a ha_em_rb_query
 *  (Event Manager Request Block: QUERY) structure.  This structure is
 *  actually defined in ha_emapi_base.h.  It is included in the ha_em_res_blk
 *  union, defined later in this file.
 */

struct ha_em_rb_query;

/*
 *  Multiple requests of a particular type are made through a ha_em_res_blk
 *  (Event Manager REquestS BLocK) union.  All the requests must be of the
 *  same type.  The type of array used depends on the type of the requests
 *  being made.  The number of elements in the array depends on the number
 *  of requests being made.  This union is included in the ha_em_cmd_blk
 *  structure, defined later in this file.
 */

union ha_em_res_blk {
    struct ha_em_rb_reg     em_rb_reg[1];   /* Used to register for events   */
    ha_em_eid_t             em_rb_unreg[1]; /* Used to unregister events     */
    struct ha_em_rb_query   em_rb_query[1]; /* Used to query for information */
};

/*
 *  A block of requests of a particular type are made through a ha_em_cmd_blk
 *  (Event Manager CoMmand BLocK) structure passed to the ha_em_send_command()
```

```
 *  routine.  All the requests must be of the same type.  The type of the
 *  requests are identified in the em_cmd and em_subcmd fields.  The requests
 *  themselves are in an array within the ha_em_res_blk union.
 */

struct ha_em_cmd_blk {
    int                 em_cmd_num_elem;    /* input:  number of requests    */
    short               em_cmd;             /* input:  type of requests      */
    short               em_subcmd;          /* input:  subtype of requests   */
    ha_em_qid_t         em_qid;             /* output: query identifier      */
    void                (*em_qcb)(int, struct ha_em_rsp_blk *, void *);
                                            /* input:  query callback routine*/
    void                *em_qcb_arg;        /* input:  query callback arg.   */
    union ha_em_res_blk em_res_blk;         /* the array of requests         */
};

/*
 *  A block of responses of a particular type are delivered through a
 *  ha_em_rsp_blk (Event Manager ReSPonse BLocK) structure passed through
 *  the ha_em_receive_command() routine.  This structure is
 *  actually defined in ha_emapi_base.h.
 */

struct ha_em_rsp_blk;

/*
 *  The ha_em_partinfo (Event Manager PARTition INFOrmation) structure is
 *  returned by the ha_em_get_partinfo() routine.  It describes how to
 *  communicate with a particular system partition.
 */

struct ha_em_partinfo {
    int  em_local_access;   /* output: the partition can be accessed locally */
    char em_hostname[256];  /* output: the partition's host name             */
    char em_address[64];    /* output: the partition's foreign address       */
    char em_process[64];    /* output: the partition's foreign process       */
};


/*--------------------------------------------------------------------------*/
/*  Event Manager API function prototypes.                                  */
/*--------------------------------------------------------------------------*/

/*
 *  Function prototypes for functions intended for Event Manager API clients.
 *  Clients should use the generic routine names defined by macros included
 *  in this header file, such as ha_em_start_session(), instead of the version
 *  specific routine names, like ha_em_start_session_1().  Those macros
 *  convert the generic names to the appropriate version specific names.
 */

#ifndef _NO_PROTO

extern int ha_em_start_session_1(       /* Start an Event Manager session    */
    char *em_part_name,                 /* input:  SP partition of session   */
    struct ha_em_err_blk *em_errb       /* output: error description block   */
);                                      /* return: session ID or -1 (error)  */
```

ha_emapi.h

```
extern int ha_em_restart_session_1(     /* Restart an Event Manager session  */
    int em_session_fd,                  /* input:  old session ID            */
    struct ha_em_err_blk *em_errb       /* output: error description block   */
);                                      /* return: new ID or -1 (error)      */

extern int ha_em_end_session_1(         /* End an Event Manager session      */
    int em_session_fd,                  /* input:  Event Manager session ID  */
    struct ha_em_err_blk *em_errb       /* output: error description block   */
);                                      /* return: 0 or -1 (error)           */

extern int ha_em_send_command_1(        /* Send an Event Manager command     */
    int em_session_fd,                  /* input:  Event Manager session ID  */
    struct ha_em_cmd_blk *em_cmdb,      /* input/output: Command block       */
    struct ha_em_err_blk *em_errb       /* output: error description block   */
);                                      /* return: 0 or -1 (error)           */

extern int ha_em_receive_response_1(    /* Receive an Event Manager response */
    int em_session_fd,                  /* input:  Event Manager session ID  */
    struct ha_em_rsp_blk **em_rsp_blk,  /* output: pointer to response block */
    struct ha_em_err_blk *em_errb       /* output: error description block   */
);                                      /* return: number of responses       */
                                        /*         (may be 0), or -1 (error) */

extern ha_em_ecgid_t ha_em_get_ecgid_1( /* Get event command group identifier*/
    ha_em_eid_t em_eid                  /* input:  event identifier          */
);                                      /* return: event command group ID    */

#endif  /* ! _NO_PROTO */

#ifdef _NO_PROTO
extern int ha_em_start_session_1();     /* Start an Event Manager session    */
extern int ha_em_restart_session_1();   /* Restart an Event Manager session  */
extern int ha_em_end_session_1();       /* End an Event Manager session      */
extern int ha_em_send_command_1();      /* Send an Event Manager command     */
extern int ha_em_receive_response_1();  /* Receive an Event Manager response */
extern ha_em_ecgid_t ha_em_get_ecgid_1();  /* Get event command group ID     */
#endif  /* _NO_PROTO */

/*
 *
 *  The ha_em_get_partinfo() routine is externalized for implementation
 *  convenience.  It should not be necessary for a client of the
 *  Event Manager to call it.
 */

#ifndef _NO_PROTO

extern int ha_em_get_partinfo_1(        /* Obtain SP partition information   */
    char *em_part_name,                 /* input:  SP partition of session   */
    int em_get_all,                     /* input:  get all partition info.   */
    struct ha_em_partinfo *em_partinfo, /* output: SP partition information  */
    struct ha_em_err_blk *em_errb       /* output: error description block   */
);                                      /* return: 0 or -1 (error)           */

#endif  /* ! _NO_PROTO */
```

```
#ifdef _NO_PROTO
extern int ha_em_get_partinfo_1();      /* Obtain SP partition information   */
#endif  /* _NO_PROTO */

#endif  /* _HA_EMAPI_H */
```

ha_emapi_base.h File

```
/*                                                          vi:set tabstop=4: */
/*============================================================================*/
/* Licensed Materials - Property of IBM                                       */
/*                                                                            */
/* 5765-529                                                                   */
/*                                                                            */
/* (C) Copyright IBM Corp. 1996 All Rights Reserved.                          */
/*                                                                            */
/* US Government Users Restricted Rights - Use, duplication or disclosure    */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                     */
/*============================================================================*/
/*                                                                            */
/* Module Name:  ha_emapi_base.h                                              */
/*                                                                            */
/* Description:                                                               */
/*                                                                            */
/*     Interface definitions of the Event Manager API needed by               */
/* Event Manager API clients, the Event Manager API library, and the          */
/* Event Manager daemon.                                                      */
/*                                                                            */
/*     This file is formatted to be viewed with tab stops set to 4.           */
/*============================================================================*/

#ifndef _HA_EMAPI_BASE_H
#define _HA_EMAPI_BASE_H

#include <sys/time.h>

#include <ha_emcommon.h>


/*----------------------------------------------------------------------------*/
/*  Event Manager API constants.                                              */
/*----------------------------------------------------------------------------*/

/*
 *  Event Manager Commands.
 */

#define HA_EM_CMD_REG       1   /* Register for events                        */
#define HA_EM_CMD_REG2      2   /* Register for events from both predicates   */
#define HA_EM_CMD_UNREG     3   /* Unregister events                          */
#define HA_EM_CMD_QUERY     4   /* Query for information                      */

#define HA_EM_CMD_RERR      5   /* Returned for registration command errors   */
#define HA_EM_CMD_R2ERR     6   /* Returned for registration 2 command errors*/
#define HA_EM_CMD_QERR      7   /* Returned for query command errors          */

/*
 *  Event Manager Subcommands for the HA_EM_CMD_REG and HA_EM_CMD_REG2 commands.
 */

#define HA_EM_SCMD_REVAL    1   /* Evaluate predicate at first observation    */
                                /*    and return result as an event           */

/*
```

```
 *  Event Manager Subcommands for the HA_EM_CMD_QUERY command. These are also
 *  returned with HA_EM_CMD_QERR to indicate the query subcommand that
 *  resulted in errors.
 */

#define HA_EM_SCMD_QCUR     1   /* Query current resource variable values    */
#define HA_EM_SCMD_QDEF     2   /* Query for defined resource variables and  */
                                /*     their default predicates              */

/*
 *  Event Manager Error Codes, returned in em_error of the various response
 *  structures. General error codes indicate the nature of the operation that
 *  resulted in the error. Specific error codes detail the actual error.
 */

#define HA_EM_RSP_EGEN_COMMAND  1   /* This general error code indicates an  */
                                    /* error was detected in the             */
                                    /* specification of a command            */
/*-------------------------------------------------------------------------*/
/*  Specific error codes associated with HA_EM_RSP_EGEN_COMMAND            */
/*-------------------------------------------------------------------------*/
#define HA_EM_RSP_ENOVARENT     1   /* The specified variable name does not  */
                                    /* exist in the EM data base             */
#define HA_EM_RSP_EPRED         2   /* The specified predicate could not be  */
                                    /* correctly parsed. The predicate       */
                                    /* parsing error is contained in the     */
                                    /* em_errinfo0 field of the              */
                                    /* ha_em_rpb_rerr structure; the         */
                                    /* position of the error within the      */
                                    /* predicate is contained in the         */
                                    /* em_errinfo1 field of the              */
                                    /* ha_em_rpb_rerr structure.             */
#define HA_EM_RSP_ERAPRED       3   /* The specified re-arm predicate could  */
                                    /* not be correctly parsed. The returned */
                                    /* value is encoded as for the           */
                                    /* HA_EM_RSP_EPRED error.                */
#define HA_EM_RSP_ENODFLTPRED   4   /* No predicate was specified and no     */
                                    /* default predicate is configured       */
#define HA_EM_RSP_ENOVECNAME    5   /* the specified instantiation vector    */
                                    /* was missing a vector element name     */
#define HA_EM_RSP_ENOVECVALUE   6   /* the specified instantiation vector    */
                                    /* was missing a vector element value    */
#define HA_EM_RSP_EVECSYNTAX    7   /* the specified instantiation vector    */
                                    /* contained a syntax error.             */
#define HA_EM_RSP_EVECVALLENGTH 8   /* a vector element value length was     */
                                    /* longer, including the terminating     */
                                    /* NULL, than                            */
                                    /* HA_EM_VECTOR_ELEM_VALUE_LEN_MAX       */
#define HA_EM_RSP_EMISSINGVEC   9   /* the specified instantiation vector    */
                                    /* was missing a required element.       */
#define HA_EM_RSP_EDUPVECENT    10  /* the specified instantiation vector    */
                                    /* contained a duplicate element.        */
#define HA_EM_RSP_ENOVECENT     11  /* the specified instantiation vector    */
                                    /* contained an element that is not      */
                                    /* defined for the specified variable.   */
#define HA_EM_RSP_EINVALEID     12  /* An invalid event ID was specified.    */
#define HA_EM_RSP_ECLASSMISMATCH    13  /* The specified class does not      */
                                        /* match the class defined for the   */
```

ha_emapi.h

```
                                    /* specified variable.              */
#define HA_EM_RSP_EVECMISMATCH  14  /* The specified instantiation vector   */
                                    /* does not match the vector defined for */
                                    /* the specified variable(s).           */
#define HA_EM_RSP_EWCVMISMATCH  15  /* The class, if specified, and/or the  */
                                    /* instantiation vector, if specified,  */
                                    /* do not match the class and/or vector */
                                    /* defined for any of the variables that */
                                    /* matched the specified wildcarded     */
                                    /* variable name.                       */
#define HA_EM_RSP_ENOCLASS      16  /* The specified class name does not    */
                                    /* exist in the EM data base            */
#define HA_EM_RSP_EVECTOOLONG   17  /* The specified instantiation vector   */
                                    /* string is too long.                  */
#define HA_EM_RSP_EINVALNODENUM 18  /* A node number implied in an instan-  */
                                    /* tiation vector isn't within the range */
                                    /* supported by the Event Manager       */
#define HA_EM_RSP_EINVALVECVAL  19  /* A vector element value of type 'int' */
                                    /* is invalid, i.e. less than 0, greater */
                                    /* than 4096 or, if the second value in */
                                    /* a value range, it is less than the   */
                                    /* first value in the range             */



#define HA_EM_RSP_EGEN_RESOURCE 2   /* This general error code indicates an */
                                    /* error was due to the temporary       */
                                    /* unavailability of an internal Event  */
                                    /* Manager resource.                    */
/*------------------------------------------------------------------------*/
/*  Specific error codes associated with HA_EM_RSP_EGEN_RESOURCE          */
/*------------------------------------------------------------------------*/
#define HA_EM_RSP_EQNONE        1   /* No response could be generated for   */
                                    /* this individual query request.       */
#define HA_EM_RSP_EQINCOMPLETE  2   /* Not used; will be removed            */
#define HA_EM_RSP_EQCOMPLETE    2   /* The query is complete; all data was  */
                                    /* returned in previous message(es)     */
#define HA_EM_RSP_EINSTNOTAVAIL 3   /* Variable instance not available      */
#define HA_EM_RSP_EINSTSTALE    4   /* Variable instance is stale, i.e. the */
                                    /* resource monitor supplying the       */
                                    /* instance has terminated unexpectedly. */
#define HA_EM_RSP_EDAEMONGONE   5   /* the Event Manager daemon supplying   */
                                    /* instances matching the instantiation */
                                    /* vector has terminated.               */



#define HA_EM_RSP_EGEN_RESMON   3   /* This general error code indicates an */
                                    /* error resulted from starting or      */
                                    /* communicating with a resource monitor */
/*------------------------------------------------------------------------*/
/*  Specific error codes associated with HA_EM_RSP_EGEN_RESMON            */
/*------------------------------------------------------------------------*/
#define HA_EM_RSP_ERMLOCKSTATUS 1   /* Error in obtaining resource monitor  */
                                    /* lock status.                         */
#define HA_EM_RSP_ERMNOTRUNNING 2   /* a resource monitor is not running and */
                                    /* it cannot be started by Event Mgt    */
#define HA_EM_RSP_ENOSTARTRM    3   /* a resource monitor could not be      */
```

```
                                    /* started by Event Management          */
#define HA_EM_RSP_ENORMCONNECT  4   /* the Event Management daemon could not */
                                    /* connect to a resource monitor.       */



#define HA_EM_RSP_EGEN_EVALERR  4   /* This general error code indicates an  */
                                    /* error was detected in the evaluation  */
                                    /* of a predicate                        */
/*------------------------------------------------------------------------*/
/*  Specific error codes associated with HA_EM_RSP_EGEN_EVALERR are defined  */
/*  in ha_emcommon.h using names of the form HA_EM_PRED_*.                */
/*------------------------------------------------------------------------*/



/*------------------------------------------------------------------------*/
/*  Event Manager API type and structure definitions.                     */
/*------------------------------------------------------------------------*/

/*
 *  Event command group identifiers are of type ha_em_ecgid_t
 *  (Event Manager Event Command Group IDentifier).
 */

typedef unsigned long ha_em_ecgid_t;

/*
 *  Event identifiers are of type ha_em_eid_t (Event Manager Event IDentifier).
 */

typedef unsigned long ha_em_eid_t;

/*
 *  Query identifiers are of type ha_em_qid_t (Event Manager Query IDentifier).
 */

typedef unsigned long ha_em_qid_t;

/*
 *  One query request is made through a ha_em_rb_query
 *  (Event Manager Request Block: QUERY) structure.  This structure is
 *  included in the ha_em_res_blk union, defined in ha_emapi.h.
 */

struct ha_em_rb_query {
    char        *em_class;          /* input:  resource variable class name  */
    char        *em_name;           /* input:  resource variable name        */
    char        *em_ivector;        /* input:  instantiation vector          */
};

/*
 *  Resource variables can have one of three types of values, signed long
 *  integer, float, or structured byte string.  The ha_em_val (Event Manager
 *  VALue) union reflects these possibilities.
 */

union ha_em_val {
```

```
    long    em_vall;                /* output: long integer value           */
    float   em_valf;                /* output: floating point value         */
    void    *em_valsbs;             /* output: structured byte string value  */
};

/*
 *  Event Manager error codes consist of a general error code and a specific
 *  error code, encoded in the ha_em_errnum union.
 */

union ha_em_errnum {
    unsigned int    em_error_number;
    unsigned short  em_error_codes[2];
};

#define em_errnum   em_error.em_error_number    /* for quick tests  */
#define em_generr   em_error.em_error_codes[0]  /* general error code */
#define em_specerr  em_error.em_error_codes[1]  /* specific error code */

/*
 *  The occurrence of an event, or the unregistration of an event, is reported
 *  through a ha_em_rpb_event (Event Manager ResPonse Block: EVENT) structure.
 *  This structure is included in the ha_em_resp_blk union, defined later in
 *  this file.
 */

struct ha_em_rpb_event {
    union ha_em_errnum  em_error;       /* output: error number             */
    ha_em_eid_t         em_event_id;    /* output: event identifier         */
    unsigned long       em_event_flags; /* output: event flags. See below   */
    struct timeval      em_timestamp;   /* output: time of event            */
    int                 em_location;    /* output: node generating event    */
    char                *em_name;       /* output: resource variable name   */
    char                *em_ivector;    /* output: instantiation vector     */
    enum ha_emData_Type em_data_type;   /* output: resource var. data type  */
    union ha_em_val     em_val;         /* output: resource variable value  */
};

#define HA_EM_EVENT_RE_ARM      0x0001  /* event generated from re-arm      */
                                        /*     predicate.                   */
#define HA_EM_EVENT_PRED_FALSE  0x0002  /* predicate evaluated to FALSE.    */
                                        /*     (response to HA_EM_SCMD_REVAL */
                                        /*     subcommand)                  */
#define HA_EM_EVENT_UNREG       0x0004  /* event has been unregistered      */

/*
 *  When the current value of a resource variable is queried, it is reported
 *  through a ha_em_rpb_qcur (Event Manager ResPonse Block: Query CURrent)
 *  structure.  This structure is included in the ha_em_resp_blk union,
 *  defined later in this file.
 */

struct ha_em_rpb_qcur {
    union ha_em_errnum  em_error;       /* output: error number             */
    int                 em_location;    /* output: node containing variable  */
    char                *em_name;       /* output: name of resource variable */
    char                *em_ivector;    /* output: instantiation vector     */
    enum ha_emData_Type em_data_type;   /* output: resource var. data type  */
```

```
    union ha_em_val     em_val;         /* output: resource variable value   */
};

/*
 *  When defined resource variables and predicates are queried, they are
 *  reported through ha_em_rpb_qdef (Event Manager ResPonse Block: Query
 *  DEFined) structure.  This structure is included in the ha_em_resp_blk union,
 *  defined later in this file.
 */

struct ha_em_rpb_qdef {
    union ha_em_errnum  em_error;   /* output: error number                  */
    char                *em_name;   /* output: name of resource variable     */
    char                *em_descrp; /* output: resource variable description */
    enum ha_emValue_Type em_value_type; /* output: Counter, Quantity, or     */
                                    /*         State                         */
    enum ha_emData_Type  em_data_type;  /* output: long, float, or structured*/
                                    /*         byte string (SBS)             */
    char    *em_sbs_format;         /* output: description of SBS format     */
    char    *em_init_value;         /* output: initial value of variable     */
    char    *em_class;              /* output: variable's resource class     */
    char    *em_ivector;            /* output: instantiation vector defn.    */
    char    *em_ivector_descrp;     /* output: instantiation vector descrip. */
    char    *em_ptx_name;           /* output: PTX shared memory name        */
    char    *em_dflt_pred;          /* output: default predicate             */
    char    *em_event_descrp;       /* output: event description             */
    char    *em_locator;            /* output: location indicator            */
    char    *em_order_group;        /* output: resource variable order group */
};

/*
 *  When the parsing of an element in an event registration request results in
 *  an error, it is reported through a ha_em_rpb_rerr (Event Manager ResPonse
 *  Block: Registration ERRor) structure.  This structure is included in the
 *  ha_em_resp_blk union, defined later in this file.
 *
 *  The fields in this structure are a reflection of the information in the
 *  request in error.
 */

struct ha_em_rpb_rerr {
    union ha_em_errnum  em_error;   /* output:  error number                 */
    char                *em_name;   /* output:  resource variable name       */
    char                *em_ivector;/* output:  instantiation vector         */
    char                *em_pred;   /* output:  predicate                    */
    char                *em_rapred; /* output:  re-arm predicate             */
    short               em_errinfo0;/* output: additional error info         */
    unsigned short      em_errinfo1;/* output: additional error info         */
    ha_em_eid_t         em_event_id;    /* output:  event identifier         */
};
#define em_pred_err em_errinfo0     /* output: predicate parsing error       */
#define em_pred_pos em_errinfo1     /* output: position of parsing error */

/*
 *  When the parsing of an element in a query request results in an error, it
 *  is reported through a ha_em_rpb_qerr (Event Manager ResPonse Block: Query
 *  ERRor) structure.  This structure is included in the ha_em_resp_blk union,
 *  defined later in this file.
```

ha_emapi.h

```
 *
 *  The fields in this structure are a reflection of the information in the
 *  request in error.
 */

struct ha_em_rpb_qerr {
    union ha_em_errnum  em_error;   /* output:  error number                  */
    char                *em_class;  /* output:  resource variable class name  */
    char                *em_name;   /* output:  resource variable name        */
    char                *em_ivector;/* output:  instantiation vector          */
    unsigned short      em_errinfo; /* output: additional error info         */
};

/*
 *  Multiple responses of a particular type are made through a ha_em_resp_blk
 *  (Event Manager RESPonse BLocK) union.  All the responses are of the
 *  same type.  The number of elements in the array depends on the number
 *  of responses being made.  This union is included in the ha_em_rsp_blk
 *  structure, defined later in this file.
 */

union ha_em_resp_blk {
    struct ha_em_rpb_event  em_rpb_event[1];    /* Used for events           */
    struct ha_em_rpb_qcur   em_rpb_qcur[1];     /* Used for current values   */
    struct ha_em_rpb_qdef   em_rpb_qdef[1];     /* Used for defined variables*/
    struct ha_em_rpb_rerr   em_rpb_rerr[1];     /* Used for reg cmd errors   */
    struct ha_em_rpb_qerr   em_rpb_qerr[1];     /* Used for query cmd errors */
};

/*
 *  A block of responses of a particular type are delivered through a
 *  ha_em_rsp_blk (Event Manager ReSPonse BLocK) structure passed through
 *  the ha_em_receive_command() routine. All the responses are of the same type.
 *  The type of the requests are identified in the em_cmd and em_subcmd
 *  fields.  The responses themselves are in an array within the ha_em_resp_blk
 *  union.
 */

struct ha_em_rsp_blk {
    int                     em_rsp_blk_len; /* output: length of responses   */
    int                     em_rsp_num_resp;/* output: number of responses   */
    short                   em_cmd;         /* output: type of responses     */
    short                   em_subcmd;      /* output: subtype of responses  */
    ha_em_qid_t             em_qid;         /* output: query identifier      */
    int                     em_qend;        /* output: query response end    */
    union ha_em_resp_blk    em_resp_blk;    /* output: the array of responses*/
};

#endif  /* _HA_EMAPI_BASE_H */
```

ha_emcommon.h File

```
/*============================================================================*/
/* Licensed Materials - Property of IBM                                       */
/*                                                                            */
/* 5765-529                                                                   */
/*                                                                            */
/* (C) Copyright IBM Corp. 1996 All Rights Reserved.                          */
/*                                                                            */
/* US Government Users Restricted Rights - Use, duplication or disclosure    */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                     */
/*============================================================================*/
/*                                                                            */
/* Module Name:  ha_emcommon.h                                                */
/*                                                                            */
/* Description:                                                               */
/*     Common definitions for the Event Management Subsystem                  */
/*                                                                            */
/*============================================================================*/

#ifndef _HA_EMCOMMON_H
#define _HA_EMCOMMON_H

#define HA_EM_MAXERRMSG  256
#define HA_EM_MAXERRLVL  16
#define HA_EM_MAXERRFN   32

struct ha_em_err_blk {
        int     em_errline;
        char    em_errlevel[HA_EM_MAXERRLVL];
        char    em_errfile[HA_EM_MAXERRFN];
        int     em_errno;
        char    em_errmsg[HA_EM_MAXERRMSG];
};

#define HA_EM_CLEAR_ERR(X)        memset(X,0,sizeof(struct ha_em_err_blk))

#define HA_EM_VECTOR_SIZE               4               /* max number of elements in an
                                                           instantiation vector */

/* Max length of a vector value including null */
#define HA_EM_VECTOR_ELEM_VALUE_LEN_MAX         32

enum ha_emValue_Type {
        ha_emVTcounter,
        ha_emVTquantity,
        ha_emVTstate
};

enum ha_emData_Type {
        ha_emDTlong,
        ha_emDTfloat,
        ha_emDTsbs
};

enum ha_emField_Type {
        ha_emFTlong,
        ha_emFTfloat,
```

```
        ha_emFTchar,
        ha_emFTbyte
};

enum ha_emElement_Type {
        ha_emETint,
        ha_emETstring
};

/* define pathnames for lock, socket, log and config files */

/* lock file to ensure single instance of a resource monitor. First %s is
 * resource monitor name, second %s is partition name
 */
#define HA_EM_RM_LOCK_FMT               "/var/ha/lck/em.RM%s.%s"

/* file used in creation of key for PTX shared memory created by resource
 * monitor. "Conceptually" a lock file, so put it in lck directory. %s
 * is resource monitor name.
 *
 * N.B. While there may be multiple resource monitors of the same name executing
 * on the CWS, such resource monitors may only supply State variables. Only
 * once instance of a resource monitor supplying Counter or Quantity variables
 * may execute on the CWS.
 */
#define HA_EM_RMSHM_KEY_FMT             "/var/ha/lck/em.RM%sSHM"

/* lock file to ensure single instance of an event management daemon.
 * %s is partition name.
 */
#define HA_EM_DAEMON_LOCK_FMT   "/var/ha/lck/em.haemd.%s"

/* name of Unix domain socket used by EMAPI to connect to event management
 * daemon. %s is partition name.
 */
#define HA_EM_CL_TO_D_SOCK_FMT  "/var/ha/soc/em.clsrv.%s"

/* name of Unix domain socket used by RMAPI to connect to event management
 * daemon. %s is partition name.
 */
#define HA_EM_RM_TO_D_SOCK_FMT  "/var/ha/soc/em.rmsrv.%s"

/* name of Unix domain socket used by event management daemon to connect to
 * resource monitor. First %s is resource monitor name, second is partition
 * name.
 */
#define HA_EM_D_TO_RM_SOCK_FMT  "/var/ha/soc/em.RM%s.%s"

/* name of event management daemon log file for tracing. %s is partition
 * name.
 */
#define HA_EM_HAEMD_TRACE_LOG_FMT       "/var/ha/log/em.trace.%s"

/* name of event management daemon log file for tracing messages between the
 * various daemon components. %s is partition name.
 */
#define HA_EM_HAEMD_MSGTRACE_LOG_FMT    "/var/ha/log/em.msgtrace.%s"
```

```
/* name of event management daemon log file for default messages. These are
 * messages produced before logging to error log is possible. %s is partition
 * name.
 */
#define HA_EM_HAEMD_DFLT_LOG_FMT        "/var/ha/log/em.default.%s"

/* name of event management configuration data base. %s is partition name. */
#define HA_EM_HAEMD_CFGDB_FMT   "/etc/ha/cfg/em.%s.cdb"

/* name of file that contains the version string of the CDB currently
 * being used by the event management daemon. This is used by the RMAPI
 * to insure it is using the same copy. %s is partition name.
 */
#define HA_EM_HAEMD_CFGDB_VERS_FMT   "/etc/ha/cfg/em.%s.cdb_vers"

/* name of directory where event management instance executes; place where
 * core files are produced. %s is partition name.
 */
#define HA_EM_HAEMD_CWD_FMT             "/var/ha/run/haem.%s"

/* define the service name for the port used by Event Management for peer
 * communication. %s is partition name.
 */
#define HA_EM_SERVICE_NAME_FMT          "haem.%s"

/*
 * Predicate parsing and evaluation errors.
 */

#define HA_EM_PRED_ERVAR                1       /* Internal resource variable error. */
#define HA_EM_PRED_EPREDLEN             2       /* Parsing table overflow. */
#define HA_EM_PRED_ESYNTAX              3       /* Error in predicate syntax. */
#define HA_EM_PRED_ESTACK               4       /* Internal stack overflow. */
#define HA_EM_PRED_EILLEGALOP           5       /* Illegal operation between types. */
#define HA_EM_PRED_ESTRING              6       /* Error in string constant syntax. */
#define HA_EM_PRED_ESBSSN               7       /* SBS out of range or missing from value. */
#define HA_EM_PRED_EFIELDTYPE           8       /* Illegal or mismatch in SBS field type. */
#define HA_EM_PRED_EDATATYPE            9       /* Invalid internal data type. */
#define HA_EM_PRED_ESBSLEN              10      /* Mismatch SBS data type and field length. */
#define HA_EM_PRED_EDIVIDE              11      /* Divide by zero. */

#endif  /* _HA_EMCOMMON_H */
```

Related Information

EMAPI subroutines: **ha_em_end_session**, **ha_em_receive_response**, **ha_em_restart_session**, **ha_em_send_command**, **ha_em_start_session**

Predicates (haempred)

Purpose

Predicates – Predicates used in Event Management

Description

In the Event Management Application Programming Interface (EMAPI), a predicate is a condition that a resource variable must meet in order to generate an event.

Predicates that represent conditions of interest are defined by the programmers of Event Management clients. Default predicates are defined by the designers of resource monitors when they design the data that their resource monitors report.

A predicate is a condition that a resource variable must meet in order to generate an event. More formally, a predicate is the relational expression between a resource variable and other elements, such as a constant or the value of a variable from the previous observation.

The predicate is applied to each instance of the resource variable as it is observed. If the predicate is true, an event is generated. More than one predicate may be applied to an instance of the same resource variable at the same observation.

An example of a simple predicate is **X < 10**, where **X** is a resource variable that represents the percentage of free space in a file system. This predicate would generate an event whenever the file system's free space was observed to be less than 10%.

For information about how resource variables are defined, see “Resource Variables (haemrviv)” on page 141.

A predicate is of the following form:

```
predicate::     expression rel_op constant
                expression rel_op expression
                predicate log_op predicate
                unary_op predicate
                unary_op expression

expression::    var_name
                var_name arith_op constant
                var_name arith_op expression

var_name::      X
                X@var_name_mod

var_name_mod::  one of
                P, R, PR, sbs_sn, Psbs_sn

unary_op::      !

rel_op::        one of
                ==  !=  <  >  <=  >=

log_op::        one of
                &&  ||

arith_op::      one of
                *  /  %  +  -

sbs_sn::        a structured field serial number
```

Operators have the same meaning and precedence as in the C language. Parentheses may be used for grouping as in C. Constants are integer or floating point, also as in C.

The variable name modifier **P** indicates that the value of the variable instance from its previous observation is used. The variable name modifier **R** indicates that the raw value of the variable instance is used; this is useful only with resource variables of type Counter. When both of these modifiers are present (**PR**), the raw value of the variable instance from its previous observation is used. If neither modifier is present for a Counter, a rate is used. The rate is calculated by subtracting the raw value of the previous observation of the variable instance from the raw value of the latest observation, and dividing it by the time, in seconds, between the two observations.

The variable name modifier *sbs_sn* is a structured field serial number. This modifier is used only with a structured byte string (SBS) resource variable and is used to select the structured field value that is to be used in the evaluation of the predicate. When used with **P**, the selected structured field is taken from the value of the variable instance from its previous observation. For more information about the format of SBS resource variables, see “Resource Variables (haemrviv)” on page 141.

Operands that are modified to select structured fields of type character string or byte string may only be used with the relational operators; both operands must be of the same type. When the operands are character strings, the implied comparison and its result are equivalent to the C library function **strcmp**(*loprn*,*roprn*), where *loprn* is the left operand in the predicate and *roprn* is

the right operand in the predicate. When the operands are byte strings, the implied comparison and its result are equivalent to the C library function **memcmp**(*loprn*,*roprn*,*N*), where *loprn* is the left operand in the predicate, *roprn* is the right operand in the predicate, and *N* is the length of the shortest byte string.

Predicates use the letter "X" to represent the resource variable name. The variable name may be repeated in the predicate and may be in any one of its modified forms.

Here are some examples of valid predicate definitions:

`X == 0`	The value of the resource variable instance is equal to zero.
`X < 20 || X > 80`	The value of the resource variable instance is less than 20 or greater than 80.
`!(X < 20 || X > 80)`	The value of the resource variable instance is neither less than 20 nor greater than 80.
`X@R > X@PR`	The current raw value of the variable instance is greater than the raw value of the variable instance from its previous observation.
`X >= X@P + 5`	The current value of the variable instance is greater than or equal to the value of the variable instance from its previous observation plus 5.
`X@2 != 100`	For a resource variable instance that is defined as an SBS, the value of the structured field number 2 is not equal to 100.
`X@0 != X@P0 || X@1 != X@P1`	For a resource variable instance that is defined as an SBS, the value of either structured field number 0 or structured field number 1 has changed since its previous observation.

Related Information

EMAPI subroutines: **ha_em_receive_response**, **ha_em_send_command**

Resource Variables (haemrviv)

Purpose

Resource Variables and Instance Vectors – Resource variables and instance vectors used in Event Management

Description

In the Event Management Application Programming Interface (EMAPI) and the Resource Monitor Application Programming Interface (RMAPI), the attributes of system resources are represented by resource variables and the copy of each resource is represented by an instance vector.

Resource variables and instance vectors are defined by the programmers of resource monitors. Unless otherwise indicated, the definitions and specifications described here apply to both the EMAPI and the RMAPI.

Resource Variables

A resource variable represents the attribute of a system resource. Associated with each resource variable is a name, a value type, a data type, an instance vector and, optionally, a location.

Resource Variable Names

A resource variable is identified by a **resource variable name**, which is a string that consists of a resource name followed by a period followed by the resource attribute.

By convention, a **resource name** represents a hierarchical organization of components, from general to specific, similar to a filename. It consists of a series of two or more components separated by periods, as follows:

- The first component is the name of the vendor that supplies the subsystem or application that manages the resource.
- The second component is the name of the product that contains the subsystem or application.
- Any additional components that are required to name the resource appear after the second component.

The **resource attribute** is a single component that follows the resource name.

Each component of the resource variable name is a character string that may contain only alphanumeric characters, underscores, or a percent sign. The first component (vendor name) must begin with an alphabetic character. All other components must begin with either an alphabetic character or a percent sign.

Here are some examples of resource variable names:

```
IBM.PSSP.aixos.CPU.gluser
IBM.PSSP.aixos.Mem.Real.size
IBM.PSSP.aixos.Mem.Virt.pagein
IBM.PSSP.aixos.Mem.Virt.pageout
IBM.PSSP.aixos.Mem.Kmem.inuse
IBM.PSSP.aixos.PagSp.%totalfree
IBM.PSSP.aixos.Disk.busy
IBM.PSSP.Membership.Node.state
```

The name space for resource variable names is local to a system partition.

Resource Variable Value Types

A resource variable belongs to one of three value types: Counter, Quantity, or State.

A **Counter** is a resource variable that represents a rate. It can have a data type of either long or float. Typically, a Counter represents throughput. Examples include paging rates, I/O rates, and transaction rates.

By default, the Event Management subsystem presents the value of an instance of a Counter as a rate. The rate represents the change in the actual contents of the instance of the Counter from one observation to the next divided by the time between the two observations. However, the Event Management subsystem can also present the **raw value** of the instance of the Counter, which is the actual contents of the instance of the Counter from the latest observation. The raw value of a Counter always increases over time.

For example, the resource variable called **IBM.PSSP.aixos.Mem.Virt.pagein** is a Counter that represents the rate at which pages are paged into virtual memory. Normally, its value is presented as a rate (number of pages per second). However, its raw value indicates the total number of pages that have been paged in since the Counter was initialized.

A Counter can have a data type of either long or float. These formats are identical to the C language types of the same names.

The Event Management subsystem assumes that instances of a Counter may be observed at an interval different from that of their update without loss of meaningful information.

A **Quantity** is a variable whose value fluctuates over time. Typically, a Quantity is used to represent a level, that is, an indicator of how many.

A Quantity can have a data type of either long or float. These formats are identical to the C language types of the same names.

The Event Management subsystem assumes that instances of a Quantity may be observed at an interval different from that of their update without loss of meaningful information.

A **State** is a variable whose value fluctuates over time, like a Quantity. Unlike a Quantity, however, the semantics of a State variable are assumed to be such that every change in value of an instance of a State variable must be observed; otherwise, meaningful information would be lost. A State variable can be used to

represent an attribute of a resource that indicates anything other than throughput or a level.

A State can have a data type of long, float, or structured byte string (SBS). The long and float formats are identical to the C language types of the same names. The format of an SBS is described in "Resource Variable Data Types."

Resource Variable Data Types

A Counter and a Quantity can have a data type of either long or float. The long and float formats are identical to the C language types of the same names.

A State can have a data type of long, float, or structured byte string. The long and float formats are identical to the C language types of the same names.

A **structured byte string** (**SBS**) is a string of bytes, where each byte may have any value from 0 through 255, that consists of a four-byte SBS length field followed by one or more structured fields. The SBS length specifies the total length of the structured fields that follow it.

A structured field consists of a four-byte header followed by a value. The first two bytes of the header are the length of the structured field value, the third byte is a structured field data type, and the fourth byte is a serial number.

The structured field type is one of long, float, character string, or byte string. Long and float are the same as in the C language. A character string consists of one or more nonzero bytes terminated by a null byte; the null byte is included in the structured field value length. A byte string consists of one or more bytes, where each byte may have any value from 0 through 255.

The serial number is a unique value that identifies the structured field. This serial number is defined for each structured field by the resource monitor that supplies the SBS resource variable. However, the set of serial numbers for the structured byte string starts with 0 and is contiguous.

Associated with each structured field in an SBS, but not included in the actual SBS, is a structured field name. This name is used to provide a short description of the structured byte field and must consist of only alphanumeric characters and underscores; the first character must be alphabetic.

Here are some examples of structured byte string definitions, taken from resource variables provided by PSSP resource monitors:

```
------------------------------------------------------------------------
SBS associated with resource variable IBM.PSSP.Prog.pcount

SBS Length = variable
SBS Fields
    Field Name   Field Length   Field Type         Field Serial Number
    CurPIDCount    4               long              0
    PrevPIDCount   4               long              1
    CurPIDList     variable        cstring           2

------------------------------------------------------------------------
SBS associated with resource variable IBM.PSSP.pm.Errlog

SBS Length = variable
SBS Fields
    Field Name          Field Length   Field Type    Field Serial Number
    sequenceNumber      variable       cstring          0
    errorID             variable       cstring          1
    errorClass          variable       cstring          2
    errorType           variable       cstring          3
    alertFlagsValue     variable       cstring          4
    resourceName        variable       cstring          5
    resourceType        variable       cstring          6
    resourceClass       variable       cstring          7
    errorLabel          variable       cstring          8
```

The Counter and Quantity types have the same definition as the value types for statistics objects that are defined by the Performance Toolbox for AIX (PTX/6000) System Performance Measurement Interface (SPMI). The SPMI is used directly by Event Management.

Resource Variable Instances and Instance Vectors

Most resources in the system have multiple copies. Each of these copies is an **instance** of the resource.

The resource variables that represent the states of these resources also have multiple copies. Each of these copies is an **instance** of the resource variable.

To uniquely identify each copy of a resource and all of its variables, each resource in the system has one, and only one, associated instance vector. An **instance vector** is a list of elements, where each element is a name/value pair. In the EMAPI, the elements are separated by semicolons. In the RMAPI, the elements are separated by commas.

A name/value pair consists of an instance vector element name followed by an equal sign followed by the value of the vector element. There are no blanks in the instance vector. An instance vector can contain up to 4 elements, as defined by the **HA_EM_VECTOR_SIZE** constant.

The instance vector element name is a string that describes the element. It must consist of only alphanumeric characters and underscores; the first character must be alphabetic. In the EMAPI, an instance vector element value may consist of a single value, a range of values, a comma-separated list of single values, or a comma-separated list of ranges. A range takes the form *a-b* and is valid only for vector elements of type integer. In the RMAPI, only single values are allowed.

An instance vector element may be wildcarded in a manner that varies by subroutine.

The set of values in the instance vector uniquely identify the copy of the resource in the system. By extension, they also uniquely identify the copy of the resource variable in the system. If there is only one copy of the resource in the system (for example, the control workstation), its instance vector is null.

The semantics of an instance vector are defined by the semantics of the resource variable with which it is associated. The names of the vector elements must be unique within a given variable's instance vector. These names may be used in other instance vectors and may or may not have the same semantics. In other words, the name space for instance vector element names is local to each defined resource variable.

Here are some examples of PSSP resource variable names and instance vectors:

```
Resource Variable Name              Instance Vector

IBM.PSSP.aixos.CPU.gluser           NodeNum=5
IBM.PSSP.aixos.CPU.kern             NodeNum=5;Cpu=cpu0
IBM.PSSP.aixos.Mem.Real.size        NodeNum=5
IBM.PSSP.aixos.Mem.Virt.pagein      NodeNum=5
IBM.PSSP.aixos.Mem.Virt.pageout     NodeNum=5
IBM.PSSP.aixos.Mem.Kmem.inuse       NodeNum=5;Type=mbuf
IBM.PSSP.aixos.PagSp.%totalfree     NodeNum=5
IBM.PSSP.aixos.Disk.busy            NodeNum=5;Name=hdisk0
IBM.PSSP.aixos.Disk.busy            NodeNum=5;Name=hdisk1
IBM.PSSP.aixos.VG.free              NodeNum=5;VG=rootvg
IBM.PSSP.aixos.VG.free              NodeNum=5;VG=spdata
IBM.PSSP.aixos.FS.%totused          NodeNum=5;VG=rootvg;LV=hd4
IBM.PSSP.aixos.FS.%totused          NodeNum=5;VG=rootvg;LV=hd4
IBM.PSSP.SP_HW.Node.powerLED        NodeNum=5
IBM.PSSP.SP_HW.Frame.frACLED        FrameNum=1
IBM.PSSP.SP_HW.Switch.powerLED      SwitchNum=1
IBM.PSSP.Membership.Node.state      NodeNum=5
IBM.DB2.Part.size                   DBname=foo;Part=3
```

Although many resources are associated with a particular node, some are not. Therefore, not all instance vectors contain a node number.

Although components of a resource variable name can be used in more than one name, resource names must uniquely identify a resource.

Dynamically Instantiable Resource Variables

In most cases, the instances of a resource that a resource monitor is responsible for tracking and reporting on are static or relatively so. Thus, a resource monitor knows from the time it starts running, after any necessary initialization occurs, all of the instances of the resources it monitors. For example, after any necessary initialization, a hardware monitor would know what frames there are in the system, a node health monitor would know what nodes there are in the system, and a disk monitor would know what disks there are in the system.

However, in some cases, a resource monitor knows how to track and report on a type of resource, but the resource monitor does not know all of the possible

instances of the resource that may occur. For example, the program resource monitor knows how to track and report on a program that is running in a system partition, but it does not know in advance the names of all possible programs that can run. For a resource like this, you can define the resource variable as **dynamically instantiable**.

If a resource variable is dynamically instantiable, the resource monitor does not register all of the instances of its resource variables. Instead, it waits for the Event Management subsystem to tell it what instances to register. As the Event Management subsystem receives requests from EM clients for events based on instances of dynamically instantiable variables, the Event Management subsystem passes the instances to the resource monitor in a control message with an "instantiate variables" command.

Location of a Resource Variable Instance

The **location** of a resource variable instance is the node on which the *resource monitor* that supplies the instance resides. The resource monitor may or may not reside on the same node as the instance of the resource.

Resource Variable Classes

The **class** of a resource variable is used to indicate the subsystem or application that manages the associated resource. It can also be used to group resource variables by observation interval. Instances of all resource variables of value type Counter and Quantity in the same class are observed at the same intervals.

By convention, class names follow the same hierarchical naming conventions that are used for resource names.

Each class name must be unique among class names within a system partition.

Here are some examples of class names:

Resource Variable Class	Resource Variables in the Class
IBM.PSSP.aixos.CPU	IBM.PSSP.aixos.CPU.gluser
	IBM.PSSP.aixos.CPU.glkern
	IBM.PSSP.aixos.CPU.glwait
	IBM.PSSP.aixos.CPU.glidle
	IBM.PSSP.aixos.CPU.user
	IBM.PSSP.aixos.CPU.kern
	IBM.PSSP.aixos.CPU.wait
	IBM.PSSP.aixos.CPU.idle
IBM.PSSP.aixos.Mem	IBM.PSSP.aixos.Mem.Real.size
	IBM.PSSP.aixos.Mem.Real.numfrb
	IBM.PSSP.aixos.Mem.Real.%free
	IBM.PSSP.aixos.Mem.Real.%pinned
	IBM.PSSP.aixos.Mem.Virt.pagein
	IBM.PSSP.aixos.Mem.Virt.pageout
	IBM.PSSP.aixos.Mem.Virt.pgspgin
	IBM.PSSP.aixos.Mem.Virt.pgspgout
	IBM.PSSP.aixos.Mem.Virt.pagexct
	IBM.PSSP.aixos.Mem.Kmem.inuse
	IBM.PSSP.aixos.Mem.Kmem.calls
	IBM.PSSP.aixos.Mem.Kmem.failures
	IBM.PSSP.aixos.Mem.Kmem.memuse
IBM.PSSP.aixos.Disk	IBM.PSSP.aixos.Disk.busy
	IBM.PSSP.aixos.Disk.xfer
	IBM.PSSP.aixos.Disk.rblk
	IBM.PSSP.aixos.Disk.wblk

Related Information

EMAPI subroutine: **ha_em_send_command**

RMAPI Errors (err_rmapi)

Purpose

RMAPI Errors – Error numbers and the error block for the Resource Monitor Application Programming Interface (RMAPI)

Description

The Resource Monitor Application Programming Interface (RMAPI) provides a class of errors that are returned synchronously by RMAPI subroutines.

Synchronous Errors Returned by RMAPI Subroutines

If an RMAPI subroutine is unsuccessful, it returns a value of -1 and other error information in the error block specified on input. The error block contains an error number and a null-terminated error message.

The RMAPI uses an error block that is common to both the EMAPI and the RMAPI. The block is defined in the **ha_emcommon.h** header file that is included by the **ha_rmapi.h** header file.

The RMAPI error numbers are defined in the **ha_rmapi.h** header file.

The Event Management error block has the following definition:

```
struct ha_em_err_blk {
    int     em_errline;
    char    em_errlevel[HA_EM_MAXERRLVL];
    char    em_errfile[HA_EM_MAXERRFN];
    int     em_errno;
    char    em_errmsg[HA_EM_MAXERRMSG];
}
```

The **em_errline**, **em_errlevel**, and **em_errfile** fields are reserved for IBM use in providing information for problem determination.

If an error occurs, the **em_errno** field contains one of the values listed below. The **em_errmsg** field contains a message text that describes the error in further detail. For information about RMAPI messages, see *PSSP Diagnosis and Messages Guide*.

In addition to indicating the reason that an RMAPI subroutine is unsuccessful, RMAPI error numbers are also used to indicate errors in individual elements of the **ha_rr_variable** array that was passed to the **ha_rr_add_var**, **ha_rr_del_var**, **ha_rr_reg_var**, or **ha_rr_unreg_var** subroutine. The error number is returned in the **rr_var_errno** field of the **ha_rr_variable** array element.

The RMAPI error numbers are:

HA_RR_ENOSDR
: A System Data Repository (SDR) function failed or an SDR class has a missing or invalid attribute.

HA_RR_EACCESS
The RMAPI has not been initialized or an error occurred while trying to open or lock a file.

HA_RR_EBADCDB
There is problem with the Event Management Configuration Database (EMCDB) file. Data in the EMCDB file does not match data in the System Data Repository (SDR) or the EMCDB file has been corrupted.

HA_RR_ESPMIFAIL
A System Performance Measurement (SPMI) routine failed.

HA_RR_ENOTSERVER
The resource monitor has not been defined as a server.

HA_RR_ENOHNDLR
No signal handler has been defined for **SIGIO**.

HA_RR_ENOSERVSOCK
The resource monitor server socket does not exist.

HA_RR_EBADPROTO
The notification protocol argument does not match the notification protocol that was specified by a previous call.

HA_RR_EEXIST
The RMAPI has already been (or has not been) initialized, the resource monitor does not exist, or the session or socket already exists. For details, see the accompanying message.

HA_RR_EAGAIN
A request was made to accept a connection on the resource monitor server socket but no connection request was found, or a request to read data from a connection failed because there was no data ready to be read.

HA_RR_EMAXSESSIONS
A request for a session was denied because the number of active sessions would exceed the maximum number that are allowed.

HA_RR_EBADSESSION
No session matching the specified file descriptor was found.

HA_RR_EDISCONNECT
The connection with a resource monitor manager was lost.

HA_RR_ENOENT
The specified entity does not exist. For details, see the accompanying message.

HA_RR_EBADVECNAME
The instance vector was incorrectly specified. The instance vector value is missing, contains blanks, or is too long (that is, its length is greater than **HA_EM_VECTOR_ELEM_VALUE_LEN_MAX**).

HA_RR_ENOVECNAME
The instance vector was incorrectly specified. The instance vector name/value pair is missing for one or more vector elements.

HA_RR_EBADVECTOR
The instance vector was incorrectly specified, due to one of the following conditions:

- The data type of the vector element is defined as an integer but the string contains one or more nonnumeric characters.
- The variable for which the vector string was specified has no vector elements defined for it.
- The instance vector value is missing.

HA_RR_EDUPVECTOR
A duplicate instance vector was found.

HA_RR_EBADCLASS
The requested resource class name was not found for this resource monitor.

HA_RR_ESYSCALL
An error was returned by a system call.

HA_RR_ENOMEM
Required memory could not be allocated.

HA_RR_EINTERNAL
The RMAPI encountered an internal error.

HA_RR_EPARM
A function argument passed to the routine is not valid.

HA_RR_EHADDR
The address of the variable handle location is null.

HA_RR_EHANDLE
The supplied variable handle is null.

HA_RR_ENOTREG
The specified variable instance is not registered.

HA_RR_EDUPREG
The specified variable instance has already been registered.

Related Information

RMAPI subroutines: **ha_rr_add_var**, **ha_rr_del_var**, **ha_rr_end_session**, **ha_rr_get_ctrlmsg**, **ha_rr_get_interval**, **ha_rr_init**, **ha_rr_makserv**, **ha_rr_reg_var**, **ha_rr_send_val**, **ha_rr_start_session**, **ha_rr_terminate**, **ha_rr_touch**, **ha_rr_unreg_var**

RMAPI header file: **ha_rmapi.h**

Messages: *PSSP Diagnosis and Messages Guide*.

ha_rmapi.h File

Purpose

ha_rmapi.h – Header file for the Resource Monitor Application Programming Interface (RMAPI)

Description

The **ha_rmapi.h** header file provides data types and structures for use with the Resource Monitor Application Programming Interface (RMAPI) subroutines, which reside in the **libha_rr.a** library. The **libha_rr.a** library is not thread-safe. Any program that uses the RMAPI subroutines must include this file, which resides in the /**usr**/**include** directory.

The following listing shows the contents of the **ha_rmapi.h** file. For the contents of the nested header file, **ha_emcommon.h**, see **ha_emapi.h**.

```
/*============================================================================*/
/* Licensed Materials - Property of IBM                                       */
/*                                                                            */
/* 5765-529                                                                   */
/*                                                                            */
/* (C) Copyright IBM Corp. 1996 All Rights Reserved.                          */
/*                                                                            */
/* US Government Users Restricted Rights - Use, duplication or disclosure    */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                     */
/*============================================================================*/
/*                                                                            */
/* Module Name:  ha_rmapi.h                                                   */
/*                                                                            */
/* Description:                                                               */
/*      Include for Event Management resource monitor api (librmapi.a)        */
/*                                                                            */
/*============================================================================*/
#ifndef _H_HA_RMAPI
#define _H_HA_RMAPI

#include <ha_emcommon.h>

/******************************************************************************
 *
 * Good and Bad return codes
 *
 ******************************************************************************/

#define HA_RR_PASS                      (int)0
#define HA_RR_FAIL                      (int)-1

/******************************************************************************
 *
 * rmapi error codes
 *
 ******************************************************************************/
```

ha_rmapi.h

```
#define HA_RR_ENOSDR            (int)101
#define HA_RR_EACCESS           (int)102
#define HA_RR_EBADCDB           (int)103
#define HA_RR_ESPMIFAIL         (int)104
#define HA_RR_ENOTSERVER        (int)105
#define HA_RR_ENOHNDLR          (int)106
#define HA_RR_ENOSERVSOCK       (int)107
#define HA_RR_EBADPROTO         (int)108
#define HA_RR_EEXIST            (int)109
#define HA_RR_EAGAIN            (int)110
#define HA_RR_EMAXSESSIONS      (int)111
#define HA_RR_EBADSESSION       (int)112
#define HA_RR_EDISCONNECT       (int)113
#define HA_RR_ENOENT            (int)114
#define HA_RR_EBADVECNAME       (int)115
#define HA_RR_ENOVECNAME        (int)116
#define HA_RR_EBADVECTOR        (int)117
#define HA_RR_EDUPVECTOR        (int)118
#define HA_RR_EBADCLASS         (int)119
#define HA_RR_ESYSCALL          (int)120
#define HA_RR_ENOMEM            (int)121
#define HA_RR_EINTERNAL         (int)122
#define HA_RR_EPARM             (int)123
#define HA_RR_EHADDR            (int)124
#define HA_RR_EHANDLE           (int)125
#define HA_RR_ENOTREG           (int)126
#define HA_RR_EDUPREG           (int)127

/******************************************************************************
 *
 * ctrl command values received from resource managers
 *
 ******************************************************************************/

#define HA_RR_CMD_ADDV          (int)201
#define HA_RR_CMD_ADDALL        (int)202
#define HA_RR_CMD_DELV          (int)203
#define HA_RR_CMD_DELALL        (int)204
#define HA_RR_CMD_REFRESH       (int)205
#define HA_RR_CMD_INSTV         (int)206
#define HA_RR_CMD_MGR_ID        (int)207

/******************************************************************************
 *
 * ctrl msg notification method HA_RR_NOTIFY_SELECT (for select or poll),
 * HA_RR_NOTIFY_SIGIO for SIGIO signal notification
 * HA_RR_MAX_SESSIONS is the max number of client sessions for server RMs
 *
 ******************************************************************************/

#define HA_RR_NOTIFY_SELECT     (int)301
#define HA_RR_NOTIFY_SIGIO      (int)302
#define HA_RR_MAX_SESSIONS      (int)8

/******************************************************************************
 *
 * rmapi structures
 *
```

```
 ****************************************************************************/

struct ha_rr_variable {
        char    *rr_var_name;
        char    *rr_var_ivector;
        union {
                int             rr_var_inst_id;
                void    **rr_var_hndl;
        } rr_varu;
#define rr_var_handle   rr_varu.rr_var_hndl
#define rr_var_iid              rr_varu.rr_var_inst_id
        void    *rr_value;
        int             rr_var_errno;
};

struct ha_rr_val {
        void    *rr_value;
        void    *rr_var_hndl;
};

struct ha_rr_ctrl_msg {
        int             rr_ctrl_msg_len;
        int             rr_ctrl_cmd;
        int             rr_ctrl_cmdarg;
        int             rr_ctrl_num_vars;
        union   ha_rr_ctrlv {
                struct ha_rr_ctrl_var {
                        char    *rr_ctrl_name;
                        char    *rr_ctrl_ivector;
                } rr_ctrl_varn[1];
                int rr_ctrl_vari[1];
        } rr_ctrlv;
#define rr_ctrl_vars    rr_ctrlv.rr_ctrl_varn
#define rr_ctrl_ids             rr_ctrlv.rr_ctrl_vari
};

/****************************************************************************
 *
 * rmapi function prototypes
 *
 ****************************************************************************/

#ifdef _NO_PROTO
int ha_rr_init();
int ha_rr_makserv();
int ha_rr_start_session();
int ha_rr_get_ctrlmsg();
int ha_rr_end_session();
int ha_rr_add_var();
int ha_rr_del_var();
int ha_rr_get_interval();
int ha_rr_send_val();
int ha_rr_reg_var();
int ha_rr_unreg_var();
int ha_rr_terminate();
int ha_rr_touch();
#else   /* NO_PROTO */
int ha_rr_init(char *, struct ha_em_err_blk *);
```

```
int ha_rr_makserv(int, struct ha_em_err_blk *);
int ha_rr_start_session(int, struct ha_em_err_blk *);
int ha_rr_get_ctrlmsg(int, struct ha_rr_ctrl_msg **, struct ha_em_err_blk *);
int ha_rr_end_session(int, struct ha_em_err_blk *);
int ha_rr_add_var(int, struct ha_rr_variable *, int, int, struct ha_em_err_blk *);
int ha_rr_del_var(int, struct ha_rr_variable *, int, struct ha_em_err_blk *);
int ha_rr_get_interval(char *, struct ha_em_err_blk *);
int ha_rr_send_val(struct ha_rr_val *pv, int, int, struct ha_em_err_blk *);
int ha_rr_reg_var(struct ha_rr_variable *, int, struct ha_em_err_blk *);
int ha_rr_unreg_var(struct ha_rr_variable *, int, struct ha_em_err_blk *);
int ha_rr_terminate(struct ha_em_err_blk *);
int ha_rr_touch(struct ha_em_err_blk *);
#endif          /* NO_PROTO */

#endif          /* _H_HA_RMAPI */
```

Related Information

RMAPI subroutines: **ha_rr_add_var**, **ha_rr_del_var**, **ha_rr_end_session**, **ha_rr_get_ctrlmsg**, **ha_rr_get_interval**, **ha_rr_init**, **ha_rr_makserv**, **ha_rr_reg_var**, **ha_rr_send_val**, **ha_rr_start_session**, **ha_rr_terminate**, **ha_rr_touch**, **ha_rr_unreg_var**

Chapter 5. Using the RMAPI: Some Resource Monitor Examples

This chapter contains the listings of three sample resource monitors and their associated files:

- "The rmapi_smpcmd.c Sample Program" on page 156 is an example of a command-based resource monitor.
- "The rmapi_smpdae.c Sample Program" on page 167 is an example of a daemon-based resource monitor that uses **select** system call notification.
- "The rmapi_smpsig.c Sample Program" on page 186 is an example of a daemon-based resource monitor that uses **SIGIO** notification.
- "The rmapi_smp.msg Message File" on page 206 is the message source file for the messages used by the examples.
- "The rmapi_smp.loadsdr Shell Script" on page 208 and "The rmapi_smp.unloadsdr Shell Script" on page 214 are shell scripts that you can use to load the definitions of the resource variables used by the samples into the SDR, and unload them, respectively.

You can find the files for these programs online in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/rmapi**.

© Copyright IBM Corp. 1996

The rmapi_smpcmd.c Sample Program

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ha_rmapi.h>

/*
 *  rmapi_smpcmd.c
 *
 *  This program presents an example of using the Resource Monitor Application
 *  Programming Interface (RMAPI).  The configuration data for this monitor
 *  is in the file RmapiSample.loadsdr.  This monitor is an example of a command
 *  based client monitor (connects to the Event Management daemon) and uses
 *  state type variables.
 *
 *  Event Management objects defined for this monitor:
 *
 *  Monitor:
 *      IBM.PSSP.SampleCmdMon                    # client monitor
 *                                               #
 *  Class:                                       #
 *      IBM.PSSP.SampleCmdClass                  # variable class
 *                                               #
 *  Variables:                                   #
 *      IBM.PSSP.SampleCmdMon.state              # Valuetype state, datatype long used to
 *                                               # reflect the state of the resource instance:
 *                                               # Configured, Unconfigured, Running.
 *        Instance Vectors:  NodeNum             # Used as locator field of variable.
 *                           NAME                # Instance name of the resource.
 *                                               #
 *    IBM.PSSP.SampleCmdMon.call                 # Valuetype state, datatype SBS used to
 *                                               # reflect the state of the last command
 *                                               # for this resource instance.
 *        Instance Vectors:  NodeNum             # Used as locator field of variable.
 *                           NAME                # Instance name of the resource.
 *        SBS Field:         Action              # Datatype long, sn=0.
 *                                               # Last action attempted for this instance.
 *                           Options             # Datatype char, sn=1.
 *                                               # Dummy options string to put into SBS.
 *                           StateChange         # Datatype long, sn=2.
 *                                               # 0 if action failed, else !0.
 *                           State               # Datatype long, sn=3.
 *                                               # Final state of the resource
 *                                                 (same as the state var).
 *
 * What this monitor does:  rmapi_smpcmd provides an example of a command which
 *   controls some imaginary resource. The command is used to change the "state"
 *   of the resource to one of the following: Unconfigured, Configured, or Running,
 *   by passing an "action" argument: config, unconfig, start, or stop. One
 *   or both of the state variables will be updated through the RMAPI on each
 *   invocation of this command.
 *
 * Legal state transitions (action):
 *    Unconfigured -> Configured    (config)
```

```
*    Configured   -> Unconfigured  (unconfig)
*    Configured   -> Running       (start)
*    Running      -> Configured    (stop)
*
* If a "legal" action is requested, the command will add both of the variables
* for this resource instance to the Event Management client session. The
* IBM.PSSP.SampleCmdMon.state
* variable will be added by the call ha_rr_add_var() with a value of TRANSITION and then
* updated to the new state value by ha_rr_send_val().
*
* If the action is "illegal" for the current state, only the IBM.PSSP.SampleCmdMon.call
* variable
* will be updated so that the IBM.PSSP.SampleCmdMon.state variable remains unchanged.
*
* Parameters:
*
*  -a <action>      Required. Possible options are config, unconfig, start or stop. This is
*                   the action requested.
*
*  -c <curr_state>  Required. Possible options are Configured, Unconfigured, or Running.
*                   Since this command does no real work, this option tells the command
*                   what "state" the resource is in before attempting the action.
*
*  -n <name>        Required. Used as the Instance Vector NAME element to uniquely
*                   identify different instances of the resource variables.
*
*  -o <option>      Optional. Any string, used to demonstrate a char datatype within an SBS
*                   variable.
*
* Eamples:
*
*  1. rmapi_smpcmd -a config -c Unconfigured -n test_resource -o test_opt_str
*
*  Will cause both variables defined for this monitor to be created
*  using "test_resource" as the NAME instance vector value. Both
*  variables will be registered and added to the RMAPI. The IBM.PSSP.SampleCmdMon.call
*  will have the following values sent to the Event Manager daemon on
*  the call to ha_rr_add():
*      action:           0                # CONFIG
*      options:          test_opt_str     # -o flag parm
*      state_change:     1                # successful state change
*      state:            1                # CONFIGURED
*
*  The IBM.PSSP.SampleCmdMon.state variable will have a initial value of TRANSITION
*  sent to the RMAPI on ha_rr_add(). It's value will then be updated to CONFIGURED
*  and sent to the RMAPI using ha_rr_send_val().
*
*  2. rmapi_smpcmd -a unconfig -c Running -n test_resource
*
*  Will cause only the IBM.PSSP.SampleCmdMon.call variable to be created
*  using "test_resource" as the NAME instance vector value. The "state"
*  variable is not sent to the RMAPI because it was unchanged. IBM.PSSP.SampleCmdMon.call
*  will have the following values sent to the Event Manager daemon on
*  the call to ha_rr_add():
*      action:           1                # UNCONFIG
*      options:          No options.      # default value for -o flag
*      state_change:     0                # state change failed
*      state:            2                # RUNNING
```

rmapi_smpcmd.c

```
 *
 *  This program can be compiled with the following command:
 *
 *      cc -O rmapi_smpcmd.c -o rmapi_smpcmd -lha_rr
 */

/* local defines
CMDNAME                  - Name of this command.
RESOURCE_MONITOR_NAME    - Name of this monitor as defined in the configuration database.
STATE_VARIABLE_NAME      - Name of the "state" variable defined for this monitor.
CALL_VARIABLE_NAME       - Name of the "call" variable defined for this monitor.
IVECTOR                  - Instance vector name.
MAX_VARIABLES            - Number of variables defined for this monitor.
DEFAULT_OPTION_STRING    - Default string for -o opt.
USAGE_MSG                - Error msg for cmd syntax errors.
*/
#define CMDNAME                   "rmapi_smpcmd"
#define RESOURCE_MONITOR_NAME     "IBM.PSSP.SampleCmdMon"
#define STATE_VARIABLE_NAME       "IBM.PSSP.SampleCmdMon.state"
#define CALL_VARIABLE_NAME        "IBM.PSSP.SampleCmdMon.call"
#define IVECTOR                   "NAME="
#define MAX_VARIABLES             2
#define DEFAULT_OPTION_STRING     "No options."
#define USAGE_MSG    "Usage: "CMDNAME" -a <action> -c <curr_state> -n <name> -o <options>\n"\
    "\t-a <action>\tResource action (config,unconfig,start or stop).\n"\
    "\t-c <curr_state>\tCurrent \"state\" of the resource (Configured,Unconfigured,Running).\n"\
    "\t-n <name>\tName of the resource instance.\n"\
    "\t-o <options>\tResource instance options.\n"

/* local functions */
void display_rmapi_err(struct ha_em_err_blk *errblk);
enum States process_action(enum States, enum Actions);
void * create_sbs(enum Actions, char *, long, enum States);
void rr_start();
void rr_reg_var();
void rr_add_var();
void rr_del_var();
void rr_end();

/* globals */
struct ha_rr_variable   Variables[MAX_VARIABLES];   /* RMAPI variable array. */
struct ha_rr_val        Values[MAX_VARIABLES];      /* RMAPI value array. */
int                     NumVariables = 0;           /* Number of RMAPI variables to use. */
struct ha_em_err_blk    ErrBlock;                   /* global error block for RMAPI calls. */
int                     ClientSock = HA_RR_FAIL;    /* Client session socket. */
enum Actions {                                      /* State processing actions. */
    CONFIG,
    UNCONFIG,
    START,
    STOP,
    NUM_ACTIONS
};
char *ActionNames[] = {"config", "unconfig", "start", "stop"};
enum States {                                  /* Resource states. */
    CONFIGURED,
    UNCONFIGURED,
    RUNNING,
    NUM_STATES,
```

```
    TRANSITION
};
char *StateNames[] = {"Configured", "Unconfigured", "Running"};

/*****************************************************************************/
/* Main - Main - Main - Main - Main - Main - Main - Main - Main - Main - Main */
/*****************************************************************************/
main(int argc, char **argv)
{
enum States new_state;
enum States curr_state;
enum States trans_state = TRANSITION;
enum Actions action;
char *action_name = NULL;
char *curr_state_name = NULL;
char *inst_name = NULL;
char *inst_opts = NULL;
char *ivec;
int c, i, rc;

    /*
     * Parse the command line arguments.
     */
    while ((c = getopt(argc, argv, "a:c:n:o:")) != EOF) {
        switch (c) {
            case 'a' : action_name = optarg;     break;
            case 'c' : curr_state_name = optarg; break;
            case 'n' : inst_name = optarg;       break;
            case 'o' : inst_opts = optarg;       break;
            default  :
                fprintf(stderr,USAGE_MSG);
                exit(1);
            }
    }

    /*
     * Check for required opts.
     */
    if ((inst_name == NULL) || (action_name == NULL) || (curr_state_name == NULL)) {
        fprintf(stderr,"%s: missing -a, -c or -n flag.\n",CMDNAME);
        fprintf(stderr,USAGE_MSG);
        exit(1);
    }

    /*
     * Check for valid action and state opts.
     */
    for (i=0; (i < NUM_ACTIONS) && strcmp(ActionNames[i],action_name); i++);
    action = i;

    for (i=0; (i < NUM_STATES) && strcmp(StateNames[i],curr_state_name); i++);
    curr_state = i;

    if ((action == NUM_ACTIONS) || (curr_state == NUM_STATES)) {
        fprintf(stderr,"%s: invalid -a or -c parameter.\n",CMDNAME);
        fprintf(stderr,USAGE_MSG);
        exit(1);
    }
```

```
/*
 * Make sure an option string is put into the sbs variable.
 */
if (inst_opts == NULL) {
    inst_opts = DEFAULT_OPTION_STRING;
}

fprintf(stdout,"%s: Current state of resource instance %s is %s. Requested action is %s.\n",
        CMDNAME,inst_name,curr_state_name,action_name);
/*
 * Dummy resource processing. Just checks that the "action" is legal for the
 * current "state" and returns the new state (new_state == curr_state if not
 * a legal action for the current state).
 */
new_state = process_action(curr_state,action);

/*
 * Init the RMAPI variables.
 */
memset(Variables, 0 , sizeof(struct ha_rr_variable) * MAX_VARIABLES);
memset(Values, 0 , sizeof(struct ha_rr_val) * MAX_VARIABLES);

/*
 * Create an ivector for the variables.
 */
ivec = malloc(strlen(inst_name) + strlen(IVECTOR) + 1);
sprintf(ivec,"%s%s",IVECTOR,inst_name);

Variables[0].rr_var_name = CALL_VARIABLE_NAME;
Variables[0].rr_var_ivector  = ivec;
Variables[0].rr_varu.rr_var_inst_id = 0;

Variables[1].rr_var_name = STATE_VARIABLE_NAME;
Variables[1].rr_var_ivector  = ivec;
Variables[1].rr_varu.rr_var_inst_id = 1;

if (new_state != curr_state) {
    /*
     * The "action" was successful indicated by the state change.
     * Send both the new "state" variable and the "call" variable,
     * otherwise only send the failed "call" variable.
     */
    fprintf(stdout,
        "%s: Action %s successful. Resource %s will change states %s->TRANSITION->%s.\n",
        CMDNAME,action_name,inst_name,curr_state_name,StateNames[new_state]);
    NumVariables = 2;
} else {
    fprintf(stderr,"%s: Cannot perform action %s when in state %s. The "
            "\"state\" variable will not be updated.\n",
            CMDNAME,action_name,curr_state_name);
    NumVariables = 1;
}

/*
 * Initialize the RMAPI and client session.
 */
rr_start();
```

```
/*
 * Register variable(s) with the RMAPI.
 */
rr_reg_var();

/*
 * Set the handle addresses and initial values in the
 * RMAPI variables and add them to the client session.
 * The initial value for IBM.PSSP.SampleCmdMon.state is
 * TRANSITION. If the state changed, the new state value
 * will then be sent by the RMAPI routine ha_rr_send_val().
 * If the state did not change, IBM.PSSP.SampleCmdMon will
 * not be added.
 */
Variables[0].rr_varu.rr_var_hndl = &(Values[0].rr_var_hndl);
Variables[0].rr_value = create_sbs(action,inst_opts,(new_state==curr_state),new_state);
Variables[1].rr_varu.rr_var_hndl = &(Values[1].rr_var_hndl);
Variables[1].rr_value = &(trans_state);

/*
 * Add the variable(s) to the RMAPI. This will also cause the initial
 * values (Variables[n].rr_value) to be sent to the Event Manager.
 */
rr_add_var();

if (new_state != curr_state) {
    /*
     * State change occurred. ha_rr_add_var() sent the initial value for
     * the "state" variable as being in TRANSITION state. Now call
     * the RMAPI to send the new state of the resource. First check that
     * an error didn't occur when register or adding the variable.
     */
    if ((NumVariables == 2) && (Variables[1].rr_var_errno == 0)) {
        /*
         * Sleep to look like were actually doing something.
         */
        sleep(2);

        /*
         * Send the new state value of variable IBM.PSSP.SampleCmdMon.state.
         */
        Values[1].rr_value = &(new_state);
        rc = ha_rr_send_val(&Values[1],1,0,&ErrBlock);
        if (rc == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
        }
    }
}

/*
 * Delete the variable(s) that were registered and added.
 */
rr_del_var();

/*
 * Close the EM session and terminate the RMAPI.
 */
```

```
    rr_end();

    /*
     * Exit return code.
     */
    exit (new_state==curr_state);
}

/*--------------------------------------------------------------------------------
 * Resource "processing" routine. Returns the new state if the action
 * is valid for the current state, otherwise returns the current state.
 * Legal state transitions:
 *    Unconfigured -> Configured    (config)
 *    Configured   -> Unconfigured  (unconfig)
 *    Configured   -> Running       (start)
 *    Running      -> Configured    (stop)
 --------------------------------------------------------------------------------*/
enum States process_action(enum States state, enum Actions action) {
enum States new_state = state;

    switch (state) {
        case CONFIGURED :
            switch (action) {
                case CONFIG   :
                case STOP     :
                    break;
                case UNCONFIG :
                    new_state = UNCONFIGURED; break;
                case START    :
                    new_state = RUNNING; break;
            }
            break;
        case UNCONFIGURED :
            switch (action) {
                case UNCONFIG :
                case START    :
                case STOP     :
                    break;
                case CONFIG   :
                    new_state = CONFIGURED; break;
            }
            break;
        case RUNNING :
            switch (action) {
                case CONFIG   :
                case UNCONFIG :
                case START    :
                    break;
                case STOP     :
                    new_state = CONFIGURED; break;
            }
            break;
    }
    return new_state;
}

/*--------------------------------------------------------------------------------
 * Creates an SBS variable value for the call variable.
```

```
 ------------------------------------------------------------------------------*/
void * create_sbs(enum Actions action, char *opts, long state_chg, enum States state) {
int buff_sz;        /* size of the buffer needed for the sbs value. */
int slen;           /* length of the opts sbs field. */
char *sbs, *p;      /* sbs buffer pointers. */

    /*
     * Get the length of the "opts" field - including terminating char.
     */
    slen = strlen(opts) + 1;

    /*
     * Calc the size of the buffer needed:
     * length of the opts field +
     * length of the 3 long fields +
     * 1 long for each sbs field header.
     */
    buff_sz = slen + (sizeof(long) * 7);

    /*
     * Allocate an sbs buffer and prefix the buffer size.
     */
    sbs = p = malloc(buff_sz);
    *((long *)p) = buff_sz;  p += sizeof(long);

    /*
     * Copy the first (Action) sbs field. The field prefixes are a
     * short field length, 1 char for the data type and 1
     * char for the field serial number.
     */
    *((short *)p) = (short)sizeof(long); p += sizeof(short);
    *p = (char)ha_emFTlong; p++;
    *p = 0; p++;
    *((long *)p) = (long)action; p += sizeof(long);

    /*
     * Copy the second (Options) field (sbs serial number 1).
     */
    *((short *)p) = (short)slen; p += sizeof(short);
    *p = (char)ha_emFTchar; p++;
    *p = 1; p++;
    strcpy(p,opts); p += slen;

    /*
     * Copy the third (StateChange) field (sbs serial number 2).
     */
    *((short *)p) = (short)sizeof(long); p += sizeof(short);
    *p = (char)ha_emFTlong; p++;
    *p = 2; p++;
    *((long *)p) = (long)state_chg; p += sizeof(long);

    /*
     * Copy the forth (State) field (sbs serial number 3).
     */
    *((short *)p) = (short)sizeof(long); p += sizeof(short);
    *p = (char)ha_emFTlong; p++;
    *p = 3; p++;
    *((long *)p) = (long)state; p += sizeof(long);
```

```
    /*
     * Return the pointer to the sbs value.
     */
    return sbs;
}

/*--------------------------------------------------------------------------------
 * Initializes the RMAPI and starts the Client session with haemd.
 --------------------------------------------------------------------------------*/
void rr_start() {
int rc;

    /*
     * Initialize the RMAPI.
     */
    rc = ha_rr_init(RESOURCE_MONITOR_NAME, &ErrBlock);
    if (rc == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
    } else {
        /*
         * Create a client session with the Event Manager daemon.
         */
        ClientSock = ha_rr_start_session(HA_RR_NOTIFY_SELECT, &ErrBlock);
        if (ClientSock == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
        }
    }
    if ((rc == HA_RR_FAIL) || (ClientSock == HA_RR_FAIL)) {
        /*
         * Init or start session failed. Set the number of variables
         * to 0 so that reg, add, send and delete are ignored.
         */
        NumVariables = 0;
    }
}

/*--------------------------------------------------------------------------------
 * Registers the variables to be added with the RMAPI.
 --------------------------------------------------------------------------------*/
void rr_reg_var() {
int i, num_registered;

    if (NumVariables) {
        num_registered = ha_rr_reg_var(Variables,NumVariables,&ErrBlock);
        if (num_registered != NumVariables) {
            if (num_registered == HA_RR_FAIL) {
                display_rmapi_err(&ErrBlock);
            } else {
                for (i=0; i < NumVariables; i++) {
                    /*
                     * Loop through the variables that were attempted
                     * to be registered and report any the errors.
                      */
                    if (Variables[i].rr_var_errno) {
                        fprintf(stderr,"ha_rr_reg_var() error with variable(%s) ivector(%s)"
                                " instid(%d) RMAPI errno=(%d).\n",
                                Variables[i].rr_var_name,
```

```
                                Variables[i].rr_var_ivector,
                                Variables[i].rr_varu.rr_var_inst_id,
                                Variables[i].rr_var_errno);
                        }
                    }
                }
            }
            if ((num_registered == HA_RR_FAIL) || (num_registered == 0)) {
                /*
                 * Register failed - set NumVariables to 0
                 * so that ha_rr_add_var() isn't attempted.
                 */
                NumVariables = 0;
            }
        } else {
            fprintf(stdout,"%s: No variables to register.\n",CMDNAME);
        }
}

/*-------------------------------------------------------------------------------
 * Adds variables to the Client session with haemd.
 -------------------------------------------------------------------------------*/
void rr_add_var() {
int i, num_added;

    if (NumVariables) {
        num_added = ha_rr_add_var(ClientSock, Variables, NumVariables, 1, &ErrBlock);
        if (num_added != NumVariables) {
            if (num_added == HA_RR_FAIL) {
                display_rmapi_err(&ErrBlock);
            } else {
                /*
                 * Loop through the variables that were attempted
                 * to be added and report the errors.
                 */
                for (i=0; i < NumVariables; i++) {
                    if (Variables[i].rr_var_errno != 0) {
                        fprintf(stderr,"Variable(%s) ivec(%s) "
                        "bad errno(%d) from ha_rr_add_var().\n",
                                Variables[i].rr_var_name,
                                Variables[i].rr_var_ivector,
                                Variables[i].rr_var_errno);
                    }
                }
            }
        }
        if ((num_added == HA_RR_FAIL) || (num_added == 0)) {
            /*
             * Add failed - set NumVariables to 0 so that ha_rr_send_var()
             * and ha_rr_del_var() are not called.
             */
            NumVariables = 0;
        }
    } else {
        fprintf(stdout,"%s: No variables to add.\n",CMDNAME);
    }
}
```

rmapi_smpcmd.c

```
/*--------------------------------------------------------------------------------
 * Deletes any variables added to the RMAPI for the Client session with haemd.
 --------------------------------------------------------------------------------*/
void rr_del_var() {
int num_deleted;

    if (NumVariables) {
        /*
         * Delete any variables that have been added.
         */
        num_deleted = ha_rr_del_var(ClientSock, Variables, NumVariables, &ErrBlock);
        if (num_deleted == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
        }
    } else {
        fprintf(stdout,"%s: No variables to delete.\n",CMDNAME);
    }
}

/*--------------------------------------------------------------------------------
 * Ends the Client session with haemd (if started) and terminates the RMAPI.
 --------------------------------------------------------------------------------*/
void rr_end() {
int rc;

    if (ClientSock != HA_RR_FAIL) {
        /*
         * Call the RMAPI to close the client session with the Event Manager.
         */
        rc = ha_rr_end_session(ClientSock, &ErrBlock);
        if (rc == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
        }
    }

    /*
     * Terminate the RMAPI.
     */
    rc = ha_rr_terminate(&ErrBlock);
    if (rc && ErrBlock.em_errno != HA_RR_EACCESS) {
        display_rmapi_err(&ErrBlock);
    }
}

/*--------------------------------------------------------------------------------
 * Displays an RMAPI error.
 --------------------------------------------------------------------------------*/
void display_rmapi_err(struct ha_em_err_blk *errblk) {
  fprintf(stderr,
          "RMAPI Error: File(%s) Version(%s) Line(%d) Errno(%d)\n\t%s",
          errblk->em_errfile,
          errblk->em_errlevel,
          errblk->em_errline,
          errblk->em_errno,
          errblk->em_errmsg);
 }
```

The rmapi_smpdae.c Sample Program

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/select.h>
#include <sys/signal.h>
#include <sys/events.h>

#include <ha_rmapi.h>

/*
 *  rmapi_smpdae.c
 *
 *  This program presents an example of using the Resource Monitor Application
 *  Programming Interface (RMAPI).  The configuration data for this monitor
 *  is in the file RmapiSample.loadsdr.  This monitor is an example of server
 *  (Event Management daemon connects to the monitor) monitor function that could
 *  be incorporated into a daemon or subsystem.
 *
 *  Event Management objects defined for this monitor:
 *
 *  Monitor:
 *      IBM.PSSP.SampleDaeMon                     # server monitor definition
 *
 *  Class:
 *      IBM.PSSP.SampleDaeClass                   # variable class
 *
 *  Variables:
 *      IBM.PSSP.SampleDaeMon.StaticVars.static_var1    # valuetype quantity, datatype long
 *          Instance Vectors:  NodeNum                  # used as locator field of var
 *      IBM.PSSP.SampleDaeMon.StaticVars.static_var2    # valuetype quantity, datatype long
 *          Instance Vectors:  NodeNum                  # used as locator field of var
 *      IBM.PSSP.SampleDaeMon.StaticVars.static_var3    # valuetype quantity, datatype long
 *          Instance Vectors:  NodeNum                  # used as locator field of var
 *      IBM.PSSP.SampleDaeMon.InstVars.inst_var1        # valuetype quantity, datatype long
 *          Instance Vectors:  NodeNum                  # used as locator field of var
 *                             Name                     # instance name of variable
 *      IBM.PSSP.SampleDaeMon.InstVars.inst_var2        # valuetype quantity, datatype long
 *          Instance Vectors:  NodeNum                  # used as locator field of var
 *                             Name                     # instance name of variable
 *
 * What this monitor does:  rmapi_smpdae provides an example of a server monitor
 *   which has 3 non-instantiable and 2 instantiable Quantity variables which
 *   are updated with random values on a fixed interval defined in the variable class.
 *   When the monitor is started, it creates and registers the static and 1 set of the
 *   instantiable variables.  To further demonstrate instantiation, new instances are
 *   created and registered on the 5th and 12th call to the function send_values().
 *
 *   The monitor will continue executing, adding/deleting and sending values for the
 *   variables as requested by control messages sent from the Event Management daemon
 *   through the RMAPI.  This monitor uses select notification protocol to determine when a
 *   connection is ready to accept on the server socket created by ha_rr_makserv() or
 *   when a message is ready to ge read though the RMAPI from one of the monitor's
```

rmapi_smpdae.c

```
 *   client sessions.
 *
 *   The monitor ends execution when it receives a terminating signal, or when all
 *   all clients have closed their connections.
 *
 *  This program can be compiled with the following command:
 *
 *      cc -O rmapi_smpdae.c -o rmapi_smpdae -lha_rr
 */

/* local defines
PROGNAME                - Name of this program.
MAX_INTERVAL            - Max number of seconds before ha_rr_send_value
                          or ha_rr_touch must be called.
RESOURCE_MONITOR_NAME   - Name of this monitor as defined in the config data.
MONITOR_CLASS_NAME      - Class name as defined in the config data.
SOCKET_TABLE_SIZE       - HA_RR_MAX_SESSIONS + 1 for the RMAPI server socket.
RMAPI_SERVER_INDEX      - Index in the socket call table for the RMAPI server socket.
STATIC_VAR_NAME_PREFIX  - Common portion of the static vars as defined in the config data.
INST_VAR_NAME_PREFIX    - Common portion of the inst vars as defined in the config data.
INST_VAR_IVEC_FORMAT    - Ivector format for inst vars.
INITIAL_VALUE           - Variable initial value used on ha_rr_add_var().
MAX_VALUE               - Maximum variable value.
NUM_STATIC_VARS         - Number of static variables defined for monitor.
NUM_INST_VARS           - Number of instantiable variables defined for monitor.
*/
#define PROGNAME                "rmapi_smpdae"
#define MAX_INTERVAL            500
#define RESOURCE_MONITOR_NAME   "IBM.PSSP.SampleDaeMon"
#define MONITOR_CLASS_NAME      "IBM.PSSP.SampleDaeClass"
#define SOCKET_TABLE_SIZE       (HA_RR_MAX_SESSIONS + 1)
#define RMAPI_SERVER_INDEX      (HA_RR_MAX_SESSIONS)
#define STATIC_VAR_NAME_PREFIX  "IBM.PSSP.SampleDaeMon.StaticVars.static_var"
#define INST_VAR_NAME_PREFIX    "IBM.PSSP.SampleDaeMon.InstVars.inst_var"
#define INST_VAR_IVEC_FORMAT    "InstName=InstSet%d"
#define INITIAL_VALUE           10
#define MAX_VALUE               500
#define NUM_STATIC_VARS         3
#define NUM_INST_VARS           2

/* local structs */
struct sock_table_entry {
    int  socket_fd;             /* fd for manager session or RMAPI server socket. */
    void (*sock_funcp)(int);    /* function to be called for this socket fd. */
};

struct local_vars {
    char var_name[128];          /* Variable name (as in the CDB). */
    char var_ivec[32];           /* Qualified ivector (for instantiable variables). */
    long value;                  /* variables value. */
    void *var_handle;            /* place for RMAPI to store the var handle. */
};

/* local functions */
void display_rmapi_err(struct ha_em_err_blk *errblk);
void server_socket_handler(int table_index);
void session_socket_handler(int table_index);
void end_session(int table_index);
```

```
void register_variables();
void add_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg);
void del_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg);
void send_values();
void refresh_values(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg);
void control_loop();
void set_signals();
void catch_alrm_signal(int);
void catch_exit_signal(int);
void mon_exit(int);

/* globals */
int                     NumMgrs = 0;             /* num of resrc managers connected. */
struct ha_rr_variable   *Variables = (struct ha_rr_variable *)0;  /* RMAPI var array. */
struct ha_rr_val        *Values = (struct ha_rr_val *)0;          /* RMAPI val array. */
struct sock_table_entry SocketTable[SOCKET_TABLE_SIZE];   /* RMAPI server */
                                                            /*and session sockets*/
struct ha_em_err_blk    ErrBlock;                   /* global err blk for RMAPI calls */
struct local_vars       *LocalVars = (struct local_vars *)0;  /* array of local vars */
static sigset_t         SignalMask;                           /* mask for signals. */
timer_t                 IntervalTimerId;          /* ID for var update interval timer. */
int                     Interval;                 /* Update interval for this monitor. */
int                     SampleTime = 0;           /* flag set by interval alrm sig. */
int                     Terminate  = 0;           /* flag set by terminate signal. */
int                     SigCaught = 0;            /* signal caught by term. */
int                     TimeOut = MAX_INTERVAL;   /* time out counter to call ha_rr_touch. */
int                     NumVariables = 0;         /* Number of variables in arrays. */

/*****************************************************************************/
/* Main - Main - Main - Main - Main - Main - Main - Main - Main - Main - Main */
/*****************************************************************************/
main(int argc, char **argv)
{
int i, rc;
struct itimerstruc_t itimer;

    fprintf(stdout,"%s: %s resource monitor started.\n",PROGNAME,RESOURCE_MONITOR_NAME);

    /*
     * Setup siganl handlers.
     */
    set_signals();

    /*
     * Initialize the socket table. Socket file descriptors are inited to -1.
     */
    for (i=0;i<SOCKET_TABLE_SIZE;i++) {
        SocketTable[i].socket_fd = -1;
        if (i == RMAPI_SERVER_INDEX) {
            SocketTable[i].sock_funcp = server_socket_handler;
        } else {
            SocketTable[i].sock_funcp = session_socket_handler;
        }
    }

    /*
     * Initialize the RMAPI.
     */
```

rmapi_smpdae.c

```
    rc = ha_rr_init(RESOURCE_MONITOR_NAME, &ErrBlock);
    if (rc == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        mon_exit(1);
    }

    /*
     * Query the update interval from the RMAPI.
     */
    Interval = ha_rr_get_interval(MONITOR_CLASS_NAME,&ErrBlock);
    if (Interval == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        mon_exit(1);
    }
    fprintf(stdout,"Interval for class %s is %d.\n",MONITOR_CLASS_NAME,Interval);

    /*
     * Register variables with the RMAPI
     */
    register_variables();

    /*
     * Make this Resource Monitor a server.
     */
    rc = SocketTable[RMAPI_SERVER_INDEX].socket_fd = ha_rr_makserv(HA_RR_NOTIFY_SELECT, &ErrBlock);
    if (rc == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        mon_exit(1);
    }
    fprintf(stdout,"Assigned RMAPI server socket_fd(%d) to table index(%d).\n",
                rc,RMAPI_SERVER_INDEX);

    /*
     * Set up sampling timer.
     */
    IntervalTimerId = gettimerid(TIMERID_REAL, DELIVERY_SIGNALS);
    itimer.it_value.tv_sec = Interval;
    itimer.it_value.tv_nsec = 0;
    itimer.it_interval.tv_sec = Interval;
    itimer.it_interval.tv_nsec = 0;

    if (incinterval(IntervalTimerId, &itimer, (struct itimerstruc_t *)0) < 0) {
        fprintf(stderr,"incinterval() failed with errno=%d.\n",errno);
        mon_exit(1);
    }

    /*
     * Start the control loop - should never return.
     */
    control_loop();

    /*
     * Should never get here.
     */
    mon_exit(0);
}

/*
```

```
 * Sets the signal handlers.
 */
void set_signals() {
struct sigaction sigactn;

    sigemptyset(&SignalMask);
    sigaddset(&SignalMask, SIGALRM);
    sigaddset(&SignalMask, SIGTERM);

    /*
     * Alrm signal caught on the interval boundary.
     * Handler sets a global flag to do sampling.
     */
    sigactn.sa_handler = catch_alrm_signal;
    sigactn.sa_mask = SignalMask;
    sigactn.sa_flags = 0;

    if (sigaction(SIGALRM, &sigactn, (struct sigaction *)0) < 0) {
        fprintf(stderr,"sigaction() failed for ALRM signal. errno=%d.\n",errno);
        mon_exit(1);
    }

    /*
     * Need terminate handler to cleanup RMAPI on exit.
     */
    sigactn.sa_handler = catch_exit_signal;
    sigactn.sa_mask = SignalMask;
    sigactn.sa_flags = 0;

    if (sigaction(SIGTERM, &sigactn, (struct sigaction *)0) < 0) {
        fprintf(stderr,"sigaction() failed for TERM signal. errno=%d.\n",errno);
        mon_exit(1);
    }

    /*
     * Set signal mask, i.e. block SIGALRM, SIGTERM
     */
    if (sigprocmask(SIG_SETMASK, &SignalMask, (sigset_t *)0) < 0) {
        fprintf(stderr,"sigprocmask() failed. errno=%d.\n",errno);
        mon_exit(1);
    }
}

/*
 * Sends new values to the RMAPI - called when sig alrm is caught.
 */
void send_values() {
int i, rc;
int num_vals_to_send = 0;
static int times_called = 0;

    times_called++;
    for (i=0;i<NumVariables;i++) {

        /*
         * Give the variable a new random value.
         */
        LocalVars[i].value = (random() % MAX_VALUE);
```

```
        /*
         * Only need to send values for vars with non-NULL handles.
         */
        if (LocalVars[i].var_handle != (void *)0) {

            /*
             * Copy the handle and value to the next available RMAPI struct.
             */
            Values[num_vals_to_send].rr_var_hndl = LocalVars[i].var_handle;
            Values[num_vals_to_send].rr_value = &(LocalVars[i].value);

            /*
             * Increment the number to send count.
             */
            num_vals_to_send++;
        }
    }

    if (num_vals_to_send) {
        /*
         * Send the Value array to the RMAPI.
         */
        rc = ha_rr_send_val(Values,num_vals_to_send,0,&ErrBlock);
        if (rc == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }

        /*
         * Reset the TimeOut counter.
         */
        TimeOut = MAX_INTERVAL;

    } else if ((TimeOut -= Interval) <= 0) {

        /*
         * No values have been sent in the TimeOut period. Need
         * to call ha_rr_touch to update the shm timestamp.
         */
        if (ha_rr_touch(&ErrBlock) == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }

        /*
         * Reset the TimeOut counter.
         */
        TimeOut = MAX_INTERVAL;

    }

    /*
     * Add some new instantiable variables
     * on the 5th and 12th time called.
     */
    if ((times_called == 5) || (times_called == 12)) {
        fprintf(stdout,"Call number %d - calling register_variables() "
```

```
                    "to create new variables.\n",
                times_called);
        register_variables();
    }
}

/*
 * Catches alrm sig and sets global flag SampleTime to send values.
 */
void catch_alrm_signal(int sig) {
    SampleTime = 1;
}

/*
 * Catches term sig and sets global flag Terminate to exit.
 */
void catch_exit_signal(int sig) {
    Terminate = 1;
    SigCaught = sig;
}

/*
 * Handles a connection request on the RMAPI server
 * socket returned by ha_rr_makserv().
 */
void server_socket_handler(int table_index) {
int i, mgr_sock, rc;

    mgr_sock = ha_rr_start_session(HA_RR_NOTIFY_SELECT,&ErrBlock);
    if (mgr_sock < 0) {
        display_rmapi_err(&ErrBlock);
        if ((ErrBlock.em_errno != HA_RR_EMAXSESSIONS) &&
                        (ErrBlock.em_errno != HA_RR_EAGAIN)) {
            /*
             * Severe error in the RMAPI - exit.
             */
            mon_exit(1);
        }
        return;
    }
    /*
     * Find an available element in the session socket table.
     */
    for (i=0;(i < HA_RR_MAX_SESSIONS)&&(SocketTable[i].socket_fd >= 0);i++);

    if (i < HA_RR_MAX_SESSIONS) {
        /*
         * Save the mgr socket fd returned by the RMAPI.
         * The socket function was initialized in main.
         */
        SocketTable[i].socket_fd = mgr_sock;
        NumMgrs++;
        fprintf(stdout,"New session accepted to index(%d) socket_fd(%d).\n",i,mgr_sock);
    } else {
        /*
         * Should never get here, RMAPI handles too many sessions.
         */
        rc = ha_rr_end_session(mgr_sock,&ErrBlock);
```

```
    }
}

/*
 * Handles a msg being received from the manager session table_index.
 * Called following select.
 */
void session_socket_handler(int table_index) {
int rc, session_sock;
struct ha_rr_ctrl_msg *ctrl_msg;

    /*
     * Get the socket fd for this session.
     */
    if ((session_sock = SocketTable[table_index].socket_fd) < 0) {
        return;
    }

    /*
     * Call ha_rr_get_ctrl_msg to read the cmd from the manager.
     */
    fprintf(stdout,"Calling ha_rr_get_ctrlmsg for session(%d) socket_fd(%d).\n",
            table_index,session_sock);
    rc = ha_rr_get_ctrlmsg(session_sock, &ctrl_msg, &ErrBlock);

    if (rc == 0) {
        /*
         * Msg was for the RMAPI or was incomplete.
         */
        return;
    }

    if (rc == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        if (ErrBlock.em_errno == HA_RR_EDISCONNECT) {
            end_session(table_index);
        } else if (ErrBlock.em_errno != HA_RR_EAGAIN) {
            /*
             * Severe RMAPI error.
             */
            mon_exit(1);
        }
        return;
    }

    switch (ctrl_msg->rr_ctrl_cmd) {
        case HA_RR_CMD_ADDALL  :
        case HA_RR_CMD_ADDV    :
            add_variables(session_sock,ctrl_msg);
            break;
        case HA_RR_CMD_DELALL  :
        case HA_RR_CMD_DELV    :
            del_variables(session_sock,ctrl_msg);
            break;
        case HA_RR_CMD_REFRESH :
            refresh_values(session_sock,ctrl_msg);
            break;
        default                :
```

```
            /*
             * Unknown or unsupported msg - ignore it.
             */
            fprintf(stderr,"Received an unexpected cmd(%d) from socket_fd(%d), ignoring...\n",
                    ctrl_msg->rr_ctrl_cmd,session_sock);
            break;
    }

    /*
     * Free the msg allocated by the RMAPI.
     */
    free(ctrl_msg);
}

/*
 * Adds the variables referenced in the ctrl_msg to the RMAPI for
 * the manager session sock_fd.
 */
void add_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg) {
int i;
int inst_id;
int num_added, num_to_add = 0;

    switch (ctrl_msg->rr_ctrl_cmd) {
        case HA_RR_CMD_ADDALL :
            /*
             * Add all variables to this manager session.
             */
            fprintf(stdout,"Processing cmd HA_RR_CMD_ADDALL "
            "to add all vars to session_fd %d.\n",sock_fd);
            for (i=0;i<NumVariables;i++) {
                /*
                 * Copy the var to the next RMAPI var struct.
                 */
                Variables[num_to_add].rr_var_name = LocalVars[i].var_name;
                Variables[num_to_add].rr_var_ivector = LocalVars[i].var_ivec;
                Variables[num_to_add].rr_varu.rr_var_hndl = &(LocalVars[i].var_handle);
                Variables[num_to_add].rr_value = &(LocalVars[i].value);

                /*
                 * Increment the count of variables to be added.
                 */
                num_to_add++;
            }
            break;
        case HA_RR_CMD_ADDV :
            /*
             * Add a vector of variables to the session.
             */
            fprintf(stdout,"Processing cmd HA_RR_CMD_ADDV "
                    "to add %d vars for session_fd %d.\n",
                    ctrl_msg->rr_ctrl_num_vars,sock_fd);
            for (i=0;i < ctrl_msg->rr_ctrl_num_vars;i++) {

                /*
                 * The inst id field in the ctrl msg is
                 * the index into the LocalVars array.
                 */
```

rmapi_smpdae.c

```
                inst_id =  ctrl_msg->rr_ctrlv.rr_ctrl_vari[i];

                if (inst_id < NumVariables) {
                    /*
                     * Copy the var to the next RMAPI var struct.
                     */
                    Variables[num_to_add].rr_var_name =
                            LocalVars[inst_id].var_name;
                    Variables[num_to_add].rr_var_ivector =
                            LocalVars[inst_id].var_ivec;
                    Variables[num_to_add].rr_varu.rr_var_hndl =
                            &(LocalVars[inst_id].var_handle);
                    Variables[num_to_add].rr_value =
                            &(LocalVars[inst_id].value);

                    num_to_add++;
                }
            }
            break;
        default :
            break;
    }
    if (num_to_add) {
        /*
         * Add the variables by calling ha_rr_add_var.
         */
        num_added = ha_rr_add_var(sock_fd, Variables, num_to_add, 1, &ErrBlock);

        if (num_added == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }

        fprintf(stdout,"ha_rr_add_var() added %d vars to session_fd %d.\n",num_added,sock_fd);
        if (num_to_add != num_added) {
            /*
             * Loop through the variables that were attempted
             * to be added and report any that had errors.
             */
            for (i=0;i<num_to_add;i++) {
                if (Variables[i].rr_var_errno != 0) {
                    fprintf(stderr,"Variable(%s) ivec(%s) had bad errno(%d) "
                    "from ha_rr_add_var().\n",

Variables[i].rr_var_name,Variables[i].rr_var_ivector,Variables[i“.rr_var_errno);
                }
            }
        }
    }
}

/*
 * Deletes the variables referenced in the ctrl_msg for the mgr
 * sock_fd session.
 */
void del_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg) {
int i;
int ctrl_cmd;
```

```
int inst_id;
int num_deleted, num_to_del = 0;

    if (ctrl_msg == (struct ha_rr_ctrl_msg *)0) {
        /*
         * Called with NULL msg when a session is ending. Need to delete
         * all vars from the session before calling ha_rr_end_session.
         */
        ctrl_cmd = HA_RR_CMD_DELALL;
    } else {
        ctrl_cmd = ctrl_msg->rr_ctrl_cmd;
    }

    switch (ctrl_cmd) {
        case HA_RR_CMD_DELALL :
            fprintf(stdout,"Processing cmd HA_RR_CMD_DELALL "
            "to delete all vars for session_fd %d.\n",
                    sock_fd);
            /*
             * Delete all variables for this manager session.
             */
            for (i=0;i<NumVariables;i++) {
                if (LocalVars[i].var_handle != (void *)0) {
                    /*
                     * Copy the var to the next RMAPI var struct.
                     */
                    Variables[num_to_del].rr_var_name = LocalVars[i].var_name;
                    Variables[num_to_del].rr_var_ivector = LocalVars[i].var_ivec;
                    Variables[num_to_del].rr_varu.rr_var_hndl = &(LocalVars[i].var_handle);

                    /*
                     * Increment the count of variables to be deleted.
                     */
                    num_to_del++;
                }
            }
            break;
        case HA_RR_CMD_DELV :
            fprintf(stdout,"Processing cmd HA_RR_CMD_DELV "
                    "to delete %d vars for session_fd %d.\n",
                    ctrl_msg->rr_ctrl_num_vars,sock_fd);
            /*
             * Delete a vector of variables for this manager session.
             */
            for (i = 0; i < ctrl_msg->rr_ctrl_num_vars; i++) {
                inst_id = ctrl_msg->rr_ctrlv.rr_ctrl_vari[i];
                if ((inst_id < NumVariables) &&
                    (LocalVars[inst_id].var_handle != (void *)0)) {

                    /*
                     * Copy the var to the next RMAPI var struct.
                     */
                    Variables[num_to_del].rr_var_name = LocalVars[inst_id].var_name;
                    Variables[num_to_del].rr_var_ivector = LocalVars[inst_id].var_ivec;
                    Variables[num_to_del].rr_varu.rr_var_hndl =
                              &(LocalVars[inst_id].var_handle);

                    /*
```

```
                    * Increment the count of variables to be deleted.
                    */
                   num_to_del++;
               }
           }
           break;
       default :
           break;
   }
   if (num_to_del) {
       /*
        * Delete the variables by calling ha_rr_del_var.
        */
       num_deleted = ha_rr_del_var(sock_fd, Variables, num_to_del, &ErrBlock);

       if (num_deleted == HA_RR_FAIL) {
           display_rmapi_err(&ErrBlock);
           mon_exit(1);
       }

       /*
        * rc>0 from ha_rr_del_var is the number of variables that no longer need
        * their values updated. The RMAPI will have set their handles to NULL.
        */
       fprintf(stdout,"%d variables no longer need to be updated.\n",num_deleted);
   }
}

/*
 * Updates values in response to a manager refresh command.
 */
void refresh_values(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg) {
int i, rc;
int inst_id;
int num_to_send = 0;

   fprintf(stdout,"Processing a refresh request of %d variables "
           "for session_fd %d.\n",
           ctrl_msg->rr_ctrl_num_vars,sock_fd);
   for (i = 0; i < ctrl_msg->rr_ctrl_num_vars; i++) {
       inst_id =  ctrl_msg->rr_ctrlv.rr_ctrl_vari[i];
       if ((inst_id < NumVariables) &&
           (LocalVars[inst_id].var_handle)) {

           /*
            * Give the variable a new bogus value.
            */
           LocalVars[inst_id].value = (random() % MAX_VALUE);

           /*
            * Copy the var handle and value to the RMAPI struct.
            */
           Values[num_to_send].rr_var_hndl = LocalVars[inst_id].var_handle;
           Values[num_to_send].rr_value = &(LocalVars[inst_id].value);
           num_to_send++;
       }
   }
   if (num_to_send) {
```

```
        /*
         * Call RMAPI with the refresh flag to send the values.
         */
        rc = ha_rr_send_val(Values, num_to_send, 1, &ErrBlock);
        if (rc) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }
        /*
         * Reset the TimeOut counter.
         */
        TimeOut = MAX_INTERVAL;
    }
}

/*
 * Ends the manager session indexed by the table_index parm
 * in the socket table.
 */
void end_session(int table_index) {
int rc;

    fprintf(stdout,"Ending session index(%d) session_fd(%d).\n",
                table_index,SocketTable[table_index].socket_fd);

    /*
     * Make sure this is a valid session.
     */
    if ((table_index < 0) || (table_index > HA_RR_MAX_SESSIONS) ||
        (SocketTable[table_index].socket_fd < 0)) {
        return;
    }

    /*
     * Delete all variables for this manager.
     */
    del_variables(SocketTable[table_index].socket_fd, (struct ha_rr_ctrl_msg *)0);

    /*
     * RMAPI call to close the session.
     */
    rc = ha_rr_end_session(SocketTable[table_index].socket_fd, &ErrBlock);
    if (rc == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        mon_exit(1);
    }

    /*
     * Reset the fd in the SocketTable and decrement the mgr count.
     */
    SocketTable[table_index].socket_fd = -1;
    NumMgrs--;

    if (NumMgrs <= 0) {
        /*
         * This was the last mgr to end - exit with a good rc.
         */
        fprintf(stdout,"Last manager session closed, exiting...\n");
```

```
        mon_exit(0);
    }
}

/*
 * Registers variables with the RMAPI.
 */
void register_variables() {
int i, j, num_var;
int num_registered, num_to_reg = 0;
static int times_called = 0;

    fprintf(stdout,"Call number %d to register_variables().\n",times_called);
    times_called++;
    if (times_called == 1) {
        fprintf(stdout,"Creating %d new variables.\n",NUM_STATIC_VARS + NUM_INST_VARS);
        /*
         * First time called allocate the LocalVars, Variables and
         * Value arrays for the StaticVars and 1 each of InstVars.
         */
        NumVariables = NUM_STATIC_VARS + NUM_INST_VARS;

        LocalVars =
            (struct local_vars *)malloc(sizeof(struct local_vars) * NumVariables);
        Variables =
            (struct ha_rr_variable *)malloc(sizeof(struct ha_rr_variable) * NumVariables);
        Values =
            (struct ha_rr_val *)malloc(sizeof(struct ha_rr_val) * NumVariables);

        /*
         * Initialize the arrays.
         */
        memset(LocalVars,0,sizeof(struct local_vars) * NumVariables);
        memset(Variables,0,sizeof(struct ha_rr_variable) * NumVariables);
        memset(Values,0,sizeof(struct ha_rr_val) * NumVariables);

        /*
         * Initialize the local variable structs and copy the name, ivector
         * and value ointers to the RMAPI variables for registration.
         */
        for (i=0;i<NUM_STATIC_VARS;i++) {

            /*
             * Create the non-instantiable variable names. The ivectors are NULL
             * strings because they are not instantiable. The only ivector, NodeNum,
             * defined for these variables is used as the locator field and does not
             * need to be sent to the Event Manager. The index of the variable in
             * the local array is used as the inst_id (Variables[n].rr_var_iid). This
             * id will be used by Event Manager in ctrl msgs to identify variables.
             */
            sprintf(LocalVars[num_to_reg].var_name,"%s%d",STATIC_VAR_NAME_PREFIX,i+1);
            LocalVars[num_to_reg].var_ivec[0] = '\0';
            LocalVars[num_to_reg].value = INITIAL_VALUE;

            Variables[num_to_reg].rr_var_name = LocalVars[num_to_reg].var_name;
            Variables[num_to_reg].rr_var_ivector = LocalVars[num_to_reg].var_ivec;
            Variables[num_to_reg].rr_var_iid = num_to_reg;
            Variables[num_to_reg].rr_value = &(LocalVars[num_to_reg].value);
```

```
        num_to_reg++;
    }
    for (i=0;i<NUM_INST_VARS;i++) {

        /*
         * Initialize the instantiable variables. The difference here is that
         * they have a instance vector that needs to be supplied. The
         * instance vector will be InstSet<n> where <n> is the number of the
         * call to this routine.
         */
        sprintf(LocalVars[num_to_reg].var_name,"%s%d",INST_VAR_NAME_PREFIX,i+1);
        sprintf(LocalVars[num_to_reg].var_ivec,INST_VAR_IVEC_FORMAT,times_called);
        LocalVars[num_to_reg].value = INITIAL_VALUE;

        Variables[num_to_reg].rr_var_name = LocalVars[num_to_reg].var_name;
        Variables[num_to_reg].rr_var_ivector = LocalVars[num_to_reg].var_ivec;
        Variables[num_to_reg].rr_var_iid = num_to_reg;
        Variables[num_to_reg].rr_value = &(LocalVars[num_to_reg].value);

        num_to_reg++;
    }
} else {
    fprintf(stdout,"Creating %d new variables.\n",NUM_INST_VARS);
    /*
     * Subsequent call to this routine. This is an example of new instantations
     * being created during normal execution. A new set of instantiable variables
     * will be registered using the current number of times this routine was called
     * in the ivector.
     */

    /*
     * Get the new number of variables.
     */
    num_var = NumVariables + NUM_INST_VARS;

    /*
     * Realloc the arrays to the new size and initialize the local array.
     */
    LocalVars =
        (struct local_vars *)realloc(LocalVars, sizeof(struct local_vars) * num_var);
    Variables =
    (struct ha_rr_variable *)realloc(Variables, sizeof(struct ha_rr_variable) * num_var);
    Values =
        (struct ha_rr_val *)realloc(Values, sizeof(struct ha_rr_val) * num_var);
    memset(LocalVars+NumVariables,0,sizeof(struct local_vars) * NUM_INST_VARS);

    for (i=NumVariables,j=0;i<num_var;i++,j++) {

        /*
         * Create a new set of variables as above and copy them to the
         * RMAPI variable struct to be registered.
         */
        sprintf(LocalVars[i].var_name,"%s%d",INST_VAR_NAME_PREFIX,j+1);
        sprintf(LocalVars[i].var_ivec,INST_VAR_IVEC_FORMAT,times_called);
        LocalVars[i].value = INITIAL_VALUE;

        Variables[num_to_reg].rr_var_name = LocalVars[i].var_name;
```

rmapi_smpdae.c

```
            Variables[num_to_reg].rr_var_ivector = LocalVars[i].var_ivec;
            Variables[num_to_reg].rr_var_iid = i;
            Variables[num_to_reg].rr_value = &(LocalVars[i].value);

            num_to_reg++;
        }
        NumVariables = num_var;
    }

    if (num_to_reg) {
        /*
         * Call the RMAPI to register the variables.
         */
        num_registered = ha_rr_reg_var(Variables,num_to_reg,&ErrBlock);
        if (num_registered != num_to_reg) {
            if (num_registered == HA_RR_FAIL) {
                display_rmapi_err(&ErrBlock);
                mon_exit(1);
            }
            fprintf(stdout,"%d variables registered with the RMAPI.\n",
                    num_registered);
            for (i=0; i<num_to_reg; i++) {
                if ((Variables[i].rr_var_errno) || (num_registered == HA_RR_FAIL)) {
                    /*
                     * Display a message for each variable with a registration error.
                     */
                    fprintf(stderr,"ha_rr_reg_var() error with variable(%s) "
                                   "instid(%d) RMAPI errno=(%d).\n",
                                Variables[i].rr_var_name,
                                Variables[i].rr_varu.rr_var_inst_id,
                                Variables[i].rr_var_errno);
                }
            }
        }
    }
}

/*
 * Called once by main to establish the control loop.
 */
void control_loop() {
int i, rc;
int sock;
int CallSockFunctions;
fd_set sockfd_set;

    for(;;) {

        /*
         * Init the socket mask for select.
         */
        FD_ZERO(&sockfd_set);
        for(i=0;i<SOCKET_TABLE_SIZE;i++) {
             if (SocketTable[i].socket_fd >= 0) FD_SET(SocketTable[i].socket_fd,&sockfd_set);
        }
        CallSockFunctions = 0;

                /*
```

```
 * Unblock the term and alrm sigs.
 */
if (sigprocmask(SIG_UNBLOCK, &SignalMask, (sigset_t *)0) < 0) {
    /*
     * sigprocmask sys call failed.
     */
    fprintf(stderr,"Cannot unblock signals. sigprocmask() errno=%d.\n",errno);
    mon_exit(1);
}

/*
 * Only select if not terminating or alrm caught to sample values.
 */
if (!(SampleTime || Terminate)) {
    sock = select(FD_SETSIZE, &sockfd_set, NULL, NULL, NULL);
    if (sock > 0) {
        CallSockFunctions = 1;
                } else if (sock < 0) {
        /*
         * select call failed - exit if not an interupt.
         */
        if (errno != EINTR) {
            fprintf(stderr,"select() failed with errno=%d.\n",errno);
            mon_exit(1);
        }
        continue;
    }
}

/*
 * Block the term and alrm sigs.
 */
if (sigprocmask(SIG_BLOCK, &SignalMask, (sigset_t *)0) < 0) {
    fprintf(stderr,"Cannot block signals. sigprocmask() errno=%d.\n",errno);
    mon_exit(1);
}

if (Terminate) {
    /*
     * sig term caught - call mon_exit to clean up.
     */
    fprintf(stdout,"Caught signal(%d)...calling exit...\n",SigCaught);
    mon_exit(SigCaught);
}

        if (CallSockFunctions) {
    /*
     * Call the socket function for each socket returned by select.
     */
    for (i=0;i<SOCKET_TABLE_SIZE;i++) {
        if (SocketTable[i].socket_fd >=0 &&
            FD_ISSET(SocketTable[i].socket_fd,&sockfd_set)) {
            fprintf(stdout,"Calling socket function for session(%d) "
                        "socket_fd(%d).\n",
                        i,SocketTable[i].socket_fd);
            (SocketTable[i].sock_funcp)(i);
        }
    }
```

```
        }

                if (SampleTime) {
            /*
             * sig alrm caught - call send_values
             * and reset the sample flag.
             */
            send_values();
            SampleTime = 0;
        }
    }
}

/*
 * Routine to terminate the monitor.
 */
void mon_exit(int s)
{
static int recursively_called = 0;
int i, rc;

    /*
     * Check for recursive call - some routines mon_exit()
     * calls can likewise call mon_exit().
     */
    if (recursively_called) return;
    recursively_called = 1;

    /*
     * Gracefully close all sessions.
     */
    for (i=0;i<HA_RR_MAX_SESSIONS;i++) {
        if (SocketTable[i].socket_fd >= 0) {
            end_session(i);
        }
    }

    /*
     * Terminate the RMAPI.
     */
    rc = ha_rr_terminate(&ErrBlock);
    if (rc) {
        display_rmapi_err(&ErrBlock);
    }
    fprintf(stdout,"%s: %s resource monitor exiting.\n",PROGNAME,RESOURCE_MONITOR_NAME);
    exit(s);
}

/*
 * Display an RMAPI error.
 */
void display_rmapi_err(struct ha_em_err_blk *errblk)
{
    fprintf(stderr,
        "RMAPI Error: File(%s) Version(%s) Line(%d) Errno(%d)\n\t%s",
        errblk->em_errfile,
        errblk->em_errlevel,
        errblk->em_errline,
```

```
        errblk->em_errno,
        errblk->em_errmsg);
}
```

The rmapi_smpsig.c Sample Program

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/events.h>
#include <ha_rmapi.h>

/*
 *  rmapi_smpsig.c
 *
 *  This program presents an example of using the Resource Monitor Application
 *  Programming Interface (RMAPI).  The configuration data for this monitor
 *  is in the file RmapiSample.loadsdr.  This monitor is an example of server
 *  (Event Management daemon connects to the monitor) monitor function that could
 *  be incorporated into a daemon or subsystem.  This example monitor demonstrates
 *  using SIGIO as the notification protocol and the use of dynamically instantiable
 *  variables.
 *
 *  Event Management objects defined for this monitor:
 *
 *  Monitor:
 *      IBM.PSSP.SampleSigMon                    # server monitor definition
 *
 *  Class:
 *      IBM.PSSP.SampleSigDynInstClass           # class for dynamically instantable vars
 *      IBM.PSSP.SampleSigNonInstClass           # class for non-instantable vars
 *
 *  Variables:
 *      IBM.PSSP.SampleSigMon.DynInstVar.var       # valuetype quantity, datatype long
 *          Instance Vectors:  NodeNum             # used as locator field of var
 *                             InstName            # instance name of variable
 *      IBM.PSSP.SampleSigMon.NonInstVars.var1     # valuetype counter, datatype long
 *          Instance Vectors:  NodeNum             # used as locator field of var
 *      IBM.PSSP.SampleSigMon.NonInstVars.var2     # valuetype counter, datatype long
 *          Instance Vectors:  NodeNum             # used as locator field of var
 *      IBM.PSSP.SampleSigMon.NonInstVars.var3     # valuetype counter, datatype long
 *          Instance Vectors:  NodeNum             # used as locator field of var
 *
 * What this monitor does:  rmapi_smpsig provides an example of a server monitor
 *   which has 1 dynamically instantable variable and 3 non-instantiable variables.
 *
 *   The monitor will continue to run until it either receives a signal to terminate, or
 *   all clients have closed their connections.
 *
 *  This program can be compiled with the following command:
 *
 *      cc -O rmapi_smpsig.c -o rmapi_smpsig -lha_rr
 */
```

```
#ifndef MIN
#define MIN(x, y)   ((x) < (y) ? (x) : (y))
#endif

/* local defines
PROGNAME                - Name of this program.
MAX_INTERVAL            - Max number of seconds before ha_rr_send_value
                          or ha_rr_touch must be called.
RESOURCE_MONITOR_NAME   - Name of this monitor as defined in the config data.
MONITOR_CLASS_NAME      - Class name as defined in the config data.
SOCKET_TABLE_SIZE       - HA_RR_MAX_SESSIONS + 1 for the RMAPI server socket.
RMAPI_SERVER_INDEX      - Index in the socket call table for the RMAPI server socket.
STATIC_VAR_NAME_PREFIX  - Common portion of the static vars as defined in the config data.
INST_VAR_NAME_PREFIX    - Common portion of the inst vars as defined in the config data.
INST_VAR_IVEC_FORMAT    - Ivector format for inst vars.
INITIAL_VALUE           - Variable initial value used on ha_rr_add_var().
MAX_VALUE               - Maximum variable value.
NUM_STATIC_VARS         - Number of static variables defined for monitor.
NUM_INST_VARS           - Number of instantiable variables defined for monitor.
*/
#define PROGNAME                  "rmapi_smpsig"
#define MAX_INTERVAL              500
#define RESOURCE_MONITOR_NAME     "IBM.PSSP.SampleSigMon"
#define DYN_INST_CLASS_NAME       "IBM.PSSP.SampleSigDynInstClass"
#define NON_INST_CLASS_NAME       "IBM.PSSP.SampleSigNonInstClass"
#define NUM_NONINST_VARS          3
#define SOCKET_TABLE_SIZE         (HA_RR_MAX_SESSIONS + 1)
#define RMAPI_SERVER_INDEX        (HA_RR_MAX_SESSIONS)
#define DYN_INST_VAR_NAME         "IBM.PSSP.SampleSigMon.DynInstVar.var"
#define NON_INST_VAR_NAME_PREFIX  "IBM.PSSP.SampleSigMon.NonInstVars.var"
#define DYN_INST_VAR_IVECTOR      "InstName"
#define INITIAL_VALUE             10
#define MAX_VALUE                 500
#define NULL_IVEC                 ""

/* local structs */
struct sock_table_entry {
    int  socket_fd;             /* fd for manager session or RMAPI server socket. */
    void (*sock_funcp)(int);    /* function to be called for this socket fd. */
};

struct local_vars {
    char var_name[128];        /* Variable name (as in config data). */
    char var_ivec[32];         /* Qualified ivector (for inst variables). */
    short registered;          /* flag set when variable is registered. */
    long value;                /* variables value. */
    void *var_handle;          /* place for RMAPI to store the var handle. */
};

/* local functions */
void display_rmapi_err(struct ha_em_err_blk *errblk);
void server_socket_handler(int table_index);
void session_socket_handler(int table_index);
void end_session(int table_index);
void register_variables();
void inst_variables(struct ha_rr_ctrl_msg *);
void add_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg);
void del_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg);
```

rmapi_smpsig.c

```
void send_values();
void refresh_values(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg);
void control_loop();
void set_signals();
void catch_io_signal(int);
void catch_alrm_signal(int);
void catch_exit_signal(int);
void mon_exit(int);

/* globals */
int                     NumMgrs = 0;                      /* num of resrc mngrs connected*/
struct ha_rr_variable   *Variables = (struct ha_rr_variable *)0;  /* RMAPI var array. */
struct ha_rr_val        *Values = (struct ha_rr_val *)0;          /* RMAPI val array. */
struct sock_table_entry SocketTable[SOCKET_TABLE_SIZE];   /* RMAPI sever */
                                                          /* and session sockets */
struct ha_em_err_blk    ErrBlock;                    /* global err blk for RMAPI calls*/
struct local_vars       *LocalVars = (struct local_vars *)0;      /* array of local vars */
static sigset_t         SignalMask;                       /* mask for signals. */
timer_t                 IntervalTimerId;          /* ID for var update interval timer. */
int                     Interval;                 /* Update interval for this monitor. */
int                     SampleTime = 0;           /* flag set by catching interval SIGALRM. */
int                     Terminate  = 0;           /* flag set by catching SIGTERM. */
int                     SigioCaught = 0;          /* flag set by catching SIGIO. */
int                     SigCaught = 0;            /* actual signal caught by term. */
int                     TimeOut = MAX_INTERVAL;   /* time out counter to call ha_rr_touch. */
int                     NumVariables = 0;         /* Number of variables available. */

/*****************************************************************************/
/* Main - Main - Main - Main - Main - Main - Main - Main - Main - Main - Main */
/*****************************************************************************/
main(int argc, char **argv)
{
int interval1, interval2;
int i, rc;
struct itimerstruc_t itimer;

    fprintf(stdout,"%s: %s resource monitor started.\n",PROGNAME,RESOURCE_MONITOR_NAME);

    /*
     * Setup siganl handlers.
     */
    set_signals();

    /*
     * Initialize the socket table. Socket file descriptors are inited to -1.
     */
    for (i=0;i<SOCKET_TABLE_SIZE;i++) {
        SocketTable[i].socket_fd = -1;
        if (i == RMAPI_SERVER_INDEX) {
            SocketTable[i].sock_funcp = server_socket_handler;
        } else {
            SocketTable[i].sock_funcp = session_socket_handler;
        }
    }

    /*
     * Initialize the RMAPI.
     */
```

```
rc = ha_rr_init(RESOURCE_MONITOR_NAME, &ErrBlock);
if (rc == HA_RR_FAIL) {
    display_rmapi_err(&ErrBlock);
    mon_exit(1);
}

/*
 * Query the update intervals for each class from the RMAPI.
 */
interval1 = ha_rr_get_interval(DYN_INST_CLASS_NAME,&ErrBlock);
if (interval1 <= 0) {
    display_rmapi_err(&ErrBlock);
    mon_exit(1);
}
fprintf(stdout,"Interval for class %s is %d.\n",DYN_INST_CLASS_NAME,interval1);

interval2 = ha_rr_get_interval(NON_INST_CLASS_NAME,&ErrBlock);
if (interval2 <= 0) {
    display_rmapi_err(&ErrBlock);
    mon_exit(1);
}
fprintf(stdout,"Interval for class %s is %d.\n",NON_INST_CLASS_NAME,interval2);

/*
 * For simplicity, the real interval will be the smallest of the two.
 */
 Interval = MIN(interval1,interval2);

/*
 * Allocate the local and RMAPI arrays for the non-instantiable variables.
 */


NumVariables = NUM_NONINST_VARS;
LocalVars =
    (struct local_vars *)malloc(sizeof(struct local_vars) * NumVariables);
Variables =
    (struct ha_rr_variable *)malloc(sizeof(struct ha_rr_variable) * NumVariables);
Values =
    (struct ha_rr_val *)malloc(sizeof(struct ha_rr_val) * NumVariables);

/*
 * Initialize the arrays.
 */
memset(LocalVars,0,sizeof(struct local_vars) * NumVariables);
memset(Variables,0,sizeof(struct ha_rr_variable) * NumVariables);
memset(Values,0,sizeof(struct ha_rr_val) * NumVariables);

/*
 * Initialize the local variable struct for the non-instantiables.
 */
for (i=0;i<NUM_NONINST_VARS;i++) {
    sprintf(LocalVars[i].var_name,"%s%d",NON_INST_VAR_NAME_PREFIX,i+1);
    LocalVars[i].var_ivec[0] = '\0';
    LocalVars[i].value = 0;
}

/*
```

rmapi_smpsig.c

```
     * Register the non-instantiable variables with the RMAPI.
     */
    register_variables();

    /*
     * Make this Resource Monitor a server.
     */
    rc = SocketTable[RMAPI_SERVER_INDEX].socket_fd =
            ha_rr_makserv(HA_RR_NOTIFY_SIGIO, &ErrBlock);
    if (rc == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        mon_exit(1);
    }
    fprintf(stdout,"Assigned rmapi server socket_fd(%d) to table index(%d).\n",
                rc,RMAPI_SERVER_INDEX);

    /*
     * Set up sampling timer.
     */
    IntervalTimerId = gettimerid(TIMERID_REAL, DELIVERY_SIGNALS);
    itimer.it_value.tv_sec = Interval;
    itimer.it_value.tv_nsec = 0;
    itimer.it_interval.tv_sec = Interval;
    itimer.it_interval.tv_nsec = 0;

    if (incinterval(IntervalTimerId, &itimer, (struct itimerstruc_t *)0) < 0) {
        fprintf(stderr,"incinterval() failed with errno=%d.\n",errno);
        mon_exit(1);
    }

    /*
     * Start the control loop - should never return.
     */
    control_loop();

    /*
     * Should never get here.
     */
    mon_exit(0);
}

/*
 * Sets the signal handlers.
 */
void set_signals() {
struct sigaction sigactn;

    sigemptyset(&SignalMask);
    sigaddset(&SignalMask, SIGALRM);
    sigaddset(&SignalMask, SIGTERM);
    sigaddset(&SignalMask, SIGIO);

    /*
     * Sigio signal handler.
     */
    sigactn.sa_handler = catch_io_signal;
    sigactn.sa_mask = SignalMask;
    sigactn.sa_flags = 0;
```

```
    if (sigaction(SIGIO, &sigactn, (struct sigaction *)0) < 0) {
        fprintf(stderr,"sigaction() failed for SIGIO signal. errno=%d.\n",errno);
        mon_exit(1);
    }

    /*
     * Alrm signal caught on the interval boundary.
     * Handler sets a global flag to do sampling.
     */
    sigactn.sa_handler = catch_alrm_signal;
    sigactn.sa_mask = SignalMask;
    sigactn.sa_flags = 0;

    if (sigaction(SIGALRM, &sigactn, (struct sigaction *)0) < 0) {
        fprintf(stderr,"sigaction() failed for ALRM signal. errno=%d.\n",errno);
        mon_exit(1);
    }

    /*
     * Need terminate handler to cleanup RMAPI on exit.
     */
    sigactn.sa_handler = catch_exit_signal;
    sigactn.sa_mask = SignalMask;
    sigactn.sa_flags = 0;

    if (sigaction(SIGTERM, &sigactn, (struct sigaction *)0) < 0) {
        fprintf(stderr,"sigaction() failed for TERM signal. errno=%d.\n",errno);
        mon_exit(1);
    }

    /*
     * Set signal mask, i.e. block SIGIO, SIGALRM, SIGTERM
     */
    if (sigprocmask(SIG_SETMASK, &SignalMask, (sigset_t *)0) < 0) {
        fprintf(stderr,"sigprocmask() failed. errno=%d.\n",errno);
        mon_exit(1);
    }
}

/*
 * Sends new values to the RMAPI - called when sig alrm is caught.
 */
void send_values() {
int i, rc;
int num_vals_to_send = 0;
static int times_called = 0;

    times_called++;
    for (i=0;i<NumVariables;i++) {

        if (i < NUM_NONINST_VARS) {
            /*
             * Randomly increment the non-instantiable variable (Counter).
             */
            LocalVars[i].value += (random() % 50);
        } else {
            /*
```

```
             * Assign a random value to the DynInstVar variable (Quantity).
             */
            LocalVars[i].value = (random() % MAX_VALUE);
        }

        /*
         * Only need to send values for vars with non-NULL handles.
         */
        if (LocalVars[i].var_handle != (void *)0) {

            /*
             * Copy the handle and value to the next available RMAPI struct.
             */
            Values[num_vals_to_send].rr_var_hndl = LocalVars[i].var_handle;
            Values[num_vals_to_send].rr_value = &(LocalVars[i].value);

            /*
             * Increment the number to send count.
             */
            num_vals_to_send++;
        }
    }

    if (num_vals_to_send) {
        /*
         * Send the Value array to the RMAPI.
         */
        rc = ha_rr_send_val(Values,num_vals_to_send,0,&ErrBlock);
        if (rc == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }

        /*
         * Reset the TimeOut counter.
         */
        TimeOut = MAX_INTERVAL;

    } else if ((TimeOut -= Interval) <= 0) {

        /*
         * No values have been sent in the TimeOut period. Need
         * to call ha_rr_touch to update the shm timestamp.
         */
        if (ha_rr_touch(&ErrBlock) == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }

        /*
         * Reset the TimeOut counter.
         */
        TimeOut = MAX_INTERVAL;

    }
}

/*
```

```
 * Catches io sig and sets global flag SigioCaught to send values.
 */
void catch_io_signal(int sig) {
    SigioCaught = 1;
}

/*
 * Catches alrm sig and sets global flag SampleTime to send values.
 */
void catch_alrm_signal(int sig) {
    SampleTime = 1;
}

/*
 * Catches term sig and sets global flag Terminate to exit.
 */
void catch_exit_signal(int sig) {
    Terminate = 1;
    SigCaught = sig;
}

/*
 * Handles a connection request on the RMAPI server
 * socket returned by ha_rr_makserv().
 */
void server_socket_handler(int table_index) {
int i, mgr_sock, rc;

    mgr_sock = ha_rr_start_session(HA_RR_NOTIFY_SIGIO,&ErrBlock);
    if (mgr_sock < 0) {
        display_rmapi_err(&ErrBlock);
        if ((ErrBlock.em_errno != HA_RR_EMAXSESSIONS) &&
                (ErrBlock.em_errno != HA_RR_EAGAIN)) {
            /*
             * Severe error in the RMAPI - exit.
             */
            mon_exit(1);
        }
        return;
    }
    /*
     * Find an available element in the session socket table.
     */
    for (i=0;(i < HA_RR_MAX_SESSIONS)&&(SocketTable[i].socket_fd >= 0);i++);

    if (i < HA_RR_MAX_SESSIONS) {
        /*
         * Save the mgr socket fd returned by the RMAPI.
         * The socket function was initialized in main.
         */
        SocketTable[i].socket_fd = mgr_sock;
        NumMgrs++;
        fprintf(stdout,"New session accepted to index(%d) socket_fd(%d).\n",i,mgr_sock);
    } else {
        /*
         * Should never get here, RMAPI handles too many sessions.
         */
        rc = ha_rr_end_session(mgr_sock,&ErrBlock);
```

```
    }
}

/*
 * Handles a msg being received from the manager session table_index.
 */
void session_socket_handler(int table_index) {
int len, session_sock;
struct ha_rr_ctrl_msg *ctrl_msg_start, *ctrl_msg;

    /*
     * Get the socket fd for this session.
     */
    if ((session_sock = SocketTable[table_index].socket_fd) < 0) {
        return;
    }

    /*
     * Call ha_rr_get_ctrl_msg to read the cmd from the manager.
     */
    fprintf(stdout,"Calling ha_rr_get_ctrlmsg for session(%d) socket_fd(%d).\n",
                table_index,session_sock);
    len = ha_rr_get_ctrlmsg(session_sock, &ctrl_msg, &ErrBlock);

    if (len == 0) {
        /*
         * Msg was for the RMAPI or was incomplete.
         */
        return;
    }

    if (len == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        if (ErrBlock.em_errno == HA_RR_EDISCONNECT) {
            end_session(table_index);
        } else if (ErrBlock.em_errno != HA_RR_EAGAIN) {
            /*
             * Severe RMAPI error.
             */
            mon_exit(1);
        }
        return;
    }

    /*
     * Since SIGIO is the notification protocol there could be
     * more than 1 message to read.
     */
    ctrl_msg_start = ctrl_msg;
    while ((char *)ctrl_msg < (((char *)ctrl_msg_start) + len)) {
        switch (ctrl_msg->rr_ctrl_cmd) {
            case HA_RR_CMD_ADDALL  :
            case HA_RR_CMD_ADDV    :
                add_variables(session_sock,ctrl_msg);
                break;
            case HA_RR_CMD_DELALL  :
            case HA_RR_CMD_DELV    :
                del_variables(session_sock,ctrl_msg);
```

```
                break;
            case HA_RR_CMD_INSTV   :
                inst_variables(ctrl_msg);
                break;
            case HA_RR_CMD_REFRESH :
                refresh_values(session_sock,ctrl_msg);
                break;
            default                :
                /*
                 * Unknown or unsupported msg - ignore it.
                 */
                fprintf(stderr,"Received an unexpected cmd(%d) "
                            "from socket_fd(%d), ignoring...\n",
                            ctrl_msg->rr_ctrl_cmd,session_sock);
                break;
        }
        ctrl_msg =
        (struct ha_rr_ctrl_msg *)(((char *)ctrl_msg) + ctrl_msg->rr_ctrl_msg_len);
    }

    /*
     * Free the msg allocated by the RMAPI.
     */
    free(ctrl_msg_start);
}

/*
 * Dynamically instantiates the variables requested in the ctrl msg. If
 * the ctrlmsg does not wildcard the ivector, it creates an instance of what
 * was requested otherwise creates several variables to simulate wildcard matching.
 */
void inst_variables(struct ha_rr_ctrl_msg *ctrl_msg) {
int num_var, num_to_create;
struct ha_rr_ctrl_var *vp;
char *inst_name;
static int inst_num = 10;
int i, j;

    fprintf(stdout,"Processing cmd HA_RR_CMD_INSTV "
    "to create with %d vars.\n",
            ctrl_msg->rr_ctrl_num_vars);

    for (i=0;i < ctrl_msg->rr_ctrl_num_vars;i++) {

        num_to_create = 0;

        /*
         * Get a pointer to the variable struct in the ctrl message.
         */
        vp = &(ctrl_msg->rr_ctrl_vars[i]);

        if (strcmp(DYN_INST_VAR_NAME, vp->rr_ctrl_name)) {
            /*
             * Variable name didn't match the dynamically instantiable var name.
             */
            fprintf(stderr,"%s: Invalid var (%s) name sent in INSTV ctrl msg.\n",
                    PROGNAME,vp->rr_ctrl_name);
            continue;
```

```
        }

        if (strcmp(vp->rr_ctrl_ivector,"") != 0) {
            /*
             * A specific instance is requested. Accommodate the request by creating
             * an the instance. If this were a real monitor, it would be created if
             * the request matched a valid variable.
             */
    if (strncmp(vp->rr_ctrl_ivector,DYN_INST_VAR_IVECTOR,strlen(DYN_INST_VAR_IVECTOR))) {
                /*
                 * The ivector in the ctrl message does not match the ivector
                 * for the variable.
                 */
                fprintf(stderr,"%s: Invalid ivector (%s) sent in INSTV ctrl msg.\n",
                        PROGNAME,vp->rr_ctrl_ivector);
                continue;
            }
            /*
             * Get a pointer passed the ivector name - should be pointing to an '=' now.
             * The format of the ivector is:  IvectorName=IvectorValue.
             */
            inst_name = vp->rr_ctrl_ivector + strlen(DYN_INST_VAR_IVECTOR);
            if (*inst_name != '=' || *(inst_name + 1) == '\0') {
                /*
                 * '=' not found or value missing in the ctrl message.
                 */
                fprintf(stderr,"%s: Invalid ivector (%s) sent in INSTV ctrl msg.\n",
                        PROGNAME,vp->rr_ctrl_ivector);
                continue;
            }
            /*
             * Adjust the instance name pointer to the value in the ctrl message.
             */
            inst_name++;
            num_to_create = 1;
        } else {
            /*
             * Ivector in the ctrl message was NULL which means wildcard the value.
             * Since this is just a sample monitor, we'll make up some variable names
             * to simulate the wildcard match. The contrived instances will have
             * ivectors: InstName=Inst<nn> where nn is the value of the local static
             * variable inst_num.
             */
            num_to_create = 5;
        }

        fprintf(stdout,"Instantiating %d new variables.\n",num_to_create);
        /*
         * Grow the local and RMAPI variable arrays to accomodate the new instances.
         */
        num_var = NumVariables + num_to_create;
        LocalVars =
            (struct local_vars *)realloc(LocalVars, sizeof(struct local_vars) * num_var);
        Variables =
        (struct ha_rr_variable *)realloc(Variables, sizeof(struct ha_rr_variable) * num_var);
        Values =
            (struct ha_rr_val *)realloc(Values, sizeof(struct ha_rr_val) * num_var);
```

```
        /*
         * Initialize the new local variables.
         */
        memset(LocalVars+NumVariables,0,sizeof(struct local_vars) * num_to_create);
        for (j=NumVariables;j<num_var;j++) {
            /*
             * Copy the dynamic variable name to the local struct.
             */
            sprintf(LocalVars[j].var_name,DYN_INST_VAR_NAME);
            if (num_to_create == 1) {
                /*
                 * Ivector was specified in the ctrl message - use the
                 * value in the new variables ivector.
                 */
                sprintf(LocalVars[j].var_ivec,"%s=%s",DYN_INST_VAR_IVECTOR,inst_name);
                fprintf(stdout,"Created dynamic instance: %s\n",LocalVars[j].var_ivec);
            } else {
                /*
                 * Ivector not specified in the ctrl message - contrive
                 * an ivector for the new instance.
                 */
                sprintf(LocalVars[j].var_ivec,"%s=Inst%d",DYN_INST_VAR_IVECTOR,inst_num);
                /*
                 * Increment the inst_num counter to keep new ivectors unique.
                 */
                inst_num++;
            }
            LocalVars[j].value = INITIAL_VALUE;
        }
        /*
         * Update the variable array size.
         */
        NumVariables = num_var;
    }
    /*
     * Register the new variables with the RMAPI.
     */
    register_variables();
}

/*
 * Adds the variables referenced in the ctrl_msg to the RMAPI for
 * the manager session sock_fd.
 */
void add_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg) {
int i;
int inst_id;
int num_added, num_to_add = 0;

    switch (ctrl_msg->rr_ctrl_cmd) {
        case HA_RR_CMD_ADDALL :
            /*
             * Add all variables to this manager session.
             */
            fprintf(stdout,"Processing cmd HA_RR_CMD_ADDALL "
            "to add all vars to session_fd %d.\n",sock_fd);
            for (i=0;i<NumVariables;i++) {
                /*
```

```
                 * Copy the var to the next RMAPI var struct.
                 */
                Variables[num_to_add].rr_var_name = LocalVars[i].var_name;
                Variables[num_to_add].rr_var_ivector = LocalVars[i].var_ivec;
                Variables[num_to_add].rr_varu.rr_var_hndl = &(LocalVars[i].var_handle);
                Variables[num_to_add].rr_value = &(LocalVars[i].value);

                /*
                 * Increment the count of variables to be added.
                 */
                num_to_add++;
            }
            break;
        case HA_RR_CMD_ADDV   :
            /*
             * Add a vector of variables to the session.
             */
            fprintf(stdout,"Processing cmd HA_RR_CMD_ADDV "
            "to add %d vars for session_fd %d.\n",
                        ctrl_msg->rr_ctrl_num_vars,sock_fd);
            for (i=0;i < ctrl_msg->rr_ctrl_num_vars;i++) {

                /*
                 * The inst id field in the ctrl msg is
                 * the index into the LocalVars array.
                 */
                inst_id =  ctrl_msg->rr_ctrlv.rr_ctrl_vari[i];

                if (inst_id < NumVariables) {
                    /*
                     * Copy the var to the next RMAPI var struct.
                     */
                    Variables[num_to_add].rr_var_name =
                        LocalVars[inst_id].var_name;
                    Variables[num_to_add].rr_var_ivector =
                        LocalVars[inst_id].var_ivec;
                    Variables[num_to_add].rr_varu.rr_var_hndl =
                        &(LocalVars[inst_id].var_handle);
                    Variables[num_to_add].rr_value =
                        &(LocalVars[inst_id].value);

                    num_to_add++;
                }
            }
            break;
        default                 :
            break;
    }
    if (num_to_add) {
        /*
         * Add the variables by calling ha_rr_add_var.
         */
        num_added = ha_rr_add_var(sock_fd, Variables, num_to_add, 1, &ErrBlock);

        if (num_added == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }
```

```
        fprintf(stdout,"ha_rr_add_var() added %d vars to session_fd %d.\n",
                        num_added,sock_fd);
        if (num_to_add != num_added) {
            /*
             * Loop through the variables that were attempted
             * to be added and report any that had errors.
             */
            for (i=0;i<num_to_add;i++) {
                if (Variables[i].rr_var_errno != 0) {
                    fprintf(stderr,"Variable(%s) ivec(%s) had bad errno(%d) "
                                   "from ha_rr_add_var().\n",

Variables[i].rr_var_name,Variables[i].rr_var_ivector,Variables[i".rr_var_errno);
                }
            }
        }
    }
}

/*
 * Deletes the variables referenced in the ctrl_msg for the mgr
 * sock_fd session.
 */
void del_variables(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg) {
int i;
int ctrl_cmd;
int inst_id;
int num_deleted, num_to_del = 0;

    if (ctrl_msg == (struct ha_rr_ctrl_msg *)0) {
        /*
         * Called with NULL msg when a session is ending. Need to delete
         * all vars from the session before calling ha_rr_end_session.
         */
        ctrl_cmd = HA_RR_CMD_DELALL;
    } else {
        ctrl_cmd = ctrl_msg->rr_ctrl_cmd;
    }

    switch (ctrl_cmd) {
        case HA_RR_CMD_DELALL :
            fprintf(stdout,"Processing cmd HA_RR_CMD_DELALL "
            "to delete all vars for session_fd %d.\n",
                    sock_fd);
            /*
             * Delete all variables for this manager session.
             */
            for (i=0;i<NumVariables;i++) {
                if (LocalVars[i].var_handle != (void *)0) {
                    /*
                     * Copy the var to the next RMAPI var struct.
                     */
                    Variables[num_to_del].rr_var_name = LocalVars[i].var_name;
                    Variables[num_to_del].rr_var_ivector = LocalVars[i].var_ivec;
                    Variables[num_to_del].rr_varu.rr_var_hndl = &(LocalVars[i].var_handle);

                    /*
```

```
                     * Increment the count of variables to be deleted.
                     */
                    num_to_del++;
                }
            }
            break;
        case HA_RR_CMD_DELV   :
            fprintf(stdout,"Processing cmd HA_RR_CMD_DELV "
            "to delete %d vars for session_fd %d.\n",
                    ctrl_msg->rr_ctrl_num_vars,sock_fd);
            /*
             * Delete a vector of variables for this manager session.
             */
            for (i = 0; i < ctrl_msg->rr_ctrl_num_vars; i++) {
                inst_id = ctrl_msg->rr_ctrlv.rr_ctrl_vari[i];
                if ((inst_id < NumVariables) &&
                    (LocalVars[i].var_handle != (void *)0)) {
                    /*
                     * Copy the var to the next RMAPI var struct.
                     */
                    Variables[num_to_del].rr_var_name =
                        LocalVars[inst_id].var_name;
                    Variables[num_to_del].rr_var_ivector =
                        LocalVars[inst_id].var_ivec;
                    Variables[num_to_del].rr_varu.rr_var_hndl =
                        &(LocalVars[inst_id].var_handle);

                    /*
                     * Increment the count of variables to be deleted.
                     */
                    num_to_del++;
                }
            }
            break;
        default                 :
            break;
    }
    if (num_to_del) {
        /*
         * Delete the variables by calling ha_rr_del_var.
         */
        num_deleted = ha_rr_del_var(sock_fd, Variables, num_to_del, &ErrBlock);

        if (num_deleted == HA_RR_FAIL) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }

        /*
         * rc>0 from ha_rr_del_var is the number of variables that no longer need
         * their values updated. The RMAPI will have set their handles to NULL.
         */
        fprintf(stdout,"%d variables no longer need to be updated.\n",num_deleted);
    }
}

/*
 * Updates values in response to a manager refresh command.
```

```
 */
void refresh_values(int sock_fd, struct ha_rr_ctrl_msg *ctrl_msg) {
int i, rc;
int inst_id;
int num_to_send = 0;

    fprintf(stdout,"Processing a refresh request of %d variables "
            "for session_fd %d.\n",
                        ctrl_msg->rr_ctrl_num_vars,sock_fd);
    for (i = 0; i < ctrl_msg->rr_ctrl_num_vars; i++) {
        inst_id =  ctrl_msg->rr_ctrlv.rr_ctrl_vari[i];
        if ((inst_id < NumVariables) &&
            (LocalVars[inst_id].var_handle)) {

            if (inst_id < NUM_NONINST_VARS) {
                /*
                 * Randomly increment the non-instantiable variable (Counter).
                 */
                LocalVars[inst_id].value += (random() % 50);
            } else {
                /*
                 * Assign a random value to the DynInstVar variable (Quantity).
                 */
                LocalVars[inst_id].value = (random() % MAX_VALUE);
            }

            /*
             * Copy the var handle and value to the RMAPI struct.
             */
            Values[num_to_send].rr_var_hndl = LocalVars[inst_id].var_handle;
            Values[num_to_send].rr_value = &(LocalVars[inst_id].value);
            num_to_send++;
        }
    }
    if (num_to_send) {
        /*
         * Call RMAPI with the refresh flag to send the values.
         */
        rc = ha_rr_send_val(Values, num_to_send, 1, &ErrBlock);
        if (rc) {
            display_rmapi_err(&ErrBlock);
            mon_exit(1);
        }
        /*
         * Reset the TimeOut counter.
         */
        TimeOut = MAX_INTERVAL;
    }
}

/*
 * Ends the manager session indexed by the table_index parm
 * in the socket table.
 */
void end_session(int table_index) {
int rc;

    fprintf(stdout,"Ending session index(%d) session_fd(%d).\n",
```

```
                table_index,SocketTable[table_index].socket_fd);

    /*
     * Make sure this is a valid session.
     */
    if ((table_index < 0) || (table_index > HA_RR_MAX_SESSIONS) ||
        (SocketTable[table_index].socket_fd < 0)) {
        return;
    }

    /*
     * Delete all variables for this manager.
     */
    del_variables(SocketTable[table_index].socket_fd, (struct ha_rr_ctrl_msg *)0);

    /*
     * RMAPI call to close the session.
     */
    rc = ha_rr_end_session(SocketTable[table_index].socket_fd, &ErrBlock);
    if (rc == HA_RR_FAIL) {
        display_rmapi_err(&ErrBlock);
        mon_exit(1);
    }

    /*
     * Reset the fd in the SocketTable and decrement the mgr count.
     */
    SocketTable[table_index].socket_fd = -1;
    NumMgrs--;

    if (NumMgrs <= 0) {
        /*
         * This was the last mgr to end - exit with a good rc.
         */
        fprintf(stdout,"Last manager session closed, exiting...\n");
        mon_exit(0);
    }
}

/*
 * Registers variables with the RMAPI.
 */
void register_variables() {
int i, j, num_var;
int num_registered, num_to_reg = 0;

    for (i=0;i<NumVariables;i++) {
        if (LocalVars[i].registered == 0) {

            /*
             * Variable not registered yet. Copy name, ivector and and value
             * pointers. Use the index of the variable as the inst id.
             */
            LocalVars[i].registered = 1;
            Variables[num_to_reg].rr_var_name = LocalVars[i].var_name;
            Variables[num_to_reg].rr_var_ivector = LocalVars[i].var_ivec;
            Variables[num_to_reg].rr_var_iid = i;
            Variables[num_to_reg].rr_value = &(LocalVars[i].value);
```

```
            num_to_reg++;
        }
    }

    if (num_to_reg) {

        /*
         * Call the RMAPI to  register the variables.
         */
        num_registered = ha_rr_reg_var(Variables,num_to_reg,&ErrBlock);
        if (num_registered != num_to_reg) {
            if (num_registered == HA_RR_FAIL) {
                display_rmapi_err(&ErrBlock);
                mon_exit(1);
            }
            fprintf(stdout,"%d variables registered with the RMAPI.\n",
                    num_registered);
            for (i=0; i < num_to_reg; i++) {
                if ((Variables[i].rr_var_errno) || (num_registered == HA_RR_FAIL)) {
                    /*
                     * Display a message for each variable with a registration error.
                     */
                    fprintf(stderr,"ha_rr_reg_var() error with variable(%s) "
                                   "instid(%d) RMAPI errno=(%d).\n",
                                Variables[i].rr_var_name,
                                Variables[i].rr_varu.rr_var_inst_id,
                                Variables[i].rr_var_errno);
                }
            }
        }
    }
}

/*
 * Called once by main to establish the control loop.
 */
void control_loop() {
int i, rc;
int sock;

    for(;;) {

        /*
         * Unblock the signals.
         */
        if (sigprocmask(SIG_UNBLOCK, &SignalMask, (sigset_t *)0) < 0) {
            /*
             * sigprocmask sys call failed.
             */
            fprintf(stderr,"Cannot unblock signals. sigprocmask() errno=%d.\n",errno);
            mon_exit(1);
        }

        /*
         * Pause to receive a signal.
         */
        pause();
```

```
        /*
         * Block the signals.
         */
        if (sigprocmask(SIG_BLOCK, &SignalMask, (sigset_t *)0) < 0) {
            fprintf(stderr,"Cannot block signals. sigprocmask() errno=%d.\n",errno);
            mon_exit(1);
        }

        if (Terminate) {
            /*
             * sig term caught - call mon_exit to clean up.
             */
            fprintf(stdout,"Caught signal(%d)...calling exit...\n",SigCaught);
            mon_exit(SigCaught);
        }

        if (SampleTime) {
            /*
             * sig alrm caught - call send_values
             * and reset the sample flag.
             */
            send_values();
            SampleTime = 0;
        }

        if (SigioCaught) {
            /*
             * Call the socket function for each valid socket.
             */
            SigioCaught = 0;
            for (i=0;i<SOCKET_TABLE_SIZE;i++) {
                if (SocketTable[i].socket_fd >=0) {
                    fprintf(stdout,"Calling socket function for session(%d) "
                                   "socket_fd(%d).\n",
                                i,SocketTable[i].socket_fd);
                    (SocketTable[i].sock_funcp)(i);
                }
            }
        }
    }
}

/*
 * Routine to terminate the monitor.
 */
void mon_exit(int s)
{
static int recursively_called = 0;
int i, rc;

    /*
     * Check for recursive call - some routines mon_exit()
     * calls can likewise call mon_exit().
     */
    if (recursively_called) return;
    recursively_called = 1;

    /*
```

```
     * Gracefully close all sessions.
     */
    for (i=0;i<HA_RR_MAX_SESSIONS;i++) {
        if (SocketTable[i].socket_fd >= 0) {
            end_session(i);
        }
    }

    /*
     * Terminate the RMAPI.
     */
    rc = ha_rr_terminate(&ErrBlock);
    if (rc) {
        display_rmapi_err(&ErrBlock);
    }
    fprintf(stdout,"%s: %s resource monitor exiting.\n",PROGNAME,RESOURCE_MONITOR_NAME);
    exit(s);
}

/*
 * Display an RMAPI error.
 */
void display_rmapi_err(struct ha_em_err_blk *errblk)
{
    fprintf(stderr,
        "RMAPI Error: File(%s) Version(%s) Line(%d) Errno(%d)\n\t%s",
        errblk->em_errfile,
        errblk->em_errlevel,
        errblk->em_errline,
        errblk->em_errno,
        errblk->em_errmsg);
}
```

The rmapi_smp.msg Message File

```
$ Prolog
$ ============================================================================*/
$                                                                             */
$ Module Name:  rmapi_smp.msg                                                 */
$                                                                             */
$ Description:                                                                */
$      Script to load Event Manager configuration data into the SDR.          */
$      This data is for the Rmapi sample monitors.                            */
$                                                                             */
$      Use runcat to generate a catalog from this file:                       */
$                                                                             */
$                  runcat rmapi_smp rmapi_smp.msg                             */
$                                                                             */
$      Then copy the rmapi_smp.cat file to the nls directory                  */
$      for the language set by your environment.                                    */
$                                                                             */
$ ============================================================================*/
$ End of Prolog
$
$ The quote signifies that the " character will be used to delimit
$ messages in the catalog.
$
$quote "
$
$ Lines beginning with a dollar sign are comments.  Lines
$ beginning with numbers are the messages.
$
$set 1
$
1 "IBM Data Suppliers"
$
2 "RMAPI Sample Monitor"
$
3 "Example non-instantiable resource variable (Counter or Quantity)."
$
4 "Example dynamically instantiable resource variable."
$
5 "Example instantiable resource variable (Counter or Quantity)."
$
6 "Example instantiable resource variable (State)."
$
7 "Example non-instantiable variables."
$
8 "Example instantiable variables."
$
9 "Example ivector NodeNum. Used to instantiate variables as node \n\
specific and as the variable locator field."
$
10 "Example instance vector for creating multiple instances of variables."
$
11 "Example SBS field: Action. Last action requested by the sample command."
$
12 "Example SBS field: Options. Last options parameter passed to the sample command."
$
13 "Example SBS field: StateChange. Success of last sample command call \
```

```
(!0==success,0==fail)."
$
14 "Example SBS field: State. State of the resource."
$
```

The rmapi_smp.loadsdr Shell Script

```
#! /bin/ksh
#*=============================================================================*/
#*                                                                             */
#* Module Name:  rmapi_smp.loadsdr                                             */
#*                                                                             */
#* Description:                                                                */
#*      Script to load Event Manager configuration data into the SDR.          */
#*      This data is for the Rmapi sample monitors.                            */
#*                                                                             */
#* Usage: rmapi_smp.loadsdr <resource_monitor_path>                            */
#*   Where <resource_monitor_path> is the path to the                          */
#*   compiled monitor samples.                                                 */
#*                                                                             */
#*=============================================================================*/
#*=============================================================================*/

#------------------------------------------------------------------------------
# Load resource monitor data for example resource monitor rmapi_smpdae
#------------------------------------------------------------------------------

if [ "$#" -ne 1 ]
then
        echo "Usage: $0 <resource_monitor_path>"
        exit 1
fi
rmpath=$1

SDRCreateObjects EM_Resource_Monitor \
        'rmName=IBM.PSSP.SampleDaeMon' \
        'rmPath='$rmpath/rmapi_smpdae \
        'rmMessage_file=rmapi_smp.cat' \
        'rmMessage_set=1' \
        'rmConnect_type=server' \
        'rmPTX_prefix=IBM/PSSP.SampleDaeMon' \
        'rmPTX_description=1,2' \
        'rmPTX_asnno=2'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Class \
        'rcClass=IBM.PSSP.SampleDaeClass' \
        'rcResource_monitor=IBM.PSSP.SampleDaeMon'  \
        'rcObservation_interval=10'  \
        'rcReporting_interval=10'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleDaeMon.StaticVars.static_var1' \
        'rvLocator=NodeNum' \
        'rvDescription=3' \
        'rvValue_type=Quantity' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=StaticVars/static_var1' \
        'rvPTX_description=7' \
        'rvPTX_min=0' \
```

```
        'rvPTX_max=500' \
        'rvClass=IBM.PSSP.SampleDaeClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleDaeMon.StaticVars.static_var2' \
        'rvLocator=NodeNum' \
        'rvDescription=3' \
        'rvValue_type=Quantity' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=StaticVars/static_var2' \
        'rvPTX_description=7' \
        'rvPTX_min=0' \
        'rvPTX_max=500' \
        'rvClass=IBM.PSSP.SampleDaeClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleDaeMon.StaticVars.static_var3' \
        'rvLocator=NodeNum' \
        'rvDescription=3' \
        'rvValue_type=Quantity' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=StaticVars/static_var3' \
        'rvPTX_description=7' \
        'rvPTX_min=0' \
        'rvPTX_max=500' \
        'rvClass=IBM.PSSP.SampleDaeClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleDaeMon.InstVars.inst_var1' \
        'rvLocator=NodeNum' \
        'rvDescription=5' \
        'rvValue_type=Quantity' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=$InstName/inst_var1' \
        'rvPTX_description=8' \
        'rvPTX_min=0' \
        'rvPTX_max=500' \
        'rvClass=IBM.PSSP.SampleDaeClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleDaeMon.InstVars.inst_var2' \
        'rvLocator=NodeNum' \
        'rvDescription=5' \
        'rvValue_type=Quantity' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=$InstName/inst_var2' \
```

```
        'rvPTX_description=8' \
        'rvPTX_min=0' \
        'rvPTX_max=500' \
        'rvClass=IBM.PSSP.SampleDaeClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleDaeMon.StaticVars' \
        'ivElement_name=NodeNum' \
        'ivElement_description=9'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleDaeMon.InstVars' \
        'ivElement_name=NodeNum' \
        'ivElement_description=9'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleDaeMon.InstVars' \
        'ivElement_name=InstName' \
        'ivElement_description=10'
if (( $? != 0 )) ; then exit 1 ; fi

#------------------------------------------------------------------------------
# Load resource monitor data for example resource monitor rmapi_smpcmd
#------------------------------------------------------------------------------

SDRCreateObjects EM_Resource_Monitor \
        'rmName=IBM.PSSP.SampleCmdMon' \
        'rmMessage_file=rmapi_smp.cat' \
        'rmMessage_set=1' \
        'rmConnect_type=client' \
        'rmPTX_prefix=dummy' \
        'rmPTX_description=2' \
        'rmPTX_asnno=1'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Class \
        'rcClass=IBM.PSSP.SampleCmdClass' \
        'rcResource_monitor=IBM.PSSP.SampleCmdMon'  \
        'rcObservation_interval=0'  \
        'rcReporting_interval=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleCmdMon.state' \
        'rvLocator=NodeNum' \
        'rvDescription=6' \
        'rvValue_type=State' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvClass=IBM.PSSP.SampleCmdClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
```

```
        'rvName=IBM.PSSP.SampleCmdMon.call' \
        'rvLocator=NodeNum' \
        'rvDescription=6' \
        'rvValue_type=State' \
        'rvInitial_value=0' \
        'rvData_type=SBS' \
        'rvClass=IBM.PSSP.SampleCmdClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleCmdMon' \
        'ivElement_name=NodeNum' \
        'ivElement_description=9'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleCmdMon' \
        'ivElement_name=NAME' \
        'ivElement_description=10'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Structured_Byte_String \
        'sbsVariable_name=IBM.PSSP.SampleCmdMon.call' \
        'sbsField_name=11' \
        'sbsField_type=long' \
        'sbsField_SN=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Structured_Byte_String \
        'sbsVariable_name=IBM.PSSP.SampleCmdMon.call' \
        'sbsField_name=12' \
        'sbsField_type=cstring' \
        'sbsField_SN=1'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Structured_Byte_String \
        'sbsVariable_name=IBM.PSSP.SampleCmdMon.call' \
        'sbsField_name=13' \
        'sbsField_type=long' \
        'sbsField_SN=2'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Structured_Byte_String \
        'sbsVariable_name=IBM.PSSP.SampleCmdMon.call' \
        'sbsField_name=14' \
        'sbsField_type=long' \
        'sbsField_SN=3'
if (( $? != 0 )) ; then exit 1 ; fi

#------------------------------------------------------------------------------
# Load resource monitor data for example resource monitor rmapi_smpsig
#------------------------------------------------------------------------------

SDRCreateObjects EM_Resource_Monitor \
        'rmName=IBM.PSSP.SampleSigMon' \
        'rmPath='$rmpath/rmapi_smpsig \
        'rmMessage_file=rmapi_smp.cat' \
```

```
        'rmMessage_set=1' \
        'rmConnect_type=server' \
        'rmPTX_prefix=IBM/PSSP.SampleSigMon' \
        'rmPTX_description=1,2' \
        'rmPTX_asnno=2'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Class \
        'rcClass=IBM.PSSP.SampleSigDynInstClass' \
        'rcResource_monitor=IBM.PSSP.SampleSigMon'  \
        'rcObservation_interval=10'  \
        'rcReporting_interval=10'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Class \
        'rcClass=IBM.PSSP.SampleSigNonInstClass' \
        'rcResource_monitor=IBM.PSSP.SampleSigMon'  \
        'rcObservation_interval=10'  \
        'rcReporting_interval=10'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleSigMon.NonInstVars.var1' \
        'rvLocator=NodeNum' \
        'rvDescription=3' \
        'rvValue_type=Counter' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=NonInstVars/var1' \
        'rvPTX_description=7' \
        'rvPTX_min=0' \
        'rvPTX_max=500000' \
        'rvClass=IBM.PSSP.SampleSigNonInstClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleSigMon.NonInstVars.var2' \
        'rvLocator=NodeNum' \
        'rvDescription=3' \
        'rvValue_type=Counter' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=NonInstVars/var2' \
        'rvPTX_description=7' \
        'rvPTX_min=0' \
        'rvPTX_max=500000' \
        'rvClass=IBM.PSSP.SampleSigNonInstClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleSigMon.NonInstVars.var3' \
        'rvLocator=NodeNum' \
        'rvDescription=3' \
        'rvValue_type=Counter' \
        'rvInitial_value=0' \
        'rvData_type=long' \
```

```
        'rvPTX_name=NonInstVars/var3' \
        'rvPTX_description=7' \
        'rvPTX_min=0' \
        'rvPTX_max=500000' \
        'rvClass=IBM.PSSP.SampleSigNonInstClass' \
        'rvDynamic_instance=0'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleSigMon.NonInstVars' \
        'ivElement_name=NodeNum' \
        'ivElement_description=9'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Resource_Variable \
        'rvName=IBM.PSSP.SampleSigMon.DynInstVar.var' \
        'rvLocator=NodeNum' \
        'rvDescription=4' \
        'rvValue_type=Quantity' \
        'rvInitial_value=0' \
        'rvData_type=long' \
        'rvPTX_name=DynInstVars/$InstName/var' \
        'rvPTX_description=8,8' \
        'rvPTX_min=0' \
        'rvPTX_max=500000' \
        'rvClass=IBM.PSSP.SampleSigDynInstClass' \
        'rvDynamic_instance=1'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleSigMon.DynInstVar' \
        'ivElement_name=NodeNum' \
        'ivElement_description=9'
if (( $? != 0 )) ; then exit 1 ; fi

SDRCreateObjects EM_Instance_Vector \
        'ivResource_name=IBM.PSSP.SampleSigMon.DynInstVar' \
        'ivElement_name=InstName' \
        'ivElement_description=10'
if (( $? != 0 )) ; then exit 1 ; fi
```

The rmapi_smp.unloadsdr Shell Script

```
#! /bin/ksh
#*=============================================================================*/
#*                                                                             */
#* Module Name:  rmapi_smp.unloadsdr                                           */
#*                                                                             */
#* Description:                                                                */
#*      Script to remove Event Manager configuration data from the SDR.        */
#*      This script is for sample resource monitors.                                  */
#*                                                                             */
#*=============================================================================*/
#-------------------------------------------------------------------------------
# Unload resource monitor data for example resource monitor rmapi_smpdae
#-------------------------------------------------------------------------------

SDRDeleteObjects EM_Resource_Monitor \
        'rmName==IBM.PSSP.SampleDaeMon'

SDRDeleteObjects EM_Resource_Class \
        'rcClass==IBM.PSSP.SampleDaeClass'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleDaeMon.StaticVars.static_var1'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleDaeMon.StaticVars.static_var2'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleDaeMon.StaticVars.static_var3'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleDaeMon.InstVars.inst_var1'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleDaeMon.InstVars.inst_var2'

SDRDeleteObjects EM_Instance_Vector \
        'ivResource_name==IBM.PSSP.SampleDaeMon.StaticVars'

SDRDeleteObjects EM_Instance_Vector \
        'ivResource_name==IBM.PSSP.SampleDaeMon.InstVars'


#-------------------------------------------------------------------------------
# Unload resource monitor data for example resource monitor rmapi_smpcmd
#-------------------------------------------------------------------------------

SDRDeleteObjects EM_Resource_Monitor \
        'rmName==IBM.PSSP.SampleCmdMon'

SDRDeleteObjects EM_Resource_Class \
        'rcClass==IBM.PSSP.SampleCmdClass'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleCmdMon.state'
```

```
SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleCmdMon.call'

SDRDeleteObjects EM_Instance_Vector \
        'ivResource_name==IBM.PSSP.SampleCmdMon'

SDRDeleteObjects EM_Structured_Byte_String \
        'sbsVariable_name==IBM.PSSP.SampleCmdMon.call'

#------------------------------------------------------------------------------
# Unload resource monitor data for example resource monitor rmapi_smpsig
#------------------------------------------------------------------------------

SDRDeleteObjects EM_Resource_Monitor \
        'rmName==IBM.PSSP.SampleSigMon'

SDRDeleteObjects EM_Resource_Class \
        'rcClass==IBM.PSSP.SampleSigDynInstClass'

SDRDeleteObjects EM_Resource_Class \
        'rcClass==IBM.PSSP.SampleSigNonInstClass'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleSigMon.NonInstVars.var1'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleSigMon.NonInstVars.var2'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleSigMon.NonInstVars.var3'

SDRDeleteObjects EM_Instance_Vector \
        'ivResource_name==IBM.PSSP.SampleSigMon.NonInstVars'

SDRDeleteObjects EM_Resource_Variable \
        'rvName==IBM.PSSP.SampleSigMon.DynInstVar.var'

SDRDeleteObjects EM_Instance_Vector \
        'ivResource_name==IBM.PSSP.SampleSigMon.DynInstVar'
```

rmapi_smp.unloadsdr

Chapter 6. Using the EMAPI: Some Event Management Client Examples

This chapter contains the listings of four sample Event Management client programs:

- "The emapi_v01_ex01.c Sample Program" on page 218 monitors four programs that are running on various nodes in two SP system partitions. Because two system partitions are involved, the program establishes two sessions with the EMAPI, one per system partition.

  Because the program is not multi-threaded, it uses the **select** system call to multiplex responses from the two EMAPI sessions.
- "The emapi_v01_ex02.c Sample Program" on page 238 provides the same function as the **emapi_v01_ex01.c** program. It differs in that it uses callback routines to receive events.
- "The emapi_v01_ex03.c Sample Program" on page 258 provides the same function as the first two programs. It differs in that it is multi-threaded. Each session is managed by a separate thread.
- "The emapi_v01_ex04.c Sample Program" on page 278 illustrates the use of the EMAPI query function.

You can find the files for these programs online in the PSSP product samples directory, **/usr/lpp/ssp/samples/haem/emapi**.

© Copyright IBM Corp. 1996

The emapi_v01_ex01.c Sample Program

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <time.h>

#include <sys/types.h>
#include <sys/select.h>
#include <sys/time.h>

#include <ha_emapi.h>

/*
 *  emapi_v01_ex01.c
 *
 *  This program presents an example of using the Event Manager Application
 *  Programming Interface (EMAPI).  The program uses the EMAPI to monitor
 *  four programs running on various nodes in two SP partitions.  Since two
 *  SP partitions are involved, the program establishes two sessions with the
 *  EMAPI, one per SP partition.  Since the program is not multi-threaded,
 *  select() is used to multiplex responses from the two EMAPI sessions.
 *
 *  This program can be compiled with the following command:
 *
 *      cc -O emapi_v01_ex01.c -o emapi_v01_ex01 -lha_em
 */

/*
 *  If a MAX macro isn't defined by one of the included header files, define
 *  it.
 */

#ifndef MAX
#define MAX(x, y)   ((x) > (y) ? (x) : (y))
#endif

/*
 *  The following macros specify the programs this program will monitor through
 *  the EMAPI.
 */

#define PART_1      "k21s"              /* 1st SP partition                 */

#define PROG_1A     "subsys_pgrm1"      /* 1st program in 1st SP partition,  */
#define USER_1A     "root"              /* ... and user running 1st program, */
#define NODE_1A     5                   /* ... and node running 1st program. */

#define PROG_1B     "subsys_pgrm2"      /* 2nd program in 1st SP partition,  */
#define USER_1B     "root"              /* ... and user running 2nd program, */
#define NODE_1B     6                   /* ... and node running 2nd program. */

#define PART_2      "k21sp2"            /* 2nd SP partition                 */
```

```
#define PROG_2A     "subsys_pgrm1"      /* 1st program in 2nd SP partition,  */
#define USER_2A     "root"              /* ... and user running 1st program, */
#define NODE_2A     14                  /* ... and node running 1st program. */

#define PROG_2B     "subsys_pgrm2"      /* 2nd program in 2nd SP partition,  */
#define USER_2B     "root"              /* ... and user running 2nd program, */
#define NODE_2B     15                  /* ... and node running 2nd program. */

/*
 *  This program saves information about each EMAPI session it has established
 *  in a session structure.
 */

struct session {
    char        *name;      /* Name of partition used by session             */
    int          fd;        /* Session file descriptor                       */
    int          restart;   /* Boolean - Session's connection has been lost  */
};

/*
 *  This program saves information about each program it is monitoring through
 *  the EMAPI in a program structure.
 */

struct program {
    char        *name;      /* Name of a program to be monitored             */
    char        *user;      /* Name of user running program                  */
    int          node;      /* Node on which program is running              */
    ha_em_eid_t  eid;       /* Event identifier for program                  */
    int          unreged;   /* Event is unregistered                         */
};

/*
 *  The terminate_requested global variable is set to a non-zero value when
 *  the user of this program requests that the program be terminated.  The
 *  user requests program termination via the interrupt key (typically Ctrl-C).
 *  The interrupt key causes a SIGINT signal to be delivered to this program.
 *  This program installs a signal handler for the SIGINT signal, which sets
 *  the terminate_requested global variable to a non-zero value when SIGINT
 *  is delivered.
 */

static int terminate_requested       = 0;

/*
 *  The unregistrations_requested global variable indicates whether this
 *  program has sent unregistration requests to the Event Manager.
 *  Unregistration requests are sent when the user requests program
 *  termination.
 */

static int unregistrations_requested = 0;

/*
 *  Function prototypes for internal functions.
 */
```

emapi_v01_ex01.c

```
static void interrupt_catch(int signo);
static void setup_signals(void);
static void start_session(struct session *sess_p);
static void register_for_events(struct session *sess_p,
        struct program *progs_p, int progs_elems, int *reg_cnt_p);
static void unregister_events(struct session *sess_p,
        struct program *progs_p, int progs_elems);
static void select_session(struct session *sess_p,
        fd_set *read_fds_p, int *fd_limit_p, int *timeout_p);
static int session_response_ready(struct session *sess_p, fd_set *read_fds_p);
static void process_response(struct session *sess_p,
         struct program *progs_p, int progs_elems, int *reg_cnt_p);
static void process_response_reg(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk);
static void process_response_rerr(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p);
static void process_response_unreg(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p);
static void format_timestamp(struct timeval *timestamp_p,
        char *fmt_timestamp_p);
static void breakdown_ivector(char *ivector_p,
        char *prog_name_p, char *user_name_p, int *node_p);
static void find_ivector_value(char *ivector_p, char *name_p,
        char **value_pp, size_t *value_len_p);
static void end_session(struct session *sess_p);


/*
 *  The main() function calls functions to initialize the program, establish
 *  EMAPI sessions, register for events, wait for Event Manager responses,
 *  process Event Manager responses, and respond to the user request for
 *  program termination.  When the program termination request is received,
 *  main() calls functions to send unregistration requests through the EMAPI.
 *  Once Event Manager responses have been received indicating that the
 *  events have been unregistered, the program ends its EMAPI sessions,
 *  and terminates.
 */

int main(int argc, char **argv)
{
    struct session sess1 = {PART_1, -1, 0};
    struct session sess2 = {PART_2, -1, 0};

    struct program progs1[] = {{PROG_1A, USER_1A, NODE_1A, 0, 0},
                               {PROG_1B, USER_1B, NODE_1B, 0, 0}};
    struct program progs2[] = {{PROG_2A, USER_2A, NODE_2A, 0, 0},
                               {PROG_2B, USER_2B, NODE_2B, 0, 0}};

    int     progs1_elems = sizeof progs1 / sizeof progs1[0];
    int     progs2_elems = sizeof progs2 / sizeof progs2[0];

    int     reg_cnt;            /* Count of registered events               */
    fd_set  read_fds;           /* select() file descriptor mask            */
    int     fd_limit;           /* select() file descriptor limit           */
    int     need_timeout;       /* Boolean - select() timeout needed        */
```

```
int     rc;                 /* Return code                              */

struct timeval  timeout_value = {15, 0};    /* Fifteen second timeout    */


setup_signals();            /* Initialize signal dispositions           */

start_session(&sess1);      /* Start EMAPI session for 1st SP partition  */
start_session(&sess2);      /* Start EMAPI session for 2nd SP partition  */

/*
 *  Register for events in both EMAPI sessions that will monitor the
 *  programs of interest.  Maintain count of registered events.
 */

reg_cnt = 0;
register_for_events(&sess1, progs1, progs1_elems, &reg_cnt);
register_for_events(&sess2, progs2, progs2_elems, &reg_cnt);

/*
 *  Continue looking for Event Manager responses as long as there are
 *  registered events.
 */

while (reg_cnt > 0) {

    /*
     *  If program termination has been requested, and unregistration
     *  requests have not been sent, send them now.
     */

    if (terminate_requested && !unregistrations_requested) {
        printf("Termination requested.\n");
        unregister_events(&sess1, progs1, progs1_elems);
        unregister_events(&sess2, progs2, progs2_elems);
        unregistrations_requested = 1;
    }

    /*
     *  Construct the parameters to be passed to select(), and then
     *  call select() to wait for Event Manager responses.
     */

    FD_ZERO(&read_fds); fd_limit = 0; need_timeout = 0;
    select_session(&sess1, &read_fds, &fd_limit, &need_timeout);
    select_session(&sess2, &read_fds, &fd_limit, &need_timeout);

    rc = select(fd_limit + 1, &read_fds, NULL, NULL,
                              need_timeout ? &timeout_value : NULL);

    if (rc == -1) {                 /* select() indicates error          */
        if (errno == EINTR) {       /* select() interrupted by signal    */
            continue;               /* Return to top of loop             */
        }
        perror("select()");         /* select() really encountered error */
        exit(1);                    /* End program                       */
    }
```

```
        /*
         *  If an Event Manager response is present for the 1st EMAPI session,
         *  process it.  Also, maintain the count of registered events.
         */

        if (session_response_ready(&sess1, &read_fds)) {
            process_response(&sess1, progs1, progs1_elems, &reg_cnt);
        }

        /*
         *  If an Event Manager response is present for the 2nd EMAPI session,
         *  process it.  Also, maintain the count of registered events.
         */

        if (session_response_ready(&sess2, &read_fds)) {
            process_response(&sess2, progs2, progs2_elems, &reg_cnt);
        }

    }

    end_session(&sess1);        /* End EMAPI session for 1st SP partition    */
    end_session(&sess2);        /* End EMAPI session for 2nd SP partition    */

    return 0;                   /* Indicate program was successful           */
}


/*
 *  The interrupt_catch() function is installed as the signal handler for the
 *  SIGINT signal.  The SIGINT signal is delivered to this process when the
 *  user presses the interrupt key (usually Ctrl-C).  When this routine is
 *  invoked, it sets the global variable terminate_requested to a non-zero
 *  value, indicating the program is to be terminated.
 */

static
void interrupt_catch(int signo)
{
    terminate_requested = 1;
    return;
}


/*
 *  The setup_signals() function initializes the disposition of a couple of
 *  signals.
 *
 *  The disposition of the SIGPIPE signal is set such that the signal is
 *  ignored.  When a process writes to a pipe or connection-oriented socket for
 *  which there is no reader, the SIGPIPE signal is delivered to the process.
 *  The default disposition for SIGPIPE is to kill the receiving process.  If
 *  SIGPIPE is ignored, the SIGPIPE signal does not kill the process, and the
 *  system call used to write to the pipe or connection-oriented socket sets
 *  errno to EPIPE and returns an error indication.  It is recommended that
 *  clients of the EMAPI ignore the SIGPIPE signal, since the EMAPI uses
 *  connection-oriented sockets to communicate with the Event Manager.
 *
 *  The interrupt_catch() function is installed as the signal handler for the
```

```
 *  SIGINT signal.  See the comments pertaining to the interrupt_catch()
 *  function to find out why this is done.
 */

static
void setup_signals(void)
{
    struct sigaction    sa;


    sa.sa_handler = SIG_IGN;               /* SIGPIPE to be ignored         */
    sigemptyset(&sa.sa_mask);              /* No signals to mask off        */
    sa.sa_flags = 0;                       /* No flags needed               */

    if (sigaction(SIGPIPE, &sa, NULL) == -1) {  /* Change SIGPIPE disposition*/
        perror("sigaction(SIGPIPE)");
        exit(1);
    }

    sa.sa_handler = interrupt_catch;       /* SIGINT signal handler         */
    sigemptyset(&sa.sa_mask);              /* No signals to mask off        */
    sa.sa_flags = SA_RESTART;              /* Restart interrupted syscalls  */

    if (sigaction(SIGINT, &sa, NULL) == -1) {   /* Change SIGINT disposition */
        perror("sigaction(SIGINT)");
        exit(1);
    }

    return;
}


/*
 *  The start_session() function establishes a session with the EMAPI by
 *  calling the EMAPI routine ha_em_start_session().  If a session cannot be
 *  established, the program is terminated.
 */

static
void start_session(struct session *sess_p)
{
    struct ha_em_err_blk errb;                 /* EMAPI error block         */


    /*
     *  The ha_em_start_session() routine receives the name of the partition
     *  with which a session is to be established, and returns a file
     *  descriptor with which the session can be used.
     */

    sess_p->fd = ha_em_start_session(sess_p->name, &errb);

    /*
     *  If ha_em_start_session() indicates a session could not be established,
     *  print the error information returned by the routine, and exit the
     *  program.
     */
```

```
    if (sess_p->fd == -1) {
        fprintf(stderr, "%s:\tha_em_start_session() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    /*
     *  A session has been established with the EMAPI, and the session's file
     *  descriptor has been saved in the session structure passed to this
     *  routine.  Set the flag in the session structure that indicates the
     *  session does not need to be restarted.
     */

    sess_p->restart = 0;

    return;
}


/*
 *  The register_for_events() function uses the EMAPI routine
 *  ha_em_send_command() to request the registration of events pertaining to
 *  programs of interest that are in the same SP partition.
 *
 *  The resource variable name specified is "IBM.PSSP.Prog.xpcount".  This is a
 *  state variable whose value indicates whether or not processes are executing
 *  a specific program.  The instantiation vector for this resource variable
 *  specifies the program name, the name of the user running the program, and
 *  the node on which the program is running.  The resource variable's value
 *  is a structured byte string of three fields.  Field 0 is of type long; its
 *  value is the number of processes currently running the specified program
 *  for the specified user on the specified node.  Field 1 is also of type
 *  long; its value is the previous number of processes running the program.
 *  Field 2 is of type character string; its value is a comma separated list
 *  of the PIDs of the processes running the program.
 *
 *  When the events are registered, two predicates are specified.  The primary
 *  predicate, "X@0 == 0", specifies that an event is to be generated when
 *  field 0 of the resource variable's structured byte string has a value of 0.
 *  Taking into account the semantics of the resource variable, the predicate
 *  causes an event to be generated when no processes are running the specified
 *  program for the specified user on the specified node.  The re-arm
 *  predicate, "X@0 > 0", specifies that after the primary predicate has
 *  generated an event, the primary predicate is to be re-armed once a process
 *  starts running the specified program for the specified user on the
 *  specified node.  Since the HA_EM_CMD_REG2 command is specified, the re-arm
 *  predicate will generate events.
 *
 *  Notice that when events are registered, the HA_EM_SCMD_REVAL subcommand is
 *  specified.  As a result, the initial value of the programs being monitored
 *  will be returned.
 */

static
void register_for_events(struct session *sess_p,
        struct program *progs_p, int progs_elems, int *reg_cnt_p)
{
    int                     num_events;      /* Number of events to register */
```

```
size_t                  alloc_size;      /* Size of memory to allocate   */
struct ha_em_cmd_blk    *cmd_blk;        /* Pointer to command block     */
struct ha_em_rb_reg     *re;             /* Pointer to registered event  */
int                     i;               /* Index                        */
struct ha_em_err_blk    errb;            /* EMAPI error block            */
int                     rc;              /* Return code                  */
char                    ivector_buf[80]; /* Instantiation vector buffer  */


num_events = progs_elems;           /* Register one event per program    */

/*
 *  Allocate enough memory to hold the command block to be given to the
 *  EMAPI.  The ha_em_cmd_blk structure contains one ha_em_rb_reg
 *  structure.  Additional space must be allocated when more than one
 *  event is to be registered.
 */

alloc_size = sizeof(struct ha_em_cmd_blk) +
                ((num_events - 1) * sizeof(struct ha_em_rb_reg));

if ((cmd_blk = malloc(alloc_size)) == NULL) {
    perror("malloc()");
    exit(1);
}

/*
 *  Fill in the em_rb_reg array entries.  There is one entry per event to
 *  be registered.
 */

for (i = 0; i < num_events; i++) {

    re = &cmd_blk->em_res_blk.em_rb_reg[i];

    re->em_name = "IBM.PSSP.Prog.xpcount";   /* Resource variable name   */

    sprintf(ivector_buf, "ProgName=%s;UserName=%s;NodeNum=%d",
            progs_p[i].name, progs_p[i].user, progs_p[i].node);
    re->em_ivector = strdup(ivector_buf);    /* Instantiation vector     */

    if (re->em_ivector == NULL) {
        perror("strdup()");
        exit(1);
    }

    re->em_pred   = "X@0 == 0";              /* Predicate                */
    re->em_rapred = "X@0 > 0";               /* Re-arm predicate         */

    re->em_cb     = NULL;                    /* Callback routine not used*/
    re->em_cb_arg = NULL;                    /* Callback routine not used*/
}

/*
 *  Fill in command block header, specifying number of elements in
 *  em_rb_reg array, command, and subcommand.
 */
```

```
    cmd_blk->em_cmd_num_elem = num_events;
    cmd_blk->em_cmd          = HA_EM_CMD_REG2;
    cmd_blk->em_subcmd       = HA_EM_SCMD_REVAL;

    /*
     *  Send the command.  A more elaborate program might check for the
     *  HA_EM_ECONNREFUSED Event Manager error number when an error is
     *  returned, and attempt to restart the session.  But, this program
     *  just terminates if ha_em_send_command() detects a lost connection.
     */

    rc = ha_em_send_command(sess_p->fd, cmd_blk, &errb);

    if (rc == -1) {
        fprintf(stderr, "%s:\tha_em_send_command() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    /*
     *  Save the event identifiers assigned to the events just registered.
     */

    for (i = 0; i < num_events; i++) {
        re = &cmd_blk->em_res_blk.em_rb_reg[i];
        progs_p[i].eid     = re->em_event_id;
        progs_p[i].unreged = 0;
        free(re->em_ivector);
    }

    free(cmd_blk);

    *reg_cnt_p += num_events;           /* Update count of registered events */

    return;
}


/*
 *  The unregister_events() function uses the EMAPI routine
 *  ha_em_send_command() to request the unregistration of events pertaining to
 *  programs of interest that are in the same SP partition.
 */

static
void unregister_events(struct session *sess_p,
        struct program *progs_p, int progs_elems)
{
    int                     num_events;      /* Number of events to unreg.   */
    size_t                  alloc_size;      /* Size of memory to allocate   */
    struct ha_em_cmd_blk    *cmd_blk;        /* Pointer to command block     */
    ha_em_eid_t             *re;             /* Pointer to unreg. event      */
    int                     i;               /* Index                        */
    struct ha_em_err_blk    errb;            /* EMAPI error block            */
    int                     rc;              /* Return code                  */


    /*
```

```
     *  Allocate enough memory to hold the command block to be given to the
     *  EMAPI.  The ha_em_cmd_blk structure contains one ha_em_eid_t
     *  element.  Additional space must be allocated when more than one
     *  event is to be unregistered.
     */

    alloc_size = sizeof(struct ha_em_cmd_blk) +
                    ((progs_elems - 1) * sizeof(ha_em_eid_t));

    if ((cmd_blk = malloc(alloc_size)) == NULL) {
        perror("malloc()");
        exit(1);
    }

    /*
     *  Fill in the event identifier array entries.  There is one entry per
     *  event to be unregistered.  Only unregister events that are not already
     *  unregistered.
     */

    num_events = 0;

    for (i = 0; i < progs_elems; i++) {
        if (!progs_p[i].unreged) {
            re  = &cmd_blk->em_res_blk.em_rb_unreg[num_events];
            *re = progs_p[i].eid;
            num_events++;
        }
    }

    /*
     *  Fill in command block header, specifying the number of elements in
     *  the event identifier array, and the unregister command.
     */

    cmd_blk->em_cmd_num_elem = num_events;
    cmd_blk->em_cmd          = HA_EM_CMD_UNREG;

    /*
     *  Send the unregister command.  A more elaborate program might check for
     *  the HA_EM_ECONNREFUSED Event Manager error number when an error is
     *  returned, and attempt to restart the session.  But, this program
     *  just terminates if ha_em_send_command() detects a lost connection.
     */

    rc = ha_em_send_command(sess_p->fd, cmd_blk, &errb);

    if (rc == -1) {
        fprintf(stderr, "%s:\tha_em_send_command() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    free(cmd_blk);

    return;
}
```

emapi_v01_ex01.c

```
/*
 *  The select_session() routine determines how the specified session affects
 *  the next call to select().  If the session is marked as needing to be
 *  restarted, ha_em_restart_session() is called to try to restart the session.
 *  If the restart attempt fails, the next call to select() should specify
 *  a timeout period - so the restart can be attempted again.  If the restart
 *  attempt succeeds, the new session file descriptor is saved, and the session
 *  is marked as not needing to be restarted.  If the session did not need
 *  to be restarted, or had been successfully restarted, the next call to
 *  select() should examine the session's file descriptor.
 */

static
void select_session(struct session *sess_p,
        fd_set *read_fds_p, int *fd_limit_p, int *timeout_p)
{
    struct ha_em_err_blk    errb;           /* EMAPI error block             */
    int                     new_fd;         /* New session file descriptor   */


    /*
     *  If the specified session needs to be restarted, try to do so.
     */

    if (sess_p->restart) {

        new_fd = ha_em_restart_session(sess_p->fd, &errb);

        if (new_fd == -1) {

            /*
             *  The session could not be restarted.  If the Event Manager
             *  error number indicates connection refused, indicate the
             *  next select() should specify a timeout value (so the restart
             *  can be tried again later), and return without putting the
             *  session's file descriptor in the select() file descriptor
             *  mask.
             */

            if (errb.em_errno == HA_EM_ECONNREFUSED) {
                *timeout_p = 1;
                return;
            }

            /*
             *  If execution reaches here, the session restart attempt failed
             *  for an unexpected reason; terminate the program.
             */

            fprintf(stderr, "%s:\tha_em_restart_session() returned EM errno "
                    "%d:\n%s", sess_p->name, errb.em_errno, errb.em_errmsg);
            exit(1);
        }

        /*
         *  If execution reaches here, a session restart was attempted and
         *  was successful.  Save the new session file descriptor, and indicate
```

```
         *  that the session no longer needs to be restarted.
         */

        sess_p->fd = new_fd;
        sess_p->restart = 0;

    }

    /*
     *  If execution reaches here, the specified session is not known to be
     *  in need of a restart.  Put the session's file descriptor in the
     *  select() file descriptor mask.  Also, adjust the maximum file
     *  descriptor to select, if appropriate.
     */

    FD_SET(sess_p->fd, read_fds_p);
    *fd_limit_p = MAX(sess_p->fd, *fd_limit_p);

    return;
}


/*
 *  The session_response_ready() function determines if a select() file
 *  descriptor mask indicates that a session's file descriptor is ready for
 *  reading.
 */

static
int session_response_ready(struct session *sess_p, fd_set *read_fds_p)
{
        return FD_ISSET(sess_p->fd, read_fds_p);
}


/*
 *  The process_response() function processes an Event Manager response
 *  in the specified EMAPI session.  First, ha_em_receive_response() is called
 *  to receive the response.  If ha_em_receive_response() indicates that the
 *  session's connection with the Event Manager has been lost, the session
 *  is marked as being in need of a restart.  Any other error will terminate
 *  the program.  The ha_em_receive_response() function may indicate that
 *  there really isn't any response for the EMAPI client to process at this
 *  time.  In that case, this function just returns.  If a response has
 *  been received, it is processed depending on what type of response it is.
 */

static
void process_response(struct session *sess_p,
         struct program *progs_p, int progs_elems, int *reg_cnt_p)
{
    struct ha_em_rsp_blk    *rsp_blk;   /* Pointer to the response block     */
    int                     rc;         /* Return code                       */
    struct ha_em_err_blk    errb;       /* Event Manager error block         */
    int                     i;          /* Index                             */


    /*
```

```
     *  Receive response from session, if there is one to receive.
     */

    rc = ha_em_receive_response(sess_p->fd, &rsp_blk, &errb);

    if (rc == -1) {

        /*
         *  If the connection with the Event Manager has been lost,
         *  mark the session as needing to be restarted, and return.
         */

        if (errb.em_errno == HA_EM_ECONNLOST) {
            printf("%s:\tconnection to session lost.\n", sess_p->name);
            sess_p->restart = 1;
            return;
        }

        /*
         *  Some error has occurred other than the loss of the connection.
         *  Terminate the program.
         */

        fprintf(stderr, "%s:\tha_em_receive_response() returned EM errno %d:\n"
                "%s", sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);

    }

    /*
     *  If the ha_em_receive_response() routine returned zero,
     *  there is no response for the EMAPI client (this program) to process
     *  at this time.  The response may have been for the EMAPI itself, or
     *  a full response may not be available yet.  Just return.
     */

    if (rc == 0) {
        return;
    }

    /*
     *  A response for the EMAPI client (this program) has been returned.
     *  The response buffer is pointed to by the rsp_blk variable.  Appropriate
     *  processing for the response depends on the command type.
     */

    switch (rsp_blk->em_cmd) {

        case HA_EM_CMD_REG:             /* Event response                    */
        case HA_EM_CMD_REG2:            /* Event response (possible re-arm)  */
            process_response_reg(sess_p, progs_p, progs_elems, rsp_blk);
            break;

        case HA_EM_CMD_RERR:            /* REG  event registration error     */
        case HA_EM_CMD_R2ERR:           /* REG2 event registration error     */
            process_response_rerr(sess_p, progs_p, progs_elems, rsp_blk,
                                  reg_cnt_p);
            break;
```

```
        case HA_EM_CMD_UNREG:           /* Event unregistration response     */
            process_response_unreg(sess_p, progs_p, progs_elems, rsp_blk,
                                   reg_cnt_p);
            break;

        default:                        /* Unexpected response               */
            fprintf(stderr, "%s:\tProgram received unexpected command "
                    "response: %d.\n", sess_p->name, rsp_blk->em_cmd);
            exit(1);
            break;

    }

    /*
     *  The EMAPI client (this program) must free the memory associated with
     *  the returned response block when it is no longer needed.
     */

    free(rsp_blk);

    return;
}


/*
 *  The process_response_reg() function processes event responses from the
 *  Event Manager.  Several points should be kept in mind:
 *
 *      - The response block may contain multiple event responses.  The
 *        number of responses included in the response block is given in
 *        the response block header.
 *
 *      - An event response may be an error response.  An event error response
 *        does not indicate that a registered event has occurred.  Instead, it
 *        indicates the Event Manager no longer knows the current value of
 *        the associated resource variable, and cannot generate events for
 *        the resource variable.  This may be a temporary condition.  If the
 *        Event Manager later obtains the current value of the associated
 *        resource variable, an event will be generated (possibly indicating
 *        that the predicate is false).
 *
 *      - The event for which an event error response is generated is still
 *        registered.  There is no need to re-register the event.
 *
 *      - Since events are registered by this program using the HA_EM_CMD_REG2
 *        command, the re-arm predicates will generate events.  Therefore, the
 *        HA_EM_EVENT_RE_ARM flag must be tested to see if the predicate or
 *        the re-arm predicate generated the event.
 *
 *      - It is possible for an event to be delivered indicating the predicate
 *        is false.  This can happen for two reasons.  First, when events are
 *        registered by this program, the HA_EM_SCMD_REVAL subcommand is
 *        specified.  That subcommand requests the initial value of the
 *        associated resource variable.  Second, the current value of the
 *        associated resource variable is returned through an event once the
 *        Event Manager obtains the current value of a resource variable after
 *        an error has occurred.
```

emapi_v01_ex01.c

```
 *
 *      - If an event response does not have the HA_EM_EVENT_RE_ARM nor
 *        HA_EM_EVENT_PRED_FALSE flags set, the event response indicates that
 *        the event's predicate is true.
 */

static
void process_response_reg(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk)
{
    struct ha_em_rpb_event  *event_p;           /* Pointer to event response */
    struct ha_em_rpb_event  *last_event_p;      /* Beyond last event response*/

    char time_stamp[80];        /* Formatted event time stamp                */
    char prog_name[80];         /* Name of program to which event pertains   */
    char user_name[80];         /* Name of user to which event pertains      */
    int  node;                  /* Node to which event pertains              */


    event_p = rsp_blk->em_resp_blk.em_rpb_event;
    last_event_p = event_p + rsp_blk->em_rsp_num_resp;

    for ( ; event_p < last_event_p; event_p++) {

        /*
         *  Format the event's time stamp; extract the program name, user
         *  name, and node number from the instantiation vector.
         */

        format_timestamp(&event_p->em_timestamp, time_stamp);
        breakdown_ivector(event_p->em_ivector, prog_name, user_name, &node);

        /*
         *  If this is an event error response, print a message which includes
         *  the general and specific error codes.
         */

        if (event_p->em_errnum != 0) {
            printf("%s:\t%s State of %s run by %s on node %d unknown (%d, %d)."
                   "\n", sess_p->name, time_stamp, prog_name, user_name, node,
                   event_p->em_generr, event_p->em_specerr);
            continue;
        }

        /*
         *  If the event response was generated for the re-arm predicate, or
         *  the event response was generated for the primary predicate but
         *  it is false, print a message indicating the program associated
         *  with the event is being run.
         */

        if ((event_p->em_event_flags & HA_EM_EVENT_RE_ARM)     ||
            (event_p->em_event_flags & HA_EM_EVENT_PRED_FALSE)) {
            printf("%s:\t%s %s being run by %s on node %d.\n",
                   sess_p->name, time_stamp, prog_name, user_name, node);
            continue;
        }
```

```
        /*
         *  If execution reaches this point, the event response was generated
         *  for the primary predicate, and it is true.  Print a message
         *  indicating the program associated with the event is not being run.
         */

        printf("%s:\t%s %s NOT being run by %s on node %d.\n",
               sess_p->name, time_stamp, prog_name, user_name, node);

    }

    return;
}


/*
 *  The process_response_rerr() function processes registration error
 *  responses.  When an event cannot be registered due to some detected error,
 *  a registration error response is sent to the EMAPI client.  This function
 *  marks the event, as tracked in a program structure, as being unregistered.
 *  This will prevent an attempt to unregister a non-registered event later
 *  in the program.
 */

static
void process_response_rerr(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p)
{
    struct ha_em_rpb_rerr   *error_p;           /* Pointer to error response */
    struct ha_em_rpb_rerr   *last_error_p;      /* Beyond last error response*/
    struct program          *prog_p;            /* Pointer to program struct */

    char prog_name[80];         /* Name of program to which error pertains   */
    char user_name[80];         /* Name of user to which error pertains      */
    int  node;                  /* Node to which error pertains              */


    error_p = rsp_blk->em_resp_blk.em_rpb_rerr;
    last_error_p = error_p + rsp_blk->em_rsp_num_resp;

    for ( ; error_p < last_error_p; error_p++) {

        /*
         *  Find the program structure which describes the program associated
         *  with the event whose registration failed.  The match is made
         *  with event identifiers.
         */

        for (prog_p = progs_p; prog_p < progs_p + progs_elems; prog_p++) {
            if (prog_p->eid == error_p->em_event_id) {
                break;
            }
        }

        if (prog_p == progs_p + progs_elems) {
            fprintf(stderr, "%s:\tUnknown event identifier encountered (0x%x)."
```

```
                    "\n", sess_p->name, error_p->em_event_id);
            exit(1);
        }

        /*
         *  Extract the program name, user name, and node number from the
         *  instantiation vector.
         */

        breakdown_ivector(error_p->em_ivector, prog_name, user_name, &node);

        /*
         *  Print a message about the registration error.
         */

        printf("%s:\tRegistration error for %s run by %s on node %d (%d, %d)."
               "\n", sess_p->name, prog_name, user_name, node,
               error_p->em_generr, error_p->em_specerr);

        /*
         *  Mark the program as not having an associated event registered.
         */

        prog_p->unreged = 1;

    }

    /*
     *  Update the number of events that are registered.
     */

    *reg_cnt_p -= rsp_blk->em_rsp_num_resp;

    return;
}


/*
 *  The process_response_unreg() function processes unregistrations responses.
 *  Note that the unregistration response uses the same type of structure
 *  as the event response, but not all the fields in the structure have
 *  meaningful values for an unregistration response.
 */

static
void process_response_unreg(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p)
{
    struct ha_em_rpb_event  *event_p;           /* Pointer to unreg response */
    struct ha_em_rpb_event  *last_event_p;      /* Beyond last unreg response*/
    struct program          *prog_p;            /* Pointer to program struct */

    char time_stamp[80];                /* Formatted unregister time stamp   */


    event_p = rsp_blk->em_resp_blk.em_rpb_event;
    last_event_p = event_p + rsp_blk->em_rsp_num_resp;
```

```
    for ( ; event_p < last_event_p; event_p++) {

        /*
         *  Find the program structure which describes the program associated
         *  with the event just unregistered.  The match is made
         *  with event identifiers.
         */

        for (prog_p = progs_p; prog_p < progs_p + progs_elems; prog_p++) {
            if (prog_p->eid == event_p->em_event_id) {
                break;
            }
        }

        if (prog_p == progs_p + progs_elems) {
            fprintf(stderr, "%s:\tUnknown event identifier encountered (0x%x)."
                    "\n", sess_p->name, event_p->em_event_id);
            exit(1);
        }

        /*
         *  Format the unregistration time stamp.
         */

        format_timestamp(&event_p->em_timestamp, time_stamp);

        /*
         *  Print a message indicating the program associated with the
         *  unregistered event is no longer being monitored.
         */

        printf("%s:\t%s no longer monitoring %s run by %s on node %d.\n",
               sess_p->name, time_stamp, prog_p->name, prog_p->user,
               prog_p->node);

        /*
         *  Note: em_errnum could indicate an error, but there is no point
         *        looking at it here, since the program would not do anything
         *        differently.
         */

        /*
         *  Mark the program as not having an associated event registered.
         */

        prog_p->unreged = 1;

    }

    /*
     *  Update the number of events that are still registered.
     */

    *reg_cnt_p -= rsp_blk->em_rsp_num_resp;

    return;
}
```

emapi_v01_ex01.c

```
/*
 *  The format_timestamp() function takes a time stamp returned by the
 *  Event Manager and converts it into 24-hour time.
 */

static
void format_timestamp(struct timeval *timestamp_p, char *fmt_timestamp_p)
{
    struct tm   *broken_down_time;


    broken_down_time = localtime((time_t *) &timestamp_p->tv_sec);
    (void) strftime(fmt_timestamp_p, 20, "(%X)", broken_down_time);

    return;
}


/*
 *  The breakdown_ivector() function takes an instantiation vector
 *  associated with the IBM.PSSP.Prog.xpcount resource variable and extracts
 *  from it the program name, user name, and node number.
 */

static
void breakdown_ivector(char *ivector_p,
        char *prog_name_p, char *user_name_p, int *node_p)
{
    char    *value_p;
    size_t  value_len;


    find_ivector_value(ivector_p, "ProgName=", &value_p, &value_len);
    strncpy(prog_name_p, value_p, value_len);
    *(prog_name_p + value_len) = '\0';

    find_ivector_value(ivector_p, "UserName=", &value_p, &value_len);
    strncpy(user_name_p, value_p, value_len);
    *(user_name_p + value_len) = '\0';

    find_ivector_value(ivector_p, "NodeNum=", &value_p, &value_len);
    *node_p = atoi(value_p);

    return;
}


/*
 *  The find_ivector_value() takes an instantiation vector and extracts
 *  a value from it.
 */

static
void find_ivector_value(char *ivector_p, char *name_p,
        char **value_pp, size_t *value_len_p)
{
```

```
    char    *bp;
    char    *ep;
    size_t  len;


    if ((bp = strstr(ivector_p, name_p)) == NULL) {
        fprintf(stderr, "Unexpected instantiation vector encountered.\n");
        exit(1);
    }

    bp += strlen(name_p);

    if ((ep = strchr(bp, ';')) != NULL) {
        len = ep - bp;
    } else {
        len = strlen(bp);
    }

    *value_pp    = bp;
    *value_len_p = len;

    return;
}


/*
 *  The end_session() function terminates a session with the EMAPI by
 *  calling the EMAPI routine ha_em_end_session().
 */

static
void end_session(struct session *sess_p)
{
    struct ha_em_err_blk errb;


    if (ha_em_end_session(sess_p->fd, &errb) == -1) {
        fprintf(stderr, "%s:\tha_em_end_session() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    return;
}
```

The emapi_v01_ex02.c Sample Program

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <time.h>

#include <sys/types.h>
#include <sys/select.h>
#include <sys/time.h>

#include <ha_emapi.h>

/*
 *  emapi_v01_ex02.c
 *
 *  This program presents an example of using the Event Manager Application
 *  Programming Interface (EMAPI).  The program uses the EMAPI to monitor
 *  four programs running on various nodes in two SP partitions.  Since two
 *  SP partitions are involved, the program establishes two sessions with the
 *  EMAPI, one per SP partition.  Since the program is not multi-threaded,
 *  select() is used to multiplex responses from the two EMAPI sessions.
 *
 *  This program provides the same function as the emapi_v01_ex01.c program.
 *  This program differs in that it uses callback routines to receive events.
 *
 *  This program can be compiled with the following command:
 *
 *      cc -O emapi_v01_ex02.c -o emapi_v01_ex02 -lha_em
 */

/*
 *  If a MAX macro isn't defined by one of the included header files, define
 *  it.
 */

#ifndef MAX
#define MAX(x, y)   ((x) > (y) ? (x) : (y))
#endif

/*
 *  The following macros specify the programs this program will monitor through
 *  the EMAPI.
 */

#define PART_1      "k21s"              /* 1st SP partition                  */

#define PROG_1A     "subsys_pgrm1"      /* 1st program in 1st SP partition,  */
#define USER_1A     "root"              /* ... and user running 1st program, */
#define NODE_1A     5                   /* ... and node running 1st program. */
```

```
#define PROG_1B     "subsys_pgrm2"      /* 2nd program in 1st SP partition,  */
#define USER_1B     "root"              /* ... and user running 2nd program, */
#define NODE_1B     6                   /* ... and node running 2nd program. */

#define PART_2      "k21sp2"            /* 2nd SP partition                  */

#define PROG_2A     "subsys_pgrm1"      /* 1st program in 2nd SP partition,  */
#define USER_2A     "root"              /* ... and user running 1st program, */
#define NODE_2A     14                  /* ... and node running 1st program. */

#define PROG_2B     "subsys_pgrm2"      /* 2nd program in 2nd SP partition,  */
#define USER_2B     "root"              /* ... and user running 2nd program, */
#define NODE_2B     15                  /* ... and node running 2nd program. */

/*
 *  This program saves information about each EMAPI session it has established
 *  in a session structure.
 */

struct session {
    char        *name;      /* Name of partition used by session             */
    int          fd;        /* Session file descriptor                       */
    int          restart;   /* Boolean - Session's connection has been lost  */
};

/*
 *  This program saves information about each program it is monitoring through
 *  the EMAPI in a program structure.
 */

struct program {
    char        *name;      /* Name of a program to be monitored             */
    char        *user;      /* Name of user running program                  */
    int          node;      /* Node on which program is running              */
    ha_em_eid_t  eid;       /* Event identifier for program                  */
    int          unreged;   /* Event is unregistered                         */
    struct callback_data *cb_data; /* Callback data allocated for program    */
};

/*
 *  Data for a callback routine is stored in a callback_data structure.
 */

struct callback_data {
    struct session  *sess_p;            /* Session structure pointer         */
    struct program  *prog_p;            /* Program structure pointer         */
    int             *reg_cnt_p;         /* Registration counter pointer      */
};

/*
 *  The terminate_requested global variable is set to a non-zero value when
 *  the user of this program requests that the program be terminated.  The
 *  user requests program termination via the interrupt key (typically Ctrl-C).
 *  The interrupt key causes a SIGINT signal to be delivered to this program.
 *  This program installs a signal handler for the SIGINT signal, which sets
 *  the terminate_requested global variable to a non-zero value when SIGINT
 *  is delivered.
 */
```

emapi_v01_ex02.c

```
static int terminate_requested       = 0;

/*
 *  The unregistrations_requested global variable indicates whether this
 *  program has sent unregistration requests to the Event Manager.
 *  Unregistration requests are sent when the user requests program
 *  termination.
 */

static int unregistrations_requested = 0;

/*
 *  Function prototypes for internal functions.
 */

static void interrupt_catch(int signo);
static void setup_signals(void);
static void start_session(struct session *sess_p);
static void register_for_events(struct session *sess_p,
        struct program *progs_p, int progs_elems, int *reg_cnt_p);
static void unregister_events(struct session *sess_p,
        struct program *progs_p, int progs_elems);
static void select_session(struct session *sess_p,
        fd_set *read_fds_p, int *fd_limit_p, int *timeout_p);
static int session_response_ready(struct session *sess_p, fd_set *read_fds_p);
static void process_response(struct session *sess_p,
         struct program *progs_p, int progs_elems, int *reg_cnt_p);
static void process_response_rerr(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p);
static void event_callback(int sess_fd, struct ha_em_rpb_event *event_p,
                           void *arg);
static void format_timestamp(struct timeval *timestamp_p,
        char *fmt_timestamp_p);
static void breakdown_ivector(char *ivector_p,
        char *prog_name_p, char *user_name_p, int *node_p);
static void find_ivector_value(char *ivector_p, char *name_p,
        char **value_pp, size_t *value_len_p);
static void end_session(struct session *sess_p);


/*
 *  The main() function calls functions to initialize the program, establish
 *  EMAPI sessions, register for events, wait for Event Manager responses,
 *  process Event Manager responses, and respond to the user request for
 *  program termination.  When the program termination request is received,
 *  main() calls functions to send unregistration requests through the EMAPI.
 *  Once Event Manager responses have been received indicating that the
 *  events have been unregistered, the program ends its EMAPI sessions,
 *  and terminates.
 */

int main(int argc, char **argv)
{
    struct session sess1 = {PART_1, -1, 0};
    struct session sess2 = {PART_2, -1, 0};
```

```
struct program progs1[] = {{PROG_1A, USER_1A, NODE_1A, 0, 0, NULL},
                           {PROG_1B, USER_1B, NODE_1B, 0, 0, NULL}};
struct program progs2[] = {{PROG_2A, USER_2A, NODE_2A, 0, 0, NULL},
                           {PROG_2B, USER_2B, NODE_2B, 0, 0, NULL}};

int     progs1_elems = sizeof progs1 / sizeof progs1[0];
int     progs2_elems = sizeof progs2 / sizeof progs2[0];

int     reg_cnt;            /* Count of registered events               */
fd_set  read_fds;           /* select() file descriptor mask            */
int     fd_limit;           /* select() file descriptor limit           */
int     need_timeout;       /* Boolean - select() timeout needed        */
int     rc;                 /* Return code                              */

struct timeval  timeout_value = {15, 0};    /* Fifteen second timeout    */


setup_signals();            /* Initialize signal dispositions           */

start_session(&sess1);      /* Start EMAPI session for 1st SP partition  */
start_session(&sess2);      /* Start EMAPI session for 2nd SP partition  */

/*
 *  Register for events in both EMAPI sessions that will monitor the
 *  programs of interest.  Maintain count of registered events.
 */

reg_cnt = 0;
register_for_events(&sess1, progs1, progs1_elems, &reg_cnt);
register_for_events(&sess2, progs2, progs2_elems, &reg_cnt);

/*
 *  Continue looking for Event Manager responses as long as there are
 *  registered events.
 */

while (reg_cnt > 0) {

    /*
     *  If program termination has been requested, and unregistration
     *  requests have not been sent, send them now.
     */

    if (terminate_requested && !unregistrations_requested) {
        printf("Termination requested.\n");
        unregister_events(&sess1, progs1, progs1_elems);
        unregister_events(&sess2, progs2, progs2_elems);
        unregistrations_requested = 1;
    }

    /*
     *  Construct the parameters to be passed to select(), and then
     *  call select() to wait for Event Manager responses.
     */

    FD_ZERO(&read_fds); fd_limit = 0; need_timeout = 0;
    select_session(&sess1, &read_fds, &fd_limit, &need_timeout);
    select_session(&sess2, &read_fds, &fd_limit, &need_timeout);
```

```
        rc = select(fd_limit + 1, &read_fds, NULL, NULL,
                                  need_timeout ? &timeout_value : NULL);

        if (rc == -1) {                 /* select() indicates error          */
            if (errno == EINTR) {       /* select() interrupted by signal    */
                continue;               /* Return to top of loop             */
            }
            perror("select()");         /* select() really encountered error */
            exit(1);                    /* End program                       */
        }

        /*
         *  If an Event Manager response is present for the 1st EMAPI session,
         *  process it.  Also, maintain the count of registered events.
         */

        if (session_response_ready(&sess1, &read_fds)) {
            process_response(&sess1, progs1, progs1_elems, &reg_cnt);
        }

        /*
         *  If an Event Manager response is present for the 2nd EMAPI session,
         *  process it.  Also, maintain the count of registered events.
         */

        if (session_response_ready(&sess2, &read_fds)) {
            process_response(&sess2, progs2, progs2_elems, &reg_cnt);
        }

    }

    end_session(&sess1);        /* End EMAPI session for 1st SP partition    */
    end_session(&sess2);        /* End EMAPI session for 2nd SP partition    */

    return 0;                   /* Indicate program was successful           */
}


/*
 *  The interrupt_catch() function is installed as the signal handler for the
 *  SIGINT signal.  The SIGINT signal is delivered to this process when the
 *  user presses the interrupt key (usually Ctrl-C).  When this routine is
 *  invoked, it sets the global variable terminate_requested to a non-zero
 *  value, indicating the program is to be terminated.
 */

static
void interrupt_catch(int signo)
{
    terminate_requested = 1;
    return;
}


/*
 *  The setup_signals() function initializes the disposition of a couple of
 *  signals.
```

```
 *
 *  The disposition of the SIGPIPE signal is set such that the signal is
 *  ignored.  When a process writes to a pipe or connection-oriented socket for
 *  which there is no reader, the SIGPIPE signal is delivered to the process.
 *  The default disposition for SIGPIPE is to kill the receiving process.  If
 *  SIGPIPE is ignored, the SIGPIPE signal does not kill the process, and the
 *  system call used to write to the pipe or connection-oriented socket sets
 *  errno to EPIPE and returns an error indication.  It is recommended that
 *  clients of the EMAPI ignore the SIGPIPE signal, since the EMAPI uses
 *  connection-oriented sockets to communicate with the Event Manager.
 *
 *  The interrupt_catch() function is installed as the signal handler for the
 *  SIGINT signal.  See the comments pertaining to the interrupt_catch()
 *  function to find out why this is done.
 */

static
void setup_signals(void)
{
    struct sigaction    sa;


    sa.sa_handler = SIG_IGN;               /* SIGPIPE to be ignored         */
    sigemptyset(&sa.sa_mask);              /* No signals to mask off        */
    sa.sa_flags = 0;                       /* No flags needed               */

    if (sigaction(SIGPIPE, &sa, NULL) == -1) {  /* Change SIGPIPE disposition*/
        perror("sigaction(SIGPIPE)");
        exit(1);
    }

    sa.sa_handler = interrupt_catch;       /* SIGINT signal handler         */
    sigemptyset(&sa.sa_mask);              /* No signals to mask off        */
    sa.sa_flags = SA_RESTART;              /* Restart interrupted syscalls  */

    if (sigaction(SIGINT, &sa, NULL) == -1) {   /* Change SIGINT disposition */
        perror("sigaction(SIGINT)");
        exit(1);
    }

    return;
}


/*
 *  The start_session() function establishes a session with the EMAPI by
 *  calling the EMAPI routine ha_em_start_session().  If a session cannot be
 *  established, the program is terminated.
 */

static
void start_session(struct session *sess_p)
{
    struct ha_em_err_blk errb;                 /* EMAPI error block         */


    /*
     *  The ha_em_start_session() routine receives the name of the partition
```

```
     *  with which a session is to be established, and returns a file
     *  descriptor with which the session can be used.
     */

    sess_p->fd = ha_em_start_session(sess_p->name, &errb);

    /*
     *  If ha_em_start_session() indicates a session could not be established,
     *  print the error information returned by the routine, and exit the
     *  program.
     */

    if (sess_p->fd == -1) {
        fprintf(stderr, "%s:\tha_em_start_session() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    /*
     *  A session has been established with the EMAPI, and the session's file
     *  descriptor has been saved in the session structure passed to this
     *  routine.  Set the flag in the session structure that indicates the
     *  session does not need to be restarted.
     */

    sess_p->restart = 0;

    return;
}


/*
 *  The register_for_events() function uses the EMAPI routine
 *  ha_em_send_command() to request the registration of events pertaining to
 *  programs of interest that are in the same SP partition.
 *
 *  The resource variable name specified is "IBM.PSSP.Prog.xpcount".  This is a
 *  state variable whose value indicates whether or not processes are executing
 *  a specific program.  The instantiation vector for this resource variable
 *  specifies the program name, the name of the user running the program, and
 *  the node on which the program is running.  The resource variable's value
 *  is a structured byte string of three fields.  Field 0 is of type long; its
 *  value is the number of processes currently running the specified program
 *  for the specified user on the specified node.  Field 1 is also of type
 *  long; its value is the previous number of processes running the program.
 *  Field 2 is of type character string; its value is a comma separated list
 *  of the PIDs of the processes running the program.
 *
 *  When the events are registered, two predicates are specified.  The primary
 *  predicate, "X@0 == 0", specifies that an event is to be generated when
 *  field 0 of the resource variable's structured byte string has a value of 0.
 *  Taking into account the semantics of the resource variable, the predicate
 *  causes an event to be generated when no processes are running the specified
 *  program for the specified user on the specified node.  The re-arm
 *  predicate, "X@0 > 0", specifies that after the primary predicate has
 *  generated an event, the primary predicate is to be re-armed once a process
 *  starts running the specified program for the specified user on the
 *  specified node.  Since the HA_EM_CMD_REG2 command is specified, the re-arm
```

```
 *  predicate will generate events.
 *
 *  Notice that when events are registered, the HA_EM_SCMD_REVAL subcommand is
 *  specified.  As a result, the initial value of the programs being monitored
 *  will be returned.
 */

static
void register_for_events(struct session *sess_p,
        struct program *progs_p, int progs_elems, int *reg_cnt_p)
{
    int                     num_events;      /* Number of events to register */
    size_t                  alloc_size;      /* Size of memory to allocate   */
    struct ha_em_cmd_blk    *cmd_blk;        /* Pointer to command block     */
    struct ha_em_rb_reg     *re;             /* Pointer to registered event  */
    int                     i;               /* Index                        */
    struct ha_em_err_blk    errb;            /* EMAPI error block            */
    int                     rc;              /* Return code                  */
    char                    ivector_buf[80]; /* Instantiation vector buffer  */
    struct callback_data    *cd_p;           /* Pointer to callback data     */


    num_events = progs_elems;           /* Register one event per program    */

    /*
     *  Allocate enough memory to hold the command block to be given to the
     *  EMAPI.  The ha_em_cmd_blk structure contains one ha_em_rb_reg
     *  structure.  Additional space must be allocated when more than one
     *  event is to be registered.
     */

    alloc_size = sizeof(struct ha_em_cmd_blk) +
                    ((num_events - 1) * sizeof(struct ha_em_rb_reg));

    if ((cmd_blk = malloc(alloc_size)) == NULL) {
        perror("malloc()");
        exit(1);
    }

    /*
     *  Fill in the em_rb_reg array entries.  There is one entry per event to
     *  be registered.
     */

    for (i = 0; i < num_events; i++) {

        re = &cmd_blk->em_res_blk.em_rb_reg[i];

        re->em_name = "IBM.PSSP.Prog.xpcount";   /* Resource variable name   */

        sprintf(ivector_buf, "ProgName=%s;UserName=%s;NodeNum=%d",
                progs_p[i].name, progs_p[i].user, progs_p[i].node);
        re->em_ivector = strdup(ivector_buf);    /* Instantiation vector     */

        if (re->em_ivector == NULL) {
            perror("strdup()");
            exit(1);
        }
```

```
    re->em_pred   = "X@0 == 0";              /* Predicate               */
    re->em_rapred = "X@0 > 0";               /* Re-arm predicate        */

    if ((cd_p = malloc(sizeof(struct callback_data))) == NULL) {
        perror("malloc()");
        exit(1);
    }

    cd_p->sess_p    = sess_p;        /* Callback gets session pointer    */
    cd_p->prog_p    = &progs_p[i];   /* Callback gets program pointer    */
    cd_p->reg_cnt_p = reg_cnt_p;     /* Callback gets reg. count pointer */

    re->em_cb     = event_callback;  /* Callback routine                */
    re->em_cb_arg = cd_p;            /* Callback routine args.          */

    progs_p[i].cb_data = cd_p;       /* Save pointer to callback data    */
}

/*
 *  Fill in command block header, specifying number of elements in
 *  em_rb_reg array, command, and subcommand.
 */

cmd_blk->em_cmd_num_elem = num_events;
cmd_blk->em_cmd          = HA_EM_CMD_REG2;
cmd_blk->em_subcmd       = HA_EM_SCMD_REVAL;

/*
 *  Send the command.  A more elaborate program might check for the
 *  HA_EM_ECONNREFUSED Event Manager error number when an error is
 *  returned, and attempt to restart the session.  But, this program
 *  just terminates if ha_em_send_command() detects a lost connection.
 */

rc = ha_em_send_command(sess_p->fd, cmd_blk, &errb);

if (rc == -1) {
    fprintf(stderr, "%s:\tha_em_send_command() returned EM errno %d:\n%s",
            sess_p->name, errb.em_errno, errb.em_errmsg);
    exit(1);
}

/*
 *  Save the event identifiers assigned to the events just registered.
 */

for (i = 0; i < num_events; i++) {
    re = &cmd_blk->em_res_blk.em_rb_reg[i];
    progs_p[i].eid     = re->em_event_id;
    progs_p[i].unreged = 0;
    free(re->em_ivector);
}

free(cmd_blk);

*reg_cnt_p += num_events;           /* Update count of registered events */
```

```
    return;
}


/*
 *  The unregister_events() function uses the EMAPI routine
 *  ha_em_send_command() to request the unregistration of events pertaining to
 *  programs of interest that are in the same SP partition.
 */

static
void unregister_events(struct session *sess_p,
        struct program *progs_p, int progs_elems)
{
    int                     num_events;      /* Number of events to unreg.   */
    size_t                  alloc_size;      /* Size of memory to allocate   */
    struct ha_em_cmd_blk    *cmd_blk;        /* Pointer to command block     */
    ha_em_eid_t             *re;             /* Pointer to unreg. event      */
    int                     i;               /* Index                        */
    struct ha_em_err_blk    errb;            /* EMAPI error block            */
    int                     rc;              /* Return code                  */


    /*
     *  Allocate enough memory to hold the command block to be given to the
     *  EMAPI.  The ha_em_cmd_blk structure contains one ha_em_eid_t
     *  element.  Additional space must be allocated when more than one
     *  event is to be unregistered.
     */

    alloc_size = sizeof(struct ha_em_cmd_blk) +
                    ((progs_elems - 1) * sizeof(ha_em_eid_t));

    if ((cmd_blk = malloc(alloc_size)) == NULL) {
        perror("malloc()");
        exit(1);
    }

    /*
     *  Fill in the event identifier array entries.  There is one entry per
     *  event to be unregistered.  Only unregister events that are not already
     *  unregistered.
     */

    num_events = 0;

    for (i = 0; i < progs_elems; i++) {
        if (!progs_p[i].unreged) {
            re  = &cmd_blk->em_res_blk.em_rb_unreg[num_events];
            *re = progs_p[i].eid;
            num_events++;
        }
    }

    /*
     *  Fill in command block header, specifying the number of elements in
     *  the event identifier array, and the unregister command.
     */
```

```
    cmd_blk->em_cmd_num_elem = num_events;
    cmd_blk->em_cmd          = HA_EM_CMD_UNREG;

    /*
     *  Send the unregister command.  A more elaborate program might check for
     *  the HA_EM_ECONNREFUSED Event Manager error number when an error is
     *  returned, and attempt to restart the session.  But, this program
     *  just terminates if ha_em_send_command() detects a lost connection.
     */

    rc = ha_em_send_command(sess_p->fd, cmd_blk, &errb);

    if (rc == -1) {
        fprintf(stderr, "%s:\tha_em_send_command() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    free(cmd_blk);

    return;
}


/*
 *  The select_session() routine determines how the specified session affects
 *  the next call to select().  If the session is marked as needing to be
 *  restarted, ha_em_restart_session() is called to try to restart the session.
 *  If the restart attempt fails, the next call to select() should specify
 *  a timeout period - so the restart can be attempted again.  If the restart
 *  attempt succeeds, the new session file descriptor is saved, and the session
 *  is marked as not needing to be restarted.  If the session did not need
 *  to be restarted, or had been successfully restarted, the next call to
 *  select() should examine the session's file descriptor.
 */

static
void select_session(struct session *sess_p,
        fd_set *read_fds_p, int *fd_limit_p, int *timeout_p)
{
    struct ha_em_err_blk    errb;           /* EMAPI error block             */
    int                     new_fd;         /* New session file descriptor   */


    /*
     *  If the specified session needs to be restarted, try to do so.
     */

    if (sess_p->restart) {

        new_fd = ha_em_restart_session(sess_p->fd, &errb);

        if (new_fd == -1) {

            /*
             *  The session could not be restarted.  If the Event Manager
             *  error number indicates connection refused, indicate the
```

```
             *  next select() should specify a timeout value (so the restart
             *  can be tried again later), and return without putting the
             *  session's file descriptor in the select() file descriptor
             *  mask.
             */

            if (errb.em_errno == HA_EM_ECONNREFUSED) {
                *timeout_p = 1;
                return;
            }

            /*
             *  If execution reaches here, the session restart attempt failed
             *  for an unexpected reason; terminate the program.
             */

            fprintf(stderr, "%s:\tha_em_restart_session() returned EM errno "
                    "%d:\n%s", sess_p->name, errb.em_errno, errb.em_errmsg);
            exit(1);
        }

        /*
         *  If execution reaches here, a session restart was attempted and
         *  was successful.  Save the new session file descriptor, and indicate
         *  that the session no longer needs to be restarted.
         */

        sess_p->fd = new_fd;
        sess_p->restart = 0;

    }

    /*
     *  If execution reaches here, the specified session is not known to be
     *  in need of a restart.  Put the session's file descriptor in the
     *  select() file descriptor mask.  Also, adjust the maximum file
     *  descriptor to select, if appropriate.
     */

    FD_SET(sess_p->fd, read_fds_p);
    *fd_limit_p = MAX(sess_p->fd, *fd_limit_p);

    return;
}


/*
 *  The session_response_ready() function determines if a select() file
 *  descriptor mask indicates that a session's file descriptor is ready for
 *  reading.
 */

static
int session_response_ready(struct session *sess_p, fd_set *read_fds_p)
{
        return FD_ISSET(sess_p->fd, read_fds_p);
}
```

emapi_v01_ex02.c

```
/*
 *  The process_response() function processes an Event Manager response
 *  in the specified EMAPI session.  First, ha_em_receive_response() is called
 *  to receive the response.  If ha_em_receive_response() indicates that the
 *  session's connection with the Event Manager has been lost, the session
 *  is marked as being in need of a restart.  Any other error will terminate
 *  the program.  The ha_em_receive_response() function may indicate that
 *  there really isn't any response for the EMAPI client to process at this
 *  time.  In that case, this function just returns.  If a response has
 *  been received, it is processed depending on what type of response it is.
 */

static
void process_response(struct session *sess_p,
         struct program *progs_p, int progs_elems, int *reg_cnt_p)
{
    struct ha_em_rsp_blk    *rsp_blk;   /* Pointer to the response block     */
    int                     rc;         /* Return code                       */
    struct ha_em_err_blk    errb;       /* Event Manager error block         */
    int                     i;          /* Index                             */


    /*
     *  Receive response from session, if there is one to receive.
     */

    rc = ha_em_receive_response(sess_p->fd, &rsp_blk, &errb);

    if (rc == -1) {

        /*
         *  If the connection with the Event Manager has been lost,
         *  mark the session as needing to be restarted, and return.
         */

        if (errb.em_errno == HA_EM_ECONNLOST) {
            printf("%s:\tconnection to session lost.\n", sess_p->name);
            sess_p->restart = 1;
            return;
        }

        /*
         *  Some error has occurred other than the loss of the connection.
         *  Terminate the program.
         */

        fprintf(stderr, "%s:\tha_em_receive_response() returned EM errno %d:\n"
                "%s", sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);

    }

    /*
     *  If the ha_em_receive_response() routine returned zero,
     *  there is no response for the EMAPI client (this program) to process
     *  at this time.  The response may have been for the EMAPI itself,
     *  a full response may not be available yet, or the response may have
```

```
     *  been handled with the callback routine.  Just return.
     */

    if (rc == 0) {
        return;
    }

    /*
     *  A response for the EMAPI client (this program) has been returned.
     *  The response buffer is pointed to by the rsp_blk variable.  Appropriate
     *  processing for the response depends on the command type.  Note that
     *  since all event registrations in this program specify the use of a
     *  callback routine, event responses and event unregistration responses
     *  are not expected by this function.
     */

    switch (rsp_blk->em_cmd) {

        case HA_EM_CMD_RERR:            /* REG  event registration error     */
        case HA_EM_CMD_R2ERR:           /* REG2 event registration error     */
            process_response_rerr(sess_p, progs_p, progs_elems, rsp_blk,
                                  reg_cnt_p);
            break;

        default:                        /* Unexpected response               */
            fprintf(stderr, "%s:\tProgram received unexpected command "
                    "response: %d.\n", sess_p->name, rsp_blk->em_cmd);
            exit(1);
            break;

    }

    /*
     *  The EMAPI client (this program) must free the memory associated with
     *  the returned response block when it is no longer needed.
     */

    free(rsp_blk);

    return;
}


/*
 *  The process_response_rerr() function processes registration error
 *  responses.  When an event cannot be registered due to some detected error,
 *  a registration error response is sent to the EMAPI client.  This function
 *  marks the event, as tracked in a program structure, as being unregistered.
 *  This will prevent an attempt to unregister a non-registered event later
 *  in the program.
 */

static
void process_response_rerr(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p)
{
    struct ha_em_rpb_rerr   *error_p;           /* Pointer to error response */
```

emapi_v01_ex02.c

```
struct ha_em_rpb_rerr   *last_error_p;      /* Beyond last error response*/
struct program          *prog_p;            /* Pointer to program struct */

char prog_name[80];         /* Name of program to which error pertains   */
char user_name[80];         /* Name of user to which error pertains      */
int  node;                  /* Node to which error pertains              */


error_p = rsp_blk->em_resp_blk.em_rpb_rerr;
last_error_p = error_p + rsp_blk->em_rsp_num_resp;

for ( ; error_p < last_error_p; error_p++) {

    /*
     *  Find the program structure which describes the program associated
     *  with the event whose registration failed.  The match is made
     *  with event identifiers.
     */

    for (prog_p = progs_p; prog_p < progs_p + progs_elems; prog_p++) {
        if (prog_p->eid == error_p->em_event_id) {
            break;
        }
    }

    if (prog_p == progs_p + progs_elems) {
        fprintf(stderr, "%s:\tUnknown event identifier encountered (0x%x)."
                "\n", sess_p->name, error_p->em_event_id);
        exit(1);
    }

    /*
     *  Extract the program name, user name, and node number from the
     *  instantiation vector.
     */

    breakdown_ivector(error_p->em_ivector, prog_name, user_name, &node);

    /*
     *  Print a message about the registration error.
     */

    printf("%s:\tRegistration error for %s run by %s on node %d (%d, %d)."
           "\n", sess_p->name, prog_name, user_name, node,
           error_p->em_generr, error_p->em_specerr);

    /*
     *  Mark the program as not having an associated event registered.
     */

    prog_p->unreged = 1;

    /*
     *  Free the callback data allocated for this program.
     */

    free(prog_p->cb_data);
    prog_p->cb_data = NULL;
```

```
    }

    /*
     *  Update the number of events that are registered.
     */

    *reg_cnt_p -= rsp_blk->em_rsp_num_resp;

    return;
}


/*
 *  The event_callback() function is called whenever an event response has been
 *  received from the Event Manager.  Several points should be kept in mind:
 *
 *      - The event response may indicate that an unregistration request has
 *        been completed.  The HA_EM_EVENT_UNREG flag indicates the event
 *        response is actually an unregistration response.
 *
 *      - An event response may be an error response.  An event error response
 *        does not indicate that a registered event has occurred.  Instead, it
 *        indicates the Event Manager no longer knows the current value of
 *        the associated resource variable, and cannot generate events for
 *        the resource variable.  This may be a temporary condition.  If the
 *        Event Manager later obtains the current value of the associated
 *        resource variable, an event will be generated (possibly indicating
 *        that the predicate is false).
 *
 *      - The event for which an event error response is generated is still
 *        registered.  There is no need to re-register the event.
 *
 *      - Since events are registered by this program using the HA_EM_CMD_REG2
 *        command, the re-arm predicates will generate events.  Therefore, the
 *        HA_EM_EVENT_RE_ARM flag must be tested to see if the predicate or
 *        the re-arm predicate generated the event.
 *
 *      - It is possible for an event to be delivered indicating the predicate
 *        is false.  This can happen for two reasons.  First, when events are
 *        registered by this program, the HA_EM_SCMD_REVAL subcommand is
 *        specified.  That subcommand requests the initial value of the
 *        associated resource variable.  Second, the current value of the
 *        associated resource variable is returned through an event once the
 *        Event Manager obtains the current value of a resource variable after
 *        an error has occurred.
 *
 *      - If an event response does not have the HA_EM_EVENT_RE_ARM nor
 *        HA_EM_EVENT_PRED_FALSE flags set, the event response indicates that
 *        the event's predicate is true.
 */

static
void event_callback(int sess_fd, struct ha_em_rpb_event *event_p, void *arg)
{
    struct callback_data *cd_p; /* Callback routine data pointer             */
    struct session  *sess_p;    /* Session structure pointer                 */
    struct program  *prog_p;    /* Program structure pointer                 */
```

emapi_v01_ex02.c

```
int             *reg_cnt_p; /* Registration counter pointer             */

char time_stamp[80];        /* Formatted event time stamp               */


/*
 *  Get the callback routine's arguments.
 */

cd_p = (struct callback_data *)arg;

sess_p    = cd_p->sess_p;
prog_p    = cd_p->prog_p;
reg_cnt_p = cd_p->reg_cnt_p;

/*
 *  Format the time stamp.
 */

format_timestamp(&event_p->em_timestamp, time_stamp);

if (event_p->em_event_flags & HA_EM_EVENT_UNREG) {

    /*
     *  If execution reaches this point, the event indicates an event
     *  unregistration has completed.
     *  Print a message indicating the program associated with the
     *  unregistered event is no longer being monitored.
     */

    printf("%s:\t%s no longer monitoring %s run by %s on node %d.\n",
           sess_p->name, time_stamp, prog_p->name, prog_p->user,
           prog_p->node);

    /*
     *  Note: em_errnum could indicate an error, but there is no point
     *        looking at it here, since the program would not do anything
     *        differently.
     */

    /*
     *  Mark the program as not having an associated event registered.
     */

    prog_p->unreged = 1;

    /*
     *  Indicate this program has one fewer registered event.
     */

    (*reg_cnt_p)--;

    /*
     *  The callback routine will no longer be called about this event,
     *  so free the data that was allocated specifically for this event.
     */

    free(prog_p->cb_data);
```

```
        prog_p->cb_data = NULL;

        return;
    }

    /*
     *  If execution reaches this point, the event response is not an
     *  unregistration response.
     */

     /*
      *  If this is an event error response, print a message which includes
      *  the general and specific error codes.
      */

     if (event_p->em_errnum != 0) {
         printf("%s:\t%s State of %s run by %s on node %d unknown (%d, %d).\n",
                sess_p->name, time_stamp, prog_p->name, prog_p->user,
                prog_p->node, event_p->em_generr, event_p->em_specerr);
         return;
     }

     /*
      *  If the event response was generated for the re-arm predicate, or
      *  the event response was generated for the primary predicate but
      *  it is false, print a message indicating the program associated
      *  with the event is being run.
      */

     if ((event_p->em_event_flags & HA_EM_EVENT_RE_ARM)     ||
         (event_p->em_event_flags & HA_EM_EVENT_PRED_FALSE)) {
         printf("%s:\t%s %s being run by %s on node %d.\n",
                sess_p->name, time_stamp, prog_p->name, prog_p->user,
                prog_p->node);
         return;
     }

     /*
      *  If execution reaches this point, the event response was generated
      *  for the primary predicate, and it is true.  Print a message
      *  indicating the program associated with the event is not being run.
      */

     printf("%s:\t%s %s NOT being run by %s on node %d.\n",
            sess_p->name, time_stamp, prog_p->name, prog_p->user,
            prog_p->node);

    return;
}


/*
 *  The format_timestamp() function takes a time stamp returned by the
 *  Event Manager and converts it into 24-hour time.
 */
```

emapi_v01_ex02.c

```
static
void format_timestamp(struct timeval *timestamp_p, char *fmt_timestamp_p)
{
    struct tm   *broken_down_time;


    broken_down_time = localtime((time_t *) &timestamp_p->tv_sec);
    (void) strftime(fmt_timestamp_p, 20, "(%X)", broken_down_time);

    return;
}


/*
 *  The breakdown_ivector() function takes an instantiation vector
 *  associated with the IBM.PSSP.Prog.xpcount resource variable and extracts
 *  from it the program name, user name, and node number.
 */

static
void breakdown_ivector(char *ivector_p,
        char *prog_name_p, char *user_name_p, int *node_p)
{
    char    *value_p;
    size_t  value_len;


    find_ivector_value(ivector_p, "ProgName=", &value_p, &value_len);
    strncpy(prog_name_p, value_p, value_len);
    *(prog_name_p + value_len) = '\0';

    find_ivector_value(ivector_p, "UserName=", &value_p, &value_len);
    strncpy(user_name_p, value_p, value_len);
    *(user_name_p + value_len) = '\0';

    find_ivector_value(ivector_p, "NodeNum=", &value_p, &value_len);
    *node_p = atoi(value_p);

    return;
}


/*
 *  The find_ivector_value() takes an instantiation vector and extracts
 *  a value from it.
 */

static
void find_ivector_value(char *ivector_p, char *name_p,
        char **value_pp, size_t *value_len_p)
{
    char    *bp;
    char    *ep;
    size_t  len;


    if ((bp = strstr(ivector_p, name_p)) == NULL) {
        fprintf(stderr, "Unexpected instantiation vector encountered.\n");
```

```
        exit(1);
    }

    bp += strlen(name_p);

    if ((ep = strchr(bp, ';')) != NULL) {
        len = ep - bp;
    } else {
        len = strlen(bp);
    }

    *value_pp    = bp;
    *value_len_p = len;

    return;
}


/*
 *  The end_session() function terminates a session with the EMAPI by
 *  calling the EMAPI routine ha_em_end_session().
 */

static
void end_session(struct session *sess_p)
{
    struct ha_em_err_blk errb;


    if (ha_em_end_session(sess_p->fd, &errb) == -1) {
        fprintf(stderr, "%s:\tha_em_end_session() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    return;
}
```

The emapi_v01_ex03.c Sample Program

```
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <time.h>

#include <ha_emapi.h>

/*
 *  emapi_v01_ex03.c
 *
 *  This program presents an example of using the Event Manager Application
 *  Programming Interface (EMAPI).  The program uses the EMAPI to monitor
 *  four programs running on various nodes in two SP partitions.  Since two
 *  SP partitions are involved, the program establishes two sessions with the
 *  EMAPI, one per SP partition.  Each session is managed by a separate thread.
 *
 *  This program can be compiled with the following command:
 *
 *      cc_r -O emapi_v01_ex03.c -o emapi_v01_ex03 -lha_em_r -lpthreads -lc_r
 */

/*
 *  The following macros specify the programs this program will monitor through
 *  the EMAPI.
 */

#define PART_1      "k21s"              /* 1st SP partition                 */

#define PROG_1A     "subsys_pgrm1"      /* 1st program in 1st SP partition,  */
#define USER_1A     "root"              /* ... and user running 1st program, */
#define NODE_1A     5                   /* ... and node running 1st program. */

#define PROG_1B     "subsys_pgrm2"      /* 2nd program in 1st SP partition,  */
#define USER_1B     "root"              /* ... and user running 2nd program, */
#define NODE_1B     6                   /* ... and node running 2nd program. */

#define PART_2      "k21sp2"            /* 2nd SP partition                 */

#define PROG_2A     "subsys_pgrm1"      /* 1st program in 2nd SP partition,  */
#define USER_2A     "root"              /* ... and user running 1st program, */
#define NODE_2A     14                  /* ... and node running 1st program. */

#define PROG_2B     "subsys_pgrm2"      /* 2nd program in 2nd SP partition,  */
#define USER_2B     "root"              /* ... and user running 2nd program, */
#define NODE_2B     15                  /* ... and node running 2nd program. */

/*
 *  The STRERROR_BUF and STRERROR macros make it a little more convenient to
 *  use the thread safe version of strerror(), strerror_r().
 */
```

```
#define STRERROR_BUF    char strerror_buf[256]

#define STRERROR(en) (                                                  \
    strerror_r(en, strerror_buf, sizeof strerror_buf) == 0              \
        ? strerror_buf : "Unknown error"                                \
)

/*
 *  This program saves information about each EMAPI session it has established
 *  in a session structure.
 */

struct session {
    char        *name;      /* Name of partition used by session            */
    int          fd;        /* Session file descriptor                      */
    int          restart;   /* Boolean - Session's connection has been lost  */
};

/*
 *  This program saves information about each program it is monitoring through
 *  the EMAPI in a program structure.
 */

struct program {
    char        *name;      /* Name of a program to be monitored            */
    char        *user;      /* Name of user running program                 */
    int          node;      /* Node on which program is running             */
    ha_em_eid_t  eid;       /* Event identifier for program                 */
    int          unreged;   /* Event is unregistered                        */
};

/*
 *  The thread_data structure is used by the initial thread to pass data to
 *  a secondary thread when it is created.
 */

struct thread_data {
    struct session  *sess;      /* EMAPI session the thread is to establish  */
    struct program  *progs;     /* Programs the thread is to monitor         */
    int              progs_cnt; /* Number of programs thread is to monitor   */
};

/*
 *  Following are handles for threads created by the initial thread.
 */

pthread_t  thread1;             /* Handle to thread 1 - a program monitor    */
pthread_t  thread2;             /* Handle to thread 2 - a program monitor    */
pthread_t  thread3;             /* Handle to thread 3 - cancellation monitor */

/*
 *  Function prototypes for internal functions.
 */

static void setup_signals(void);
static void create_thread(pthread_t *thread_p, void *(* thread_rtn)(void *),
```

emapi_v01_ex03.c

```
                          void *data_p);
static void cancel_thread(pthread_t thread);
static void wait_for_thread(pthread_t thread);
static void *program_thread_main(void *parm_p);
static void program_thread_main_cleanup(void *parm_p);
static void *cancel_thread_main(void *parm_p);
static void start_session(struct session *sess_p);
static void register_for_events(struct session *sess_p,
        struct program *progs_p, int progs_elems, int *reg_cnt_p);
static void ensure_session_connected(struct session *sess_p);
static void process_response(struct session *sess_p,
         struct program *progs_p, int progs_elems, int *reg_cnt_p);
static void process_response_reg(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk);
static void process_response_rerr(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p);
static void format_timestamp(struct timeval *timestamp_p,
        char *fmt_timestamp_p);
static void breakdown_ivector(char *ivector_p,
        char *prog_name_p, char *user_name_p, int *node_p);
static void find_ivector_value(char *ivector_p, char *name_p,
        char **value_pp, size_t *value_len_p);
static void end_session(struct session *sess_p);


/*
 *  The main() function is executed by this program's initial thread.
 *  The function creates three other threads.  The first two threads
 *  use the EMAPI to monitor other programs.  One thread is used per
 *  EMAPI session, and one EMAPI session is used per SP partition.
 *  The third thread waits for the user to request the termination of this
 *  program.  When the third thread detects the user would like this program
 *  to terminate, it cancels the two threads using the EMAPI.  When those
 *  two threads terminate, the initial thread cancels the third thread, waits
 *  for the third thread to terminate, and terminates the program.
 */

int main(int argc, char **argv)
{
    struct session sess1 = {PART_1, -1, 0};
    struct session sess2 = {PART_2, -1, 0};

    struct program progs1[] = {{PROG_1A, USER_1A, NODE_1A, 0, 0},
                               {PROG_1B, USER_1B, NODE_1B, 0, 0}};
    struct program progs2[] = {{PROG_2A, USER_2A, NODE_2A, 0, 0},
                               {PROG_2B, USER_2B, NODE_2B, 0, 0}};

    int     progs1_elems = sizeof progs1 / sizeof progs1[0];
    int     progs2_elems = sizeof progs2 / sizeof progs2[0];

    struct thread_data td1 = {&sess1, progs1, progs1_elems};
    struct thread_data td2 = {&sess2, progs2, progs2_elems};


    setup_signals();                    /* Initialize signal dispositions    */
```

```
    /*
     *  Create 2 threads that will monitor programs with the EMAPI.
     */

    create_thread(&thread1, program_thread_main, &td1);  /* Create thread 1  */
    create_thread(&thread2, program_thread_main, &td2);  /* Create thread 2  */

    /*
     *  Create a thread that will wait for the user to request termination.
     */

    create_thread(&thread3, cancel_thread_main,  NULL);  /* Create thread 3  */

    /*
     *  Wait for the 2 threads that are monitoring programs to end.  They
     *  will end due to errors or they will be canceled by thread 3 when
     *  the user requests termination.
     */

    wait_for_thread(thread1);           /* Wait for thread 1 to terminate    */
    wait_for_thread(thread2);           /* Wait for thread 2 to terminate    */

    /*
     *  Cancel the thread waiting for the user to request termination, and
     *  then wait for that thread to terminate.
     */

    cancel_thread(thread3);             /* Cancel thread 3                   */
    wait_for_thread(thread3);           /* Wait for thread 3 to terminate    */

    return 0;                           /* Indicate program was successful   */
}


/*
 *  The setup_signals() function initializes the disposition of a couple of
 *  signals.
 *
 *  The disposition of the SIGPIPE signal is set such that the signal is
 *  ignored.  When a process writes to a pipe or connection-oriented socket for
 *  which there is no reader, the SIGPIPE signal is delivered to the process.
 *  The default disposition for SIGPIPE is to kill the receiving process.  If
 *  SIGPIPE is ignored, the SIGPIPE signal does not kill the process, and the
 *  system call used to write to the pipe or connection-oriented socket sets
 *  errno to EPIPE and returns an error indication.  It is recommended that
 *  clients of the EMAPI ignore the SIGPIPE signal, since the EMAPI uses
 *  connection-oriented sockets to communicate with the Event Manager.
 *
 *  The SIGINT signal is blocked in the initial thread's signal mask.
 *  Threads created by the initial thread will inherit the signal mask from
 *  the initial thread.  One thread will wait for the SIGINT signal
 *  to be delivered by using sigwait().
 */

static
void setup_signals(void)
{
    struct sigaction    sa;             /* Signal action description         */
```

emapi_v01_ex03.c

```
    int                 rc;             /* Return code (error number)        */
    STRERROR_BUF;                       /* Buffer for strerror_r()           */
    sigset_t            sigset;         /* Signal mask to block SIGINT       */


    sa.sa_handler = SIG_IGN;            /* SIGPIPE to be ignored             */
    sigemptyset(&sa.sa_mask);           /* No signals to mask off            */
    sa.sa_flags = 0;                    /* No flags needed                   */

    if (sigaction(SIGPIPE, &sa, NULL) == -1) {  /* Change SIGPIPE disposition*/
        perror("sigaction(SIGPIPE)");
        exit(1);
    }

    sigemptyset(&sigset);               /* Initialize signal set             */
    sigaddset(&sigset, SIGINT);         /* Add SIGINT to signal set          */

    rc = sigthreadmask(SIG_BLOCK, &sigset, NULL);  /* Block SIGINT signal    */
    if (rc != 0) {
        fprintf(stderr, "sigthreadmask(): %s\n", STRERROR(rc));
        exit(1);
    }

    return;
}


/*
 *  The create_thread() function creates a thread, passing along the specified
 *  main function address and parameter address.  The function returns the
 *  thread's handle.  The thread is created such that it can be joined later
 *  on.
 */

static
void create_thread(pthread_t *thread_p, void *(* thread_rtn)(void *),
                   void *data_p)
{
    pthread_attr_t  thread_attr;        /* Thread attributes                 */
    int             rc;                 /* Return code (error number)        */
    STRERROR_BUF;                       /* Buffer for strerror_r()           */


    rc = pthread_attr_init(&thread_attr);
    if (rc != 0) {
        fprintf(stderr, "pthread_attr_init(): %s\n", STRERROR(rc));
        exit(1);
    }

    rc = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_UNDETACHED);
    if (rc != 0) {
        fprintf(stderr, "pthread_attr_setdetachstate(): %s\n",
                STRERROR(rc));
        exit(1);
    }

    rc = pthread_create(thread_p, &thread_attr, thread_rtn, data_p);
    if (rc != 0) {
```

```
        fprintf(stderr, "pthread_create(): %s\n", STRERROR(rc));
        exit(1);
    }

    rc = pthread_attr_destroy(&thread_attr);
    if (rc != 0) {
        fprintf(stderr, "pthread_attr_destroy(): %s\n", STRERROR(rc));
        exit(1);
    }

    return;
}


/*
 *  The cancel_thread() function requests the cancellation of the specified
 *  thread.
 */

static
void cancel_thread(pthread_t thread)
{
    int             rc;                 /* Return code (error number)        */
    STRERROR_BUF;                       /* Buffer for strerror_r()           */


    rc = pthread_cancel(thread);
    if (rc != 0) {
        fprintf(stderr, "pthread_cancel(): %s\n", STRERROR(rc));
        exit(1);
    }

    return;
}


/*
 *  The wait_for_thread() function waits for the specified thread to
 *  terminate.
 */

static
void wait_for_thread(pthread_t thread)
{
    int             rc;                 /* Return code (error number)        */
    STRERROR_BUF;                       /* Buffer for strerror_r()           */


    rc = pthread_join(thread, NULL);
    if (rc != 0) {
        fprintf(stderr, "pthread_join(): %s\n", STRERROR(rc));
        exit(1);
    }

    return;
}
```

emapi_v01_ex03.c

```
/*
 *  The program_thread_main() function is the main function for threads that
 *  are created to monitor programs in a SP partition.  The function is told
 *  what programs are to be monitored in what partition through the function
 *  parameter, which is a pointer to a thread_data structure.  The function
 *  calls functions to establish an EMAPI session, register for events, wait
 *  for process Event Manager responses, and cleanup when the thread is
 *  canceled.  The thread cleanup function that is installed ends the EMAPI
 *  session that was established for the thread.
 */

static
void *program_thread_main(void *parm_p)
{
    struct thread_data *td;     /* Thread data (parameters)                 */
    int     reg_cnt;            /* Count of registered events               */
    int     rc;                 /* Return code                              */


    td = (struct thread_data *) parm_p; /* Get parameters for the thread     */

    start_session(td->sess);    /* Start EMAPI session for SP partition     */

    /*
     *  Install a clean up handler that will end the session when the thread
     *  is canceled.
     */

    pthread_cleanup_push(program_thread_main_cleanup, (void *)td->sess);

    /*
     *  Register for events that will monitor the programs of interest.
     *  Maintain count of registered events.
     */

    reg_cnt = 0;
    register_for_events(td->sess, td->progs, td->progs_cnt, &reg_cnt);

    /*
     *  Continue looking for Event Manager responses as long as there are
     *  registered events.
     */

    while (reg_cnt > 0) {

        /*
         *  Ensure the session is still connected to the Event Manager.
         */

        ensure_session_connected(td->sess);

        /*
         *  Process the next response from the Event Manager.
         */

        process_response(td->sess, td->progs, td->progs_cnt, &reg_cnt);

    }
```

```
    /*
     *  Pop and execute the thread cleanup handler installed by this routine.
     *  Note that the cleanup handler will end the EMAPI session.
     */

    pthread_cleanup_pop(1);

    return NULL;               /* The thread is finished.                  */
}


/*
 *  The program_thread_main_cleanup() function is the thread cancellation
 *  cleanup handler for the program_thread_main() function.  When this
 *  function is installed as a thread cancellation cleanup handler, the
 *  thread has an EMAPI session.  This function ends that session.  The
 *  function receives as input a pointer to the session structure describing
 *  the session.
 */

static
void program_thread_main_cleanup(void *parm_p)
{
    struct session  *sess_p;


    sess_p = (struct session *) parm_p;

    end_session(sess_p);

    printf("%s:\tno longer monitoring programs in this partition.\n",
           sess_p->name);

    return;
}


/*
 *  The cancel_thread_main() function waits for the user to request
 *  the termination of this program.  The user requests program termination
 *  by pressing the interrupt key (usually Ctrl-C).  Pressing the interrupt
 *  key will cause this program to receive the SIGINT signal.  This function
 *  waits for the SIGINT signal to be delivered.  Once the SIGINT signal has
 *  been delivered, this function cancels the two threads monitoring programs
 *  through the EMAPI.
 */

static
void *cancel_thread_main(void *parm_p)
{
    int             rc;                /* Return code (error number)       */
    sigset_t        sigset;            /* Signal mask to unblock SIGINT    */
    int             sig;               /* Received signal                  */
    STRERROR_BUF;                      /* Buffer for strerror_r()          */


    sigemptyset(&sigset);              /* Initialize signal set            */
```

emapi_v01_ex03.c

```
    sigaddset(&sigset, SIGINT);         /* Add SIGINT to signal set          */

    for ( ; ; ) {

        rc = sigwait(&sigset, &sig);
        if (rc != 0) {
            fprintf(stderr, "sigwait(): %s\n", STRERROR(rc));
            exit(1);
        }

        if (sig == SIGINT) {
            printf("Termination requested.\n");
            cancel_thread(thread1);     /* Request cancellation of thread 1  */
            cancel_thread(thread2);     /* Request cancellation of thread 2  */
            break;                      /* No longer wait for SIGINT         */
        }

        fprintf(stderr, "sigwait() reported unexpected signal: %d.\n", sig);
        exit(1);

    }

    return NULL;                        /* The thread is finished            */
}


/*
 *  The start_session() function establishes a session with the EMAPI by
 *  calling the EMAPI routine ha_em_start_session().  If a session cannot be
 *  established, the program is terminated.
 */

static
void start_session(struct session *sess_p)
{
    struct ha_em_err_blk errb;                  /* EMAPI error block         */


    /*
     *  The ha_em_start_session() routine receives the name of the partition
     *  with which a session is to be established, and returns a file
     *  descriptor with which the session can be used.
     */

    sess_p->fd = ha_em_start_session(sess_p->name, &errb);

    /*
     *  If ha_em_start_session() indicates a session could not be established,
     *  print the error information returned by the routine, and exit the
     *  program.
     */

    if (sess_p->fd == -1) {
        fprintf(stderr, "%s:\tha_em_start_session() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }
```

```
    /*
     *  A session has been established with the EMAPI, and the session's file
     *  descriptor has been saved in the session structure passed to this
     *  routine.  Set the flag in the session structure that indicates the
     *  session does not need to be restarted.
     */

    sess_p->restart = 0;

    return;
}


/*
 *  The register_for_events() function uses the EMAPI routine
 *  ha_em_send_command() to request the registration of events pertaining to
 *  programs of interest that are in the same SP partition.
 *
 *  The resource variable name specified is "IBM.PSSP.Prog.xpcount".  This is a
 *  state variable whose value indicates whether or not processes are executing
 *  a specific program.  The instantiation vector for this resource variable
 *  specifies the program name, the name of the user running the program, and
 *  the node on which the program is running.  The resource variable's value
 *  is a structured byte string of three fields.  Field 0 is of type long; its
 *  value is the number of processes currently running the specified program
 *  for the specified user on the specified node.  Field 1 is also of type
 *  long; its value is the previous number of processes running the program.
 *  Field 2 is of type character string; its value is a comma separated list
 *  of the PIDs of the processes running the program.
 *
 *  When the events are registered, two predicates are specified.  The primary
 *  predicate, "X@0 == 0", specifies that an event is to be generated when
 *  field 0 of the resource variable's structured byte string has a value of 0.
 *  Taking into account the semantics of the resource variable, the predicate
 *  causes an event to be generated when no processes are running the specified
 *  program for the specified user on the specified node.  The re-arm
 *  predicate, "X@0 > 0", specifies that after the primary predicate has
 *  generated an event, the primary predicate is to be re-armed once a process
 *  starts running the specified program for the specified user on the
 *  specified node.  Since the HA_EM_CMD_REG2 command is specified, the re-arm
 *  predicate will generate events.
 *
 *  Notice that when events are registered, the HA_EM_SCMD_REVAL subcommand is
 *  specified.  As a result, the initial value of the programs being monitored
 *  will be returned.
 */

static
void register_for_events(struct session *sess_p,
        struct program *progs_p, int progs_elems, int *reg_cnt_p)
{
    int                     num_events;      /* Number of events to register */
    size_t                  alloc_size;      /* Size of memory to allocate   */
    struct ha_em_cmd_blk    *cmd_blk;        /* Pointer to command block     */
    struct ha_em_rb_reg     *re;             /* Pointer to registered event  */
    int                     i;               /* Index                        */
    struct ha_em_err_blk    errb;            /* EMAPI error block            */
    int                     rc;              /* Return code                  */
```

emapi_v01_ex03.c

```
char                    ivector_buf[80]; /* Instantiation vector buffer  */


num_events = progs_elems;           /* Register one event per program    */

/*
 *  Allocate enough memory to hold the command block to be given to the
 *  EMAPI.  The ha_em_cmd_blk structure contains one ha_em_rb_reg
 *  structure.  Additional space must be allocated when more than one
 *  event is to be registered.
 */

alloc_size = sizeof(struct ha_em_cmd_blk) +
                 ((num_events - 1) * sizeof(struct ha_em_rb_reg));

if ((cmd_blk = malloc(alloc_size)) == NULL) {
    perror("malloc()");
    exit(1);
}

/*
 *  Fill in the em_rb_reg array entries.  There is one entry per event to
 *  be registered.
 */

for (i = 0; i < num_events; i++) {

    re = &cmd_blk->em_res_blk.em_rb_reg[i];

    re->em_name = "IBM.PSSP.Prog.xpcount";   /* Resource variable name   */

    sprintf(ivector_buf, "ProgName=%s;UserName=%s;NodeNum=%d",
            progs_p[i].name, progs_p[i].user, progs_p[i].node);
    re->em_ivector = strdup(ivector_buf);    /* Instantiation vector     */

    if (re->em_ivector == NULL) {
        perror("strdup()");
        exit(1);
    }

    re->em_pred   = "X@0 == 0";              /* Predicate                */
    re->em_rapred = "X@0 > 0";               /* Re-arm predicate         */

    re->em_cb     = NULL;                    /* Callback routine not used*/
    re->em_cb_arg = NULL;                    /* Callback routine not used*/
}

/*
 *  Fill in command block header, specifying number of elements in
 *  em_rb_reg array, command, and subcommand.
 */

cmd_blk->em_cmd_num_elem = num_events;
cmd_blk->em_cmd          = HA_EM_CMD_REG2;
cmd_blk->em_subcmd       = HA_EM_SCMD_REVAL;

/*
 *  Send the command.  A more elaborate program might check for the
```

```
     *  HA_EM_ECONNREFUSED Event Manager error number when an error is
     *  returned, and attempt to restart the session.  But, this program
     *  just terminates if ha_em_send_command() detects a lost connection.
     */

    rc = ha_em_send_command(sess_p->fd, cmd_blk, &errb);

    if (rc == -1) {
        fprintf(stderr, "%s:\tha_em_send_command() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    /*
     *  Save the event identifiers assigned to the events just registered.
     */

    for (i = 0; i < num_events; i++) {
        re = &cmd_blk->em_res_blk.em_rb_reg[i];
        progs_p[i].eid     = re->em_event_id;
        progs_p[i].unreged = 0;
        free(re->em_ivector);
    }

    free(cmd_blk);

    *reg_cnt_p += num_events;           /* Update count of registered events */

    return;
}


/*
 *  The ensure_session_connected() function ensures the specified session
 *  is not known to be disconnected.  If the session is marked as needing
 *  to be restarted, the function does not return until the session has been
 *  successfully restarted.  Restart attempts are tried every 15 seconds.
 */

static
void ensure_session_connected(struct session *sess_p)
{
    struct ha_em_err_blk    errb;           /* EMAPI error block             */
    int                     new_fd;         /* New session file descriptor   */


    /*
     *  If the specified session needs to be restarted, try to do so.
     */

    while (sess_p->restart) {

        new_fd = ha_em_restart_session(sess_p->fd, &errb);

        if (new_fd == -1) {

            /*
             *  The session could not be restarted.  If the Event Manager
```

```
             *  error number indicates connection refused, wait 15 seconds
             *  before trying the restart again.
             */

            if (errb.em_errno == HA_EM_ECONNREFUSED) {
                sleep(15);
                continue;
            }

            /*
             *  If execution reaches here, the session restart attempt failed
             *  for an unexpected reason; terminate the program.
             */

            fprintf(stderr, "%s:\tha_em_restart_session() returned EM errno "
                    "%d:\n%s", sess_p->name, errb.em_errno, errb.em_errmsg);
            exit(1);
        }

        /*
         *  If execution reaches here, a session restart was attempted and
         *  was successful.  Save the new session file descriptor, and indicate
         *  that the session no longer needs to be restarted.
         */

        sess_p->fd = new_fd;
        sess_p->restart = 0;

    }

    /*
     *  If execution reaches here, the specified session is not known to be
     *  in need of a restart.  Just return.
     */

    return;
}


/*
 *  The process_response() function processes an Event Manager response
 *  in the specified EMAPI session.  First, ha_em_receive_response() is called
 *  to receive the response.  If ha_em_receive_response() indicates that the
 *  session's connection with the Event Manager has been lost, the session
 *  is marked as being in need of a restart.  Any other error will terminate
 *  the program.  The ha_em_receive_response() function may indicate that
 *  there really isn't any response for the EMAPI client to process at this
 *  time.  In that case, this function just returns.  If a response has
 *  been received, it is processed depending on what type of response it is.
 */

static
void process_response(struct session *sess_p,
         struct program *progs_p, int progs_elems, int *reg_cnt_p)
{
    struct ha_em_rsp_blk    *rsp_blk;   /* Pointer to the response block     */
    int                     rc;         /* Return code                       */
    struct ha_em_err_blk    errb;       /* Event Manager error block         */
```

```
int                     i;          /* Index                              */


/*
 *  Receive response from session, if there is one to receive.
 */

rc = ha_em_receive_response(sess_p->fd, &rsp_blk, &errb);

if (rc == -1) {

    /*
     *  If the connection with the Event Manager has been lost,
     *  mark the session as needing to be restarted, and return.
     */

    if (errb.em_errno == HA_EM_ECONNLOST) {
        printf("%s:\tconnection to session lost.\n", sess_p->name);
        sess_p->restart = 1;
        return;
    }

    /*
     *  Some error has occurred other than the loss of the connection.
     *  Terminate the program.
     */

    fprintf(stderr, "%s:\tha_em_receive_response() returned EM errno %d:\n"
            "%s", sess_p->name, errb.em_errno, errb.em_errmsg);
    exit(1);

}

/*
 *  If the ha_em_receive_response() routine returned zero,
 *  there is no response for the EMAPI client (this program) to process
 *  at this time.  The response may have been for the EMAPI itself, or
 *  a full response may not be available yet.  Just return.
 */

if (rc == 0) {
    return;
}

/*
 *  A response for the EMAPI client (this program) has been returned.
 *  The response buffer is pointed to by the rsp_blk variable.  Appropriate
 *  processing for the response depends on the command type.
 */

switch (rsp_blk->em_cmd) {

    case HA_EM_CMD_REG:             /* Event response                     */
    case HA_EM_CMD_REG2:            /* Event response (possible re-arm)  */
        process_response_reg(sess_p, progs_p, progs_elems, rsp_blk);
        break;

    case HA_EM_CMD_RERR:            /* REG  event registration error     */
```

```
        case HA_EM_CMD_R2ERR:           /* REG2 event registration error     */
            process_response_rerr(sess_p, progs_p, progs_elems, rsp_blk,
                                  reg_cnt_p);
            break;

        default:                        /* Unexpected response               */
            fprintf(stderr, "%s:\tProgram received unexpected command "
                    "response: %d.\n", sess_p->name, rsp_blk->em_cmd);
            exit(1);
            break;

    }

    /*
     *  The EMAPI client (this program) must free the memory associated with
     *  the returned response block when it is no longer needed.
     */

    free(rsp_blk);

    return;
}


/*
 *  The process_response_reg() function processes event responses from the
 *  Event Manager.  Several points should be kept in mind:
 *
 *      - The response block may contain multiple event responses.  The
 *        number of responses included in the response block is given in
 *        the response block header.
 *
 *      - An event response may be an error response.  An event error response
 *        does not indicate that a registered event has occurred.  Instead, it
 *        indicates the Event Manager no longer knows the current value of
 *        the associated resource variable, and cannot generate events for
 *        the resource variable.  This may be a temporary condition.  If the
 *        Event Manager later obtains the current value of the associated
 *        resource variable, an event will be generated (possibly indicating
 *        that the predicate is false).
 *
 *      - The event for which an event error response is generated is still
 *        registered.  There is no need to re-register the event.
 *
 *      - Since events are registered by this program using the HA_EM_CMD_REG2
 *        command, the re-arm predicates will generate events.  Therefore, the
 *        HA_EM_EVENT_RE_ARM flag must be tested to see if the predicate or
 *        the re-arm predicate generated the event.
 *
 *      - It is possible for an event to be delivered indicating the predicate
 *        is false.  This can happen for two reasons.  First, when events are
 *        registered by this program, the HA_EM_SCMD_REVAL subcommand is
 *        specified.  That subcommand requests the initial value of the
 *        associated resource variable.  Second, the current value of the
 *        associated resource variable is returned through an event once the
 *        Event Manager obtains the current value of a resource variable after
 *        an error has occurred.
 *
```

```
 *      - If an event response does not have the HA_EM_EVENT_RE_ARM nor
 *        HA_EM_EVENT_PRED_FALSE flags set, the event response indicates that
 *        the event's predicate is true.
 */

static
void process_response_reg(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk)
{
    struct ha_em_rpb_event  *event_p;           /* Pointer to event response */
    struct ha_em_rpb_event  *last_event_p;      /* Beyond last event response*/

    char time_stamp[80];        /* Formatted event time stamp                */
    char prog_name[80];         /* Name of program to which event pertains   */
    char user_name[80];         /* Name of user to which event pertains      */
    int  node;                  /* Node to which event pertains              */


    event_p = rsp_blk->em_resp_blk.em_rpb_event;
    last_event_p = event_p + rsp_blk->em_rsp_num_resp;

    for ( ; event_p < last_event_p; event_p++) {

        /*
         *  Format the event's time stamp; extract the program name, user
         *  name, and node number from the instantiation vector.
         */

        format_timestamp(&event_p->em_timestamp, time_stamp);
        breakdown_ivector(event_p->em_ivector, prog_name, user_name, &node);

        /*
         *  If this is an event error response, print a message which includes
         *  the general and specific error codes.
         */

        if (event_p->em_errnum != 0) {
            printf("%s:\t%s State of %s run by %s on node %d unknown (%d, %d)."
                   "\n", sess_p->name, time_stamp, prog_name, user_name, node,
                   event_p->em_generr, event_p->em_specerr);
            continue;
        }

        /*
         *  If the event response was generated for the re-arm predicate, or
         *  the event response was generated for the primary predicate but
         *  it is false, print a message indicating the program associated
         *  with the event is being run.
         */

        if ((event_p->em_event_flags & HA_EM_EVENT_RE_ARM)     ||
            (event_p->em_event_flags & HA_EM_EVENT_PRED_FALSE)) {
            printf("%s:\t%s %s being run by %s on node %d.\n",
                   sess_p->name, time_stamp, prog_name, user_name, node);
            continue;
        }
```

```
        /*
         *  If execution reaches this point, the event response was generated
         *  for the primary predicate, and it is true.  Print a message
         *  indicating the program associated with the event is not being run.
         */

        printf("%s:\t%s %s NOT being run by %s on node %d.\n",
               sess_p->name, time_stamp, prog_name, user_name, node);

    }

    return;
}


/*
 *  The process_response_rerr() function processes registration error
 *  responses.  When an event cannot be registered due to some detected error,
 *  a registration error response is sent to the EMAPI client.  This function
 *  marks the event, as tracked in a program structure, as being unregistered.
 *  This will prevent an attempt to unregister a non-registered event later
 *  in the program.
 */

static
void process_response_rerr(struct session *sess_p,
        struct program *progs_p, int progs_elems,
        struct ha_em_rsp_blk *rsp_blk, int *reg_cnt_p)
{
    struct ha_em_rpb_rerr   *error_p;           /* Pointer to error response */
    struct ha_em_rpb_rerr   *last_error_p;      /* Beyond last error response*/
    struct program          *prog_p;            /* Pointer to program struct */

    char prog_name[80];         /* Name of program to which error pertains   */
    char user_name[80];         /* Name of user to which error pertains      */
    int  node;                  /* Node to which error pertains              */


    error_p = rsp_blk->em_resp_blk.em_rpb_rerr;
    last_error_p = error_p + rsp_blk->em_rsp_num_resp;

    for ( ; error_p < last_error_p; error_p++) {

        /*
         *  Find the program structure which describes the program associated
         *  with the event whose registration failed.  The match is made
         *  with event identifiers.
         */

        for (prog_p = progs_p; prog_p < progs_p + progs_elems; prog_p++) {
            if (prog_p->eid == error_p->em_event_id) {
                break;
            }
        }

        if (prog_p == progs_p + progs_elems) {
            fprintf(stderr, "%s:\tUnknown event identifier encountered (0x%x)."
                    "\n", sess_p->name, error_p->em_event_id);
```

```
            exit(1);
        }

        /*
         *  Extract the program name, user name, and node number from the
         *  instantiation vector.
         */

        breakdown_ivector(error_p->em_ivector, prog_name, user_name, &node);

        /*
         *  Print a message about the registration error.
         */

        printf("%s:\tRegistration error for %s run by %s on node %d (%d, %d)."
               "\n", sess_p->name, prog_name, user_name, node,
               error_p->em_generr, error_p->em_specerr);

        /*
         *  Mark the program as not having an associated event registered.
         */

        prog_p->unreged = 1;

    }

    /*
     *  Update the number of events that are registered.
     */

    *reg_cnt_p -= rsp_blk->em_rsp_num_resp;

    return;
}


/*
 *  The format_timestamp() function takes a time stamp returned by the
 *  Event Manager and converts it into 24-hour time.
 */

static
void format_timestamp(struct timeval *timestamp_p, char *fmt_timestamp_p)
{
    struct tm   broken_down_time;


    localtime_r((time_t *) &timestamp_p->tv_sec, &broken_down_time);
    (void) strftime(fmt_timestamp_p, 20, "(%X)", &broken_down_time);

    return;
}


/*
 *  The breakdown_ivector() function takes an instantiation vector
 *  associated with the IBM.PSSP.Prog.xpcount resource variable and extracts
 *  from it the program name, user name, and node number.
```

emapi_v01_ex03.c

```
 */

static
void breakdown_ivector(char *ivector_p,
        char *prog_name_p, char *user_name_p, int *node_p)
{
    char    *value_p;
    size_t  value_len;


    find_ivector_value(ivector_p, "ProgName=", &value_p, &value_len);
    strncpy(prog_name_p, value_p, value_len);
    *(prog_name_p + value_len) = '\0';

    find_ivector_value(ivector_p, "UserName=", &value_p, &value_len);
    strncpy(user_name_p, value_p, value_len);
    *(user_name_p + value_len) = '\0';

    find_ivector_value(ivector_p, "NodeNum=", &value_p, &value_len);
    *node_p = atoi(value_p);

    return;
}


/*
 *  The find_ivector_value() takes an instantiation vector and extracts
 *  a value from it.
 */

static
void find_ivector_value(char *ivector_p, char *name_p,
        char **value_pp, size_t *value_len_p)
{
    char    *bp;
    char    *ep;
    size_t  len;


    if ((bp = strstr(ivector_p, name_p)) == NULL) {
        fprintf(stderr, "Unexpected instantiation vector encountered.\n");
        exit(1);
    }

    bp += strlen(name_p);

    if ((ep = strchr(bp, ';')) != NULL) {
        len = ep - bp;
    } else {
        len = strlen(bp);
    }

    *value_pp    = bp;
    *value_len_p = len;

    return;
}
```

```
/*
 *  The end_session() function terminates a session with the EMAPI by
 *  calling the EMAPI routine ha_em_end_session().
 */

static
void end_session(struct session *sess_p)
{
    struct ha_em_err_blk errb;


    if (ha_em_end_session(sess_p->fd, &errb) == -1) {
        fprintf(stderr, "%s:\tha_em_end_session() returned EM errno %d:\n%s",
                sess_p->name, errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    return;
}
```

The emapi_v01_ex04.c Sample Program

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

#include <ha_emapi.h>

/*
 *  emapi_v01_ex04.c
 *
 *  This program presents an example of using the Event Manager Application
 *  Programming Interface (EMAPI).  The program uses the EMAPI to obtain
 *  the definition of some resource variables.
 *
 *  This program can be compiled with the following command:
 *
 *      cc -O emapi_v01_ex04.c -o emapi_v01_ex04 -lha_em
 */


/*
 *  The following array contains query requests that will be sent to the
 *  EMAPI.  Each array element is of type struct ha_em_rb_query, which is
 *  defined by the EMAPI.  Each element specifies a class, a resource
 *  variable, and an instance vector.
 *
 *  Request 1 requests the definitions of resource variables in the
 *  IBM.PSSP.Prog class.
 *
 *  Request 2 requests the definitions of resource variables in the
 *  IBM.PSSP.SP_HW class whose resource variable names start with
 *  IBM.PSSP.SP_HW.Node.
 *
 *  Request 3 requests the definitions of resource variables in all
 *  classes whose instance vectors include an instance vector element
 *  named CPU.
 */

static struct ha_em_rb_query qry_array[] = {
    {"IBM.PSSP.Prog", "", "*"},                              /* Request 1     */
    {"IBM.PSSP.SP_HW", "IBM.PSSP.SP_HW.Node", "*"},          /* Request 2     */
    {"", "", "CPU;*"}                                        /* Request 3     */
};

static int qry_cnt = sizeof qry_array / sizeof qry_array[0];


/*
 *  Function prototypes for internal functions.
 */
```

```
static void setup_signals(void);
static int start_session(void);
static ha_em_qid_t send_defn_request(int sess_fd,
                                     struct ha_em_rb_query *qry_array,
                                     int qry_cnt);
static int recv_defn_reply(int sess_fd, ha_em_qid_t qid);
static int process_response_query(ha_em_qid_t qid,
                                  struct ha_em_rsp_blk *rsp_blk);
static int process_response_qerr(ha_em_qid_t qid,
                                 struct ha_em_rsp_blk *rsp_blk);
static void end_session(int sess_fd);


/*
 *  The main() function calls functions to initialize the program, establish
 *  an EMAPI session, send query requests, wait for and process Event Manager
 *  responses, and terminate the EMAPI session.
 */

int main(int argc, char **argv)
{
    int         sess_fd;                        /* Session file descriptor   */
    ha_em_qid_t qid;                            /* Query identifier          */
    int         qend;                           /* Query is finished         */


    setup_signals();                            /* Set signal dispositions   */

    sess_fd = start_session();                  /* Start EMAPI session       */

    qid = send_defn_request(sess_fd, qry_array, qry_cnt);
                                                /* Send query request        */

    /*
     *  Receive query replies until all replies have been received.
     */

    while ((qend = recv_defn_reply(sess_fd, qid)) == 0) {
        /* Nothing to do inside the loop */
    }

    end_session(sess_fd);                       /* End EMAPI session         */

    return 0;
}


/*
 *  The setup_signals() function initializes the disposition of a signal.
 *
 *  The disposition of the SIGPIPE signal is set such that the signal is
 *  ignored.  When a process writes to a pipe or connection-oriented socket for
 *  which there is no reader, the SIGPIPE signal is delivered to the process.
 *  The default disposition for SIGPIPE is to kill the receiving process.  If
 *  SIGPIPE is ignored, the SIGPIPE signal does not kill the process, and the
 *  system call used to write to the pipe or connection-oriented socket sets
 *  errno to EPIPE and returns an error indication.  It is recommended that
 *  clients of the EMAPI ignore the SIGPIPE signal, since the EMAPI uses
```

emapi_v01_ex04.c

```
 *  connection-oriented sockets to communicate with the Event Manager.
 */

static
void setup_signals(void)
{
    struct sigaction    sa;


    sa.sa_handler = SIG_IGN;               /* SIGPIPE to be ignored         */
    sigemptyset(&sa.sa_mask);              /* No signals to mask off        */
    sa.sa_flags = 0;                       /* No flags needed               */

    if (sigaction(SIGPIPE, &sa, NULL) == -1) {  /* Change SIGPIPE disposition*/
        perror("sigaction(SIGPIPE)");
        exit(1);
    }

    return;
}


/*
 *  The start_session() function establishes a session with the EMAPI by
 *  calling the EMAPI routine ha_em_start_session().  If a session cannot be
 *  established, the program is terminated.
 */

static
int start_session(void)
{
    int sess_fd;                                /* Session file descriptor   */
    struct ha_em_err_blk errb;                  /* EMAPI error block         */


    /*
     *  Since no specific partition name is passed to the
     *  ha_em_start_session() routine, it attempts to establish a session
     *  with the default partition.  The routine returns a file
     *  descriptor with which the session can be used.
     */

    sess_fd = ha_em_start_session("", &errb);

    /*
     *  If ha_em_start_session() indicates a session could not be established,
     *  print the error information returned by the routine, and exit the
     *  program.
     */

    if (sess_fd == -1) {
        fprintf(stderr, "ha_em_start_session() returned EM errno %d:\n%s",
                errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    return sess_fd;                             /* Return session fd         */
}
```

```
/*
 *  The send_defn_request() function uses the EMAPI routine
 *  ha_em_send_command() to send the query defined resource variables requests
 *  specified by the qry_array and qry_cnt parameters.
 */

static
ha_em_qid_t send_defn_request(int sess_fd,
                              struct ha_em_rb_query *qry_array, int qry_cnt)
{
    size_t                  alloc_size;      /* Size of memory to allocate   */
    struct ha_em_cmd_blk    *cmd_blk;        /* Pointer to command block     */
    struct ha_em_err_blk    errb;            /* EMAPI error block            */
    int                     rc;              /* Return code                  */
    ha_em_qid_t             qid;             /* Query identifier             */


    /*
     *  Allocate enough memory to hold the command block to be given to the
     *  EMAPI.  The ha_em_cmd_blk structure contains one ha_em_rb_query
     *  structure.  Additional space must be allocated when more than one
     *  query is to be made.
     */

    alloc_size = sizeof(struct ha_em_cmd_blk) +
                    ((qry_cnt - 1) * sizeof(struct ha_em_rb_query));

    if ((cmd_blk = malloc(alloc_size)) == NULL) {
        perror("malloc()");
        exit(1);
    }

    /*
     *  Fill in command block header, specifying number of elements in
     *  em_rb_query array, command, and subcommand.  Indicate a callback
     *  routine will not be used to receive the responses.
     */

    cmd_blk->em_cmd_num_elem = qry_cnt;
    cmd_blk->em_cmd          = HA_EM_CMD_QUERY;
    cmd_blk->em_subcmd       = HA_EM_SCMD_QDEF;
    cmd_blk->em_qcb          = NULL;
    cmd_blk->em_qcb_arg      = NULL;

    /*
     *  Copy the array of query requests into the command block.
     */

    memcpy(cmd_blk->em_res_blk.em_rb_query, qry_array,
           qry_cnt * sizeof(struct ha_em_rb_query));

    /*
     *  Send the command.  A more elaborate program might check for the
     *  HA_EM_ECONNREFUSED Event Manager error number when an error is
     *  returned, and attempt to restart the session.  But, this program
     *  just terminates if ha_em_send_command() detects a lost connection.
```

emapi_v01_ex04.c

```
     */

    rc = ha_em_send_command(sess_fd, cmd_blk, &errb);

    if (rc == -1) {
        fprintf(stderr, "ha_em_send_command() returned EM errno %d:\n%s",
                errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    /*
     *  Save the query identifier assigned to the query request just sent.
     */

    qid = cmd_blk->em_qid;

    free(cmd_blk);          /* Free the allocated command block.             */

    return qid;             /* Return the query identifier                   */
}

/*
 *  The recv_defn_reply() function processes an Event Manager response
 *  in the specified EMAPI session.  First, ha_em_receive_response() is called
 *  to receive the response.  If ha_em_receive_response() returns an error
 *  the program is terminated.  The ha_em_receive_response() function may
 *  indicate that there really isn't any response for the EMAPI client to
 *  process at this time.  In that case, this function just returns.  If a
 *  response has been received, it is processed depending on what type of
 *  response it is.
 *
 *  This function returns 0 if all query responses have not been received yet.
 *  If all query responses have been received, 1 is returned.
 */

static
int recv_defn_reply(int sess_fd, ha_em_qid_t qid)
{
    struct ha_em_rsp_blk    *rsp_blk;   /* Pointer to the response block     */
    int                     rc;         /* Return code                       */
    struct ha_em_err_blk    errb;       /* Event Manager error block         */
    int                     qend;       /* Query end flag                    */


    /*
     *  Receive response from session, if there is one to receive.
     */

    rc = ha_em_receive_response(sess_fd, &rsp_blk, &errb);

    if (rc == -1) {

        /*
         *  Some error has occurred receiving a response; terminate the
         *  program.  A more elaborate program might check for the
         *  HA_EM_ECONNREFUSED Event Manager error number when an error is
         *  returned, and attempt to restart the session.  But, this program
         *  just terminates if ha_em_receive_response() detects a lost
```

```
         *  connection.
         */

        fprintf(stderr, "ha_em_receive_response() returned EM errno %d:\n%s",
                errb.em_errno, errb.em_errmsg);
        exit(1);

    }

    /*
     *  If the ha_em_receive_response() routine returned zero,
     *  there is no response for the EMAPI client (this program) to process
     *  at this time.  The response may have been for the EMAPI itself, or
     *  a full response may not be available yet.  Just return.
     */

    if (rc == 0) {
        return 0;           /* Return indicating query is not ended          */
    }

    /*
     *  A response for the EMAPI client (this program) has been returned.
     *  The response buffer is pointed to by the rsp_blk variable.  Appropriate
     *  processing for the response depends on the command type.
     */

    switch (rsp_blk->em_cmd) {

        case HA_EM_CMD_QUERY:           /* Query response                    */
            qend = process_response_query(qid, rsp_blk);
            break;

        case HA_EM_CMD_QERR:            /* Query error response              */
            qend = process_response_qerr(qid, rsp_blk);
            break;

        default:                        /* Unexpected response               */
            fprintf(stderr, "Program received unexpected command "
                    "response: %d.\n", rsp_blk->em_cmd);
            exit(1);
            break;

    }

    /*
     *  The EMAPI client (this program) must free the memory associated with
     *  the returned response block when it is no longer needed.
     */

    free(rsp_blk);

    return qend;
}


/*
 *  The process_response_query() function processes query responses from the
 *  Event Manager.  Several points should be kept in mind:
```

```
 *
 *      - The response block may contain multiple query responses.  The
 *        number of responses included in the response block is given in
 *        the response block header.
 *
 *      - An query response may be an error response.  Such a response
 *        may indicate an error of a temporary nature.
 *
 *      - This routine compares the query identifier in the response block
 *        with the query identifier assigned to the request sent by
 *        this program to the EMAPI.  This should not really be necessary.
 *        If the program had sent multiple query requests to the EMAPI,
 *        through multiple calls to ha_em_send_command(), the query identifiers
 *        in the response blocks could be used to associate responses with
 *        requests.
 *
 *      - Multiple response blocks may be associated with one request block.
 *        The response to a query request is not complete until a response
 *        block is received with the em_qend field set to a non-zero value.
 */

static
int process_response_query(ha_em_qid_t qid, struct ha_em_rsp_blk *rsp_blk)
{
    struct ha_em_rpb_qdef   *qp;           /* Pointer to query response      */
    struct ha_em_rpb_qdef   *last_qp;      /* Beyond last query response     */


    /*
     *  Check query identifier in response block.
     */

    if (rsp_blk->em_qid != qid) {
        fprintf(stderr, "Query response block contains unexpected "
               "query identifier: %d.\n", rsp_blk->em_qid);
        exit(1);
    }

    /*
     *  Look at all the response elements in the response block.
     */

    qp = rsp_blk->em_resp_blk.em_rpb_qdef;
    last_qp = qp + rsp_blk->em_rsp_num_resp;

    for ( ; qp < last_qp; qp++) {

        printf("============================================================"
               "\n\n");

        /*
         *  Check for error code.
         */

        if (qp->em_errnum != 0) {
            printf("Query response for class \"%s\", "
                   "resource variable \"%s\", instance vector \"%s\" "
                   "unexpectedly contains an error (%d, %d).\n\n",
```

```
                qp->em_class, qp->em_name, qp->em_ivector,
                qp->em_generr, qp->em_specerr);
            continue;
        }

        /*
         *  Print definition of a resource variable.
         */

        printf("Resource Variable Name: \"%s\"\n"
               "Variable Value Type:    %s\n"
               "Variable Data  Type:    %s\n"
               "Variable SBS Format:    \"%s\"\n"
               "Variable Initial Value: \"%s\"\n"
               "Variable Class:         \"%s\"\n"
               "Instance Vector:        \"%s\"\n"
               "PTX Name:               \"%s\"\n"
               "Default Predicate:      \"%s\"\n"
               "Locator:                \"%s\"\n"
               "\n"
               "Variable Description\n"
               "--------------------\n"
               "%s\n"
               "\n"
               "Instance Vector Desciption\n"
               "--------------------------\n"
               "%s\n"
               "\n"
               "Event Description\n"
               "-----------------\n"
               "%s\n"
               "\n",

               qp->em_name,
               qp->em_value_type == ha_emVTcounter  ? "Counter"  :
               qp->em_value_type == ha_emVTquantity ? "Quantity" :
               qp->em_value_type == ha_emVTstate    ? "State"    : "Unknown",
               qp->em_data_type  == ha_emDTlong     ? "long"     :
               qp->em_data_type  == ha_emDTfloat    ? "float"    :
               qp->em_data_type  == ha_emDTsbs      ? "SBS"      : "Unknown",
               qp->em_sbs_format,
               qp->em_init_value,
               qp->em_class,
               qp->em_ivector,
               qp->em_ptx_name,
               qp->em_dflt_pred,
               qp->em_locator,
               qp->em_descrp,
               qp->em_ivector_descrp,
               qp->em_event_descrp);

    }

    return rsp_blk->em_qend;
}


/*
```

emapi_v01_ex04.c

```
 *  The process_response_qerr() function processes query error
 *  responses.
 */

static
int process_response_qerr(ha_em_qid_t qid, struct ha_em_rsp_blk *rsp_blk)
{
    struct ha_em_rpb_qerr   *error_p;           /* Pointer to error response */
    struct ha_em_rpb_qerr   *last_error_p;      /* Beyond last error response*/


    /*
     *  Check query identifier in response block.
     */

    if (rsp_blk->em_qid != qid) {
        fprintf(stderr, "Query response block contains unexpected "
               "query identifier: %d.\n", rsp_blk->em_qid);
        exit(1);
    }

    error_p = rsp_blk->em_resp_blk.em_rpb_qerr;
    last_error_p = error_p + rsp_blk->em_rsp_num_resp;

    for ( ; error_p < last_error_p; error_p++) {

        printf("============================================================"
               "\n\n");

        /*
         *  Print error information for a resource variable.
         */

        printf("Resource Variable Name: \"%s\"\n"
               "Variable Class:         \"%s\"\n"
               "Instance Vector:        \"%s\"\n"
               "General  Error:         %d\n"
               "Specific Error:         %d\n"
               "Additional Error Info:  %d\n",

               error_p->em_name,
               error_p->em_class,
               error_p->em_ivector,
               error_p->em_generr,
               error_p->em_specerr,
               error_p->em_errinfo);


    }

    return rsp_blk->em_qend;
}


/*
 *  The end_session() function terminates a session with the EMAPI by
 *  calling the EMAPI routine ha_em_end_session().
 */
```

```
static
void end_session(int sess_fd)
{
    struct ha_em_err_blk errb;


    if (ha_em_end_session(sess_fd, &errb) == -1) {
        fprintf(stderr, "ha_em_end_session() returned EM errno %d:\n%s",
                errb.em_errno, errb.em_errmsg);
        exit(1);
    }

    return;
}
```

emapi_v01_ex04.c

Glossary of Terms and Abbreviations

This glossary includes terms and definitions from:

- The *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.
- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies can be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The *ANSI/EIA Standard - 440A: Fiber Optic Terminology*, copyright 1989 by the Electronics Industries Association (EIA). Copies can be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue N.W., Washington, D.C. 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

The following cross-references are used in this glossary:

Contrast with. This refers to a term that has an opposed or substantively different meaning.

See. This refers the reader to multiple-word terms in which this term appears.

See also. This refers the reader to terms that have a related, but not synonymous, meaning.

Synonym for. This indicates that the term has the same meaning as a preferred term, which is defined in the glossary.

This section contains some of the terms that are commonly used in the SP publications.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard *Vocabulary for Information Processing* (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. ANSI definitions are preceded by an asterisk (*).

Other definitions in this glossary are taken from *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems* (SC20-1699) and *IBM DATABASE 2 Application Programming Guide for TSO Users* (SC26-4081).

© Copyright IBM Corp. 1996

A

address. A character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

AFS. A distributed file system that provides authentication services as part of its file system creation.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high function graphics and floating point computations.

AIX Version 3. The IBM AIX Version 3 for RS/6000 (also known as AIX/6000). It is based on UNIX System V, conforms with POSIX IEEE Standard 1003.1, is compatible with Berkeley Software Distribution 4.3 (4.3 BSD), and is designed to meet Trusted Computing Base level C2 security.

Amd. Berkeley Software Distribution automount daemon.

API. Application Programming Interface. A set of programming functions and routines that provide access between the Application layer of the OSI seven-layer model and applications that want to use the network. It is a software interface.

application. The use to which a data processing system is put; for example, a payroll application, an airline reservation application.

application data. The data that is produced using an application program.

ARP. Address Resolution Protocol.

ATM. Asynchronous Transfer Mode. (See *TURBOWAYS 100 ATM Adapter*.)

Authentication. The process of validating the identity of a user or server.

Authorization. The process of obtaining permission to perform specific actions.

B

batch processing. * (1) The processing of data or the accomplishment of jobs accumulated in advance in such a manner that each accumulation thus formed is processed or accomplished in the same run. * (2) The processing of data accumulating over a period of time. * (3) Loosely, the execution of computer programs serially. (4) Computer programs executed in the background.

BMCA. Block Multiplexer Channel Adapter. The block multiplexer channel connection allows the RS/6000 to communicate directly with a host System/370 or System/390; the host operating system views the system unit as a control unit.

C

call home function. The ability of a system to call the IBM support center and open a PMR to have a repair scheduled.

charge feature. An optional feature for either software or hardware for which there is a charge.

class. See *resource class*.

CLI. Command Line Interface.

client. * (1) A function that requests services from a server and makes them available to the user. * (2) A term used in an environment to identify a machine that uses the resources of the network.

Client Input/Output Sockets (CLIO/S). A software package that enables high-speed data and tape access between SP systems, AIX systems, and ES/9000 mainframes.

CLIO/S. Client Input/Output Sockets.

CMI. Centralized Management Interface provides a series of SMIT menus and dialogues used for defining and querying the SP system configuration.

connectionless. A communication process that takes place without first establishing a connection.

connectionless network. A network in which the sending logical node must have the address of the receiving logical node before information interchange can begin. The packet is routed through nodes in the network based on the destination address in the packet. The originating source does not receive an acknowledgment that the packet was received at the destination.

control workstation. A single point of control allowing the administrator or operator to monitor and manage the SP system using the IBM AIX Parallel System Support Programs.

Counter. In Event Management, the value type of a resource variable that represents a rate. Instances of a Counter can be observed at an interval different than their update without loss of meaningful information. It is typically used to represent an attribute of a resource

that is an indicator of throughput. See also *Quantity*, *State*.

D

daemon. A process, not associated with a particular user, that performs system-wide functions such as administration and control of networks, execution of time-dependent activities, line printer spooling and so forth.

DASD. Direct Access Storage Device. Storage for input/output data.

DCE. Distributed Computing Environment.

DFS. distributed file system. A subset of the IBM Distributed Computing Environment.

DNS. Domain Name Service. A hierarchical name service which maps high level machine names to IP addresses.

E

EMAPI. Event Management Application Programming Interface.

Error Notification Object. An object in the SDR that is matched with an error log entry. When an error log entry occurs that matches the Notification Object, a user-specified action is taken.

ESCON. Enterprise Systems Connection. The ESCON channel connection allows the RS/6000 to communicate directly with a host System/390; the host operating system views the system unit as a control unit.

Ethernet. (1) Ethernet is the standard hardware for TCP/IP local area networks in the UNIX marketplace. It is a 10-megabit per second baseband type LAN that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by collision detection (CSMA/CD). (2) A passive coaxial cable whose interconnections contain devices or components, or both, that are all active. It uses CSMA/CD technology to provide a best-effort delivery system.

Ethernet network. A baseband LAN with a bus topology in which messages are broadcast on a coaxial cabling using the carrier sense multiple access/collision detection (CSMA/CD) transmission method.

event. An action that takes place in the operation of the system software.

event. In Event Management, the notification that a predicate evaluated to true. This evaluation occurs each time an instance of a resource variable is observed. See also *predicate*, *resource variable*.

Event Management Application Programming Interface (EMAPI). A set of routines by which an application that is interested in changes in system resources can receive the information from the Event Management subsystem.

expect. Programmed dialogue with interactive programs.

expression. In SP Perspectives, the relationship between a resource variable and other elements (such as constants or the previous value of the variable) that, when true, generates an event. An example of an expression is `X < 10` where X represents the resource variable `IBM.PSSP.aixos.PagSp.%totalfree` (the percentage of total free paging space). When the predicate is true, that is, when the total free paging space is observed to be less than 10%, the Event Management subsystem generates an event to notify the appropriate application. Synonymous with *predicate*. See also *event*, *rearm expression*, *resource variable*.

F

Failover. Also called fallover, the sequence of events when a primary or server machine fails and a secondary or backup machine assumes the primary workload. This is a disruptive failure with a short recovery time.

fall back. Also called fall back, the sequence of events when a primary or server machine takes back control of its workload from a secondary or backup machine.

FDDI. Fiber Distributed Data Interface.

Fiber Distributed Data Interface (FDDI). An American National Standards Institute (ANSI) standard for 100-megabit-per-second LAN using optical fiber cables. An FDDI local area network (LAN) can be up to 100 km (62 miles) and can include up to 500 system units. There can be up to 2 km (1.24 miles) between system units and/or concentrators.

File Transfer Protocol (FTP). The Internet protocol (and program) used to transfer files between hosts. It is an application layer protocol in TCP/IP that uses TELNET and TCP protocols to transfer bulk-data files between machines or hosts.

file. * A set of related records treated as a unit, for example, in stock control, a file could consist of a set of invoices.

file server. A centrally located computer that acts as a storehouse of data and applications for numerous users of a local area network.

foreign host. Any host on the network other than the local host.

FTP. File transfer protocol.

G

gateway. An intelligent electronic device interconnecting dissimilar networks and providing protocol conversion for network compatibility. A gateway provides transparent access to dissimilar networks for nodes on either network. It operates at the session presentation and application layers.

H

HACMP/6000. AIX High Availability Cluster Multi-Processing/6000.

HACWS. High Availability Control Workstation function, based on HACMP/6000, provides for a backup control workstation for the SP system.

help key. In the SP graphical interface, the key that gives you access to the SP graphical interface help facility.

High Availability Cluster Multi-Processing/6000. An IBM facility to cluster nodes or components to provide high availability by eliminating single points of failure.

High Performance Switch. An IBM multi-stage packet switch for high-performance communication between processor nodes.

HiPPI. High Performance Parallel Interface. RS/6000 units can attach to a HiPPI network as defined by the ANSI specifications. The HiPPI channel supports burst rates of 100 Mbps over dual simplex cables; connections can be up to 25 km in length as defined by the standard and can be extended using third-party HiPPI switches and fiber optic extenders.

home directory. The directory associated with an individual user.

host. A computer connected to a network, and providing an access method to that network. A host provides end-user services.

HSD. The data striping device for the IBM Virtual Shared Disk. The device driver lets application programs stripe data across physical disks in multiple IBM Virtual Shared Disks, thus reducing I/O bottlenecks and hot spots.

I

instance vector. In Event Management, a set of elements, where each element is a name/value pair of the form `name=value`, whose values uniquely identify the copy of the resource (and by extension, the copy of the resource variable) in the system. Synonymous with *resource identifier*.

Internet. A specific inter-network consisting of large national backbone networks such as APARANET, MILNET, and NSFnet, and a myriad of regional and campus networks all over the world. The network uses the TCP/IP protocol suite.

Intermediate Switch Board. Switches mounted in the High Performance Switch expansion frame.

IP address. A 32-bit address assigned to devices or hosts in an IP internet that maps to a physical address. The IP address is composed of a network and host portion.

Internet Protocol (IP). (1) A protocol that routes data through a network or interconnected networks. IP acts as an interface between the higher logical layers and the physical network. This protocol, however, does not provide error recovery, flow control, or guarantee the reliability of the physical network. IP is a connectionless protocol. (2) A protocol used to route data from its source to it destination in an Internet environment.

ISB. Intermediate Switch Board.

K

Kerberos. A service for authenticating users in a network environment.

kernel. The core portion of the UNIX operating system which controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in “kernel mode” and is protected from user tampering by the hardware.

L

LAN. (1) Acronym for Local Area Network, a data network located on the user's premises in which serial transmission is used for direct data communication among data stations. (2) Physical network technology that transfers data a high speed over short distances. (3) A network in which a set of devices is connected to another for communication and that can be connected to a larger network.

local host. The computer to which a user's terminal is directly connected.

location. See *resource variable location*.

log database. A persistent storage location for the logged information.

log event. The recording of an event.

log event type. A particular kind of log event that has a hierarchy associated with it.

logging. The writing of information to persistent storage for subsequent analysis by humans or programs.

M

mask. To use a pattern of characters to control retention or elimination of portions of another pattern of characters.

menu. A display of a list of available functions for selection by the user.

Motif. The graphical user interface for OSF, incorporating the X Window System. Also called OSF/Motif.

MTBF. Mean time between failure. This is a measure of reliability.

MTTR. Mean time to repair. This is a measure of serviceability.

N

naive application. An application with no knowledge of a server that fails over to another server. Client to server retry methods are used to reconnect.

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

NFS. Network File System. NFS allows different systems (UNIX or non-UNIX), different architectures, or vendors connected to the same network, to access remote files in a LAN environment as though they were local files.

NIM. Network Installation Management is provided with AIX 4.1 to install AIX on the nodes.

NIM client. An AIX system installed and managed by a NIM master. NIM supports three types of clients:

- Standalone
- Diskless
- Dataless

NIM master. An AIX system that can install one or more NIM clients. An AIX system must be defined as a NIM master before defining any NIM clients on that system. A NIM master manages the configuration database containing the information for the NIM clients.

NIM object. A representation of information about the NIM environment. NIM stores this information as objects in the NIM database. The types of objects are:

- Network
- Machine
- Resource

NIS. Network Information System.

node. In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network.

Node Switch Board. Switches mounted on frames that contain nodes.

NSB. Node Switch Board.

NTP. Network Time Protocol.

P

parallel environment. A system environment where message passing or SP resource manager services are used by they application.

Parallel Environment. A licensed IBM program used for message passing applications on the SP or RS/6000 platforms.

parallel processing. A multiprocessor architecture which allows processes to be allocated to tightly coupled multiple processors in a cooperative processing environment, allowing concurrent execution of tasks.

parameter. * (1) A variable that is given a constant value for a specified application and that may denote the application. * (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. * (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. * (4) A particular piece of information that a system or application program needs to process a request.

partition. See system partition.

Perl. Practical Extraction and Report Language.

pipe. A UNIX utility allowing the output of one command to be the input of another. Represented by the | symbol. It is also referred to as filtering output.

PMR. Problem Management Report.

port. (1) An end point for communication between devices, generally referring to physical connection. (2) A 16-bit number identifying a particular TCP or UDP resource within a given TCP/IP node.

predicate. In Event Management, the relational expression between a resource variable and other elements (such as constants or the previous value of an instance of the variable) that, when true, generates an event. An example of a predicate is `X < 10` where X represents the resource variable `IBM.PSSP.aixos.PagSp.%totalfree` (the percentage of total free paging space). When the predicate is true, that is, when the total free paging space is observed to be less than 10%, the Event Management subsystem generates an event to notify the appropriate application. Synonymous with *expression*. See also *event*, *rearm predicate*, *resource variable*.

Primary node or machine. (1) A device that runs a workload and has a standby device ready to assume the primary workload if that primary node fails or is taken out of service. (2) A node on the High Performance Switch that initializes, provides diagnosis and recovery services, and performs other operations to the switch network. (3) In IBM Virtual Shared Disk function, the node at which the logical volume is actually local and that acts as server node to I/O requests from other nodes.

Problem Management Report. The number in the IBM support mechanism that represents a service incident with a customer.

process. * (1) A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. * (2) Any operation or combination of operations on data. * (3) A function being performed or waiting to be performed. * (4) A program in operation. For example, a daemon is a system process that is always running on the system.

protocol. A set of semantic and syntactic rules that defines the behavior of functional units in achieving communication.

PTX/6000. Performance Toolbox 1.2 and 2.1 for AIX.

Q

Quantity. In Event Management, the value type of a resource variable whose value fluctuates over time. Instances of a Quantity can be observed at an interval different than their update without loss of meaningful information. It is typically used to represent an attribute of a resource that is a level, that is, an indicator of how many. See also *Counter*, *State*.

R

RAID. Redundant array of independent disks.

raw value. In Event Management, the actual contents of an instance of a resource variable of data type Counter from the latest observation. By default, a Counter is interpreted as a rate.

rearm expression. In SP Perspectives, an expression used to generate an event that alternates with an original event expression in the following way: the event expression is used until it is true, then the rearm expression is used until it is true, then the event expression is used, and so on. The rearm expression is commonly the inverse of the event expression (for example, a resource variable is on or off). It can also be used with the event expression to define an upper and lower boundary for a condition of interest. Synonymous with *rearm predicate*. See also *event*, *expression*, *resource variable*.

rearm predicate. In Event Management, a predicate used to generate an event that alternates with an original event predicate in the following way: the event predicate is used until it is true, then the rearm predicate is used until it is true, then the event predicate is used, and so on. The rearm predicate is commonly the inverse of the event predicate (for example, a resource variable is on or off). It can also be used with the event predicate to define an upper and lower boundary for a condition of interest. Synonymous with *rearm expression*. See also *event*, *predicate*, *resource variable*.

remote host. *See foreign host*.

resource. In Event Management, an entity in the system that provides a set of services. Examples of resources include hardware entities such as processors, disk drives, memory, and adapters, and software entities such as database applications, processes, and file systems. Each resource in the system has one or more attributes that define the state of the resource.

resource attribute. In Event Management, a characteristic of a resource that is being monitored, specified as the last component of a resource variable. In the resource variable `IBM.AIX.PagSp.%totalfree`, `%totalfree` specifies the resource attribute. See also *resource name*, *resource variable*.

resource class. In Event Management, the application or subsystem that manages the associated resource, used to group resource variables.

resource identifier. In SP Perspectives, a set of fields, where each field is a name/value pair of the form `name=value`, whose values uniquely identify the copy of the resource (and by extension, the copy of the

resource variable) in the system. Synonymous with *instance vector*.

Resource Monitor Application Programming Interface (RMAPI). A set of routines by which an application that monitors changes in system resources can communicate those changes to the Event Management subsystem.

resource name. In Event Management, the name of a resource, specified as all but the last component of a resource variable. In the resource variable `IBM.AIX.PagSp.%totalfree`, `IBM.AIX.PagSp` specifies the resource name. See also *resource attribute*, *resource variable*.

resource variable. In Event Management, the representation of an attribute of a resource. An example of a resource variable is `IBM.AIX.PagSp.%totalfree`, which represents the percentage of total free paging space. `IBM.AIX.PagSp` specifies the resource name and `%totalfree` specifies the resource attribute.

resource variable location. In Event Management, the node on which the resource monitor that supplies the instance of the resource variable resides.

resource variable name. In Event Management, a string consisting of a resource name followed by a period followed by a resource attribute, similar to a UNIX file name, that identifies a resource variable.

RISC. Reduced Instruction Set Computing (RISC), the technology for today's high performance personal computers and workstations, was invented in 1975. Uses a small simplified set of frequently used instructions for rapid execution.

rlogin (remote LOGIN). A service offered by Berkeley UNIX systems that allows authorized users of one machine to connect to other UNIX systems across a network and interact as if their terminals were connected directly. The rlogin software passes information about the user's environment (for example, terminal type) to the remote machine.

RMAPI. Resource Monitor Application Programming Interface.

RPC. Acronym for Remote Procedure Call, a facility that a client uses to have a server execute a procedure call. This facility is composed of a library of procedures plus an XDR.

RSH. A variant of RLOGIN command that invokes a command interpreter on a remote UNIX machine and passes the command line arguments to the command interpreter, skipping the LOGIN step completely. See also *rlogin*.

S

SBS. structured byte string

SCSI. Small Computer System Interface.

server. (1) A function that provides services for users. A machine may run client and server processes at the same time. (2) A machine that provides resources to the network. It provides a network service, such as disk storage and file transfer, or a program that uses such a service. (3) A device, program, or code module on a network dedicated to providing a specific service to a network. (4) On a LAN, a data station that provides facilities to other data stations. Examples are file server, print server, and mail server.

Small Computer System Interface (SCSI). An input and output bus that provides a standard interface for the attachment of various direct access storage devices (DASD) and tape drives to the RS/6000.

shell. The shell is the primary user interface for the UNIX operating system. It serves as command language interpreter, programming language, and allows foreground and background processing. There are three different implementations of the shell concept: Bourne, C and Korn.

Small Computer Systems Interface Adapter (SCSI Adapter). An adapter that supports the attachment of various direct-access storage devices (DASD) and tape drives to the RS/6000.

SMIT. The System Management Interface Toolkit is a set of menu driven utilities for AIX that provides functions such as transaction login, shell script creation, automatic updates of object database, and so forth.

SNMP. Simple Network Management Protocol. (1) An IP network management protocol that is used to monitor attached networks and routers. (2) A TCP/IP-based protocol for exchanging network management information and outlining the structure for communications among network devices.

socket. (1) An abstraction used by Berkeley UNIX that allows an application to access TCP/IP protocol functions. (2) An IP address and port number pairing. (3) In TCP/IP, the Internet address of the host computer on which the application runs, and the port number it uses. A TCP/IP application is identified by its socket.

standby node or machine. A device that waits for a failure of a primary node in order to assume the identity of the primary node. The standby machine then runs the primary's workload until the primary is back in service.

State. In Event Management, the value type of a resource variable whose value fluctuates over time. Every change in the value of an instance of a State must be observed to avoid loss of meaningful information. A State variable can be used to represent an attribute of a resource that indicates anything other than throughput or a level. See also *Counter*, *Quantity*.

structured byte string (SBS). In Event Management, a string of bytes that consists of a four-byte SBS length followed by one or more structured fields.

subnet. Shortened form of subnetwork.

subnet mask. A bit template that identifies to the TCP/IP protocol code the bits of the host address that are to be used for routing for specific subnetworks.

subnetwork. Any group of nodes that have a set of common characteristics, such as the same network ID.

SUP. Software Update Protocol.

Sysctl. Secure System Command Execution Tool. An authenticated client/server system for running commands remotely and in parallel.

System Administrator. The user who is responsible for setting up, modifying, and maintaining the SP system.

subsystem. A software component that is not usually associated with a user command. It is usually a daemon process. A subsystem will perform work or provide services on behalf of a user request or operating system request.

syslog. A BSD logging system used to collect and manage other subsystem's logging data.

system partition. A group of nonoverlapping nodes on a switch chip boundary that act as a logical SP system.

T

tar. Tape ARchive, is a standard UNIX data archive utility for storing data on tape media.

Tcl. Tool Command Language.

TclX. Tool Command Language Extended.

TCP. Acronym for Transmission Control Protocol, a stream communication protocol that includes error recovery and flow control.

TCP/IP. Acronym for Transmission Control Protocol/Internet Protocol, a suite of protocols designed to allow communication between networks regardless of the technologies implemented in each network. TCP provides a reliable host-to-host protocol between hosts in packet-switched communications networks and in interconnected systems of such networks. It assumes that the underlying protocol is the Internet Protocol.

Telnet. Terminal Emulation Protocol, a TCP/IP application protocol that allows interactive access to foreign hosts.

Tk. Tcl-based Tool Kit for X Windows.

TMPCP. Tape Management Program Control Point.

token-ring. (1) Network technology that controls media access by passing a token (special packet or frame) between media-attached machines. (2) A network with a ring topology that passes tokens from one attaching device (node) to another. (3) The IBM Token-Ring LAN connection allows the RS/6000 system unit to participate in a LAN adhering to the IEEE 802.5 Token-Passing Ring standard or the ECMA standard 89 for Token-Ring, baseband LANs.

transaction. An exchange between the user and the system. Each activity the system performs for the user is considered a transaction.

transceiver (transmitter-receiver). A physical device that connects a host interface to a local area network, such as Ethernet. Ethernet transceivers contain electronics that apply signals to the cable and sense collisions.

transfer. To send data from one place and to receive the data at another place. Synonymous with move.

transmission. * The sending of data from one place for reception elsewhere.

TURBOWAYS 100 ATM Adapter. An IBM high-performance, high-function intelligent adapter that provides dedicated 100 Mbps ATM (asynchronous transfer mode) connection for high-performance servers and workstations.

U

UDP. User Datagram Protocol.

User Datagram Protocol (UDP). (1) In TCP/IP, a packet-level protocol built directly on the Internet Protocol layer. UDP is used for application-to-application programs between TCP/IP host systems. (2) A transport protocol in the Internet suite of protocols that provides unreliable, connectionless datagram service. (3) The Internet Protocol that enables an application programmer on one machine or process to send a datagram to an application program on another machine or process.

UNIX operating system. An operating system developed by Bell Laboratories that features multiprogramming in a multiuser environment. The UNIX operating system was originally developed for use on minicomputers, but has been adapted for mainframes and microcomputers. **Note:** The AIX operating system is IBM's implementation of the UNIX operating system.

user. Anyone who requires the services of a computing system.

user ID. A nonnegative integer, contained in an object of type *uid_t*, that is used to uniquely identify a system user.

V

Virtual Shared Disk, IBM. The function that allows application programs executing at different nodes of a system partition to access a raw logical volume as if it were local at each of the nodes. In actuality, the logical volume is local at only one of the nodes (the server node).

W

workstation. * (1) A configuration of input/output equipment at which an operator works. * (2) A terminal or microcomputer, usually one that is connected to a mainframe or to a network, at which a user can perform applications.

X

X Window System. A graphical user interface product.

Index

A

abnormal error condition
  releasing PTX shared memory manually 31
about this book ix
adding resource variables
  resource monitor 16
application programming interface
  Event Management (EMAPI) 4
  Resource Monitor (RMAPI) 4
application types
  command-based resource monitor 13
  daemon-based resource monitor 13
  EM client 2
  resource monitor 2
    considerations for choosing 14
  resource monitor added to another program 13
audience of this book ix

C

callback routine (Event Management)
  and event errors 61, 62
  and query errors 63
  and unregister errors 63
  for handling events 29, 57
  for query responses 29, 59, 61
  for unregister responses 29, 58
class, resource variable
  definition 11
  examples 11
  naming convention 11
  observation interval 11
command-based resource monitor
  characteristics 13
  configuring 13
  pseudocode example 17
commands
  dspcat
    displaying event management descriptions 23
  dspmsg
    displaying event management descriptions 23
  haemcfg
    configuring Event Management 15
  ipcrm 31
    using to release PTX shared memory manually 31
  SDRCreateObjects
    configuring Event Management 15
  SDRGetObjects
    finding out what resource data is available 20
    output example 22
commands *(continued)*
  spevent
    getting resource variable information 23
    output example 24
communication across nodes
  Event Management 3
components
  Event Management 1
concepts
  Event Management 1
configuration
  command-based resource monitor 13
  daemon-based resource monitor 13
  Event Management 14
    haemcfg command 15
    SDR classes 15
    use of haemloadcfg command 15
    use of SDRCreateObjects command 15
    use of System Data Repository (SDR) 14
  resource monitor logic 13
  SDR data for EMAPI 34
considerations
  for choosing a resource monitor type 14
control message commands
  HA_RR_CMD_ADDALL 18
  HA_RR_CMD_ADDV 18
  HA_RR_CMD_DELALL 19
  HA_RR_CMD_DELV 19
  HA_RR_CMD_INSTV 18
  HA_RR_CMD_REFRESH 19
Counter value type
  and class, resource variable 11
  and Event Management subsystem 7
  and Performance Monitor subsystem 7
  and PTX objects 7
  and PTX shared memory 30
  definition 6
  rate 6
  raw value 6
creating a server session with a resource monitor manager 49

D

daemon-based resource monitor
  characteristics 13
  configuring 13
  pseudocode example 19
data types, Event Management
  float 7
  long 7
  structured byte string (SBS) 7

© Copyright IBM Corp. 1996

DDS shared memory
See PTX shared memory
default predicate
definition 27
defining an Event Management callback routine
See callback routine (Event Management)
defining predicates
examples 26
rules 26
defining resource data to be monitored 5
deleting resource variable values
resource monitor 16
deleting resource variables from the RMAPI 50
descriptions, displaying event management
using the dspcat command 23
using the dspmsg command 23
displaying event management descriptions
using the dspcat command 23
using the dspmsg command 23
dspcat command
displaying event management descriptions 23
dspmsg command
displaying event management descriptions 23
dynamic data-supplier
definition (from PTX) 30
dynamic instantiation
definition 10

E

EM client
application types 2
definition 2
diagram 3
ending a session 30
function 2
multiple sessions 4
receiving responses 29
registering for events 28
sending event management queries 29
SIGPIPE signal handling 78
starting a session 27
tasks for writing
coding and testing 27
defining predicates 25
finding out what resource data is available 20
overview 20
unregistering for events 29
EM_Instance_Vector SDR class 43
EM_Resource_Class SDR class 44
EM_Resource_Monitor SDR class 45
EM_Resource_Variable SDR class 37
EM_Structured_Byte_String SDR class 42
EMAPI (Event Management Application Programming Interface)
configuration data (SDR) 34
EMAPI (Event Management Application Programming Interface) *(continued)*
error reference 116
predicate reference 138
resource variable reference 141
use by EM client 2
using 4
EMAPI subroutines
ha_em_get_ecgid subroutine 28
ha_em_receive_response subroutine 29
ha_em_send_command subroutine
HA_EM_CMD_QUERY command 29
HA_EM_CMD_REG command 28
HA_EM_CMD_REG2 command 28
HA_EM_CMD_UNREG command 29
ha_em_start_session subroutine 27
ha_rr_end_session subroutine 30
EMCDB
See Event Management Configuration Database
ending a session
EM client 30
resource monitor 16, 18
ending a session with a resource monitor manager 50
ending a session with Event Management 49
error condition
releasing PTX shared memory manually 31
error responses (EM client)
response block 29
errors
EMAPI 116
RMAPI 148
establishing a session with Event Management 49
event
definition 1, 25
event command group
definition 28
event generation
description 25
example 25
Event Management
application programming interfaces 4
communication across nodes 3
component
EM client 2
Event Management subsystem 2
resource monitor 1
configuration 14
event generation 25
graphical user interface 4
introduction 1
multiple sessions 4
name spaces 12
performance considerations 30
SDR classes
EM_Instance_Vector 43
EM_Resource_Class 44
EM_Resource_Monitor 45

Event Management *(continued)*
SDR classes *(continued)*
EM_Resource_Variable 37
EM_Structured_Byte_String 42
software components 1
system partitioning 3
Event Management Application Programming Interface (EMAPI)
See EMAPI
Event Management client
See EM client
Event Management components
diagram 3
Event Management configuration
haemcfg command 15
SDR classes 15
use of haemloadcfg command 15
use of SDRCreateObjects command 15
use of System Data Repository (SDR) 14
Event Management Configuration Database (EMCDB)
definition 15
refreshing 15
version number 15
Event Management subroutines
ending a session 49
establishing a session 49
get an event command group ID 49
ha_em_end_session 51
ha_em_get_ecgid 53
ha_em_receive_response 55
ha_em_restart_session 66
ha_em_send_command 68
ha_em_start_session 77
ha_rr_add_var 80
ha_rr_del_var 83
ha_rr_end_session 86
ha_rr_get_ctrlmsg 88
ha_rr_get_interval 93
ha_rr_init 95
ha_rr_makserv 97
ha_rr_reg_var 99
ha_rr_send_val 102
ha_rr_start_session 104
ha_rr_terminate 107
ha_rr_touch 109
ha_rr_unreg_var 111
receiving a response from the Event Management subsystem 49
restarting a session 49
send a command to the Event Management subsystem 49
Event Management subsystem
concepts 1
definition 2
diagram 3
function 2
Event Management subsystem *(continued)*
summary of subroutines 49
Event Management terms
Counter value type 6
default predicate 27
dynamic instantiation 10
EM client 2
event 1, 25
event command group 28
Event Management subsystem 2
event predicate 26
float data type 7
instance 2, 8
instance vector 8
instance vector element 8
instance vector element name 9
instance vector element value 9
location, resource variable instance 11
long data type 7
observation 2
predicate 2, 25
Quantity value type 7
raw value 6, 139
rearm predicate 26
resource 6
resource attribute 6
resource monitor 1
resource name 6
resource variable 1, 6
resource variable class 11
resource variable description 6
resource variable name 6
serial number, structured byte string 8
State value type 7
structured byte string (SBS) data type 7
structured field 7
system resource 1
Event Perspective 5
event predicate
definition 26
event responses
callback routine 29
response block 29
examples
class, resource variable 11
event generation 25
event management example
emapi_v01_ex01.c 218
emapi_v01_ex02.c 238
emapi_v01_ex03.c 258
emapi_v01_ex04.c 278
instance vector 9
location, resource variable instance 11
predicates 26
rearm predicate 27
resource data
output from the SDRGetObjects command 22

examples *(continued)*
resource data *(continued)*
using SP Perspectives 24
resource monitor example
rmapi_smp.loadsdr shell script 208
rmapi_smp.msg message file 206
rmapi_smp.unloadsdr shell script 214
rmapi_smpcmd.c 156
rmapi_smpdae.c 167
rmapi_smpsig.c 186
resource monitor pseudocode
command-based resource monitor 17
daemon-based resource monitor 19
resource variable name 9
resource variables 6
structured bytes string (SBS) definition 8
system resources 6
expression
synonymous with predicate 23

F

failure recovery
See recovery
finding out what resource data is available
using the SDRGetObjects command 20
float data type
definition 7

G

getting a control message
resource monitor 18
getting a control message from the RMAPI 49
getting an event command group ID 49
getting resource variable information
from SP Perspectives 23
getting the reporting interval for a class of resource variables 49
graphical user interface
Event Management 4

H

ha_em_end_session subroutine
ending a session 30
reference 51
ha_em_get_ecgid subroutine
getting the event command group 28
reference 53
ha_em_receive_response subroutine
receiving responses 29
reference 55
ha_em_restart_session subroutine
reference 66
ha_em_send_command subroutine
reference 68
registering for events
HA_EM_CMD_REG command 28
HA_EM_CMD_REG2 command 28
HA_EM_SCMD_REVAL subcommand 28
sending event management queries
HA_EM_CMD_QUERY command 29
unregistering for events
HA_EM_CMD_UNREG command 29
ha_em_start_session subroutine
reference 77
starting a session 27
ha_emapi_base.h header file
reference 128
ha_emapi.h header file
reference 122
ha_emcommon.h header file
reference 135
ha_rmapi.h header file
reference 151
ha_rr_add_var subroutine
adding resource variables 16
reference 80
HA_RR_CMD_ADDALL command 18
HA_RR_CMD_ADDV command 18
HA_RR_CMD_DELALL command 19
HA_RR_CMD_DELV command 19
HA_RR_CMD_INSTV command 18
HA_RR_CMD_REFRESH command 19
ha_rr_del_var subroutine
deleting resource variable values 16
reference 83
ha_rr_end_session subroutine
ending a session 16, 18
reference 86
ha_rr_get_ctrlmsg subroutine
getting a control message 18
reference 88
ha_rr_get_interval subroutine
reference 93
ha_rr_init subroutine
initializing 16, 17
reference 95
ha_rr_makserv subroutine
making a server session 17
reference 97
ha_rr_reg_var subroutine
reference 99
registering resource variables and instances 16, 17
ha_rr_send_val subroutine
reference 102
sending resource variable values 16
ha_rr_start_session subroutine
reference 104
starting a session 16, 17

ha_rr_terminate subroutine
and PTX shared memory 30
reference 107
terminating 17, 18
ha_rr_touch subroutine
reference 109
ha_rr_unreg_var subroutine
reference 111
haemcfg command
Event Management configuration 15
haemloadcfg command
use for Event Management configuration 15
haemloadlist file 15
header files
ha_emapi_base.h 128
ha_emapi.h 122
ha_emcommon.h 135
ha_rmapi.h 151

I

initializing
resource monitor 16, 17
instance
definition 8
instance vector
definition 8
examples 9
wildcards 9
instance vector element
definition 8
instance vector element name
definition 9
instance vector element value
definition 9
instantiation, dynamic
definition 10
ipcrm command
using to release PTX shared memory manually 31

K

killing a resource monitor
releasing PTX shared memory manually 31

L

load list file 15
local data-consumer program
definition (from PTX) 30
location, resource variable instance
definition 11
examples 11
logic, resource monitor
characteristics 13
configuring 13
long data type
definition 7

M

making a server session
resource monitor 17
manual pages for public code xvii
maximum
number of instance vector elements 144
meeting "send" frequency requirements 49
multiple sessions
Event Management 4

N

name spaces
Event Management 12
naming convention
class, resource variable 11
resource attribute 6
resource variable name 6

O

observation
definition (Event Management) 2
observation interval
and resource variable class 11
and resource variable value type 11
observation, previous
specifying in a predicate 26

P

partition, system
See system partition
performance
of Event Management operation 30
Performance Monitor subsystem
and Counter value type 7
and Quantity value type 7
and State value type 7
Performance Toolbox for AIX (PTX/6000)
See PTX
Perspectives graphical user interface
See SP Perspectives
predicate
creating 25
definition 2, 25
EMAPI 138
examples 26
rules for defining 26
prerequisite knowledge for this book ix
processing a control message
resource monitor 18

PTX object
and Counter value type 7
and Quantity value type 7
PTX shared memory
and Quantity and Counter value types 30
and resource monitors 30
releasing manually 31
PTX/6000
See PTX

Q

Quantity value type
and class, resource variable 11
and Event Management subsystem 7
and Performance Monitor subsystem 7
and PTX objects 7
and PTX shared memory 30
definition 7
query responses
callback routine 29
response block 29

R

raw value 139
definition 6
specifying in a predicate 26
rearm predicate
definition 26
example 27
using for inverse conditions 26
using for upper and lower boundaries 27
receiving a response from Event Management 49
receiving responses
EM client 29
registering a resource variable instance with the RMAPI 49
registering for events
EM client 28
registering resource variables
resource monitor 16, 17
registering to use the RMAPI 49
resource
definition 6
resource attribute
definition 6
resource data
defining what will be monitored 5
finding out what is available
using SP Perspectives 23
using the SDRGetObjects command 20
resource identifier
synonymous with instance vector 23
resource instance location
See location, resource instance
resource monitor
abnormal termination
releasing PTX shared memory manually 31
added to another program 13
adding resource variables 16
and PTX shared memory 30
application types 2
command-based 13
control message commands
HA_RR_CMD_ADDALL 18
HA_RR_CMD_ADDV 18
HA_RR_CMD_DELALL 19
HA_RR_CMD_DELV 19
HA_RR_CMD_INSTV 18
HA_RR_CMD_REFRESH 19
daemon-based 13
definition 1
deleting resource variable values 16
diagram 3
ending a session 16, 18
examples
pseudocode for command-based resource monitor 17
pseudocode for daemon-based resource monitor 19
function 1
getting a control message 18
initializing 16, 17
making a server session 17
processing a control message 18
registering resource variables and instances 16, 17
sending resource variable values 16
starting a session 16, 17
tasks for writing
choosing the resource monitor type 12
coding and testing 16
configuring 14
defining the resource data 5
overview 5
terminating 17, 18
testing 16
Resource Monitor Application Programming Interface (RMAPI)
See RMAPI
resource name
definition 6
resource variable
definition 1, 6
instance 2
naming convention 6
resource variable class
See class, resource variable
resource variable description
definition 6
resource variable information, displaying
using the spevent command 23

resource variable name
definition 6
examples 9
resource variables
dynamically instantiable 10
EMAPI 141
RMAPI 141
response block (Event Management)
for error responses 29
for handling events 29
for query responses 29
for unregister responses 29
restarting a session with Event Management 49
RMAPI (Resource Monitor Application Programming Interface)
error reference 148
pseudocode for command-based resource monitor 17
pseudocode for daemon-based resource monitor 19
resource variable reference 141
summary of subroutines 49—50
use by resource monitor 2
using 4
RMAPI control message commands
HA_RR_CMD_ADDALL command 18
HA_RR_CMD_ADDV command 18
HA_RR_CMD_DELALL command 19
HA_RR_CMD_DELV command 19
HA_RR_CMD_INSTV command 18
HA_RR_CMD_REFRESH command 19
RMAPI subroutines
creating a server session with a resource monitor manager 49
deleting resource variables from the RMAPI 50
ending a session with a resource monitor manager 50
getting a control message from the RMAPI 49
getting the reporting interval for a class of resource variables 49
ha_rr_add_var subroutine 16
ha_rr_del_var subroutine 16
ha_rr_end_session subroutine 16, 18
ha_rr_get_ctrlmsg subroutine 18
ha_rr_init subroutine 16, 17
ha_rr_makserv subroutine 17
ha_rr_reg_var subroutine 16, 17
ha_rr_send_val subroutine 16
ha_rr_start_session subroutine 16, 17
ha_rr_terminate subroutine 17, 18
meeting "send" frequency requirements 49
registering a resource variable instance with the RMAPI 49
registering to use the RMAPI 49
sending variable values to the RMAPI 49
starting a session with a resource monitor manager 49
RMAPI subroutines *(continued)*
terminating use of the RMAPI 50
unregistering a resource variable instance 50

S

SBS (structured byte string)
See structured byte string
SDR (System Data Repository)
classes for Event Management configuration 15
EM configuration data 34
use for Event Management configuration 14
SDR classes
Event Management
EM_Instance_Vector 43
EM_Resource_Class 44
EM_Resource_Monitor 45
EM_Resource_Variable 37
EM_Structured_Byte_String 42
SDRCreateObjects command
configuring Event Management 15
SDRGetObjects command
example of output 22
finding out what resource data is available 20
sending a command to Event Management 49
sending event management queries
EM client 29
sending resource variable values
resource monitor 16
sending variable values to the RMAPI 49
serial number, structured byte string
definition 8
session with Event Management
ending 49
restarting 49
starting 49
shared memory
See PTX shared memory
signal handling
EM client 78
SIGPIPE signal
handling for EM clients 78
SP Perspectives
example of displaying resource data 24
getting resource variable information 23
terminology differences from APIs 23
using for Event Management 5
spevent command
example of output 24
getting resource variable information 23
starting a session
EM client 27
resource monitor 16, 17
starting a session with a resource monitor manager 49
State value type
and Event Management subsystem 7

State value type *(continued)*
and Performance Monitor subsystem 7
definition 7
structured byte field
specifying in a predicate 26
structured byte string (SBS)
example 8
structured byte string (SBS) data type
definition 7
structured field
definition 7
summary of subroutines
Event Management 49
Resource Monitor API 49—50
System Data Repository
See SDR
system partition
support by Event Management 3
system resource
definition 1

T

tasks
writing EM clients
coding and testing 27
defining predicates 25
finding out what resource data is available 20
overview 20
writing resource monitors
choosing the resource monitor type 12
coding and testing 16
configuring 14
defining the resource data 5
overview 5
terminating
resource monitor 17, 18
use of the RMAPI 50
termination, abnormal
resource monitor
releasing PTX shared memory manually 31
terminology differences
between APIs and SP Perspectives 23
terms, Event Management
Counter value type 6
default predicate 27
dynamically instantiable resource variables 10
EM client 2
EMCDB (Event Management Configuration Database) 15
event 1, 25
event command group 28
Event Management subsystem 2
event predicate 26
float data type 7
instance 2, 8
terms, Event Management *(continued)*
instance vector 8
instance vector element 8
instance vector element name 9
instance vector element value 9
location, resource variable instance 11
long data type 7
observation 2
predicate 2, 25
Quantity value type 7
raw value 6, 139
rearm predicate 26
resource 6
resource attribute 6
resource monitor 1
resource name 6
resource variable 1, 6
resource variable class 11
resource variable description 6
resource variable name 6
serial number, structured byte string 8
State value type 7
structured byte string (SBS) data type 7
structured field 7
system resource 1
testing
new resource monitor 16
trademarks vii

U

unregister responses
callback routine 29
response block 29
unregistering a resource variable instance 50
unregistering for events
EM client 29
using application programming interfaces
EMAPI 4
RMAPI 4
using graphical user interfaces
SP Perspectives 5

V

value types, Event Management
Counter 6
Quantity 7
State 7
version number
EMCDB 15

W

wildcard
instance vector element 9