NAME

saf – Service Access Facility

AVAILABILITY

SUNWcsr

DESCRIPTION

The SAF generalizes the procedures for service access so that login access on the local system and network access to local services are managed in similar ways. Under the SAF, systems may access services using a variety of port monitors, including ttymon, the listener, and port monitors written expressly for a user's application. The manner in which a port monitor observes and manages access ports is specific to the port monitor and not to any component of the SAF. Users may therefore extend their systems by developing and installing their own port monitors. One of the important features of the SAF is that it can be extended in this way by users.

Relative to the SAF, a service is a process that is started. There are no restrictions on the functions a service may provide. The SAF consists of a controlling process, the service access controller (SAC), and two administrative levels corresponding to two levels in the supporting directory structure. The top administrative level is concerned with port monitor administration, the lower level with service administration. The SAC is documented in the **sac**(1M) man page. The administrative levels and associated utilities are documented in the *System Administration Guide - Volume II*. The requirements for writing port monitors and the functions a port monitor must perform to run under the SAF and the SAC are documented here.

Port Monitors

A port monitor is a process that is responsible for monitoring a set of homogeneous, incoming ports on a machine. A port monitor's major purpose is to detect incoming service requests and to dispatch them appropriately.

A port is an externally seen access point on a system. A port may be an address on a network (TSAP or PSAP), a hardwired terminal line, an incoming phone line, etc. The definition of what constitutes a port is strictly a function of the port monitor itself.

A port monitor performs certain basic functions. Some of these are required to conform to the SAF; others may be specified by the requirements and design of the port monitor itself. Port monitors have two main functions:

- Managing ports
- Monitoring ports for indications of activity

Port Management

The first function of a port monitor is to manage a port. The actual details of how a port is managed are defined by the person who defines the port monitor. A port monitor is not restricted to handling a single port; it may handle multiple ports simultaneously.

Note: Some examples of port management are setting the line speed on incoming phone connections, binding an appropriate network address, reinitializing the port when the service terminates, outputting a prompt, etc.

Activity Monitoring

The second function of a port monitor is to monitor the port or ports for which it is responsible for indications of activity. Two types of activity may be detected.

- The first is an indication to the port monitor to take some port monitor-specific action. Pressing the break key to indicate that the line speed should be cycled is an example of a port monitor activity. Not all port monitors need to recognize and respond to the same indications. The indication used to attract the attention of the port monitor is defined by the person who defines the port monitor.
- The second is an incoming service request. When a service request is received, a port monitor must be able to determine which service is being requested from the port on which the request is received. Note that the same service may be available on more than one port.

Other Port Monitor Functions

This section briefly describes other port monitor functions.

Restricting Access to the System

A port monitor must be able to restrict access to the system without disturbing services that are still running. In order to do this, a port monitor must maintain two internal states: enabled and disabled. The port monitor starts in the state indicated by the ISTATE environment variable provided by the sac. See sac(1M) for details. Enabling or disabling a port monitor affects all ports for which the port monitor is responsible. If a port monitor is responsible for a single port, only that port will be affected. If a port monitor is responsible for multiple ports, the entire collection of ports will be affected. Enabling or disabling a port monitor is a dynamic operation: it causes the port monitor to change its internal state. The effect does not persist across new invocations of the port monitor. Enabling or disabling an individual port, however, is a static operation: it causes a change to an administrative file. The effect of this change will persist across new invocations of the port monitor.

Creating **utmp** Entries

Port monitors are responsible for creating **utmp** entries with the type field set to USER_PROCESS for services they start, if this action has been specified (that is, if −**fu** was specified in the **pmadm** command line that added the service). These **utmp** entries may in turn be modified by the service. When the service terminates, the **utmp** entry must be set to DEAD_PROCESS.

Port Monitor Process IDs and Lock Files

When a port monitor starts, it writes its process id into a file named **_pid** in the current directory and places an advisory lock on the file.

Changing the Service Environment: Running **doconfig**(3N)

Before invoking the service designated in the port monitor administrative file, **_pmtab**, a port monitor must arrange for the per-service configuration script to be run, if one exists, by calling the library function **doconfig**(3N). Because the per-service configuration script may specify the execution of restricted commands, as well as for other security reasons, port monitors are invoked with root permissions. The details of how services are invoked are specified by the person who defines the port monitor.

Terminating a Port Monitor

A port monitor must terminate itself gracefully on receipt of the signal SIGTERM. The termination sequence is the following:

1. The port monitor enters the stopping state; no further service requests are accepted.
2. Any attempt to re-enable the port monitor will be ignored.
3. The port monitor yields control of all ports for which it is responsible. It must be possible for a new instantiation of the port monitor to start correctly while a previous instantiation is stopping.
4. The advisory lock on the process id file is released. Once this lock is released, the contents of the process id file are undefined and a new invocation of the port monitor may be started.

SAF Files

This section briefly covers the files used by the SAF.

The Port Monitor Administrative File

A port monitor's current directory contains an administrative file named **_pmtab**; **_pmtab** is maintained by the **pmadm** command in conjunction with a port monitor-specific administrative command.

Note: The port monitor administrative command for a listen port monitor is **nlsadmin**(1M); the port monitor administrative command for ttymon is **ttyadm**(1M). Any port monitor written by a user must be provided with an administrative command specific to that port monitor to perform similar functions.

Per-Service Configuration Files

A port monitor's current directory also contains the per-service configuration scripts, if they exist. The names of the per-service configuration scripts correspond to the service tags in the **_pmtab** file.

Private Port Monitor Files

A port monitor may create private files in the directory **/var/saf/***tag*, where *tag* is the name of the port monitor. Examples of private files are log files or temporary files.

The SAC/Port Monitor Interface

The SAC creates two environment variables for each port monitor it starts:

- PMTAG
- ISTATE

This variable is set to a unique port monitor tag by the SAC. The port monitor uses this tag to identify itself in response to **sac** messages. ISTATE is used to indicate to the port monitor what its initial internal state should be. ISTATE is set to "enabled" or "disabled" to indicate that the port monitor is to start in the enabled or disabled state respectively.

The SAC performs a periodic sanity poll of the port monitors. The SAC communicates with port monitors through FIFOs. A port monitor should open _pmpipe, in the current directory, to receive messages from the SAC and **../_sacpipe** to send return messages to the SAC.

Message Formats

This section describes the messages that may be sent from the SAC to a port monitor (**sac** messages), and from a port monitor to the SAC (port monitor messages). These messages are sent through FIFOs and are in the form of C structures.

sac Messages

The format of messages from the SAC is defined by the structure **sacmsg**:

```
struct sacmsg
{
        int sc_size; /* size of optional data portion */
        char sc_type; /* type of message */
};
```

The SAC may send four types of messages to port monitors. The type of message is indicated by setting the **sc_type** field of the **sacmsg** structure to one of the following:

SC_STATUS	status request
SC_ENABLE	enable message
SC_DISABLE	disable message
SC_READDB	message indicating that the port monitor's _pmtab file should be read

The **sc_size** field indicates the size of the optional data part of the message. See "Message Classes." For Solaris, **sc_size** should always be set to 0. A port monitor must respond to every message sent by the sac.

Port Monitor Messages

The format of messages from a port monitor to the SAC is defined by the structure **pmmsg**:

```
struct pmmsg {
        char pm_type;        /* type of message */
        unchar pm_state;     /* current state of port monitor */
        char pm_maxclass;    /* maximum message class this port
                                monitor understands */
        char pm_tag[PMTAGSIZE + 1];  /* port monitor's tag */
        int pm_size;         /* size of optional data portion */
};
```

Port monitors may send two types of messages to the SAC. The type of message is indicated by setting the **pm_type** field of the **pmmsg** structure to one of the following:

PM_STATUS state information

PM_UNKNOWN	negative acknowledgment

For both types of messages, the **pm_tag** field is set to the port monitor's tag and the **pm_state** field is set to the port monitor's current state. Valid states are:

PM_STARTING	starting
PM_ENABLED	enabled
PM_DISABLED	disabled
PM_STOPPING	stopping

The current state reflects any changes caused by the last message from the SAC. The status message is the normal return message. The negative acknowledgment should be sent only when the message received is not understood. **pm_size** indicates the size of the optional data part of the message. **pm_maxclass** is used to specify a message class. Both are discussed under "Message Classes." In Solaris, always set **pm_maxclass** to 1 and **sc_size** to 0. Port monitors may never initiate messages; they may only respond to messages that they receive.

Message Classes

The concept of message class has been included to accommodate possible SAF extensions. The messages described above are all class 1 messages. None of these messages contains a variable data portion; all pertinent information is contained in the message header. If new messages are added to the protocol, they will be defined as new message classes (for example, class 2). The first message the SAC sends to a port monitor will always be a class 1 message. Since all port monitors, by definition, understand class 1 messages, the first message the SAC sends is guaranteed to be understood. In its response to the SAC, the port monitor sets the **pm_maxclass** field to the maximum message class number for that port monitor. The SAC will not send messages to a port monitor from a class with a larger number than the value of **pm_maxclass**. Requests that require messages of a higher class than the port monitor can understand will fail. For Solaris, always set **pm_maxclass** to 1.

Note: For any given port monitor, messages of class **pm_maxclass** and messages of all classes with values lower than **pm_maxclass** are valid. Thus, if the **pm_maxclass** field is set to 3, the port monitor understands messages of classes 1, 2, and 3. Port monitors may not generate messages; they may only respond to messages. A port monitor's response must be of the same class as the originating message. Since only the SAC can generate messages, this protocol will function even if the port monitor is capable of dealing with messages of a higher class than the SAC can generate. **pm_size** (an element of the pmmsg structure) and **sc_size** (an element of the **sacmsg** structure) indicate the size of the optional data part of the message. The format of this part of the message is undefined. Its definition is inherent in the type of message. For Solaris, always set both **sc_size** and **pm_size** to 0.

Administrative Interface

This section discusses the port monitor administrative files available under the SAC.

The SAC Administrative File _sactab

The service access controller's administrative file contains information about all the port monitors for which the SAC is responsible. This file exists on the delivered system. Initially, it is empty except for a single comment line that contains the version number of the SAC. Port monitors are added to the system by making entries in the SAC's administrative file. These entries should be made using the administrative command **sacadm**(1M) with a **–a** option. **sacadm**(1M) is also used to remove entries from the SAC's administrative file. Each entry in the SAC's administrative file contains the following information.

PMTAG — A unique tag that identifies a particular port monitor. The system administrator is responsible for naming a port monitor. This tag is then used by the SAC to identify the port monitor for all administrative purposes. PMTAG may consist of up to 14 alphanumeric characters.

PMTYPE — The type of the port monitor. In addition to its unique tag, each port monitor has a type designator. The type designator identifies a group of port monitors that are different invocations of the same entity. ttymon and listen are examples of valid port monitor types. The type designator is used to facilitate the administration of groups of related port monitors. Without a type designator, the system administrator has no way of knowing which port monitor tags correspond to port monitors of the same type. PMTYPE may consist of up to 14 alphanumeric characters.

FLGS — The flags that are currently defined are:

- d — When started, do not enable the port monitor.
- x — Do not start the port monitor.

If no flag is specified, the default action is taken. By default a port monitor is started and enabled.

RCNT — The number of times a port monitor may fail before being placed in a failed state. Once a port monitor enters the failed state, the SAC will not try to restart it. If a count is not specified when the entry is created, this field is set to 0. A restart count of 0 indicates that the port monitor is not to be restarted when it fails.

COMMAND — A string representing the command that will start the port monitor. The first component of the string, the command itself, must be a full path name.

The Port Monitor Administrative File _pmtab

Each port monitor will have two directories for its exclusive use. The current directory will contain files defined by the SAF (**_pmtab**, **_pid**) and the per-service configuration scripts, if they exist. The directory **/var/saf/***pmtag,* where *pmtag* is the tag of the port monitor, is available for the port monitor's private files. Each port monitor has its own administrative file. The **pmadm**(1M) command should be used to add, remove, or modify service entries in this file. Each time a change is made using **pmadm**(1M), the corresponding port monitor rereads its administrative file. Each entry in a port monitor's administrative file defines how the port monitor treats a specific port and what service is to be invoked on that port. Some fields must be present for all types of port monitors. Each entry must include a service tag to identify the service uniquely and an identity to be assigned to the service when it is started (for example, root).

Note: The combination of a service tag and a port monitor tag uniquely define an instance of a service. The same service tag may be used to identify a service under a different port monitor. The record must also contain port monitor specific data (for example, for a ttymon port monitor, this will include the prompt string which is meaningful to ttymon). Each type of port monitor must provide a command that takes the necessary port monitor-specific data as arguments and outputs these data in a form suitable for storage in the file. The **ttyadm**(1M) command does this for ttymon and **nlsadmin**(1M) does it for listen. For a user-defined port monitor, a similar administrative command must also be supplied. Each service entry in the port monitor administrative file must have the following format and contain the information listed below:

svctag:flgs:id:reserved:reserved:reserved:pmspecific# comment

SVCTAG is a unique tag that identifies a service. This tag is unique only for the port monitor through which the service is available. Other port monitors may offer the same or other services with the same tag. A service requires both a port monitor tag and a service tag to identify it uniquely. SVCTAG may consist of up to 14 alphanumeric characters. The service entries are defined as:

FLGS Flags with the following meanings may currently be included in this field:

- x Do not enable this port. By default the port is enabled.
- u Create a utmp entry for this service. By default no utmp entry is created for the service.

ID The identity under which the service is to be started. The identity has the form of a login name as it appears in **/etc/passwd**.

PMSPECIFIC
Examples of port monitor information are addresses, the name of a process to execute, or the name of a STREAMS pipe to pass a connection through. This information will vary to meet the needs of each different type of port monitor.

COMMENT
A comment associated with the service entry. Note: Port monitors may ignore the *u* flag if creating a utmp entry for the service is not appropriate to the manner in which the service is to be invoked. Some services may not start properly unless utmp entries have been created for them (for example, login). Each port monitor administrative file must contain one special comment of the form:

VERSION=value

where *value* is an integer that represents the port monitor's version number. The version number defines the format of the port monitor administrative file. This comment line is created automatically when a port monitor is added to the system. It appears on a line by itself, before the service entries.

Monitor-Specific Administrative Command

Previously, two pieces of information included in the **_pmtab** file were described: the port monitor's version number and the port monitor part of the service entries in the port monitor's **_pmtab** file. When a new port monitor is added, the version number must be known so that the **_pmtab** file can be correctly initialized. When a new service is added, the port monitor part of the **_pmtab** entry must be formatted correctly. Each port monitor must have an administrative command to perform these two tasks. The person who defines the port monitor must also define such an administrative command and its input options. When the command is invoked with these options, the information required for the port monitor part of the service entry must be correctly formatted for inclusion in the port monitor's **_pmtab** file and must be written to the standard output. To request the version number the command must be invoked with a −**V** option; when it is invoked in this way, the port monitor's current version number must be written to the standard output. If the command fails for any reason during the execution of either of these tasks, no data should be written to standard output.

The Port Monitor/Service Interface

The interface between a port monitor and a service is determined solely by the service. Two mechanisms for invoking a service are presented here as examples.

New Service Invocations

The first interface is for services that are started anew with each request. This interface requires the port monitor to first **fork**(2) a child process. The child will eventually become the designated service by performing an **exec**(1). Before the **exec**(1) happens, the port monitor may take some port monitor-specific action; however, one action that must occur is the interpretation of the per-service configuration script, if one is present. This is done by calling the library routine **doconfig**(3N).

Standing Service Invocations

The second interface is for invocations of services that are actively running. To use this interface, a service must have one end of a stream pipe open and be prepared to receive connections through it.

Port Monitor Requirements

To implement a port monitor, several generic requirements must be met. This section summarizes these requirements. In addition to the port monitor itself, an administrative command must be supplied.

Initial Environment

When a port monitor is started, it expects an initial execution environment in which:

- It has no file descriptors open
- It cannot be a process group leader
- It has an entry in **/etc/utmp** of type LOGIN_PROCESS

- An environment variable, ISTATE, is set to "enabled" or "disabled" to indicate the port monitor's correct initial state
- An environment variable, PMTAG, is set to the port monitor's assigned tag
- The directory that contains the port monitor's administrative files is its current directory
- The port monitor is able to create private files in the directory **/var/saf/***tag*, where *tag* is the port monitor's tag
- The port monitor is running with user id 0 (root)

Important Files

Relative to its current directory, the following key files exist for a port monitor.

_config The port monitor's configuration script. The port monitor configuration script is run by the SAC. The SAC is started by **init**(1M) as a result of an entry in **/etc/inittab** that calls **sac**(1M).

_pid The file into which the port monitor writes its process id.

_pmtab The port monitor's administrative file. This file contains information about the ports and services for which the port monitor is responsible.

_pmpipe The FIFO through which the port monitor will receive messages from the SAC.

svctag The per-service configuration script for the service with the tag *svctag*.

../_sacpipe
The FIFO through which the port monitor will send messages to **sac**(1M).

Port Monitor Responsibilities

A port monitor is responsible for performing the following tasks in addition to its port monitor function:

- Write its process id into the file **_pid** and place an advisory lock on the file
- Terminate gracefully on receipt of the signal SIGTERM
- Follow the protocol for message exchange with the SAC

A port monitor must perform the following tasks during service invocation:

- Create a **utmp** entry if the requested service has the **u** flag set in **_pmtab**

  Note: Port monitors may ignore this flag if creating a **utmp** entry for the service does not make sense because of the manner in which the service is to be invoked. On the other hand, some services may not start properly unless utmp entries have been created for them.
- Interpret the per-service configuration script for the requested service, if it exists, by calling the **doconfig**(3N) library routine

Configuration Files and Scripts

The library routine **doconfig**(3N), defined in **libnsl.so**, interprets the configuration scripts contained in the files **/etc/saf/_sysconfig** (the per-system configuration file), and **/etc/saf/***pmtag***/_config** (per-port monitor configuration files); and in **/etc/saf/***pmtag***/svctag** (per-service configuration files). Its syntax is:

```
#include <sac.h>
int doconfig (int fd, char *script, long rflag);
```

script is the name of the configuration script; *fd* is a file descriptor that designates the stream to which stream manipulation operations are to be applied; *rflag* is a bitmask that indicates the mode in which script is to be interpreted. *rflag* may take two values, NORUN and NOASSIGN, which may be or'd. If *rflag* is zero, all commands in the configuration script are eligible to be interpreted. If *rflag* has the NOASSIGN bit set, the assign command is considered illegal and will generate an error return. If *rflag* has the NORUN bit set, the run and runwait commands are considered illegal and will generate error returns. If a command in the script fails, the interpretation of the script ceases at that point and a positive integer is returned; this number indicates which line in the script failed. If a system error occurs, a value of −1 is returned. If a script fails, the process whose environment was being established should not be started. In the example, **doconfig**(3N) is used to interpret a per-service configuration script.

```
. . .
        if ((i = doconfig (fd, svctag, 0)) != 0){
        error ("doconfig failed on line %d of script %s",i,svctag);
}
```

The Per-System Configuration File

The per-system configuration file, **/etc/saf/_sysconfig**, is delivered empty. It may be used to customize the environment for all services on the system by writing a command script in the interpreted language described in this chapter and on the **doconfig**(3N) manpage. When the SAC is started, it calls the **doconfig**(3N) function to interpret the per-system configuration script. The SAC is started when the system enters multiuser mode.

Per-Port Monitor Configuration Files

Per-port monitor configuration scripts (**/etc/saf/***pmtag***/_config)** are optional. They allow the user to customize the environment for any given port monitor and for the services that are available through the ports for which that port monitor is responsible. Per-port monitor configuration scripts are written in the same language used for per-system configuration scripts. The per-port monitor configuration script is interpreted when the port monitor is started. The port monitor is started by the SAC after the SAC has itself been started and after it has run its own configuration script, **/etc/saf/_sysconfig**. The per-port monitor configuration script may override defaults provided by the per-system configuration script.

Per-Service Configuration Files

Per-service configuration files allow the user to customize the environment for a specific service. For example, a service may require special privileges that are not available to the general user. Using the language described in the **doconfig**(3N) manpage, you can write a script that will grant or limit such special privileges to a particular service offered through a particular port monitor. The per-service configuration may override defaults provided by higher-level configuration scripts. For example, the per-service configuration script may specify a set of STREAMS modules other than the default set.

The Configuration Language

The language in which configuration scripts are written consists of a sequence of commands, each of which is interpreted separately. The following reserved keywords are defined: **assign**, **push**, **pop**, **runwait**, and **run**. The comment character is #. Blank lines are not significant. No line in a command script may exceed 1024 characters.

assign *variable=value*

Used to define environment variables; *variable* is the name of the environment variable and *value* is the value to be assigned to it. The value assigned must be a string constant; no form of parameter substitution is available. *value* may be quoted. The quoting rules are those used by the shell for defining environment variables. **assign** will fail if space cannot be allocated for the new variable or if any part of the specification is invalid.

push *module1*[, *module2, module3*, . . .]

Used to push STREAMS modules onto the stream designated by **fd**; *module1* is the name of the first module to be pushed, *module2* is the name of the second module to be pushed, and so on. The command will fail if any of the named modules cannot be pushed. If a module cannot be pushed, the subsequent modules on the same command line will be ignored and modules that have already been pushed will be popped.

pop [*module*]

Used to pop STREAMS modules off the designated stream. If **pop** is invoked with no arguments, the top module on the stream is popped. If an argument is given, modules will be popped one at a time until the named module is at the top of the stream. If the named module is not on the designated stream, the stream is left as it was and the command fails. If *module* is the special keyword **ALL**, then all modules on the stream will be popped. Note that only modules above the topmost driver are affected.

runwait *command*

The **runwait** command runs a command and waits for it to complete; *command* is the path name of the command to be run. The command is run with **/bin/sh −c** prepended to it; shell scripts may thus be executed from configuration scripts. The **runwait** command will fail if command cannot be found or cannot be executed, or if *command* exits with a nonzero status.

run *command*

The **run** command is identical to **runwait** except that it does not wait for command to complete; *command* is the path name of the command to be run. **run** will not fail unless it is unable to create a child process to execute the command. Although they are syntactically indistinguishable, some of the commands available to **run** and **runwait** are interpreter built-in commands. Interpreter built-ins are used when it is necessary to alter the state of a process within the context of that process. The **doconfig** interpreter built-in commands are similar to the shell special commands and, like these, they do not spawn another process for execution. See the **sh**(1) man page. The initial set of built-in commands is: **cd**, **ulimit**, **umask**.

Sample Port Monitor Code

This example shows an example of a "null" port monitor that simply responds to messages from the SAC.

```
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>
# include <fcntl.h>
# include <signal.h>
# include <sac.h>

char Scratch[BUFSIZ]; /* scratch buffer */
char Tag[PMTAGSIZE + 1]; /* port monitor's tag */
FILE *Fp; /* file pointer for log file */
FILE *Tfp; /* file pointer for pid file */
char State; /* portmonitor's current state*/

main(argc, argv)
          int argc;
          char *argv[];
{
          char *istate;
          strcpy(Tag, getenv("PMTAG"));
/*
 * open up a log file in port monitor's private directory
 */
          sprintf(Scratch, "/var/saf/%s/log", Tag);
          Fp = fopen(Scratch, "a+");
          if (Fp == (FILE *)NULL)
                    exit(1);
          log(Fp, "starting");
/*
 * retrieve initial state (either "enabled" or "disabled") and set
 * State accordingly
 */
          istate = getenv("ISTATE");
```

```
        sprintf(Scratch, "ISTATE is %s", istate);
        log(Fp, Scratch);
        if (!strcmp(istate, "enabled"))
                State = PM_ENABLED;
        else if (!strcmp(istate, "disabled"))
                State = PM_DISABLED;
        else {
                log(Fp, "invalid initial state");
                exit(1);
        }
        sprintf(Scratch, "PMTAG is %s", Tag);
        log(Fp, Scratch);
/*
 * set up pid file and lock it to indicate that we are active
 */
        Tfp = fopen("_pid", "w");
        if (Tfp == (FILE *)NULL) {
                log(Fp, "couldn't open pid file");
                exit(1);
        }
        if (lockf(fileno(Tfp), F_TEST, 0) < 0) {
                log(Fp, "pid file already locked");
                exit(1);
        }

        log(Fp, "locking file");
        if (lockf(fileno(Tfp), F_LOCK, 0) < 0) {
                log(Fp, "lock failed");
                exit(1);
        }
        fprintf(Tfp, "%d", getpid());
        fflush(Tfp);

/*
 * handle poll messages from the sac ... this function never returns
 */
        handlepoll();
        pause();
        fclose(Tfp);
        fclose(Fp);
}

handlepoll()
{
        int pfd; /* file descriptor for incoming pipe */
```

```
        int sfd; /* file descriptor for outgoing pipe */
        struct sacmsg sacmsg; /* incoming message */
        struct pmmsg pmmsg; /* outgoing message */
/*
 * open pipe for incoming messages from the sac
 */
        pfd = open("_pmpipe", O_RDONLY | O_NONBLOCK);
        if (pfd < 0) {
                log(Fp, "_pmpipe open failed");
                exit(1);
        }
/*
 * open pipe for outgoing messages to the sac
 */
        sfd = open("../_sacpipe", O_WRONLY);
        if (sfd < 0) {
                log(Fp, "_sacpipe open failed");
                exit(1);
        }
/*
 * start to build a return message; we only support class 1 messages
 */
        strcpy(pmmsg.pm_tag, Tag);
        pmmsg.pm_size = 0;
        pmmsg.pm_maxclass = 1;
/*
 * keep responding to messages from the sac
 */
        for (;;) {
                if (read(pfd, &sacmsg, sizeof(sacmsg)) != sizeof(sacmsg)) {
                        log(Fp, "_pmpipe read failed");
                        exit(1);
                }
/*
 * determine the message type and respond appropriately
 */
                switch (sacmsg.sc_type) {
                        case SC_STATUS:
                                log(Fp, "Got SC_STATUS message");
                                pmmsg.pm_type = PM_STATUS;
                                pmmsg.pm_state = State;
                                break;
                        case SC_ENABLE:
                                /*note internal state change below*/
                                log(Fp, "Got SC_ENABLE message");
```

```
                                        pmmsg.pm_type = PM_STATUS;
                                        State = PM_ENABLED;
                                        pmmsg.pm_state = State;
                                        break;
                                case SC_DISABLE:
                                        /*note internal state change below*/
                                        log(Fp, "Got SC_DISABLE message");
                                        pmmsg.pm_type = PM_STATUS;
                                        State = PM_DISABLED;
                                        pmmsg.pm_state = State;
                                        break;
                                case SC_READDB:
                                        /*
                                         * if this were a fully functional port
                                         * monitor it would read _pmtab here
                                         * and take appropriate action
                                         */
                                        log(Fp, "Got SC_READDB message");
                                        pmmsg.pm_type = PM_STATUS;
                                        pmmsg.pm_state = State;
                                        break;
                                default:
                                        sprintf(Scratch, "Got unknown message <%d>",
                                        sacmsg.sc_type);
                                        log(Fp, Scratch);
                                        pmmsg.pm_type = PM_UNKNOWN;
                                        pmmsg.pm_state = State;
                                        break;
                        }
/*
 * send back a response to the poll
 * indicating current state
 */
                        if (write(sfd, &pmmsg, sizeof(pmmsg)) != sizeof(pmmsg))
                                log(Fp, "sanity response failed");
                }
}
/*
 * general logging function
 */
log(fp, msg)
        FILE *fp;
        char *msg;
{
        fprintf(fp, "%d; %s\n", getpid(), msg);
```

```
        fflush(fp);
}
```

The sac.h Header File

The following example shows the sac.h header file.

```
/* length in bytes of a utmp id */
# define IDLEN 4
/* wild character for utmp ids */
# define SC_WILDC 0xff
/* max len in bytes for port monitor tag */
# define PMTAGSIZE 14
/*
 * values for rflag in doconfig()
 */
/* don't allow assign operations */
# define NOASSIGN 0x1
/* don't allow run or runwait operations */
# define NORUN 0x2
/*
 * message to SAC (header only). This header is forever fixed. The
 * size field (pm_size) defines the size of the data portion of the
 * message, which follows the header. The form of this optional data
 * portion is defined strictly by the message type (pm_type).
 */
struct pmmsg {
        char pm_type;           /* type of message */
        unchar pm_state;        /* current state of pm */
        char pm_maxclass;       /* max message class this port monitor
                                        understands */
        char pm_tag[PMTAGSIZE + 1]; /* pm's tag */
        int pm_size;            /* size of opt data portion */
};
/*
 * pm_type values
 */
# define PM_STATUS 1 /* status response */
# define PM_UNKNOWN 2 /* unknown message was received */
/*
 * pm_state values
 */
/*
 * Class 1 responses
 */
# define PM_STARTING 1  /* monitor in starting state */
# define PM_ENABLED 2   /* monitor in enabled state */
```

```
# define PM_DISABLED 3   /* monitor in disabled state */
# define PM_STOPPING 4   /* monitor in stopping state */
/*
 * message to port monitor
 */
struct sacmsg {
        int sc_size;        /* size of optional data portion */
        char sc_type;       /* type of message */
};
/*
 * sc_type values
 * These represent commands that the SAC sends to a port monitor.
 * These commands are divided into "classes" for extensibility. Each
 * subsequent "class" is a superset of the previous "classes" plus
 * the new commands defined within that "class". The header for all
 * commands is identical; however, a command may be defined such that
 * an optional data portion may be sent in addition to the header.
 * The format of this optional data piece is self-defining based on
 * the command. Important note: the first message sent by the SAC
 * will always be a class 1 message. The port monitor response
 * indicates the maximum class that it is able to understand. Another
 * note is that port monitors should only respond to a message with
 * an equivalent class response (i.e. a class 1 command causes a
 * class 1 response).
 */
/*
 * Class 1 commands (currently, there are only class 1 commands)
 */
# define SC_STATUS 1    /* status request *
# define SC_ENABLE 2    /* enable request */
# define SC_DISABLE 3   /* disable request */
# define SC_READDB 4    /* read pmtab request */
/*
 * 'errno' values for Saferrno, note that Saferrno is used by both
 * pmadm and sacadm and these values are shared between them
 */
# define E_BADARGS 1   /* bad args/ill-formed cmd line */
# define E_NOPRIV 2    /* user not priv for operation */
# define E_SAFERR 3    /* generic SAF error */
# define E_SYSERR 4    /* system error */
# define E_NOEXIST 5   /* invalid specification */
# define E_DUP 6       /* entry already exists */
# define E_PMRUN 7     /* port monitor is running */
# define E_PMNOTRUN 8  /* port monitor is not running */
# define E_RECOVER 9   /* in recovery */
```

Directory Structure

This section gives a description of the SAF files and directories.

/etc/saf/_sysconfig
The per-system configuration script.

/etc/saf/_sactab
The SAC's administrative file. Contains information about the port monitors for which the SAC is responsible.

/etc/saf/*pmtag*
The home directory for port monitor *pmtag*.

/etc/saf/*pmtag***/_config**
The per-port monitor configuration script for port monitor pmtag. **/etc/saf/***pmtag***/_pmtab** Port monitor pmtag's administrative file. Contains information about the services for which *pmtag* is responsible. **/etc/saf/***pmtag***/***svctag* The file in which the per-service configuration script for service *svctag* (available through port monitor **pmtag**) is placed. **/etc/saf/***pmtag***/_pid** The file in which a port monitor writes its process id in the current directory and places an advisory lock on the file. **/etc/saf/ pmtag /_pmpipe** The file in which the port monitor receives messages from the SAC and **../_sacpipe** and sends return messages to the SAC. **/var/saf/_log** The SAC's log file. **/var/saf/***pmtag* The directory for files created by port monitor *pmtag,* for example its log file.

LIST OF COMMANDS

The following administrative commands relate to SAF.

sacadm(1M)
port monitor administrative command

pmadm(1M)
service administration command

SEE ALSO

exec(1), **sh**(1), **init**(1M), **nlsadmin**(1M), **pmadm**(1M), **sac**(1M), **sacadm**(1M), **ttyadm**(1M), **fork**(2), **doconfig**(3N)