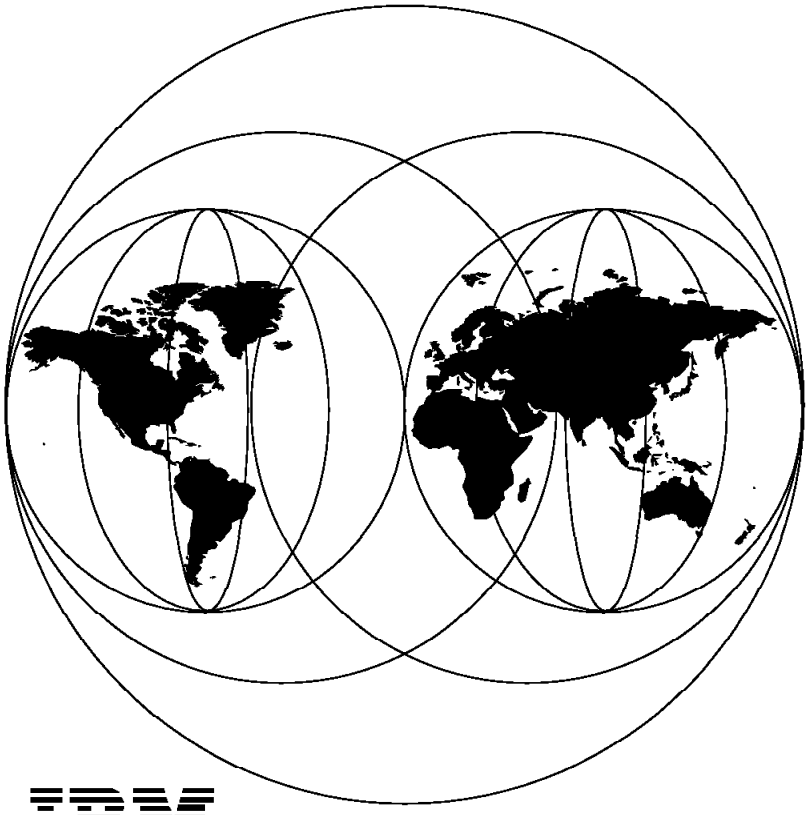
SG24-4946-00

Writing Reliable AIX Daemons

October 1996



IBM

International Technical Support Organization
Poughkeepsie Center

International Technical Support Organization

SG24-4946-00

Writing Reliable AIX Daemons

October 1996

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix D, “Special Notices” on page 399.

First Edition (October 1996)

This edition applies to Release 4.1 of AIX, 5765-393.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HYJ Mail Station P099
522 South Road
Poughkeepsie, New York 12601-5400

Contents

Figures

Preface

This redbook describes how to write reliable AIX daemons. It covers the reasons for having high reliability in the daemon processes, and it describes techniques for achieving the required reliability through using a standard set of routines that remove many reliability considerations from the concerns of the person who writes the daemon. The document includes discussions of how the required reliability is achieved when the daemons are controlled by the various resources available to perform the control task in AIX.

This redbook is valuable for administrators, program designers, and daemon writers who need to originate or understand daemon logic.

Several practical examples are presented as implementations of the techniques described. Examples of using standard routines to achieve the required reliability are provided.

The book assumes the reader is familiar with UNIX commands, UNIX system calls and C language programming. Some knowledge of network programming is also assumed.

How This Redbook Is Organized

This redbook contains 414 pages. It is organized as follows:

- Chapter 1, “Introduction to the Redbook” provides an introduction and “roadmaps” to the various chapters for those who need only a subset of the topics included.
- Chapter 2, “UNIX Implementations and Standards” provides some information about UNIX implementations and standards. The information in this chapter is important, since several issues facing a daemon are influenced by the history of UNIX implementations and standards.
- Chapter 3, “Daemon Initialization” describes how a daemon can be coded to initialize itself properly, regardless of how it is started.
- Chapter 4, “Managing a Daemon with the `init` Process” describes the `init` configuration file, `/etc/inittab`, and how entries in the file control what daemons are started when the system is booted or changes run levels.
- Chapter 5, “Managing a Daemon with the SRC” describes how a daemon is managed by the System Resource Controller (SRC). Section 5.1, “The SRC Commands” on page 50 provides an introduction to the SRC commands made available to the system administrator. Section 5.2, “Subsystem Definition” on page 62 explains how a daemon is defined as an SRC subsystem. Section 5.4, “Subsystem Failure, Respawning, and Notification” on page 73 provides an introduction to the facilities available in the SRC to recover from the abnormal termination of a subsystem. Section 5.6, “Programming for the SRC in a Daemon” on page 84 describes the coding changes that may be needed for a daemon to be placed under SRC control. Finally, section 5.7, “Programming an SRC Notification Method” on page 126 provides information about writing an SRC notification method.
- Chapter 6, “Managing a Daemon with the Internet Superserver” describes the `inetd` configuration file, `/etc/inetd.conf`, and how entries in the file

determine how daemons behave under inetd control. The chapter also describes the relationships between inetd and the SRC on AIX systems. Finally, the chapter includes information about inetd daemons and controlling terminals.

- Chapter 7, “Signal Handling” discusses many reliability issues associated with signals. Daemons tend to be more sophisticated users of signals than other types of processes, but improperly handling signals can lead to reliability problems.
- Chapter 8, “AIX Paging Space Allocation” discusses the AIX paging space allocation policies, and how daemons can avoid termination when the system runs low on paging space.
- Chapter 9, “Ensuring Exclusivity” discusses how a daemon may enforce whatever requirements it has with regard to exclusivity. For example, many daemon programs will want to only allow one process to run the program at any one time.
- Chapter 10, “Daemon Support Routines” describes the daemon support routines provided as part of the this redbook. Many of the issues discussed in this document are handled by the daemon support routines. Examples are included in this chapter.
- Chapter 11, “SRC Notification Support Programs” describes programs and shell scripts that are provided as part of this redbook to allow for consistent SRC notification methods.
- Chapter 12, “New SRC Program Support” describes a model program that is provided as part of this redbook that simplifies the task of creating a program to send a subsystem-defined request to SRC subsystems.
- Appendix A, “Daemon Support Routines man Pages,” provides detailed program instructions for the use of the Daemon Support Routines. The level of detail is consistent with that commonly provided in UNIX systems man pages.
- Appendix B, “SRC Notification Support Programs man Pages,” provides detailed program instructions for the use of the SRC Notification Support Programs. The level of detail is consistent with that commonly provided in UNIX systems man pages.
- Appendix C, “Loading the Diskette,” tells how to use the supplied diskette to create files that contain the text of figures in this redbook, the source code for the daemon support routines, the source code for the SRC notification support programs, and the source code for a model program that sends subsystem-defined requests to an SRC-controlled daemon.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists in conjunction with the International Technical Support Organization Poughkeepsie Center.

Eric Agar is a development programmer in the Scalable POWERparallel Systems organization in Poughkeepsie, New York. He has worked at IBM for 14 years. For the last seven years, he has concentrated on various AIX development projects. Prior to that time, he developed control software for machine tools used in hardware manufacture.

Thanks to the following people for their invaluable contributions to this project:

Deepak Advani

Peter Badovinatz

Larry Brenner

April Brown

Michael Browne

Mike Ferrell

Ted Kirby

George Murdock

Michael Schmidt

Steve Tovcimak

Jan Baisden
ITSO, Poughkeepsie

Comments Welcome

We want our redbooks to be as helpful as possible. Should you have any comments about this or other redbooks, please send us a note at the following address:

redbook@vnet.ibm.com

Your comments are important to us!

Chapter 1. Introduction to the Redbook

This redbook is an adaptation of a document specifying guidelines for developing reliable daemons in the Scalable POWERparallel Systems organization, a development group within the IBM RS/6000 Division. The redbook includes any sections of that document that will enable the reader to understand the recommendations made by the guidelines. The redbook should help the reader develop reliable daemons.

This chapter states the assumptions made by the redbook, describes the reasons for using a set of standard daemon support routines (provided with the redbook) that can be used to support the recommendations, and identifies chapters that apply to particular types of daemons.

1.1 Assumptions and Disclaimers

This document assumes the reader is familiar with using a UNIX system and with programming C language programs to run on a UNIX system. It assumes that the reader is familiar with the C standard library and with the system calls provided by a UNIX system for a C language program (for example, familiarity with the standard I/O routines; system calls such as `open`, `read`, `write`, `close`, `signal`, `select`, `fork`, `wait`, and `waitpid` and familiarity with the exec family of routines). An excellent text on the programming environment provided by UNIX systems is *Advanced Programming in the UNIX Environment* by W. Richard Stevens (1992, Addison-Wesley Publishing Company).

This document assumes the reader is familiar with network programming. An excellent text on network programming is *UNIX Network Programming* by W. Richard Stevens (1990, Prentice Hall, Englewood Cliffs).

When discussing the use of system routines in daemon programs, this document concentrates on providing working examples. Little attempt is made to describe the syntax or the full semantics of the system routines used. It is assumed that the reader will refer to the `man` pages in *AIX Version 4.1: Technical Reference Volume 1*, SC23-2614 and *AIX Version 4.1: Technical Reference Volume 2*, SC23-2615, as necessary. Similarly, little attempt is made to describe the full functionality of system commands used in the examples provided. The reader should refer to *AIX Version 4.1: Command Reference*, SBOF-1851, as needed.

Unless specified otherwise, references to AIX in this document are intended to be references to AIX 4.1.2. Statements made in this document about AIX may or may not apply to other versions of AIX.

This document includes many source code examples. Some examples are complete, and can be compiled and run as presented. Other examples are code fragments that would require modifications before they could be compiled and executed. Every effort has been made to make the examples error-free.

The purpose of the code examples presented in this document is to illustrate how to write reliable daemons. The examples might not follow other recommended techniques. For instance, the Scalable POWERparallel Systems organization places message numbers in the text of messages to facilitate reference to message documentation. Messages are also placed in message

catalogs, allowing easier translation to different languages. If these techniques were used in the examples, the principles of the examples would be less clear.

1.1.1 RS/6000 SP Considerations

The information presented in this redbook applies to any RS/6000 running AIX. However, since the redbook is adapted from a document written in the Scalable POWERparallel Systems organization, the RS/6000 SP system is discussed occasionally. The following information is sufficient to understand those discussions:

- The RS/6000 SP is a collection of nodes. Each node is an RS/6000 with its own processors, memory, and disks. Each node runs the AIX operating system. Nodes may communicate efficiently over a high-speed switch.
- The nodes of an RS/6000 SP may be used to run parallel programs, serial programs, batch jobs, and interactive jobs.
- The nodes of an RS/6000 SP are managed from a single location. This is a separate RS/6000, known as the control workstation, connected to the RS/6000 SP.
- Software known as the PSSP (IBM Parallel System Support Programs for AIX) allows the system administrator to manage the RS/6000 SP.
- Using the PSSP, an RS/6000 SP system can be partitioned. Each node belongs to a single partition. Each partition can be managed as though it is an independent RS/6000 SP system.

1.2 How Daemons Are Managed: A Brief Introduction

For the purposes of this redbook, a daemon is defined to be a process that is intended to run “in the background” without a controlling terminal. That is, the life span of the daemon process does not depend on the life span of a terminal session, the daemon life span does not depend on any control key sequences generated from a terminal session, and a daemon does not interact with users through a terminal. Note that this definition of a daemon does not depend on how the daemon was started. A daemon can be started by the `init` process, the System Resource Controller, the `inetd` process, or a shell.

Some daemons wait for requests from client processes, then satisfy those requests, while other daemons periodically monitor the status of a system; some daemons combine both functions. It is common for a daemon to start when a system is booted, although that is not necessary. Once a daemon is started, it will usually continue to run until the system is shut down, although it may terminate before that time.

Traditionally, the `init` process starts a daemon when the system is booted. A configuration file or a shell script tells the `init` process which daemons to start. When a configuration file is used, that file may instruct the `init` process to start a daemon directly, or it may instruct the `init` process to run a shell script, which in turn may start one or more daemons.

Once started, a daemon may spend most of its time waiting to receive a request from a client. The mechanism through which the daemon receives requests from clients depends on the implementation of the daemon; typical mechanisms include TCP/IP, UDP/IP, Berkeley sockets, and System V message queues.

A daemon may be designed to support multiple clients simultaneously, or only one at a time.

If the daemon supports only one client at a time, it is typically implemented as an iterative server. In the iterative server model, the daemon process receives a request from a client, processes that request for the client (which may involve several messages being passed between client and server), and then waits for the next request from the next client. While processing a request for one client, other client requests are not processed. Iterative servers often communicate with their clients through UDP/IP.

If the daemon supports multiple clients simultaneously, it is typically implemented as a concurrent server. In the concurrent server model, the daemon process waits for a request for service from a new client. When such a request is detected, the daemon process creates a child process whose responsibility is to handle the request from that specific client[1]. Once the child process has been created, the daemon process waits for the next request for service from another client. Multiple child processes can be running simultaneously, with each process servicing a unique client. A child process will terminate once the requests from its particular client have been satisfied. Concurrent servers often communicate with their clients through TCP/IP.

It has been observed that many daemon processes using the socket interface for network programming with TCP/IP and UDP/IP used similar code to wait for requests from new clients. As a result, the Internet Superserver (`inetd`) was written. The `inetd` process is a daemon started at system boot time. The `inetd` daemon waits for requests for service provided by many servers. When a request comes in for a particular type of service, the `inetd` daemon may handle the request itself, or it may create a child process to handle the request. The child process runs the daemon designed to provide the particular service. When a daemon is placed under `inetd` control, it does not have to contain code that waits for new clients; in fact, it should not wait for new clients. The daemons that run under `inetd` control are described to `inetd` in a configuration file. These daemons may be designed as iterative servers or concurrent servers. The information provided to `inetd` about a daemon indicates if multiple instances of the daemon should run concurrently. For iterative servers, `inetd` is told to allow only one instance of the daemon at any time. For concurrent servers, `inetd` is told to allow multiple instances. A concurrent server under `inetd` control does not have to create child processes to allow for the concurrent processing of requests from multiple clients; `inetd` provides that function.

System administrators might influence the behavior of a running daemon by sending signals to it. Daemons often install signal handlers for specific signals. For example, it is common for a daemon to install a signal handler for the `SIGHUP` signal that would cause the daemon to reread its configuration file. Anytime the system administrator wants the daemon to reread its configuration file, the process identifier (PID) of the daemon is determined using the `ps` command, and the `SIGHUP` signal is sent to the process identified by that PID with the `kill` command. If the system administrator mistakenly used an incorrect PID in the `kill` command, the `SIGHUP` signal might be sent to some other running process. If that process had not changed the disposition of the `SIGHUP` signal from the

[1] This a common implementation of a concurrent server involving multiple processes. It is also possible to implement a concurrent server using only one process.

default behavior, the process would terminate. A system administrator might also send the SIGTERM signal to a daemon to have it terminate gracefully.

On AIX, a new mechanism was introduced to control daemons; this new mechanism improves the ability of system administrators to influence daemon behavior. The new mechanism is the System Resource Controller (SRC). When a daemon is under the control of the SRC, SRC commands can be used by the system administrator to influence the behavior of the daemon. For example, one command can be used to cause the daemon to reread its configuration file. Another command can be used to get status from the daemons under SRC control. Yet another command can be used to request that a daemon terminate gracefully. These commands are generally safer and easier to use than the ps and kill commands. It is less likely that a system administrator's mistake will terminate a process unintentionally.

A daemon that is started directly by init, from a shell script, or from the command line can be changed to run under the control of the SRC. When put under the control of the SRC, the daemon is started by the SRC, but the request to start the daemon can still come from init, a shell script, or the command line. In addition to the benefits described previously, putting a daemon under the control of the SRC provides the following benefits:

- The SRC can be instructed to respawn the daemon if it terminates abnormally. This is similar to the capability of the init process to respawn a daemon that it directly started. However, under the SRC, even daemons that were started in response to a request from a shell script or a shell command line can be respawned.
- When a daemon running under the SRC terminates abnormally, the SRC writes an entry in the AIX error log. The entry identifies the daemon that terminated, and includes the exit status of the daemon process. An entry for a daemon not under SRC control is only placed in the AIX error log when the process generates a core file.
- When a daemon is put under SRC control, an SRC notification method can be associated with the daemon. If such a daemon terminates abnormally, and the SRC is not instructed to respawn the daemon, the notification method is executed by the SRC. The notification method can be any executable program or shell script. It receives the name of the daemon as a parameter. The notification method can start recovery actions for the daemon and take notification actions. The notification method of a daemon is also executed if the daemon cannot be respawned properly.
- The SRC can be instructed whether multiple instances of a particular daemon are permitted to run on a system at the same time. If multiple instances of a daemon are not allowed, an attempt to start a second instance of the daemon with the SRC will result in an error message.
- The SRC can be instructed which files should be associated with the standard input, standard output, and standard error of the daemon. The SRC makes these associations without the danger of establishing a controlling terminal for the daemon[2].

[2] On AIX, a daemon started by init can have files associated with the standard input, standard output, and standard error of a daemon. However, init does not prevent such an association from establishing a controlling terminal for the daemon.

- The SRC can be given a default set of arguments with which a particular daemon is always started. These arguments will be provided to the daemon even when the system administrator tells SRC to start the daemon from a shell command line. When a daemon is started through the SRC, additional arguments can be specified. The daemon will receive the default arguments and the additional arguments.
- The SRC can be instructed to start a daemon with a particular priority.
- The SRC can be instructed to run a daemon in a process associated with a specific user ID.
- Daemons under SRC control can be defined in groups. A single SRC command can be applied to all daemons in a group.

1.3 Reasons for Using a Set of Standard Daemon Support Routines

This redbook provides a documented set of routines intended to lessen the effort required for daemon writers and maintainers to follow the guidelines presented. The routines are designed to be general enough to be useful to a daemon started by the System Resource Controller (SRC), the `init` process, the Internet Superserver (`inetd`), or a shell command. The routines implement the procedures recommended in this redbook. They implement most of what is needed to allow signal, message queue, or socket communication between the SRC and a daemon. They implement methods to protect a daemon from termination during a low paging space situation. They also support a method for allowing a daemon to enforce its exclusivity requirements.

These routines are referred to as the daemon support routines. Their use is illustrated in Chapter 10, “Daemon Support Routines” on page 245. The interfaces to the routines are specified in Appendix A, “Daemon Support Routines `man` Pages” on page 315.

When this document discusses a particular topic, it will also indicate how the daemon support routines support the daemon writer in dealing with the issues related to the topic.

A daemon writer can develop a reliable daemon without the daemon support routines. However, it is hoped that the routines will enjoy widespread use because they help the daemon writer concentrate on the unique function provided by the daemon, instead of the general implementation overhead of writing a daemon.

1.4 Chapter Road Maps

A daemon programmer does not need to read this entire document to develop a reliable daemon. This section provides some road maps to guide the programmer through the sections that apply to his daemon. Certain chapters only apply to daemons that will be placed under the control of the SRC. Other chapters only apply to daemons placed under the control of the `inetd` process. Some chapters may apply to a daemon regardless of whether it is under the control of the SRC or the `inetd` process.

1.4.1 Chapters Applicable to All Daemons

The following chapters may apply to any daemon:

- Chapter 2, “UNIX Implementations and Standards” on page 9
- Chapter 7, “Signal Handling” on page 141

The following chapters may apply to any daemon that does not utilize the daemon support routines:

- Chapter 3, “Daemon Initialization” on page 19
- Chapter 8, “AIX Paging Space Allocation” on page 203
- Chapter 9, “Ensuring Exclusivity” on page 231

Chapter 10, “Daemon Support Routines” on page 245 applies to any daemon utilizing the daemon support routines. The issues related to daemon initialization are dealt with by the `dae_init` routine and other routines with the `dae_init_` prefix. The issues related to the AIX paging space allocation policies are dealt with by the `dae_init_psalloc`, `dae_init_lowps`, and `dae_init` routines. The issues related to daemon exclusivity are dealt with by the `dae_init_exclusive` and `dae_init` routines.

1.4.2 Chapters Applicable to SRC Controlled Daemons

The following sections may be helpful when placing a daemon under SRC control:

- Chapter 4, “Managing a Daemon with the `init` Process” on page 45, if the SRC controlled daemon is to be started when the system is booted.
- 5.1, “The SRC Commands” on page 50, for an understanding of how SRC commands are used to control an SRC subsystem.
- 5.2, “Subsystem Definition” on page 62, to understand how the daemon can be defined as an SRC subsystem.
- 5.3, “Subsystem User ID” on page 68, to understand the limitations of the users under which an SRC subsystem may run.
- 5.4, “Subsystem Failure, Respawning, and Notification” on page 73, to understand how the daemon may be respawned by the SRC.
- 5.5, “Subsystem Groups” on page 83, for a discussion concerning subsystem groups.
- 5.6, “Programming for the SRC in a Daemon” on page 84, to understand the changes needed in the daemon code. This section is only required if the daemon support routines will not be used.
- Chapter 10, “Daemon Support Routines” on page 245, to understand how to use the daemon support routines to allow the daemon to operate under SRC control.

If a daemon will take advantage of the SRC notification capabilities, the following sections may be helpful:

- 5.7, “Programming an SRC Notification Method” on page 126
- Chapter 11, “SRC Notification Support Programs” on page 295

If a daemon under SRC control will support a subsystem-defined SRC request (a request delivered to a subsystem by the SRC, but not defined by the SRC), Chapter 12, “New SRC Program Support” on page 303 may be helpful.

1.4.3 Chapters Applicable to `inetd` Controlled Daemons

For a daemon to be placed under the control of `inetd`, Chapter 6, “Managing a Daemon with the Internet Superserver” on page 129 should be helpful. Particular attention should be paid to the sections describing the interaction between `inetd` and the SRC on an AIX system. Section 6.5, “Programming a Daemon under `inetd` Control: A Word of Caution” on page 139 is important if the daemon support routines will not be used. Of course, if the daemon support routines will be used, Chapter 10, “Daemon Support Routines” on page 245 is important.

Chapter 2. UNIX Implementations and Standards

UNIX systems have evolved within two major branches: the branch of development started within AT&T Bell Laboratories, and the branch of development within the University of California at Berkeley. Recent systems developed within the AT&T branch are referred to as releases of System V. Recent systems developed within the Berkeley branch are identified with names like 4.xBSD, such as 4.3BSD and 4.4BSD[3]. BSD systems owe their origin to the predecessors of System V, resulting in many similarities between the two branches. However, many differences have developed.

Differences in the major branches of UNIX systems inspire attempts at standardization. The goal of the standards bodies is to standardize some subset of UNIX systems, allowing for more portability of code and skills among various UNIX systems. UNIX-related standards have different scopes. The standards most relevant to this document are:

- The ANSI C standard
- The IEEE POSIX standards
- The X/Open standards

The ANSI C standard, produced by the American National Standards Institute, standardizes the definition of the C language and a set of C library routines. The goal of the standardization is to allow code to be written in C and ported among operating systems supporting ANSI C. These systems are not limited to UNIX systems.

The IEEE (Institute of Electrical and Electronics Engineers) produces the POSIX (Portable Operating System Interface for Computer Environments) standards. The POSIX standards describe the features that must be present in an operating system for it to claim to be "POSIX compliant." The POSIX standards are inspired by UNIX systems, but non-UNIX systems have also complied with some of the standards[4]. The POSIX standard of most importance to programmers is POSIX 1003.1. POSIX 1003.1 defines operating system interfaces available to an application programmer. This is the standard that standardizes such system calls as `fork`. In many ways, the POSIX 1003.1 standard can be thought of as a superset of the ANSI C standard.

X/Open produces standards that are supersets of the POSIX standards. Issues of X/Open standards that are relevant to this document are XPG3 and XPG4. Recently, the X/Open Company obtained the UNIX trademark. As a result, the most recent version of XPG4 (Issue 4, Version 2) includes many routines from System V that were not previously included. Version 2 of XPG4 allows a system to claim either BASE conformance or X/OPEN UNIX conformance. The criteria specified for X/OPEN UNIX conformance includes the System V additions just mentioned.

The importance of standards relates to portability. If a program is implemented solely within the specification of a standard, it should be easily ported to another

[3] BSD stands for "Berkeley Software Distribution."

[4] IBM's MVS Open Edition complies with at least one of the POSIX standards.

system claiming conformance to that standard. The smaller the scope of a standard, the more portable an application written to that standard is. The larger the scope of a standard, the easier it is to write an application written to that standard.

UNIX implementations that claim conformance to these standards are not usually limited to the facilities specified in the standards. For example, AIX 4.1 conforms to, among other standards, POSIX 1003.1 and XPG4. However, AIX is not limited by these standards. For example, AIX 4.1 supports Berkeley sockets; sockets are not specified by any of these standards.

2.1 Standards and AIX

UNIX implementations that claim conformance to standards usually have histories that affect their externals. For example, AIX is based on the System V branch of UNIX development. So, it tends to support System V methods for doing things, in addition to the standard methods. However, AIX has also included BSD features to allow BSD programs to be ported to AIX. As a result, there can be three or more routines designed to accomplish the same function in AIX. To add to the confusion, these different routines sometimes have the same name. For example, on AIX, a program can install a signal handler with the System V `signal` routine, the BSD `signal` routine, the System V `sigset` routine, the AIX `sigset` routine, the BSD `sigvec` routine, and the POSIX and X/Open `sigaction` routine.

When AIX provides different versions of a routine with the same name, it supports the versions in different libraries. Routines with different origins are found in the `libc.a` library: routines specified in standards, routines from System V, and many BSD routines that are unique to BSD. Routines that are BSD variants of routines in the `libc.a` library are found in the `libbsd.a` library. A routine in the `libbsd.a` library may have a different interface than the same routine in the `libc.a` library, or a routine in the `libbsd.a` library may have the same interface, but the behavior may differ. Some `libbsd.a` routines may differ in both interface and behavior. For example, AIX supports two versions of the `fcntl` routine. The `libbsd.a` routine supports making a socket file descriptor non-blocking, while the `libc.a` routine does not provide that support.

On AIX, a programmer should determine whether his program requires BSD behavior. This decision is often easy to make. The AIX man page for a function being used may indicate that BSD functionality should be compiled into the program when using a function. For instance, if the program uses sockets, BSD behavior is required. If BSD behavior is required, the program should be compiled with the `_BSD` macro defined to `43` or `44`. The `43` value specifies 4.3BSD behavior; the `44` value specifies 4.4BSD behavior. When the program is linked, it should be linked such that the `libbsd.a` library is searched before the `libc.a` library. For example, you can create a program named `example` that will exhibit BSD behavior by compiling a source file named `example.c` with a command similar to that shown in Figure 1 on page 11. Using `xlc` causes an ANSI C compiler to be used. The `-D_ALL_SOURCE` flag allows AIX extensions to be used in the program. The `-D_BSD=44` flag indicates BSD behavior is desired. The `-lbsd` flag causes the `libbsd.a` library to be searched in the link step. The `libc.a` library is searched by default, after the libraries explicitly listed on the command line.

```
$ xlc -O -D_ALL_SOURCE -D_BSD=44 -lbsd -o example example.c
```

Figure 1. Compiling on AIX When BSD Behavior Is Required

When a program is linked with the libbsd.a library, some surprises may occur.

For example, if a program is linked with libbsd.a for BSD socket support, and the program calls the sleep function, the sleep function will execute with BSD behavior. The standard version of sleep in libc.a returns the number of seconds not slept[5]. However, the BSD version of sleep does not return any meaningful value; any value returned should be ignored.

The program in Figure 2 on page 12 takes as a parameter, the number of seconds to delay. It loops through calls to the sleep function to accomplish this. If the program does not receive a SIGINT signal while blocked in the sleep function, only one call to sleep is required. If the SIGINT signal interrupts the sleep function, the function is called again to attempt to delay the remaining time.

In the example shown in Figure 3 on page 13, two versions of the program are created. The delay program calls the libc.a version of sleep. The delay_bsd program calls the libbsd.a version of sleep. As the example shows, delay can process SIGINT signals and delay for the proper amount of time. However, delay_bsd does not delay for the proper amount of time when sleep is interrupted by the SIGINT signal.

[5] The sleep function may return before the specified number of seconds, due to the delivery of a signal to the process.

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <sys/times.h>

void sig_handler(int signo)
{
      return;
}

int main(int argc, char **argv)
{
      int delay_time, ticks_per_sec;
      struct sigaction sa;
      clock_t start_time, end_time;
      struct tms start_tms, end_tms;


      if (argc != 2) {
            exit(1);
      }

      delay_time = atoi(argv[1]);

      ticks_per_sec = sysconf(_SC_CLK_TCK);

      sa.sa_handler = sig_handler;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = 0;

      (void) sigaction(SIGINT, &sa, NULL);

      start_time = times(&start_tms);

      while (delay_time > 0) {
            delay_time = sleep(delay_time);
      }

      end_time = times(&end_tms);

      printf("Sleep loop lasted %d seconds.\n",
            (end_time - start_time) / ticks_per_sec);

      return 0;
}
```

Figure 2. Source for the delay *and* delay_bsd *Programs*

```
$ xlc -O -D_ALL_SOURCE -o delay delay.c
$ xlc -O -D_ALL_SOURCE -D_BSD=44 -lbsd -o delay_bsd delay.c

$ ./delay 10
^C^C^C^C^C^C^CSleep loop lasted 10 seconds.

$ ./delay_bsd 10
^CSleep loop lasted 2 seconds.
```

Figure 3. Compiling and Executing the delay *and* delay_bsd *Programs*

On AIX systems, libbsd.a and libc.a are shared libraries. When a program calls a routine in a shared library, the program's loader area indicates where the routine can be found. The dump command can be used with the -n and -v flags to find out how external references to shared libraries are resolved. In Figure 4, the dump command shows that the delay program uses the sleep function in libc.a. In Figure 5 on page 14, the dump command shows that the delay_bsd program uses the sleep function in libbsd.a.

```
$ dump -nv delay

delay:

                      ***Loader Section***
                    Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x0000000a       0x00000014       0x0000001e

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000002       0x00000200       0x0000001a       0x0000021e


                      ***Import File Strings***
INDEX  PATH                           BASE               MEMBER
0      /usr/lib:/lib
1                                     libc.a             shr.o

                      ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x00000000    undef      IMP     RW EXTref   libc.a(shr.o) errno
[1]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) exit
[2]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) atoi
[3]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) sysconf
[4]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) sigemptyset
[5]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) sigaction
[6]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) times
[7]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) sleep
[8]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[9]     0x00000010    .data    ENTpt     DS SECdef        [noIMid] __start
```

Figure 4. Display of Loader Section of the delay *Program*

```
$ dump -nv delay_bsd

delay_bsd:

                       ***Loader Section***
                     Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x0000000a       0x00000014       0x0000002e

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000003       0x00000200       0x0000001a       0x0000022e


                       ***Import File Strings***
INDEX  PATH                          BASE               MEMBER
0      /usr/lib:/lib
1                                    libc.a             shr.o
2                                    libbsd.a           shr.o

                       ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]      0x00000000    undef      IMP     RW EXTref   libc.a(shr.o) errno
[1]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) exit
[2]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) atoi
[3]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) sysconf
[4]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) sigemptyset
[5]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) sigaction
[6]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) times
[7]      0x00000000    undef      IMP     DS EXTref libbsd.a(shr.o) sleep
[8]      0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[9]      0x00000010    .data    ENTpt     DS SECdef        [noIMid] __start
```

Figure 5. Display of Loader Section of the delay_bsd *Program*

2.2 Writing Programs for Portability

When writing programs that may be ported to other systems, it is helpful to assume a certain standard as a base. As the program is written, try to limit yourself to the features of the standard you have chosen as the base. If certain features that are not included in the standard are to be used when the program is compiled on specific systems, conditional compilation can be used.

Take the daemon support routines as an example of this strategy. When compiled on an AIX system, the routines will support use of the SRC. When compiled on a non-AIX system, the routines will not support use of the SRC. This can be managed through the preprocessor. The code fragment in Figure 6 on page 15 looks for the definition, or at the values of, several preprocessor manifest constants to determine the type of system on which it is being compiled. The code recognizes the following:

- The C compiler on AIX systems defines the manifest constant _AIX.
- If a program requires BSD behavior on AIX systems, the _BSD manifest constant must be defined.

- If a system meets the requirements specified for XPG4 X/OPEN UNIX conformance, the _XOPEN_UNIX manifest constant must be defined with a value other than -1.
- If a system complies with XPG4, the _XOPEN_VERSION manifest constant must be defined with a value of 4.
- If a system complies with XPG3, the _XOPEN_VERSION manifest constant must be defined with a value of 3.
- If a system supports POSIX job control, the _POSIX_JOB_CONTROL manifest constant must be defined.

```
#include <unistd.h>
#include <signal.h>

/*
 *  Macros to determine the type of system the code is running on.
 */

#define DAE_SYS_AIX_BSD          1        /* AIX with BSD compatibility    */
#define DAE_SYS_AIX              2        /* AIX without BSD compatibility */
#define DAE_SYS_XOPEN_UNIX       3        /* X/OPEN UNIX with job control  */
#define DAE_SYS_XOPEN_JOB        4        /* X/OPEN BASE with job control  */

#if defined(_AIX)

        #if !defined(_ALL_SOURCE)
                #error "daemon support code requires _ALL_SOURCE on AIX"
        #endif
        #if defined(_BSD)
                #define DAE_SYS  DAE_SYS_AIX_BSD
        #else
                #define DAE_SYS  DAE_SYS_AIX
        #endif

#else

        #if   defined(_XOPEN_UNIX) && (_XOPEN_UNIX != -1)
                #define DAE_SYS  DAE_SYS_XOPEN_UNIX
        #elif defined(_XOPEN_VERSION) && (_XOPEN_VERSION >= 3) && \
              defined(_POSIX_JOB_CONTROL)
                #define DAE_SYS  DAE_SYS_XOPEN_JOB
        #else
                #error "daemon support code only implemented for at least XPG3
systems with job control"
        #endif

#endif
```

Figure 6. Determining the Type of System

The daemon support routines assume a XPG3-compliant system that supports POSIX job control as a base. Therefore, if the system on which the code is being compiled does not meet these assumptions, the #error directive is used to generate an error message. The code fragment in Figure 6 sets DAE_SYS to a value that represents the type of system on which the code is compiled. In turn, the code fragment in Figure 7 on page 16 uses the value of DAE_SYS to determine whether or not to define other manifest constants. Each of these manifest

constants has a specific meaning to the daemon support routines. For example, if DAE_SRC_SUPPORT is defined, the daemon support routines will compile with code designed to support running a daemon under the SRC; if DAE_SRC_SUPPORT is not defined, the daemon support routines will not include the code designed to support the SRC. Notice that DAE_SRC_SUPPORT is only defined if the code is being compiled on an AIX system.

```
/*
 *  Set macros indicating support for inetd, SRC, and SRC with sockets.
 */
#if   (DAE_SYS == DAE_SYS_AIX_BSD)
        #define DAE_INETD_SUPPORT       /* Supporting inetd               */
        #define DAE_SRC_SUPPORT         /* Supporting SRC                 */
        #define DAE_SRC_SOCK_SUPPORT    /* Supporting SRC with sockets    */
        #define DAE_PSALLOC_SUPPORT     /* Supports AIX paging space alloc.*/
        #define DAE_FCNTL_CLOSE_SUPPORT /* Supports fcntl F_CLOSEM        */
        #define DAE_NOTTY_IOCTL_SUPPORT /* Supports ioctl TIOCNOTTY       */
        #define DAE_IGN_ZOMBIES_SUPPORT /* Supports ignoring zombies      */
#elif (DAE_SYS == DAE_SYS_AIX)
        #undef  DAE_INETD_SUPPORT       /* Not supporting inetd           */
        #define DAE_SRC_SUPPORT         /* Supporting SRC                 */
        #undef  DAE_SRC_SOCK_SUPPORT    /* Not supporting SRC with sockets */
        #define DAE_PSALLOC_SUPPORT     /* Supports AIX paging space alloc.*/
        #define DAE_FCNTL_CLOSE_SUPPORT /* Supports fcntl F_CLOSEM        */
        #define DAE_NOTTY_IOCTL_SUPPORT /* Supports ioctl TIOCNOTTY       */
        #define DAE_IGN_ZOMBIES_SUPPORT /* Supports ignoring zombies      */
#elif (DAE_SYS == DAE_SYS_XOPEN_UNIX)
        #undef  DAE_INETD_SUPPORT       /* Not supporting inetd           */
        #undef  DAE_SRC_SUPPORT         /* Not supporting SRC             */
        #undef  DAE_SRC_SOCK_SUPPORT    /* Not supporting SRC with sockets */
        #undef  DAE_PSALLOC_SUPPORT     /* Does not support AIX PSALLOC   */
        #undef  DAE_FCNTL_CLOSE_SUPPORT /* Does not support fcntl F_CLOSEM */
        #undef  DAE_NOTTY_IOCTL_SUPPORT /* No support for ioctl TIOCNOTTY  */
        #define DAE_IGN_ZOMBIES_SUPPORT /* Supports ignoring zombies      */
#elif (DAE_SYS == DAE_SYS_XOPEN_JOB)
        #undef  DAE_INETD_SUPPORT       /* Not supporting inetd           */
        #undef  DAE_SRC_SUPPORT         /* Not supporting SRC             */
        #undef  DAE_SRC_SOCK_SUPPORT    /* Not supporting SRC with sockets */
        #undef  DAE_PSALLOC_SUPPORT     /* Does not support AIX PSALLOC   */
        #undef  DAE_FCNTL_CLOSE_SUPPORT /* Does not support fcntl F_CLOSEM */
        #undef  DAE_NOTTY_IOCTL_SUPPORT /* No support for ioctl TIOCNOTTY  */
        #undef  DAE_IGN_ZOMBIES_SUPPORT /* No support for ignoring zombies */
#else
        #error "DAE_SYS not defined as expected"
#endif
```

Figure 7. Determining the Functions to Support

Figure 8 on page 17 shows other uses for DAE_SYS in the daemon support routines. POSIX and X/OPEN define the getpgrp routine such that it takes no parameters and returns the process group ID of the current process. Traditional BSD systems defined getpgrp such that it takes one parameter, the PID of the process whose process group ID is to be returned. On these systems, a value of 0 means the current process. The various definitions of DAE_GETPGRP deal with these differences. The remainder of the daemon support code uses DAE_GETPGRP to obtain the process group ID of the current process. The code fragment also defines the macro DAE_IS_SESSION_LEADER. The easiest way to determine if a

process is a session leader is to compare the return value of getsid with the return value of getpid. However, systems that do not claim XPG4 X/OPEN UNIX conformance may not provide a getsid routine. The definitions of the DAE_IS_SESSION_LEADER macro deal with this.

```
/*
 *  Set macro to get the process group ID of the current process.
 */
#if   (DAE_SYS == DAE_SYS_AIX_BSD)
        #define DAE_GETPGRP()   getpgrp((pid_t)0)
#elif (DAE_SYS == DAE_SYS_AIX)
        #define DAE_GETPGRP()   getpgrp()
#elif (DAE_SYS == DAE_SYS_XOPEN_UNIX)
        #define DAE_GETPGRP()   getpgrp()
#elif (DAE_SYS == DAE_SYS_XOPEN_JOB)
        #define DAE_GETPGRP()   getpgrp()
#else
        #error "DAE_SYS not defined as expected"
#endif

/*
 *  Set macro to determine if the current process is a session leader.
 */
#if   (DAE_SYS == DAE_SYS_AIX_BSD)
        #define DAE_IS_SESSION_LEADER() (getsid((pid_t)0) == getpid())
#elif (DAE_SYS == DAE_SYS_AIX)
        #define DAE_IS_SESSION_LEADER() (getsid((pid_t)0) == getpid())
#elif (DAE_SYS == DAE_SYS_XOPEN_UNIX)
        #define DAE_IS_SESSION_LEADER() (getsid((pid_t)0) == getpid())
#elif (DAE_SYS == DAE_SYS_XOPEN_JOB)
        /*
         *  The setpgid() call below attempts to put this process in the
         *  process group to which it already belongs.  If this fails because
         *  this process is a session leader, errno will be set to EPERM.
         *  Other conditions that would return EPERM do not apply for this
         *  specific process.  Other failures should not happen.
         */
        #define DAE_IS_SESSION_LEADER() \
                  ((setpgid((pid_t)0, DAE_GETPGRP()) == -1) && (errno == EPERM))
#else
        #error "DAE_SYS not defined as expected"
#endif
```

Figure 8. Macro Definitions Depending on System Type

Chapter 3. Daemon Initialization

There is a set of generally recognized rules for how a daemon process should initialize itself. These rules are described in section 2.6, “Daemon Processes,” in *UNIX Network Programming*, W. Richard Stevens, 1990, Prentice Hall, Englewood Cliffs; and Chapter 13, “Daemon Processes,” in *Advanced Programming in the UNIX Environment*, W. Richard Stevens, 1992, Addison-Wesley Publishing Company.

UNIX Network Programming describes how the rules are traditionally implemented in System V and BSD systems. This can be useful in understanding the steps taken by existing daemons.

Advanced Programming in the UNIX Environment describes how to implement the rules in a standards-compliant manner. This is useful for writing new daemons, or converting existing daemons to use standard methods of daemon initialization.

This section discusses the steps of daemon initialization, including AIX-specific steps. Sometimes it may not be necessary for the daemon process to take all of the steps discussed. For example, on AIX, if the process was started by the `init` process, the process will already be a session leader of a session with no controlling terminal. Unless otherwise noted, it is not harmful for a daemon to take a step when it is not needed. If the daemon takes all the steps, it will be more likely to behave correctly regardless of how it was started. Following all of the steps will allow the process to behave correctly as a daemon, even if it was started by a shell.

The initialization steps that should be taken in a daemon process are as follow:

1. Determine how the daemon was started.
2. Ignore terminal-generated signals.
3. Migrate the daemon to another process, if appropriate.
4. Disassociate the daemon from its controlling terminal.
5. Deal with non-terminal generated signals.
6. Close unnecessary file descriptors.
7. Ensure certain file descriptors are open.
8. Change the current working directory.
9. Set the process file mode creation mask.

3.1 Determine How the Daemon Was Started

A daemon program can be started in a variety of ways. The ways that are significant for this discussion are:

- By the Internet Superserver (`inetd`)
- By the `init` process
- By the System Resource Controller (SRC)
- By a shell

Some of the daemon initialization steps are only appropriate when the daemon is started in a certain manner. For example, when a daemon is started by inetd, init, or the SRC, it is not appropriate to migrate the daemon to another process. When a daemon is started from a shell, it is appropriate to migrate the daemon to another process. Therefore, it is necessary for the daemon to determine how it was started. This is not as straightforward as it may first appear.

UNIX systems do not provide a straightforward, portable, reliable method for a process to determine how it was started. For example, the PID of a daemon process's parent process, obtainable with the getppid routine, is of limited use. It is literally the PID of the current parent process of the daemon process. On most UNIX systems, the PID of the init process is 1. Also, on most UNIX systems, when a process terminates, any existing child processes of the terminating process are inherited by the init process. Therefore, if a daemon process determines that the PID of its parent process is 1, there are two possible reasons. One possible reason is that the init process started the daemon process. Another possible reason is the process that started the daemon process terminated, and the daemon process was inherited by the init process.

Various daemons use different methods for determining how they were started. Many of these methods make assumptions about how certain programs work. It is common for a daemon to assume that it can only be started in certain ways; if the daemon is started any other way, unintended results may occur.

The remainder of this section discusses how the daemon support code determines how a daemon was started. The techniques are recommended as fairly general and useful.

First, determine if the daemon was started by inetd. This should only be done when the daemon is running on a system that supports inetd. The inetd daemon may be supported on a system derived from 4.3BSD, or a later BSD release. A system that does not support Berkeley sockets probably does not support inetd.

When inetd starts another daemon, it sets up the standard input, standard output, and standard error file descriptors of the daemon to refer to the socket through which the daemon is to communicate with its client. Since it is unusual for a process to start with its file descriptors in such a state, a process can reasonably test these file descriptors to determine if it was started by inetd. See the code in Figure 9 on page 21 for an example of this testing.

The is_an_inetd_process routine returns a constant value TRUE if the process appears to have been started by inetd. Otherwise, it returns FALSE. The routine attempts to get information about the sockets that might be associated with file descriptors 0, 1, and 2. If getsockname returns with an errno value of EBADF, the file descriptor under consideration is not open. If getsockname returns with an errno value of ENOTSOCK, the file descriptor is open, but it is not associated with a socket. In either case, inetd did not start the daemon. Other error values from getsockname are serious errors, so processing is terminated.

If a socket address returned by getsockname is not in the Internet domain, represented by the constant AF_INET, the daemon was not started by inetd, because inetd only supports that domain. If all three file descriptors refer to sockets with the same host addresses and the same port numbers, the daemon was started by inetd. If different host addresses or port numbers are encountered, the daemon was not started by inetd.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int is_an_inetd_process(void)
{
      int fd;
      struct sockaddr_in sockaddr;
      int sockaddrlen;
      unsigned long  addr;
      unsigned short port;

      for (fd = 0; fd <= 2; fd++) {

            sockaddrlen = sizeof sockaddr;
            if (getsockname(fd, (struct sockaddr *) &sockaddr,
                      &sockaddrlen) == -1) {
                  if ((errno == EBADF) || (errno == ENOTSOCK)) {
                        return FALSE;
                  } else {
                        abort();  /* daemon support code does not call  abort */
                  }
            }

            if (sockaddr.sin_family != AF_INET) {
                  return FALSE;
            }

            if (fd == 0) {
                  addr = ntohl(sockaddr.sin_addr.s_addr);
                  port = ntohs(sockaddr.sin_port);
            } else {
                  if ((addr != ntohl(sockaddr.sin_addr.s_addr)) ||
                      (port != ntohs(sockaddr.sin_port))) {
                        return FALSE;
                  }
            }

      }

      return TRUE;

}
```

Figure 9. Determining If inetd *Started the Daemon*

If a daemon determines it was started by inetd, but it is not designed to perform in that environment, the daemon should log an error message, and terminate with an error code.

If the daemon was not started by inetd, the next step is to determine if the daemon was started by the SRC. This should be done only when the daemon is running on a system that supports the SRC, namely AIX systems. Unlike inetd, the SRC does not leave every daemon it starts in a distinguishable state which

would allow a daemon process to test for that state. So, to determine whether SRC started a daemon, the daemon support routines get the PID of the daemon's parent process, and use an AIX-specific method to determine if the parent process is the SRC daemon, srcmstr. The AIX-specific method is to get information about the process with the AIX getprocs routine. It is acceptable to use a non-standard method here, since the code will only be compiled on AIX systems. See Figure 10 on page 23 for the definition of the get_parent_info routine, which uses getprocs. See Figure 11 on page 24 for the definition of the is_parent_SRC routine, which calls get_parent_info, and then uses the returned information to determine whether the parent of the daemon process is srcmstr.

If a daemon running on an AIX system determines it was not started by inetd nor by the SRC, it might log an error message, and terminate with an error code. This action is consistent with the practice that, on an AIX system, daemons shipped by Scalable POWERparallel Systems, RS/6000 Division, are under the control of inetd or the SRC. Daemons might decide not to follow this practice when in debugging mode.

Note that if a daemon is started by the SRC, and the srcmstr process terminates before the daemon determines how it was started, the daemon may not realize it was started by SRC. This is an acceptable exposure. If the daemon process is inherited by the init process before the daemon process calls the getppid routine in get_parent_info, the daemon process will believe it was not started by SRC, but by the init process. This ends up being acceptable. The daemon will either terminate itself, or it will proceed. If it terminates itself, no harm is done. If it proceeds under the assumption that it was started by the init process, it will function correctly, except it will not respond to some requests made by SRC system administration commands. No harm will be done. If the daemon process is inherited by the init process after the getppid call is made in get_parent_info, but before the getprocs call is made in the same routine, the daemon should terminate. Again, no harm is done.

If a daemon was not started by inetd, nor started by the SRC, and it does not terminate as a result (because it is not running on an AIX system, or it is running on an AIX system in debug mode), it should determine if its current parent is the init process. This can be done on most UNIX systems by calling the getppid routine and comparing the returned PID to 1. If the PID of the current parent process is 1, the daemon process was either started by the init process, or the process that started it has terminated. If the original parent process was srcmstr, it will be acceptable for the daemon process to continue under the assumption it was started by init, as discussed above. If the original parent process was a shell, proceeding under the assumption that the daemon was started by the init process is also acceptable. In this case, the daemon process may not be able to initialize itself, and therefore it will terminate[6]. It is also possible it will be able to initialize itself, and continue to run. In either case, no harm will be done.

[6] If the process was started by a job control shell, it may be a process group leader. This may cause the attempt to disassociate from the controlling terminal to fail.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#include <procinfo.h>
#include <sys/proc.h>

int get_parent_info(struct procsinfo **pipp)
{
      static int got_pi = FALSE;
      static struct procsinfo pi;

      pid_t ppid;
      pid_t scratch_ppid;

      if (got_pi == FALSE) {

             ppid = getppid();
             scratch_ppid = ppid;

             if (getprocs(&pi, sizeof pi, NULL, 0, &scratch_ppid, 1) != 1) {
                   /* Look at errno for an error code. */
                   return 1;
             }

             got_pi = TRUE;

             if ((pi.pi_state == SNONE) || (pi.pi_state == SIDL) ||
                 (pi.pi_state == SZOMB)) {
                    /* This entry does not describe a running process. */
                    return 1;
             }

             if (pi.pi_pid != ppid) {
                   /* This entry does not describe the parent process. */
                   return 1;
             }

      }

      *pipp = &pi;
      return 0;

}
```

Figure 10. Use of AIX getprocs *Routine to Obtain Information about the Parent Process*

3.2 Ignore Terminal-Generated Signals

Section 3.4, “Disassociate the Daemon from Its Controlling Terminal” on page 25 discusses the need to disassociate a daemon process from its controlling terminal. The section describes what a controlling terminal is, what it means to be disassociated from one, and why a daemon should disassociate from its controlling terminal. Briefly stated, the rationale for the disassociation is that a

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#include <procinfo.h>
#include <sys/proc.h>

int is_parent_SRC(int *it_is)
{
      struct procsinfo *pip;
      int rc;


      *it_is = FALSE;

      if ((rc = get_parent_info(&pip)) != 0) {
            return rc;
      }

      if (strcmp(pip->pi_comm, "srcmstr") != 0) {
            return 0;
      }

      if (pip->pi_uid != 0) {
            return 0;
      }

      if ((pip->pi_ppid != 1) || (pip->pi_pgrp != pip->pi_pid) ||
          (pip->pi_sid != pip->pi_pid) || (pip->pi_ttyp != 0)) {
            return 0;
      }

      *it_is = TRUE;

      return 0;
}
```

Figure 11. Determining if SRC Started the Daemon

controlling terminal can cause the delivery of a signal to the daemon process whose default action is to terminate the process.

Before being able to successfully disassociate the daemon from a controlling terminal, it may be necessary for the daemon to migrate itself to a child process; this is described in section 3.3, “Migrate the Daemon to Another Process.” This section describes how a daemon process should migrate itself to another process, and under what conditions it should do so. The important point here is that while the daemon is migrating itself, the processes involved may have a controlling terminal. If the processes do have a controlling terminal, some of the actions performed during the migration can cause a terminal-generated signal to be delivered to the process to which the daemon is migrating. If the disposition of the delivered signal in the receiving process is the default, the process may terminate.

It is simple to guard against this problem. The daemon should simply change the disposition for all terminal-generated signals so they are ignored. If a child

process is created, it will inherit these changes to signal dispositions. This will protect the process from termination due to the delivery of terminal-generated signals.

There is another advantage to ignoring terminal-generated signals. If, after the daemon process has disassociated itself from its controlling terminal, it accidentally acquires another controlling terminal, the delivery of terminal-generated signals will not cause the daemon to mysteriously terminate.

As discussed in section 3.4, "Disassociate the Daemon from Its Controlling Terminal," the terminal-generated signals are SIGHUP, SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, and SIGTTOU. The ignore_terminal_signals routine in Figure 12 on page 26 shows how these signals can be ignored.

3.3 Migrate the Daemon to Another Process

When the daemon program is started by a shell, the shell calls fork to create a child process, and the child process uses an exec routine to run the daemon program. Unless the shell was instructed to run the daemon program in the background (with &), the shell process waits for the child process to terminate before doing anything else. This can be a problem, since daemons tend to run for a long time. If the daemon was started from a login shell, the shell will be prevented from processing any more commands entered at its terminal. If the daemon was started from a shell script, the shell will be prevented from processing the rest of the shell script.

To prevent these problems, it is customary for the daemon program to migrate to another process when started from a shell. To do this, the daemon program calls the fork routine. When the fork routine returns in the parent process (the child process, from the shell process's point of view), that process terminates itself by calling exit. When the fork routine returns in the child process (a grandchild process, from the shell process's point of view), that process continues to run as the daemon. See Figure 13 on page 27. Since the shell process's child process has terminated, the shell process is free to process the next shell command from its input.

When the daemon program is started by init, the SRC, or inetd, the originating process calls the fork routine to create a child process. The code in the child process calls an exec routine to run the daemon program. The parent process considers the child process that it created to be the daemon. When the child process terminates, the parent process considers the daemon to have terminated. Therefore, it would be a mistake for the daemon program to execute the code in Figure 13 on page 27 if the daemon was started by init, the SRC, or inetd.

3.4 Disassociate the Daemon from Its Controlling Terminal

A process is usually associated with a controlling terminal. A controlling terminal can affect a process in ways that are generally considered undesirable for a daemon. Therefore, part of daemon initialization is disassociating the daemon process from its controlling terminal. Section 3.4.1, "Introduction to Sessions and Controlling Terminals" describes the function and features of controlling terminals. Section 3.4.2, "How to Disassociate from a Controlling Terminal" on page 30 describes how a daemon can disassociate from its

```
#include <unistd.h>
#include <signal.h>

int ignore_terminal_signals(void)
{

      /*
       *  The following terminal generated signals will be ignored:
       *
       *  SIGHUP  - Terminal hangup.
       *  SIGINT  - Terminal interrupt.
       *  SIGQUIT - Terminal quit.
       *  SIGTSTP - Interactive stop.
       *  SIGTTIN - Background read attempt from controlling terminal.
       *  SIGTTOU - Background write attempt to controlling terminal.
       *
       */

      int sig[] = {SIGHUP, SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, SIGTTOU};

      int num_sigs = (sizeof sig) / (sizeof sig[0]);
      int i;
      struct sigaction new_action;


      /*
       *  Set up generic ignore signal action structure.
       */
      new_action.sa_handler = SIG_IGN;
      (void) sigemptyset(&(new_action.sa_mask));
      new_action.sa_flags = 0;

      /*
       *  Ignore terminal generated signals.
       */
      for (i = 0; i < num_sigs; i++) {

            if (sigaction(sig[i], &new_action, NULL) == -1) {
                  return 1;
            }

      }

      return 0;
}
```

Figure 12. The ignore_terminal_signals *Routine*

controlling terminal. A process can obtain another controlling terminal after having disassociated itself from its original controlling terminal. Section 3.4.4, “How to Prevent Reacquisition of a Controlling Terminal” on page 31 describes how to avoid acquiring another controlling terminal.

```
#include <unistd.h>

      .
      .
      .

      pid_t pid;

      if ((pid = fork()) == (pid_t)-1) {
            /* parent process: fork() failed */
            /* Log some error message */
            exit(1);
      }

      if (pid != (pid_t)0) {
            /* parent process: exit */
            exit(0);
      }

      /* child process: continue as daemon */

      .
      .
      .
```

Figure 13. Migrating Daemon to Another Process

3.4.1 Introduction to Sessions and Controlling Terminals

Controlling terminals, and their behavior, are related to groupings of processes called process groups and sessions.

Each process in a POSIX-compliant UNIX system belongs to a process group. A process group is a collection of one or more processes. When a process group is created, it contains one process, and that process is known as the process group leader. The process group is identified by a process group ID. The process group ID of a process group is the PID of its process group leader. When a process creates a child process, the child process starts out in the process group of the parent process. A process group exists until there are no more processes in the process group. It is possible for a process group leader to terminate, and for the process group to remain in existence; this will occur if there are still processes in the process group when the process group leader terminates. In this case, the process group ID remains the PID of the process that had been the process group leader.

A process may determine its process group ID with the getpgrp function. If the process group ID returned is the PID of the process, the process can conclude that it is the process group leader of the process group. If a process is not a session leader (described in the following paragraph), it may leave its current process group and join another process group within the same session (described in the following paragraph) with the setpgid function. If the process group being joined does not exist, it is created. When a process forms a new process group with setpgid, it becomes the process group leader of the new process group. Most processes do not change the process group to which they belong. Job control shells like the Korn shell are the most common manipulators of process group membership.

Each process in a POSIX-compliant UNIX system also belongs to a session. A session is a collection of one or more process groups. Since process groups are composed of processes, a session may also be viewed as a collection of one or more processes. When a session is created, it contains one process, and that process is known as the session leader. At creation time, the session also contains one process group; the session leader is the only process in the process group, and it is the process group leader. On some systems, the session is identified by a session ID[7]. The session ID of a session is the PID of its session leader. When a process creates a child process, the child process is in the session of the parent process. A session exists until there are no more processes in the session. It is possible for a session leader to terminate, and for the session to remain in existence; this will occur if there are still processes in the session when the session leader terminates. In this case, the session ID remains the PID of the process that had been the session leader.

A process may determine its session ID with the `getsid` function[7]. If the session ID returned is the PID of the process, the process can conclude that it is the session leader of the session. If a process is not a process group leader[8], it may leave its current session and form a new session with the `setsid` function. When a process forms a new session with `setsid`, it becomes the session leader of the new session, and the process group leader of a new process group that is the sole process group in the session. Most processes do not change the session to which they belong. However, in order to disassociate from a controlling terminal, a daemon must become a session leader.

A session may or may not have a controlling terminal. When a session is created, it starts without a controlling terminal. If the session leader opens a terminal device, and several conditions are met, the session obtains the terminal as its controlling terminal. The conditions that must be met are:

1. The session does not already have a controlling terminal.
2. The `O_NOCTTY` value was not specified in the `oflag` parameter in the call of the `open` function.
3. The terminal device is not the controlling terminal for any other session on the system.

Once a session has a controlling terminal, the session leader is also known as the controlling process of that terminal. Also, the terminal is considered the controlling terminal of each process in the session. When a new process is created within a session, it inherits the controlling terminal of the session. A process may determine if the process and its session have a controlling terminal by attempting to open the path name returned by the `ctermid` function[9]. This path name can be used by a process to open its controlling terminal. If the call to the `open` function fails, and sets `errno` to the value `ENXIO`, the process does not have a controlling terminal.

---

7 POSIX 1003.1, XPG3, and XPG4 Base Conformance do not define a session ID or the `getsid` routine. However, they are defined in XPG4 XOPEN UNIX Conformance and AIX 4.1.

8 Notice that the `setsid` restriction is the process cannot be a group leader. This is a stronger restriction than might be expected, which is that the process cannot be a session leader. Since a session leader is always a process group leader, the actual restriction covers the expected restriction.

9 This path name is commonly `/dev/tty` on UNIX systems. It is frequently hard-coded in UNIX programs.

Once a session has obtained a controlling terminal, one process group in the session is considered a foreground process group, and all other process groups in the session are considered background process groups. This should be familiar to shell users who use job control in a shell such as the Korn shell. When a login shell is a job control shell, the shell is a session leader. The shell makes each job a separate process group. The shell controls which process group is the current foreground process group of the session using the `tcgetpgrp` and `tcsetpgrp` functions.

A process in the foreground process group of a session is permitted to read from the session's controlling terminal. A process in one of a session's background process groups is not permitted to read from the session's controlling terminal. If such a process attempts to read from the session's controlling terminal, it is sent the `SIGTTIN` signal. The default action for the `SIGTTIN` signal is to stop the process. The process will remain stopped until the `SIGCONT` signal is sent to the process. The `SIGCONT` signal is usually sent to a stopped process by a job control shell after the shell has called `tcsetpgrp` to make the process's process group the foreground process group.

The user of a shell can control whether processes in a background process group are permitted to write to the controlling terminal. This is done with the `stty` command[10]. If processes in the background process groups are not permitted to write to the controlling terminal, and such a process attempts to do so, the process is sent the `SIGTTOU` signal. Like `SIGTTIN`, the default action for `SIGTTOU` is to stop the process. Such a process is restarted in the same manner as described for `SIGTTIN`.

Certain key sequences entered on a controlling terminal cause signals to be sent to all processes in the session's foreground process group. POSIX defines three of these key sequences: the interrupt character, the quit character, and the suspend character.

Pressing the interrupt character, typically Control-c[11], on a controlling terminal may cause the `SIGINT` signal to be sent to all processes in the session's foreground process group. When the `SIGINT` signal is delivered to a process, the default action is to terminate the process.

Pressing the quit character, typically Control-[12], on a controlling terminal may cause the `SIGQUIT` signal to be sent to all processes in the session's foreground process group. When the `SIGQUIT` signal is delivered to a process, the default action is to generate a core file of the process and to terminate the process.

Pressing the suspend character, typically Control-z[13], on a controlling terminal may cause the `SIGTSTP` signal to be sent to all processes in the session's foreground process group. When the `SIGTSTP` signal is delivered to a process, the default action is to stop the process. Typically, the process is continued when the shell sends the process group the `SIGCONT` signal.

---

10 The command `stty tostop` will prevent processes in a background process group from writing to the controlling terminal. The command `stty -tostop` will allow processes in a background process group to write to the controlling terminal.

11 The interrupt character is defined to be Control-c with the command `stty INTR ^c`.

12 The quit character is defined to be Control- with the command `stty QUIT ^`.

13 The suspend character is defined to be Control-z with the command `stty SUSP ^z`.

If a modem disconnect is detected on a controlling terminal by the terminal interface, the SIGHUP signal is sent to the controlling process (the session leader). When the SIGHUP signal is delivered to a process, the default action is for the process to terminate.

When the session leader of a session with a controlling terminal terminates, the session may continue to exist. However, all processes in the foreground process group are sent the SIGHUP signal by the system. This may cause the processes in the foreground process group to terminate. If the session leader was a job control shell, it may also send the SIGHUP signal to processes in the session's background process groups. When the Korn shell is terminating, it sends the SIGHUP signal to all processes in the shell's session, except for processes in process groups that were created with the nohup command.

3.4.2 How to Disassociate from a Controlling Terminal

The most important points from the previous section are:

- A process in a session with a controlling terminal may be affected by the terminal.
  - A modem disconnect on the controlling terminal may result in the process being sent a signal whose default action is to terminate the process.
  - Certain key sequences typed on the controlling terminal may result in a signal to the process whose default action is to terminate the process or to suspend the execution of the process.
  - Attempts by the process to read from or write to the controlling terminal may result in a signal to the process whose default action is to stop the process.
- A process in a session with a controlling terminal may be affected by the session leader.
  - If the session leader terminates, the process may be sent a signal whose default action is to terminate the process.

The consequences of having a controlling terminal are valuable for most processes. However, the consequences are undesirable for a daemon process. Therefore, a daemon process should disassociate from any controlling terminal it may start with, by becoming the session leader of a new session. The daemon process must also take care not to re-acquire a controlling terminal. That is discussed in a later section.

It is important to note that when the daemon process is initializing itself, it may already be the session leader of a session without a controlling terminal. This is the case on AIX systems when the process is started by the init process and by the System Resource Controller (SRC). This may also be the case when the daemon process is started by the Internet Superserver (inetd).

Figure 14 on page 32 shows a recommended method for a daemon process to ensure that it is the session leader of a session without a controlling terminal. The create_session routine first calls the setsid function. The call to setsid will fail if the process is a process group leader. The daemon process will be a process group leader if it was made a session leader by init, the SRC, or inetd. Therefore, the return value from setsid is ignored. Instead of using the setsid return value, the create_session routine verifies that the daemon process is now a session leader without a controlling terminal. The DAE_IS_SESSION_LEADER

macro is invoked to determine if the process is a session leader. The macro was defined in Figure 8 on page 17. The has_controlling_terminal function, presented in Figure 15 on page 33, is called to determine if the process has a controlling terminal. If the routine can open the path name returned by ctermid, the process has a controlling terminal. If the routine does not have a controlling terminal, the open routine will set errno to ENXIO, and return -1.

If the daemon process is a session leader, but has a controlling terminal, after the setsid call, the create_session routine tries another method to disassociate the session from the controlling terminal. This method is coded in the release_controlling_terminal routine of Figure 16 on page 34. It involves calling the ioctl function, specifying both the file descriptor through which the controlling terminal has been opened and the TIOCNOTTY command. This is a BSD technique for disassociating a controlling terminal from a process. It is supported in the daemon support routines when compiled on an AIX system with BSD compatibility. Refer to Figure 7 on page 16 for the definition of DAE_NOTTY_IOCTL_SUPPORT, which determines whether the release_controlling_terminal routine actually attempts the ioctl call.

Under what conditions might a daemon process be a session leader, but have a controlling terminal after the call to setsid in create_session? If the daemon process was started by init, the SRC, or inetd, it may have been made into a session leader by the starting process. If the daemon process opens a terminal device before the call to create_session, it may enter create_session as a session leader with a controlling terminal. The setsid call in create_session will not disassociate the process from the controlling terminal, because the process is a process group leader. The ioctl call in release_controlling_terminal may disassociate the process from the controlling terminal.

3.4.3 Daemon Migration and Losing the Controlling Terminal

Section 3.3, “Migrate the Daemon to Another Process” on page 25 discusses the importance of migrating a daemon started by a shell to another process, so the shell is not prevented from processing its next command. When a daemon is started from a shell, it is essential that the daemon be migrated to another process before an attempt is made to disassociate from the controlling terminal.

The setsid routine will not make a process a session leader if it is already a process group leader. When a job control shell executes a command, it may make the process executing the command a process group leader. Such a process cannot use setsid to become a session leader and lose its controlling terminal. However, a child process of such a process will not be a process group leader, and will be able to use setsid to become a session leader and lose its controlling terminal.

3.4.4 How to Prevent Reacquisition of a Controlling Terminal

Once a daemon process has disassociated from its controlling terminal, it should be very careful not to acquire one again. This can be done simply enough by specifying O_NOCTTY in the oflag parameter in any open call that might be opening a terminal device. For an example, see the open call in the has_controlling_terminal function in Figure 15 on page 33.

Be careful with any functions that may in turn call open. For example, if the fopen routine is used to open a terminal device, the daemon process may acquire a controlling terminal, because fopen calls open without specifying O_NOCTTY.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/stat.h>

#if defined(DAE_NOTTY_IOCTL_SUPPORT)
#include <sys/ioctl.h>
#endif


int create_session(void)
{
      /*
       *  Attempt to make this process the session leader of a new session.
       *  If this succeeds, the process loses its controlling terminal.
       *  The attempt will fail if the process is already a session
       *  leader.  So, the return code is ignored.
       */
      (void) setsid();
      /*
       *  Make sure this process is a session leader.
       */
      if (!DAE_IS_SESSION_LEADER()) {
            return 1;
      }

      /*
       *  If this process is a session leader, but has a controlling
       *  terminal, try one more time to get rid of the terminal.
       *  If the controlling terminal cannot be released, give up.
       */
      if (has_controlling_terminal()) {
            release_controlling_terminal();

            if (has_controlling_terminal()) {
                  return 1;
            }
      }

      return 0;
}
```

Figure 14. The create_session *Routine*

Instead of using the fopen routine, a daemon may open the terminal device with open, specifying O_NOCTTY, and then call fdopen to associate a standard I/O stream with the file descriptor returned by open. The daemon support routine dae_fopen uses this technique to provide the functionality and interface of fopen, without the danger of acquiring a controlling terminal.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/stat.h>

#if defined(DAE_NOTTY_IOCTL_SUPPORT)
#include <sys/ioctl.h>
#endif


int has_controlling_terminal()
{
      int   fd;
      char  ctermid_name[L_ctermid];


      /*
       *  See if this process has a controlling terminal.  On error, assume
       *  it does.
       */

      if (ctermid(ctermid_name) == NULL) {
            abort();    /* daemon support code does not really call abort */
      }

      fd = open(ctermid_name, O_RDWR | O_NOCTTY);

      if (fd != -1) {
            (void) close(fd);
            return TRUE;
      }

      if ((fd == -1) && (errno != ENXIO)) {
            abort();    /* daemon support code does not really call abort */
      }

      /*
       *  This process does not have a controlling terminal.
       */

      return FALSE;
}
```

Figure 15. The has_controlling_terminal *Routine*

3.4.5 Non-Standard Ways of Disassociating from a Controlling Terminal

The notion of a session is a more recent concept in UNIX systems than the notion of a process group or the notion of a controlling terminal. Before sessions were introduced, there were techniques for a process to disassociate itself from its controlling terminal that did not involve the notion of a session. The techniques differed between System V-based systems and BSD-based systems. Many programs that use the older techniques still exist. This section describes the older techniques and explains why they still work on a system

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/stat.h>

#if defined(DAE_NOTTY_IOCTL_SUPPORT)
#include <sys/ioctl.h>
#endif


#if !defined(DAE_NOTTY_IOCTL_SUPPORT)
void release_controlling_terminal(void)
{
      return;
}
#endif


#if defined(DAE_NOTTY_IOCTL_SUPPORT)
void release_controlling_terminal(void)
{
      int   fd;
      char  ctermid_name[L_ctermid];


      if (ctermid(ctermid_name) == NULL) {
            return;
      }

      fd = open(ctermid_name, O_RDWR | O_NOCTTY);

      if (fd == -1) {
            return;
      }

      (void) ioctl(fd, TIOCNOTTY, 0);

      (void) close(fd);

      return;
}
#endif
```

Figure 16. The release_controlling_terminal *Routine*

supporting the notion of sessions and the setsid routine. For new programs, it is recommended that you use the newer techniques previously described.

The older System V and BSD techniques for disassociating from a controlling terminal are described in *UNIX Network Programming*, W. Richard Stevens, 1990, Prentice Hall, Englewood Cliffs. Before discussing the techniques, the book points out that a process should become a process group leader so it does not receive signals directed to the process group that the daemon process inherited. The book points out that the routine to call to become a process group leader on

both types of systems is the setpgrp routine. It further points out that the syntax of the setpgrp routine differs between the two types of systems. On System V systems, the setpgrp routine takes no parameters. On BSD systems, the setpgrp routine takes two parameters: the process ID of the process whose process group membership is to change, and the process group ID of the process group to be joined or created. The book also discusses the need to migrate the daemon to a child process, if the daemon was started by a shell. Then, it discusses disassociating from the controlling terminal as follows:

> Under System V, a process disassociates itself from its control terminal (if it has one) by calling the setpgrp system call. In addition to making the calling process a process group leader, as described above, if the process is not already a process group leader when it calls setpgrp, this call disassociates the process from its control terminal. Therefore we can use this system call to do two things - remove ourself from the inherited process group and disassociate from the inherited control terminal. But the only way a process can guarantee that it is not already a process group leader when it calls setpgrp is to do a fork and then call setpgrp from the child process. Since the process group ID is copied from the parent to the child across the fork, and since the child will have a different process ID from the parent, we are certain the child is not a process group leader. By doing the fork described previously (to run in the "background") we handle this condition. The System V code is
>
> ```
> if (fork() != 0)
>    exit(0);      /* parent process */
>
> /* first child process */
>
> setpgrp();  /* change process group and lose control tty */
> ```
>
> Under 4.3BSD a process removes its association with its controlling terminal by opening the file /dev/tty and issuing a TIOCNOTTY ioctl for the device. The 4.3BSD code is
>
> ```
> if (fork() != 0)
>    exit(0);            /* parent process */
>
> /* first child process */
>
> setpgrp(0, getpid());  /* change process group */
>
> if ( (fd = open("/dev/tty", O_RDWR)) != 0) {
>    ioctl(fd, TIOCNOTTY, (char *) 0);   /* lose control tty */
>    close(fd);
> }
> ```

The techniques described previously for System V and BSD systems will work on an AIX system. AIX supports a System V compatible version of setpgrp in the library libc.a. It also supports a BSD compatible version of setpgrp in libbsd.a. It also supports the TIOCNOTTY ioctl command. The libc.a version of setpgrp will cause the calling process to become a session leader if it is not already a process group leader. As a result, the process will lose its controlling terminal. The libbsd.a version of setpgrp will just change the process group of the specified process. However, the TIOCNOTTY ioctl command will cause the calling process to become a session leader. As a result, the process will lose its controlling terminal. Therefore, a process becomes a session leader without a

controlling terminal on an AIX system when it executes the System V or BSD code fragments shown previously[14].

3.5 Deal with Non-Terminal Generated Signals

Daemon initialization might want to deal with signals that are not generated by controlling terminals. The three likely candidates are SIGCHLD, SIGPIPE, and SIGDANGER.

3.5.1 SIGCHLD and Zombie Processes

It is common for a daemon that acts as a concurrent server to create child processes, and to not care about the exit status of the child processes. If this is the case, the daemon should not allow the child processes to remain zombies for long. When a process terminates before its parent, unless the proper arrangements are made, the child process becomes a zombie process. A zombie process still occupies an entry in the kernel's process table, so its parent process can collect its status with the wait or waitpid function. Once the parent process has collected the status of a child process, the process table entry used by that child process is freed. If the parent process is not interested in collecting the exit status of its child processes, it would be appropriate to use one of the techniques described in the following sections.

When the daemon is running on System V, on AIX, or on a system that complies with XPG4 X/OPEN UNIX, setting the disposition of the SIGCHLD signal so it is ignored has special semantics. On such systems, when the SIGCHLD signal is ignored in a process, child processes of the process do not become zombies. A routine similar to prevent_zombies in Figure 17 on page 37 is used in the daemon support routines to prevent the creation of zombies when the code is compiled on AIX systems and systems compliant to XPG4 X/OPEN UNIX. Refer to Figure 7 on page 16 for the code that determines if DAE_IGN_ZOMBIES_SUPPORT is defined.

On systems that do not support the special semantics of ignoring the SIGCHLD signal, the daemon can install a signal handler for the signal that simply calls waitpid to collect, and ignore, the exit status of the child process that has terminated. Routines similar to prevent_zombies and zombie_killer in Figure 18 on page 38 are used in the daemon support routines to prevent long term zombies. The zombie_killer routine, which is the SIGCHLD signal handler installed by prevent_zombies, loops through calls to waitpid while the waitpid routine returns positive numbers, which are the PIDs of child processes that have terminated. Thus, waitpid will be called as long at it reports the death of another child process. The routine also loops while waitpid indicates it has been interrupted[15] . The first parameter to waitpid, -1, means the routine should report any child process that has terminated. The second parameter to waitpid, NULL, indicates the exit status of the child process is of no interest to the caller. The third parameter to waitpid, WNOHANG, indicates that if there is no child process whose termination is to be reported, return immediately.

[14] To use the BSD code, the program must be compiled with the _BSD macro defined to 43 or 44, and the program must be linked with both the libbsd.a and libc.a libraries.

[15] Interrupted system calls are discussed in 7.5, "Interrupted System Calls" on page 170.

```
#include <signal.h>
#include <errno.h>


#if defined(DAE_IGN_ZOMBIES_SUPPORT)
int prevent_zombies(void)
{

      /*
       *  In ignoring SIGCHLD signals, not only will SIGCHLD signals
       *  not be delivered, but child processes will not become zombies,
       *  and the daemon cannot wait() for the termination of a particular
       *  child.  If the daemon process wants notification when its child
       *  processes die or are stopped, it should catch this signal.
       *  If the daemon process does not want notification when its child
       *  processes die or are stopped, but it wants to wait() for
       *  child processes, the action for SIGCHLD should be set to
       *  SIG_DFL.
       */

      struct sigaction new_action;


      /*
       *  Set up generic ignore signal action structure.
       */
      new_action.sa_handler = SIG_IGN;
      (void) sigemptyset(&(new_action.sa_mask));
      new_action.sa_flags = 0;

      if (sigaction(SIGCHLD, &new_action, NULL) == -1) {
            /* Look at errno for details about the error. */
            return 1;
      }

      return 0;
}
#endif
```

Figure 17. The prevent_zombies *Routine*

3.5.2 SIGPIPE and Broken Connections

If a daemon is involved in connection-oriented communications using pipes or sockets, it should ignore, or install a signal handler for, the SIGPIPE signal. The SIGPIPE signal is generated when a process writes to a pipe or a socket, and there is no reader process. This condition is often dealt with more effectively by looking for an error indication from a function like write, than it is in a signal handler.

The default action associated with receiving a SIGPIPE signal is process termination. If the signal is not ignored, or a signal handler is not installed, the termination of a reader process may cause the writer process to terminate. This is seldom the desired behavior when the writer process is a daemon. Therefore, the dae_init routine, of the daemon support routines, changes the disposition of the SIGPIPE signal so it is ignored.

```
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>


#if !defined(DAE_IGN_ZOMBIES_SUPPORT)
void zombie_killer(int signo)
{
      pid_t pid;


      do {
            pid = waitpid((pid_t)-1, NULL, WNOHANG);
      } while ((pid > (pid_t)0) || (pid == (pid_t)-1 && errno == EINTR));

      return;
}
#endif


#if !defined(DAE_IGN_ZOMBIES_SUPPORT)
int prevent_zombies(void)
{

      struct sigaction new_action;


      /*
       *  Set up signal action structure.
       */
      new_action.sa_handler = zombie_killer;
      (void) sigemptyset(&(new_action.sa_mask));
      new_action.sa_flags = SA_RESTART | SA_NOCLDSTOP;

      if (sigaction(SIGCHLD, &new_action, NULL) == -1) {
            /* Look at errno for information about the error. */
            return 1;
      }

      return 0;
}
#endif
```

Figure 18. The prevent_zombies *and* zombie_killer *Routines*

3.5.3 SIGDANGER and AIX Paging

When an AIX system runs low on paging space, it attempts to alleviate the problem by killing processes. A daemon process that is crucial to the proper functioning of a system should protect itself from being killed in such a situation. One of the methods available is to install a signal handler for the SIGDANGER signal. This is discussed in detail in Chapter 8, “AIX Paging Space Allocation” on page 203.

3.6 Close Unnecessary File Descriptors

Since a daemon process can be started in a variety of ways, it has become a custom for a daemon to close any file descriptors it does not need. There is a practical purpose to this exercise. While a process has a file opened, the file system in which the file exists usually cannot be unmounted. Since a typical daemon process runs for a long time, if it had any unneeded files open, it could unnecessarily prevent the system administrator from unmounting a file system.

Care should be taken with file descriptors 0, 1, and 2. If the daemon process was started by the `inetd` process, these file descriptors probably should not be closed; `inetd` associated them with the socket through which the daemon should communicate with its client. If the daemon was started by the SRC, the file descriptors may be associated with something meaningful. The SRC definition of a daemon specifies how these file descriptors should be set up. It is also possible for the `init` process to set these descriptors. If the daemon process was started by a shell, these file descriptors are probably not meaningful. They are usually associated with the shell session's controlling terminal.

The daemon support routines include a routine similar to `close_files` in Figure 19 on page 40. A parameter is passed to the routine, indicating whether the daemon was started by `inetd` or the SRC. If the daemon was started by `inetd` or SRC, all file descriptors except file descriptors 0, 1, and 2 will be closed; otherwise, all file descriptors will be closed. The daemon support routines do not preserve file descriptors 0, 1, and 2 when the daemon is started by `init` for the following reasons:

- It is difficult, if not impossible, to distinguish between a process started by `init` and a process inherited by `init`. Refer to the discussion in 3.1, "Determine How the Daemon Was Started" on page 19.
- Starting a process with `init` and having it redirect input or output can lead to the acquisition of a controlling terminal. Therefore, it should not be encouraged.

The `close_files` routine uses one of two methods to close unneeded file descriptors. If `DAE_FCNTL_CLOSE_SUPPORT` is defined, an AIX-specific method is used. Otherwise, a generic POSIX-compliant method is used. Referring to Figure 7 on page 16, we see that `DAE_FCNTL_CLOSE_SUPPORT` is only defined if the code is compiled on an AIX system.

The POSIX method involves determining the maximum number of files that can be opened by a process on this system, and calling the `close` function for each of the file descriptors that may potentially be open. The maximum number of files that can be opened by a process varies from system to system. The value can be determined by a call to `sysconf` specifying the `_SC_OPEN_MAX` parameter. If the `sysconf` routine returns `-1`, but does not change the value of `errno`, the system does not have a defined maximum number of files that can be opened by a process. In that case, a guess at a reasonable maximum must be made. This code guesses 2000, which happens to be the maximum on an AIX system. Once a maximum is established, a loop of calls to `close` is executed. This loop will probably be executed thousands of times. Most of the `close` calls will probably return an error indication, because most of the file descriptors were probably not open in the first place.

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>


int close_files(int inetd_or_SRC)
{

      int closemin;
      int closemax;
      int i;


      closemin = (inetd_or_SRC) ? 3 : 0;

#if defined(DAE_FCNTL_CLOSE_SUPPORT)

      if (fcntl(closemin, F_CLOSEM, 0) == -1) {
            return 1;
      }

#else
      #define OPEN_MAX_GUESS  2000

      errno = 0;
      if ((closemax = sysconf(_SC_OPEN_MAX)) == -1) {
            if (errno == 0) {
                  closemax = OPEN_MAX_GUESS;
            } else {
                  return 1;
            }
      }

      for (i = closemin; i < closemax; i++) {
            (void) close(i);
      }

      #undef OPEN_MAX_GUESS

#endif

      return 0;

}
```

Figure 19. The close_files *Routine*

The AIX-specific method is to call fcntl with the F_CLOSEM command. This causes all file descriptors in the process, starting with the one referred to by the first parameter, to be closed. This method has two advantages over a user level loop of calls to close. First, only one system call is made, instead of thousands of system calls. Second, the AIX kernel keeps track of the largest file descriptor currently opened by the process; this allows the close loop in the kernel to avoid iterating thousands of times in normal situations.

3.7 Ensure Certain File Descriptors are Open

When the open routine succeeds in opening a file, the file descriptor used is the lowest available file descriptor[16] in the process. In a daemon process that has just closed most, if not all, of its file descriptors, the lowest available file descriptor may be file descriptor 0, 1, or 2. The convention that file descriptors 0, 1, and 2 are used for a process's standard input, standard output, and standard error output is so established on UNIX systems that when these descriptors refer to files that a shell did not set up as standard input, output, and error output, confusion can result.

To avoid such confusion, a daemon process should ensure that these three file descriptors refer to an open file when daemon initialization is complete. If one of the descriptors does not refer to something meaningful, opening /dev/null on the file descriptor seems to be rational. Read attempts from /dev/null return 0 bytes. Data written to /dev/null is discarded.

Once the daemon has ensured that file descriptors 0, 1, and 2 are associated with open files, successful calls to the open and creat routines within the daemon will always return file descriptors larger than 2.

Figure 20 on page 42 shows the null_files routine, which is similar to a routine in the daemon support routines. For each of the file descriptors 0, 1, and 2, the null_files routine uses the fstat function to determine if the file descriptor is associated with an open file. If a file descriptor is not associated with an open file, the fstat routine should set errno to EBADF, and return -1. If one of these descriptors is not associated with an open file, the open routine is called on /dev/null. The file descriptor returned by open should be the same as the file descriptor being considered, since the file descriptor being considered is always the lowest available file descriptor in the process. The code makes sure this is always the case.

3.8 Change the Current Working Directory

When a daemon process is first started, its current working directory depends on how it was started. If it was started by inetd, init, or the SRC, the current working directory is probably the root directory. If started by a shell, it could be any directory. If the daemon did not change its current working directory, it may prevent a file system from being unmounted, since the process has the directory open. Therefore, the daemon initialization code should change its current working directory to either the root directory, /, or to a directory that is meaningful to the daemon.

It may be useful to keep in mind that when a process generates a core file, the file is placed in the current working directory. If the daemon has a directory dedicated to it, it may make sense for the daemon to run with that directory as the current working directory. If the daemon ever generated a core file, this would reduce the chances of the core file being overwritten by another process generating a core file.

[16] An available file descriptor is one that is not associated with an open file.

```
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>


int null_files(void)
{

      int ifd;
      int ofd;
      int rc;
      struct stat statbuf;


      for (ifd = 0; ifd <= 2; ifd++) {

            rc = fstat(ifd, &statbuf);

            if (rc != -1) {
                  continue;   /* File descriptor useful; consider next one. */
            }

            if (errno != EBADF) {   /* Error was not the expected one. */
                  /* The errno value describes the error. */
                  return 1;
            }

            /*
             *  The stat routine returned EBADF.  The file descriptor
             *  does not refer to an open file.  Open /dev/null on it.
             */
            if ((ofd = open("/dev/null", O_RDWR)) == -1) {
                  /* The errno value describes the error. */
                  return 1;
            }

            if (ofd != ifd) {
                  return 1;   /* The returned file descriptor was unexpected. */
            }

      }

      return 0;
}
```

Figure 20. The null_files *Routine*

The daemon support routines contain a routine similar to misc_stuff in Figure 21 on page 43.

```
#include <unistd.h>


int misc_stuff(void)
{

      if (chdir("/") == -1) {
            /* errno gives specific information about the error */
            return 1;
      }

      (void) umask(0);

      return 0;
}
```

Figure 21. The misc_stuff *Routine*

3.9 Set the Process File Mode Creation Mask

Each process has a file mode creation mask, the umask. The umask value is a bit mask where each bit represents a file permission bit. The file permission bits represent the read, write, and execute permissions for a file owner, group, and others. When a file is created by a process, the umask value of the process is used to mask off permission bits. If the umask value is non-zero, and an open or creat function call is used to create a file, not all of the permission bits specified on the open or creat call may be applied to the created file.

Since a daemon may be started in a variety of ways, it may inherit any valid umask value[17]. Therefore, the daemon should reset its umask value. A umask value of 0 allows the daemon code to directly control the permissions that will apply to created files. The process umask value may be set with the umask function. See the misc_stuff routine in Figure 21 for an example.

[17] The umask value is inherited across the fork and exec functions.

Chapter 4. Managing a Daemon with the init Process

The init process is the first user level process run on a UNIX system. All other user processes are descendants of it. The init process spawns other processes when the system is booted, and when the system's run level is changed. It may also terminate processes when the system run level is changed.

A system run level is really just a tag representing a collection of processes that will run when the init process is told to bring the system into that run level. System run levels that can be specified in AIX are 0 through 9, S, s, M, and m.

- Run levels 0 and 1 are reserved for future use.
- Run level 2 is intended to mean the full multiuser environment.
- Run levels 3 through 9 can be defined by the system's administrator.
- Run levels S, s, M, and m represent maintenance mode.

Daemons to be started at system boot time are started directly or indirectly by the init process. On a system without the System Resource Controller, init may directly spawn a daemon process, or a shell spawned by init to run a shell script may spawn a daemon process. On a system with the System Resource Controller, the SRC daemon, srcmstr, is started by the init process. Daemons under SRC control are started by srcmstr, but the init process is still involved. The init process directly or indirectly executes a startsrc command for every SRC-controlled daemon to be started when the system is booted or changes run levels. The startsrc command sends a start request to srcmstr, causing srcmstr to start the daemon.

On AIX systems, the init process is run with real and effective user IDs of 0 (root), and with real and effective group IDs of 0 (system). A process that is spawned by init runs with these user and group IDs, unless the program being run is a set-user-ID or set-group-ID program, or the program explicitly changes the IDs under which it is running. Therefore, a daemon directly started by init, without the SRC, usually runs with superuser privileges.

4.1 The /etc/inittab File

On AIX, and other systems derived from System V, the actions of the init process can be customized through the /etc/inittab file. When the system is booted, or is to enter one of the run levels 0 through 9, entries in the /etc/inittab file are processed by the init process. The entries are generally processed in the order in which they are found in the file.

Each entry in the /etc/inittab file has the following format:

```
Identifier:RunLevel:Action:Command
```

- The Identifier field is a tag that uniquely identifies the entry.
- The RunLevel field specifies the system run levels on which the command is run. The run levels that can be specified are 0 through 9[18].
- The Command field specifies the command to run.

[18] Run levels a, b, and c can also be specified, but they are not important for this discussion.

- The Action field specifies how the command is to be run.

For full documentation on the meaning of the fields and their possible values, refer to the “inittab File” entry in Chapter 1, “System Files” of *AIX Version 4.1: Files Reference*, SC23-2512.

The contents of an /etc/inittab file from an AIX 4.1 system appears in Figure 22. The entry with the init identifier makes run level 2 the default run level. The keyword initdefault in this entry indicates that the function of the entry is to set the default run level.

```
init:2:initdefault:
brc::sysinit:/sbin/rc.boot 3 >/dev/console 2>&1 # Phase 3 of system boot
powerfail::powerfail:/etc/rc.powerfail 2>&1 | alog -tboot > /dev/console
rc:2:wait:/etc/rc 2>&1 | alog -tboot > /dev/console # Multi-User checks
fbcheck:2:wait:/usr/sbin/fbcheck 2>&1 | alog -tboot > /dev/console
srcmstr:2:respawn:/usr/sbin/srcmstr # System Resource Controller
rctcpip:2:wait:/etc/rc.tcpip > /dev/console 2>&1 # Start TCP/IP daemons
rcnfs:2:wait:/etc/rc.nfs > /dev/console 2>&1 # Start NFS Daemons
cron:2:respawn:/usr/sbin/cron
nimclient:2:wait:/usr/sbin/nimclient -S running
cons:0123456789:respawn:/usr/sbin/getty /dev/console
qdaemon:2:wait:/usr/bin/startsrc -sqdaemon
writesrv:2:wait:/usr/bin/startsrc -swritesrv
uprintfd:2:respawn:/usr/sbin/uprintfd
diagd:2:once:/usr/lpp/diagnostics/bin/diagd >/dev/console 2>&1
```

Figure 22. A Sample /etc/inittab *File*

The uprintfd entry is an example of an entry causing the init process to start a daemon process. The daemon’s executable file is /usr/sbin/uprintfd, as specified in the Command field. The daemon will be started when the system enters run level 2, which, as mentioned earlier, is the system’s default run level. The Action field specifies respawn. This is an important value for daemons. It instructs the init process to start the daemon if it is not already running, to not wait for the daemon to terminate before processing the next entry in the /etc/inittab file, and to restart the daemon when it terminates. The frequency with which the init process is willing to respawn a command is limited. See the init man page entry in *AIX Version 4.1: Command Reference*, SBOF-1851, for details.

The rctcpip entry is an example of an /etc/inittab entry that causes the init process to run a shell script. The entry causes init to run the /etc/rc.tcpip shell script. This shell script starts the TCP/IP daemons. The wait value specified in the Action field causes init to run the command once when entering one of the specified run levels, and to wait for the shell script to terminate before proceeding to the next entry in /etc/inittab. The RunLevel value of 2 causes init to run the shell script when the system enters run level 2.

Several entries in the inittab file interact with the System Resource Controller (SRC). The srcmstr entry starts the SRC daemon, srcmstr. The writesrv and qdaemon entries execute a startsrc command to start a daemon under the control of SRC. Notice that the Action value specified for these entries is wait, not respawn. The startsrc command is a transient program. It just sends a start request to the srcmstr daemon, causing the srcmstr daemon to spawn the daemon identified in the startsrc arguments. Therefore, it would be

inappropriate to respawn the startsrc command when it terminates. When a daemon is placed under SRC control, it can be specified whether the daemon itself is to be respawned or not (refer to 5.2, “Subsystem Definition” on page 62). The /etc/rc.tcpip shell script mentioned above is written such that the TCP/IP daemons can be started under SRC control if the SRC is available.

4.2 Updating the /etc/inittab File

The traditional way to update /etc/inittab is to use a text editor. However, on AIX, the /etc/inittab file is simply the traditional representation of information that is stored in the Object Data Manager (ODM). The file's contents are generated from information in the ODM. If the /etc/inittab file is modified with a text editor on AIX, the changes will disappear the next time the file's contents are updated from the ODM. Refer to Chapter 17, “Object Data Manager(ODM)” in *AIX Version 4.1: General Programming Concepts: Writing and Debugging Programs*, SC23-2533, for information about the ODM.

AIX supplies four commands to manipulate /etc/inittab entries. They are:

mkitab, to add a new entry
chitab, to change an existing entry
lsitab, to list an existing entry
rmitab, to remove an existing entry

The mkitab, chitab, and rmitab commands modify both the ODM and the /etc/inittab file. Refer to the man pages for these commands in *AIX Version 4.1: Command Reference*, SBOF-1851 for detailed information. Figure 23 presents some examples of using these commands to modify the /etc/inittab entries.

```
$ mkitab "testentry:2:respawn:/tmp/nonexistingfile"
$ lsitab testentry
testentry:2:respawn:/tmp/nonexistingfile
$ chitab "testentry:2:once:/tmp/nonexitingfile -d argument"
$ lsitab testentry
testentry:2:once:/tmp/nonexitingfile -d argument
$ grep testentry /etc/inittab
testentry:2:once:/tmp/nonexitingfile -d argument
$ rmitab testentry
$ lsitab testentry
```

Figure 23. Examples of the mkitab, chitab, rmitab, *and* lsitab *Commands*

The lsitab command can be used to list the entire contents of /etc/inittab, as shown in Figure 24 on page 48.

```
$ lsitab -a
init:2:initdefault:
brc::sysinit:/sbin/rc.boot 3 >/dev/console 2>&1 # Phase 3 of system boot
powerfail::powerfail:/etc/rc.powerfail 2>&1 | alog -tboot > /dev/console
rc:2:wait:/etc/rc 2>&1 | alog -tboot > /dev/console # Multi-User checks
fbcheck:2:wait:/usr/sbin/fbcheck 2>&1 | alog -tboot > /dev/console
srcmstr:2:respawn:/usr/sbin/srcmstr # System Resource Controller
rctcpip:2:wait:/etc/rc.tcpip > /dev/console 2>&1 # Start TCP/IP daemons
rcnfs:2:wait:/etc/rc.nfs > /dev/console 2>&1 # Start NFS Daemons
cron:2:respawn:/usr/sbin/cron
nimclient:2:wait:/usr/sbin/nimclient -S running
cons:0123456789:respawn:/usr/sbin/getty /dev/console
qdaemon:2:wait:/usr/bin/startsrc -sqdaemon
writesrv:2:wait:/usr/bin/startsrc -swritesrv
uprintfd:2:respawn:/usr/sbin/uprintfd
diagd:2:once:/usr/lpp/diagnostics/bin/diagd >/dev/console 2>&1
$
```

Figure 24. Listing the Contents of the /etc/inittab *File*

4.3 Changing the System Run Level

The telinit or init commands can be used to change the system run level. When this is done, the init process sends the SIGTERM signal to all processes that are not defined to run in the target run level. Processes can catch this signal, clean themselves up, and terminate in a controlled fashion. If the disposition for the SIGTERM signal is the default, the process will be terminated immediately when the SIGTERM signal arrives. Twenty seconds after the SIGTERM signals have been sent, the SIGKILL signal is sent to processes that have not terminated. A process cannot ignore or catch the SIGKILL signal; it will always terminate a process receiving it.

A similar sequence of events occurs when a system is shutdown with the shutdown command. SIGTERM is sent to a process, and if the process has not terminated in thirty seconds, SIGKILL is sent to the process.

If a daemon wants the opportunity to clean up when it is to be terminated, it should install a signal handler for the SIGTERM signal. This applies to daemons regardless of how they were started: with init, SRC, inetd, or a shell.

Chapter 5. Managing a Daemon with the SRC

On AIX, many daemons that would traditionally be managed by the `init` process are managed instead by the System Resource Controller (SRC). As discussed in Chapter 4, "Managing a Daemon with the `init` Process" on page 45, the SRC has a daemon, `srcmstr`. The `srcmstr` daemon is controlled by the `init` process. Daemons under SRC control are started and managed by the `srcmstr` daemon. Commands are provided that allow the system administrator to control the execution of daemons through the SRC.

The SRC can be used to control two types of daemons, subsystems and subservers. A *subsystem* is a daemon that is directly controlled by the SRC. When an SRC command targets a subsystem, the request is processed by `srcmstr` or the subsystem itself. A *subserver* is a daemon that is directly controlled by a subsystem, and is defined to the SRC. When an SRC command targets a subserver, the request is processed by the subsystem that controls the subserver. Any SRC subsystem can have subservers, but the only subsystem that is known to exist that supports subservers is the `inetd` subsystem. This chapter only describes the interaction between the SRC and subsystems. How the SRC interacts with `inetd` subservers is described in Chapter 6, "Managing a Daemon with the Internet Superserver" on page 129. This document does not discuss the topic of how a subsystem can be written to support subservers.

When a daemon is placed under SRC control as a subsystem, characteristics of the daemon are defined to the SRC. Part of the subsystem's definition indicates how `srcmstr` should communicate requests to the daemon. The choices available are through signals, through a System V message queue, or through a BSD socket. When signal communication is used, the fewest changes are needed in the daemon source code. However, signal communication allows the daemon to support only a subset of SRC functionality. System V message queue communication and BSD socket communication require more changes to the daemon source code, while allowing the daemon to support full SRC functionality. The daemon support routines are designed to minimize the changes needed for a daemon to take advantage of message queue and socket communications. They are not described here, but in Chapter 10, "Daemon Support Routines" on page 245.

The first five sections of this chapter introduce the SRC's capabilities with regard to subsystems. Section 5.1, "The SRC Commands" on page 50 is an introduction to the commands available to the system administrator to control SRC subsystems. Section 5.2, "Subsystem Definition" on page 62 discusses the commands that are used to define SRC subsystems. Section 5.3, "Subsystem User ID" on page 68 discusses the limitations as to the user under which an SRC subsystem may run. Section 5.4, "Subsystem Failure, Respawning, and Notification" on page 73 is an introduction to the support in the SRC for dealing with failed subsystems. Section 5.5, "Subsystem Groups" on page 83 discusses considerations to take into account when organizing SRC subsystems into groups.

The remaining sections of this chapter discuss programming and SRC subsystems. Section 5.6, "Programming for the SRC in a Daemon" on page 84 describes programming techniques that allow a daemon to be managed by the SRC as a subsystem. Section 5.7, "Programming an SRC Notification Method"

on page 126 describes programming issues pertaining to SRC notification methods, which can be invoked when an SRC subsystem fails.

5.1 The SRC Commands

This section provides an introduction to the SRC commands that pertain to subsystem execution. The SRC commands that pertain to subsystem definition are discussed in 5.2, “Subsystem Definition” on page 62.

5.1.1 The `lssrc` Command

The `lssrc` command can be used to list information about daemons under SRC control. It is the one SRC command that can be run by any user. Figure 25 shows the `-a` flag being used to display information about all the daemons on the system defined to be under SRC control. Each line of output represents one daemon. Four columns of information are presented. The first column gives the name of the subsystem. Subsystem is the SRC term for a daemon. The subsystem name can be, but does not have to be, the name of the program that implements the daemon. How the subsystem name is associated with the program that is executed will be discussed in 5.2, “Subsystem Definition” on page 62. When a particular daemon is referenced in an SRC command, the subsystem name is usually used, although the PID can be used.

Notice from the `lssrc` output that `inetd` is controlled by the SRC. The second column of output indicates the group to which each subsystem belongs. When a subsystem is defined, it can be placed into a group, although that is not required. In many SRC commands, all the subsystems in a group can be referred to by the group name. Notice that the `inetd` subsystem belongs to the `tcpip` group, along with the `gated`, `named`, `routed`, `rwhod`, `iptrace`, `timed`, and `snmpd` subsystems. The third column gives the PID of the subsystem, if it is running. The `inetd` subsystem is running in a process whose PID is 5496. The fourth column indicates the status of the subsystem. The common values are `active` and `inoperative`. An `active` subsystem is one which is running normally. An `inoperative` subsystem is one that is not running. There are other possible values that describe various intermediate states.

```
# lssrc -a
Subsystem         Group            PID     Status
 syslogd          ras              4960    active
 sendmail         mail             3430    active
 portmap          portmap          5234    active
 inetd            tcpip            5496    active
 qdaemon          spooler          4236    active
 writesrv         spooler          4754    active
 lpd              spooler                  inoperative
 gated            tcpip                    inoperative
 named            tcpip                    inoperative
 routed           tcpip                    inoperative
 rwhod            tcpip                    inoperative
 iptrace          tcpip                    inoperative
 timed            tcpip                    inoperative
 snmpd            tcpip                    inoperative
```

Figure 25. Output from `lssrc -a` *Command*

The lssrc command can be used to list information about a particular group of subsystems, or about one particular subsystem. Figure 26 on page 51 illustrates how the -g flag can be used to list information about a group of subsystems, and the -s flag can be used to list information about a particular subsystem.

```
# lssrc -g tcpip
Subsystem         Group            PID     Status
 inetd            tcpip            5496    active
 gated            tcpip                    inoperative
 named            tcpip                    inoperative
 routed           tcpip                    inoperative
 rwhod            tcpip                    inoperative
 iptrace          tcpip                    inoperative
 timed            tcpip                    inoperative
 snmpd            tcpip                    inoperative

# lssrc -s inetd
Subsystem         Group            PID     Status
 inetd            tcpip            5496    active

# lssrc -s qdaemon
Subsystem         Group            PID     Status
 qdaemon          spooler          4236    active
```

Figure 26. Output of lssrc -g *and* lssrc -s *Commands*

The information presented by lssrc in the examples presented so far is referred to by the SRC documentation as subsystem short status. This short status information is provided to the lssrc command by the srcmstr daemon. The subsystems themselves are not involved in responding to requests for short status.

An SRC subsystem has the option of providing long status in response to an lssrc command. Long status is requested with the -l flag. The information returned by a particular subsystem in response to a long status request is entirely up to the subsystem. Information that is meaningful to the system administrator and relevant to the subsystem is appropriate. Since use of the lssrc command is not restricted, information whose access should be restricted should not be included in long status output. A subsystem must be defined to use message queue or socket communication in order to return long status.

Figure 27 on page 52 shows the long status output provided by qdaemon and inetd. The output for qdaemon is just an error message indicating that long status is not available from qdaemon because it only supports signal communication from srcmstr. The message is actually being generated by srcmstr; the qdaemon subsystem was not involved. The inetd subsystem can provide long status, because it supports socket communication from srcmstr.

In Figure 27 on page 52 the inetd long status begins with the same information that is provided when short status is requested; the name of the subsystem, the group to which it belongs, its PID, and the state of the subsystem. This is followed by an indication as to whether the inetd daemon is running in debug mode or not; currently it is not. Next, the long status output documents how inetd handles certain signals. Finally, it provides information about the services

inetd provides; much of that information is also found in the inetd configuration file, /etc/inetd.conf. The inetd daemon writer decided what information to supply in response to a long status request. The inetd daemon writer had to include code in his daemon to provide this status.

```
# lssrc -l -s qdaemon
0513-005 The Subsystem, qdaemon, only supports signal communication.


# lssrc -l -s inetd
Subsystem         Group            PID     Status
 inetd            tcpip            5496    active

Debug         Inactive

Signal        Purpose
 SIGALRM      Establishes socket connections for failed services
 SIGHUP       Rereads configuration database and reconfigures services

 SIGCHLD      Restarts service in case the service dies abnormally

Service       Command                  Arguments                Status
 ttdbserverd  /usr/dt/bin/rpc.ttdbserverd rpc.ttdbserverd 100083   active
 time         internal                                           active
 daytime      internal                                           active
 chargen      internal                                           active
 discard      internal                                           active
 time         internal                                           active
 daytime      internal                                           active
 chargen      internal                                           active
 discard      internal                                           active
 echo         internal                                           active
 pcnfsd       /usr/sbin/rpc.pcnfsd     pcnfsd 150001 1-2         active
 rwalld       /usr/lib/netsvc/rwall/rpc.rwalld rwalld 100008 1          active
 rusersd      /usr/lib/netsvc/rusers/rpc.rusersd rusersd 100002 1-2       active

 rstatd       /usr/sbin/rpc.rstatd     rstatd 100001 1-3         active
 ntalk        /usr/sbin/talkd          talkd                     active
 exec         /usr/sbin/rexecd         rexecd                    active
 login        /usr/sbin/rlogind        rlogind                   active
 telnet       /usr/sbin/telnetd        telnetd                   active
 ftp          /usr/sbin/ftpd           ftpd                      active
```

Figure 27. Output of lssrc -l *Command*

There are other flags defined for the lssrc command. See the man page in *AIX Version 4.1: Command Reference*, SBOF-1851 for more information.

5.1.2 The stopsrc Command

The stopsrc command is the SRC command that is used by the system administrator to stop a daemon that is running under SRC control. SRC defines three types of stop requests: normal, forced, and cancel. The SRC defines the semantics of the three types of stop requests, and it is up to the SRC daemon, srcmstr, and daemons running under SRC control to support these semantics. A normal stop request asks the target daemon to not accept new requests for

work, complete all current processing, wait for all application activity to complete, release resources, and terminate. For a concurrent server, this would mean not accepting new client requests, continuing to process the requests of current clients, and terminating when all client requests have been satisfied. A forced stop request asks the target daemon to release resources and terminate quickly. The target daemon should not wait for current client processing to finish.

The cancel stop request is similar to the forced stop request with regard to the responsibilities of the target daemon. However, the cancel stop request includes a threat: after a predetermined amount of time, if the target daemon has not terminated voluntarily, the srcmstr daemon will send a SIGKILL signal to it. The SIGKILL signal will terminate the target daemon.

Figure 28 shows an example where the stopsrc command is used to terminate the inetd daemon. First, the lssrc command shows that the inetd subsystem is running. A ps command then shows that the PID reported by lssrc really represents a process running inetd. The stopsrc command is run, specifying the inetd subsystem. The lssrc and ps commands are then used to show that the inetd daemon is no longer running. Finally, a telnet request from another system fails, because the telnet daemon is a daemon that runs under inetd control.

```
[it1n13] # lssrc -s inetd
Subsystem         Group            PID     Status
 inetd            tcpip            5496    active

[it1n13] # ps -p 5496
   PID    TTY  TIME CMD
  5496      -  0:21 inetd

[it1n13] # stopsrc -s inetd
0513-044 The stop of the /usr/sbin/inetd Subsystem was completed successfully.

[it1n13] # ps -p 5496
   PID    TTY  TIME CMD

[it1n13] # lssrc -s inetd
Subsystem         Group            PID     Status
 inetd            tcpip                    inoperative




[it1n14] # telnet it1n13.aix.kingston.ibm.com
Trying...
telnet: connect: A remote host refused an attempted connect operation.
telnet> quit
[it1n14] #
```

Figure 28. Using stopsrc *to Stop a Subsystem*

The default stop request generated by the stopsrc command is for a normal stop. A forced stop can be specified by using the -f flag. A cancel stop can be specified by using the -c flag.

A group of subsystems can be stopped by specifying the -g flag. All the subsystems on the system can be stopped by specifying the -a flag.

See the stopsrc man page in *AIX Version 4.1: Command Reference*, SBOF-1851 for more details.

5.1.3 The startsrc Command

The startsrc command is the SRC command that is used to start a daemon under SRC control. The command can be specified in a shell script or in a configuration file that allows shell commands, like /etc/inittab. The command can also be used by a system administrator to start an SRC subsystem. Figure 29 shows an example of using the startsrc command to start the inetd daemon.

```
[it1n13] # startsrc -s inetd
0513-059 The inetd Subsystem has been started. Subsystem PID is 5584.

[it1n13] # lssrc -s inetd
Subsystem         Group            PID     Status
 inetd            tcpip            5584    active
```

Figure 29. Using startsrc *to Start a Subsystem*

An SRC subsystem definition specifies if the subsystem allows SRC to start multiple instances of the subsystem. Many subsystems do not allow multiple instances; for example, inetd does not. Figure 30 shows the error displayed when SRC is used to try to start multiple instances of a subsystem that does not allow it.

```
[it1n13] # startsrc -s inetd
0513-029 The inetd Subsystem is already active.
Multiple instances are not supported.
```

Figure 30. Attempting to Start Multiple Instances of a Subsystem

The startsrc command allows for the specification of arguments that are to be passed to the daemon that will be started. Arguments for the daemon are specified with the startsrc -a flag. The arguments themselves should be protected from interpretation by the shell with double quotes. Figure 31 on page 55 shows an example of passing an argument to a subsystem. The subsys01 subsystem is started, and the -p flag is passed to the subsystem. When a subsystem is defined to the SRC, the definition can contain arguments that are always passed to the subsystem. This will be discussed in 5.2, “Subsystem Definition” on page 62. Any arguments that should always be passed to the subsystem when it is started with SRC should be specified in the definition. Additional arguments can be specified with the startsrc command.

Environment variables can also be passed to the daemon through startsrc. Environment variables for the daemon are specified with the startsrc -e flag. The variables should be protected from interpretation by the shell using double quotes. Figure 31 on page 55 shows an example of passing environment variables to the daemon. The subsys01 subsystem is started, and values are

specified for two environment variables, TEST1 and TEST2. The subsys01 subsystem is written to return the values of all environment variables when long status is requested. The output shows that the TEST1 and TEST2 variables have been set up correctly.

```
[it1n13] # startsrc -s subsys01 -a "-p" -e "TEST1=value1 TEST2=value2"
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 9996.

[it1n13] # lssrc -ls subsys01
Subsystem         Group            PID     Status
 subsys01         daes             9996    active

TERM=dumb
AUTHSTATE=compat
SHELL=/bin/ksh
HOME=/
USER=root
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin
TZ=EST5EDT
LANG=C
LOCPATH=/usr/lib/nls/loc
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
LC__FASTMSG=true
ODMDIR=/etc/objrepos
LOGNAME=root
LOGIN=root
TEST1=value1
TEST2=value2
```

Figure 31. Passing Parameters and Environment Variables to a Subsystem

When an environment variable is to contain a value which includes white space, care should be taken. The environment variable, the equal sign, and the value should be enclosed in escaped double quotes. Including only the value in escaped double quotes does not work properly. See Figure 32 on page 56 for an example of correct operation.

```
[it1n13] # stopsrc -s subsys01
0513-044 The stop of the subsys01 Subsystem was completed successfully.

[it1n13] # startsrc -s subsys01 -e "\"TEST1=multi word value\" TEST2=value2"
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 10004.

[it1n13] # lssrc -ls subsys01
Subsystem         Group            PID     Status
 subsys01         daes             10004   active

TERM=dumb
AUTHSTATE=compat
SHELL=/bin/ksh
HOME=/
USER=root
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin
TZ=EST5EDT
LANG=C
LOCPATH=/usr/lib/nls/loc
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
LC__FASTMSG=true
ODMDIR=/etc/objrepos
LOGNAME=root
LOGIN=root
TEST1=multi word value
TEST2=value2
```

Figure 32. Passing a Multi-Word Environment Value to a Subsystem

See the startsrc man page in *AIX Version 4.1: Command Reference*, SBOF-1851 for detailed information about the startsrc command.

5.1.4 The refresh Command

The refresh command allows a system administrator to request that a subsystem refresh itself. The meaning of the refresh operation is subsystem-dependent. However, a common meaning of the operation is for the subsystem to re-read its configuration file. For example, when the refresh command is applied to the inetd subsystem, the inetd daemon re-reads its configuration file, /etc/inetd.conf. This is illustrated in Figure 33 on page 57. The first egrep command shows that the ftp service is defined in /etc/inetd.conf. Then, the vi editor is used to comment out the service definition. The second egrep command shows this was done in the vi edit session. However, the ftp command on the it1n14 system can communicate with the ftp subsystem on the it1n13 system. Next, the refresh command is applied to the inetd subsystem. This causes inetd to re-read /etc/inetd.conf. Now, the ftp command on the it1n14 system cannot communicate with the ftp subsystem on the it1n13 system.

```
[it1n13] # egrep "^ftp|^#ftp" /etc/inetd.conf
ftp     stream  tcp     nowait  root    /usr/sbin/ftpd         ftpd

[it1n13] # vi /etc/inetd.conf

[it1n13] # egrep "^ftp|^#ftp" /etc/inetd.conf
#ftp     stream  tcp     nowait  root    /usr/sbin/ftpd         ftpd

[it1n14] # ftp it1n13.aix.kingston.ibm.com
Connected to it1n13.aix.kingston.ibm.com.
220 it1n13.aix.kingston.ibm.com FTP server (Version 4.1 Sat Aug 27 17:18:21 CDT
1994) ready.
Name (it1n13.aix.kingston.ibm.com:agar):
331 Password required for agar.
Password:
230 User agar logged in.
ftp> ls -C
200 PORT command successful.
150 Opening data connection for /bin/ls.
dae             daes.xopen      logging.tar     sig_stdio       timings
dae.msgq        daes_scripts    parentid        test
dae.xopen       freeterm        sig_reentr      tests
daes            info            sig_reentr.tar  threads
daes.msgq       logging         sig_restart     threads.tar
226 Transfer complete.
ftp> quit
221 Goodbye.
[it1n14] #

[it1n13] # refresh -s inetd
0513-095 The request for subsystem refresh was completed successfully.

[it1n14] # ftp it1n13.aix.kingston.ibm.com
ftp: connect: A remote host refused an attempted connect operation.
ftp> quit
```

Figure 33. Example of the refresh *Command*

The refresh command only succeeds when it is applied to a subsystem supporting message queue or socket communication from srcmstr.

See the refresh man page in *AIX Version 4.1: Command Reference*, SBOF-1851 for detailed information about the refresh command.

5.1.5 The traceson and tracesoff Commands

The traceson command can be used to ask a subsystem to trace itself. Two types of tracing are supported by traceson: short tracing and long tracing. The tracesoff command can be used to ask a subsystem to stop tracing itself. The meaning of tracing is subsystem-dependent.

The inetd subsystem supports a debugging mode that can be turned on with traceson, and off with tracesoff. The output from a long status request to inetd indicates whether inetd is running in debug mode. See Figure 34. When inetd is running in debug mode, it logs debug entries into the system log. See the example in Figure 35 on page 59. Before placing inetd in debug mode, the syslogd configuration is changed so daemon log entries are placed in a file named /tmp/daemon.log. The file is created with the touch command[19]; the syslogd configuration file, /etc/syslog.conf, is updated; and the refresh command is applied to the syslogd daemon, causing it to re-read /etc/syslog.conf. Next, long status output shows that syslogd will now place daemon log entries in the desired file. Next, the traceson command is used to put inetd into debug mode. From another system, an ftp client connects to the ftp daemon on the system whose inetd daemon is in debug mode. Finally, the contents of /tmp/daemon.log are examined. The inetd daemon has logged debug entries into the file through syslogd.

```
# lssrc -ls inetd | grep "^Debug"
Debug         Inactive

# traceson -s inetd
0513-091 The request to turn on tracing was completed successfully.

# lssrc -ls inetd | grep "^Debug"
Debug         Active

# tracesoff -s inetd
0513-093 The request to turn off tracing was completed successfully.

# lssrc -ls inetd | grep "^Debug"
Debug         Inactive
```

Figure 34. Example of the traceson *and* tracesoff *Commands*

[19] Before starting to log to a file, the file must exist. The syslogd daemon will not create a log file.

```
[it1n13] # lssrc -ls syslogd
Subsystem         Group            PID     Status
 syslogd          ras              4960    active

[it1n13] # grep "^daemon" /etc/syslog.conf
[it1n13] # touch /tmp/daemon.log
[it1n13] # vi /etc/syslog.conf
[it1n13] # grep "^daemon" /etc/syslog.conf
daemon.debug    /tmp/daemon.log

[it1n13] # refresh -s syslogd
0513-095 The request for subsystem refresh was completed successfully.

[it1n13] # lssrc -ls syslogd
Subsystem         Group            PID     Status
 syslogd          ras              4960    active

Syslogd Config   daemon.debug   /tmp/daemon.log

[it1n13] # traceson -s inetd
0513-091 The request to turn on tracing was completed successfully.

[it1n13] # lssrc -ls inetd | grep "^Debug"
Debug         Active

[it1n14] # ftp it1n13.aix.kingston.ibm.com
Connected to it1n13.aix.kingston.ibm.com.
220 it1n13.aix.kingston.ibm.com FTP server (Version 4.1 Sat Aug 27 17:18:21 CDT
1994) ready.
Name (it1n13.aix.kingston.ibm.com:agar):
331 Password required for agar.
Password:
230 User agar logged in.
ftp> quit
221 Goodbye.

[it1n13] # cat /tmp/daemon.log
  .
  .
Apr 13 23:26:18 it1n13 inetd[5376]: REDO: ttdbserverd proto=tcp, wait=1,
user=root builtin=0 server=/usr/dt/bin/rpc.ttdbserverd
Apr 13 23:27:28 it1n13 inetd[5376]: someone wants ftp
Apr 13 23:27:28 it1n13 inetd[5376]: accept, ctrl 34
Apr 13 23:27:36 it1n13 inetd[5376]: 13376 reaped
```

Figure 35. Using traceson *to Turn on Daemon Debugging*

Many daemons that use TCP/IP sockets to communicate with clients allow for the generation of TCP trace records. Such a daemon generates TCP trace records when in debugging mode. To generate TCP trace records, a daemon turns on the SO_DEBUG socket option for its TCP/IP sockets. Such a daemon usually allows for an argument that, when specified, will cause the daemon to start in debugging mode. When such a daemon is running under the control of the SRC, the daemon should also support the traceson and tracesoff commands. When the traceson command is applied to the daemon, the daemon enters debugging mode, turning on SO_DEBUG for its TCP/IP sockets. When the tracesoff

command is applied to the daemon, the daemon leaves debugging mode, turning off SO_DEBUG for its TCP/IP sockets. The superuser can use the trpt command to query the TCP trace records.

Here is a partial list of AIX daemons that support the generation of TCP trace records when placed in debug mode by traceson: inetd, rwhod, routed, named, iptrace, lpd, sendmail, and syslogd. Many other daemons support this feature indirectly as a result of being under inetd control.

In Figure 36 on page 61, the subsys01 daemon is a concurrent server that reads ASCII text supplied by clients, edits the text, and returns the results to the clients. Communications between the server and the clients is through TCP/IP. In the figure, subsys01 is started by the SRC. The traceson command is then used to put the daemon in debugging mode; the subsys01 daemon turns on the SO_DEBUG socket option for the socket on which it waits for client connections. The trpt command is then run to follow TCP trace records. When a client connects with subsys01 (not shown), trpt starts displaying TCP trace records.

```
[it1n13] # startsrc -s subsys01
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 8620

[it1n13] # traceson -l -s subsys01
0513-091 The request to turn on tracing was completed successfully.

[it1n13] # trpt -f -s -a

  .
  .
  .

522 ESTABLISHED:user SEND -> ESTABLISHED
        rcv_nxt 1d05455 rcv_wnd 4000 snd_una 9549e95c snd_nxt 9549e990 snd_max 9
549e990
        snd_wl1 1d0544f snd_wl2 9549e95c snd_wnd 4000

522 ESTABLISHED:input (src=9.117.10.14,2070,
dst=129.40.93.113,25)1d05455@9549e9
90(win=4000)<ACK,FIN> -> CLOSE_WAIT
        rcv_nxt 1d05456 rcv_wnd 4000 snd_una 9549e990 snd_nxt 9549e990 snd_max 9
549e990
        snd_wl1 1d05455 snd_wl2 9549e990 snd_wnd 4000
522 CLOSE_WAIT:output (src=129.40.93.113,25,
dst=9.117.10.14,2070)9549e990@1d054
56(win=4000)<ACK> -> CLOSE_WAIT
        rcv_nxt 1d05456 rcv_wnd 4000 snd_una 9549e990 snd_nxt 9549e990 snd_max 9
549e990
        snd_wl1 1d05455 snd_wl2 9549e990 snd_wnd 4000
532 LAST_ACK:output (src=129.40.93.113,25,
dst=9.117.10.14,2070)9549e990@1d05456
(win=4000)<ACK,FIN> -> LAST_ACK
        rcv_nxt 1d05456 rcv_wnd 4000 snd_una 9549e990 snd_nxt 9549e991 snd_max 9
549e991
        snd_wl1 1d05455 snd_wl2 9549e990 snd_wnd 4000
532 CLOSE_WAIT:user DISCONNECT -> LAST_ACK
        rcv_nxt 1d05456 rcv_wnd 4000 snd_una 9549e990 snd_nxt 9549e991 snd_max 9
549e991
        snd_wl1 1d05455 snd_wl2 9549e990 snd_wnd 4000
532 LAST_ACK:user DETACH -> LAST_ACK
        rcv_nxt 1d05456 rcv_wnd 4000 snd_una 9549e990 snd_nxt 9549e991 snd_max 9
549e991
        snd_wl1 1d05455 snd_wl2 9549e990 snd_wnd 4000
```

Figure 36. Using traceson *in Conjunction with the* trpt *Command*

The traceson and tracesoff commands only succeed when applied to a subsystem that supports message queue or socket communication from srcmstr.

See the traceson and tracesoff man pages in *AIX Version 4.1: Command Reference*, SBOF-1851 for detailed information about these commands.

5.1.6 Targeting Another Host

The SRC commands support the -h flag, whose argument is used to specify the host name or IP address of another system to which the SRC request is to be made. Scalable POWERparallel Systems, RS/6000 Division, does not use this flag, for the following reasons:

- Requests and replies are sent in UDP/IP datagrams, which are not reliable.
- The mechanism requires entries in /etc/hosts.equiv, which is not considered very secure.

Remote requests can be made through the distributed shell instead.

5.2 Subsystem Definition

SRC subsystems are defined in an ODM object class named SRCsubsys. The SRC provides a command to define a subsystem, mkssys; a command to change the definition of a subsystem, chssys; and a command to remove the definition of a subsystem; rmssys. However, it does not provide a command to look at the definition of a subsystem. To look at the definition of a subsystem, the ODM command odmget is used.. Figure 37 shows the definition of the SRCsubsys object class, as revealed by the odmshow command.

```
# odmshow SRCsubsys
class SRCsubsys {
        char subsysname[30];                    /* offset: 0xc ( 12) */
        char synonym[30];                       /* offset: 0x2a ( 42) */
        char cmdargs[200];                      /* offset: 0x48 ( 72) */
        char path[200];                         /* offset: 0x110 ( 272) */
        long uid;                               /* offset: 0x1d8 ( 472) */
        long auditid;                           /* offset: 0x1dc ( 476) */
        char standin[200];                      /* offset: 0x1e0 ( 480) */
        char standout[200];                     /* offset: 0x2a8 ( 680) */
        char standerr[200];                     /* offset: 0x370 ( 880) */
        short action;                           /* offset: 0x438 ( 1080) */
        short multi;                            /* offset: 0x43a ( 1082) */
        short contact;                          /* offset: 0x43c ( 1084) */
        long svrkey;                            /* offset: 0x440 ( 1088) */
        long svrmtype;                          /* offset: 0x444 ( 1092) */
        short priority;                         /* offset: 0x448 ( 1096) */
        short signorm;                          /* offset: 0x44a ( 1098) */
        short sigforce;                         /* offset: 0x44c ( 1100) */
        short display;                          /* offset: 0x44e ( 1102) */
        short waittime;                         /* offset: 0x450 ( 1104) */
        char grpname[30];                       /* offset: 0x452 ( 1106) */
        };
/*
        columns:        20
        structsize:     0x470 (1136) bytes
        data offset:    0x514
        population:     24 objects (24 active, 0 deleted)
*/
```

Figure 37. The Definition of the SRCsubsys *ODM Object Class*

The fields have the following meanings:

- The subsysname field specifies the name of the subsystem. The name cannot exceed 30 bytes, including the null terminator. A value for this field is required. The value is specified with the -s flag of the mkssys command.
- The synonym field specifies a character string to be used as an alternate name for the subsystem. The character string cannot exceed 30 bytes, including the null terminator. A value for this field is not required. The value is specified with the -t flag of the mkssys command.
- The cmdargs field specifies any arguments that must be passed to the command that starts the subsystem. The arguments cannot exceed 200 bytes, including the null terminator. The arguments are parsed by the srcmstr daemon according to the same rules used by shells. For example, a quoted string is passed as a single argument, and blanks outside quoted strings delimit arguments. A value for this field is not required. The value is specified with the -a flag of the mkssys command.
- The path field specifies the full path name for the program executed by the subsystem start command. The path name cannot exceed 200 bytes, including the null terminator. A value for this field is required. The value is specified with the -p flag of the mkssys command.
- The uid field specifies the user ID (numeric) under which the subsystem is run. A value of 0 indicates the root user. A value for this field is required. The value is specified with the -u flag of the mkssys command.
- The auditid field specifies the subsystem audit ID. Created automatically by the srcmstr daemon when a subsystem is defined, this field is used by the security system, if configured. This field cannot be set or changed by a program.
- The standin field specifies the file or device associated with file descriptor 0 when the subsystem's process is started by srcmstr. The default is /dev/console. The value of this field cannot exceed 200 bytes, including the null terminator. This field is ignored if the communication type is sockets. A value for this field is specified with the -i flag of the mkssys command.
- The standout field specifies the file or device associated with file descriptor 1 when the subsystem's process is started by srcmstr. The default is /dev/console. The value of this field cannot exceed 200 bytes, including the null terminator. A value for this field is specified with the -o flag of the mkssys command.
- The standerr field specifies the file or device associated with file descriptor 2 when the subsystem's process is started by srcmstr. The default is /dev/console. The value of this field cannot exceed 200 bytes, including the null terminator. A value for this field is specified with the -e flag of the mkssys command.
- The action field specifies whether the srcmstr daemon should restart the subsystem after an abnormal termination. A value of 1 specifies the srcmstr daemon should restart the subsystem. A value of 2 specifies the srcmstr daemon should not attempt to restart the failed subsystem. There is a respawn limit of two restarts within the time specified by the waittime field. If the failed subsystem cannot be successfully restarted, a notification method may be executed (see 5.4, "Subsystem Failure, Respawning, and Notification" on page 73). The default value is 2; meaning no respawn is attempted. A value for this field is specified with the -R flag (meaning respawn) or the -0 flag (meaning execute once) of the mkssys command.

- The multi field specifies whether multiple instances of a subsystem are allowed to run at one time. A value of 0 specifies that only one instance of the subsystem is allowed to run at one time. Attempts to start this subsystem when it is already running will fail, as will attempts to start a subsystem on the same IPC message queue key. A value of 1 specifies that multiple subsystems are allowed to use the same IPC message queue and that multiple instances of the same subsystem are allowed. The default value is 0. A value for this field is specified with the -Q flag (meaning only one instance) or the -q flag (meaning multiple instances) of the mkssys command.
- The contact field specifies the communication method used by the srcmstr daemon to send requests to the subsystem. A value of 1 specifies System V message queue communication. A value of 2 specifies signal communication. A value of 3 specifies BSD socket communication. The default value is 3. A value for this field is specified with the -I flag (meaning message queues), the -S flag (meaning signals), or the -K flag (meaning sockets) of the mkssys command.
- The svrkey field specifies a decimal value that corresponds to the IPC message queue key that the srcmstr daemon uses to communicate to the subsystem. A value for this field is required for subsystems that specify message queue communication in the contact field. Use the ftok subroutine with a fully qualified path name and an ID parameter to ensure that this key is unique. The srcmstr daemon creates the message queue prior to starting the subsystem. A value for this field is specified with the -I flag of the mkssys command.
- The svrmtype field specifies the message type of the message that is placed on the subsystem's message queue. The subsystem uses this value to retrieve messages by using the msgrcv or msgxrcv subroutine. A value for this field is required if the subsystem specifies message queue communication in the contact field. A value for this field is specified with the -m flag of the mkssys command.
- The priority field specifies the nice value with which the subsystem is to run. The default value is 20. A value for this field is specified with the -E flag of the mkssys command[20].
- The signorm field specifies the signal to be sent to the subsystem when a stop normal request is to be made. A value for this field is required for subsystems that specify signal communication in the contact field. A value for this field is specified with the -n flag of the mkssys command.
- The sigforce field specifies the signal to be sent to the subsystem when a stop forced request is to be made. A value for this field is required for subsystems that specify signal communication in the contact field. A value for this field is specified with the -f flag of the mkssys command.
- The display field value indicates whether the status of an inoperative subsystem can be displayed in lssrc -a or lssrc -g output. A value of 1 allows the status to be displayed; a value of 0 does not allow the status to be displayed. The default value is 1. A value for this field is specified with the -d flag (meaning display) or the -D flag (meaning do not display) of the mkssys command.

[20] When the priority field of a subsystem is set to 0, the subsystem is run with the default nice value, 20. This has been reported as a defect.

- The waittime field specifies the time in seconds that a subsystem has to complete a stop cancel request before being sent the SIGKILL signal. The value also has an effect on attempts to respawn the subsystem; see the description of the action field. The default value is 20 seconds. A value for this field can be specified with the -w flag of the mkssys command.
- The grpname field designates the subsystem as a member of a group of subsystems. The value of this field cannot exceed 30 bytes, including the null terminator. If the value of this field is the null string, the subsystem does not belong to a group. A value for this field can be specified with the -G flag of the mkssys command

When a subsystem is defined to accept signal communication from srcmstr, the requests that can be sent to the subsystem are:

- Stop Normal. This request is made of the subsystem by sending a signal, whose number is provided in the signorm field of the subsystem's definition, to the subsystem's process. This request can be made as a result of executing the stopsrc command without specifying the -f flag or the -c flag.
- Stop Forced. This request is made of the subsystem by sending a signal, whose number is provided in the sigforce field of the subsystem's definition, to the subsystem's process. This request can be made as a result of executing the stopsrc command, specifying the -f flag.
- Stop Cancel. This request is made of the subsystem by sending the SIGTERM signal to the subsystem's process. If the subsystem has not terminated in a certain number of seconds, specified in the waittime field of the subsystem's definition, the subsystem's process is sent the SIGKILL signal. This request can be made as a result of executing the stopsrc command, specifying the -c flag.

When a subsystem is defined to accept socket communication from srcmstr, most of the requests that are sent to the subsystem are sent through a UNIX domain socket. The srcmstr daemon creates this socket and associates it with file descriptor 0 of the subsystem process before calling an exec function to execute the subsystem's program. The subsystem's program must read requests from the socket it finds associated with file descriptor 0. The requests that can be sent from srcmstr to such a subsystem are:

- Stop Normal. This request is made of the subsystem by sending a packet to the subsystem's SRC request socket. This request can be made as a result of executing the stopsrc command without specifying the -f flag or the -c flag.
- Stop Forced. This request is made of the subsystem by sending a packet to the subsystem's SRC request socket. This request can be made as a result of executing the stopsrc command, specifying the -f flag.
- Stop Cancel. This request is made of the subsystem by sending the SIGTERM signal to the subsystem's process. If the subsystem has not terminated in a certain number of seconds, specified in the waittime field of the subsystem's definition, the subsystem's process is sent the SIGKILL signal. Notice that signals are used to send the stop cancel request, even when the subsystem is defined to accept requests through a socket. This request is made as a result of executing the stopsrc command, specifying the -c flag.
- Long Status. This request is made of the subsystem by sending a packet to the subsystem's SRC request socket. This request can be made as a result of executing the lssrc command, specifying the -l flag.

- Start Tracing. This request is made of the subsystem by sending a packet to the subsystem's SRC request socket. This request can be made as a result of executing the traceson command. If the -l flag is specified with the traceson command, the request is for long tracing; otherwise, the request is for short tracing.
- End Tracing. This request is made of the subsystem by sending a packet to the subsystem's SRC request socket. This request can be made as a result of executing the tracesoff command.
- Refresh. This request is made of the subsystem by sending a packet to the subsystem's SRC request socket. This request can be made as a result of executing the refresh command.

When a subsystem is defined to accept message queue communication from srcmstr, most of the requests that are sent to the subsystem are sent through a message queue. The srcmstr daemon creates this message queue using the key provided by the svrkey key field in the subsystem's definition. The subsystem's program must read requests from this queue. The requests that can be sent from srcmstr to such a subsystem are identical to the requests that can be sent to a subsystem supporting socket communication from srcmstr. The only difference between subsystems defined with message queue communication and those defined with socket communication is the mechanism through which the SRC requests are made. As for all other subsystems, srcmstr makes a stop cancel request using the SIGTERM and SIGKILL signals.

Figure 38 on page 67 shows the definition of the writesrv subsystem, and a mkssys command that could have created that definition. The odmget command is used to retrieve the definition from the SRCsubsys object class. The argument to the -q flag limits the output to the writesrv subsystem definition. The action field has a value of 1, indicating that the SRC will attempt to respawn writesrv if it terminates unexpectedly. The multi field has a value of 0, indicating the SRC will only allow one instance of writesrv to run at a time. The contact field has a value of 2, indicating srcmstr will send requests to writesrv using signals. The value of the signorm field is 30; stop normal requests will be made to writesrv by sending signal 30, SIGUSR1. The value of the sigforce field is 31; stop forced requests will be made to writesrv by sending signal 31, SIGUSR2. As with all SRC subsystems, stop cancel requests will be made to writesrv by sending the SIGTERM signal, followed by the SIGKILL signal, if needed. Since the value of the waittime field is 20, writesrv will have at least 20 seconds to terminate itself when receiving the SIGTERM signal.

```
# odmget -q "subsysname = 'writesrv'" SRCsubsys

SRCsubsys:
        subsysname = "writesrv"
        synonym = ""
        cmdargs = ""
        path = "/usr/sbin/writesrv"
        uid = 0
        auditid = 0
        standin = "/dev/console"
        standout = "/dev/console"
        standerr = "/dev/console"
        action = 1
        multi = 0
        contact = 2
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 30
        sigforce = 31
        display = 1
        waittime = 20
        grpname = "spooler"

# mkssys -s writesrv -p /usr/sbin/writesrv -u 0 -R -S -n 30 -f 31 -G spooler
0513-075 The new subsystem name is already on file.
```

Figure 38. The Definition of the writesrv *Subsystem*

Figure 39 on page 68 shows the definition of the inetd subsystem, and a mkssys command that could have created that definition. The action field has a value of 2, indicating the SRC will not attempt to respawn inetd if it terminates unexpectedly. The multi field has a value of 0, indicating the SRC will only allow one instance of inetd to run at a time. The contact field has a value of 3, indicating srcmstr will send requests to inetd using a UNIX domain socket. As with all SRC subsystems, stop cancel requests will be made to inetd by sending the SIGTERM signal, followed by the SIGKILL signal, if needed. Since the value of the waittime field is 20, inetd will have at least 20 seconds to terminate itself when receiving the SIGTERM signal.

```
# odmget -q "subsysname = 'inetd'" SRCsubsys

SRCsubsys:
        subsysname = "inetd"
        synonym = ""
        cmdargs = ""
        path = "/usr/sbin/inetd"
        uid = 0
        auditid = 0
        standin = "/dev/console"
        standout = "/dev/console"
        standerr = "/dev/console"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = "tcpip"

# mkssys -s inetd -p /usr/sbin/inetd -u 0 -G tcpip
0513-075 The new subsystem name is already on file.
```

Figure 39. The Definition of the inetd *Subsystem*

It is anticipated that most subsystems defined by Scalable POWERparallel Systems, RS/6000 Division, will be defined by executing the mkssys command in post-installation scripts. Some subsystems may be defined at other times. For example, the creation of an additional partition in an RS/6000 SP system may involve the definition of additional subsystems that are to run on a control workstation to support the new partition.

It is anticipated that usually subsystem definitions will be manipulated using the mkssys, chssys, and rmssys commands. However, the SRC also provides C language routines to manipulate subsystem definitions. The routines are addssys, chssys, defssys, delssys, and getssys; they are in the libsrc.a library.

5.3 Subsystem User ID

When a subsystem is defined, the user under which the subsystem is to run is specified. This user ID of the user is specified with the -u flag of the mkssys command, and stored in the uid field of the SRCsubsys object class. Often, the user ID that is specified is 0, which is the user ID for the superuser, root. It may be desirable for the subsystem to run under a user other than root, but there are some restrictions to what is possible.

If the subsystem is defined to receive requests from the SRC through signals, there are no limitations originating from the SRC as to the user under which the subsystem may successfully run.

If the subsystem is defined to receive requests from the SRC through a message queue or a socket, the SRC does create limitations on the users that may successfully run the subsystem. In these cases, the subsystem must be run by the superuser (the user with user ID 0) or by a user in the system group (the group with group ID 0). These limitations originate from the mechanisms used to reply to SRC requests and to send requests to the SRC daemon. When a subsystem replies to SRC requests delivered from a message queue or socket, it uses the srcsrpy routine to send the reply[21]. The srcsrpy routine creates a socket in a directory named /dev/.SRC-unix. A process must be running with an effective user ID of 0 or an effective group ID of 0 in order to be able to create the socket in this directory. Sometimes a subsystem may need to send a request to the SRC daemon. For example, it is common for a subsystem to get the short status output for itself from the SRC daemon. The SRC daemon verifies that the socket through which it is receiving a request was created by user ID 0 or group ID 0. If the socket was not created by user ID 0 or group ID 0, the request is rejected.

Figure 40 on page 70 illustrates what happens when a subsystem is run under various users. In Figure 40, part **1**, the odmget command shows the definition of the subsys01 subsystem. The odmget output indicates that the subsys01 subsystem is defined to run under the root user (the value for the uid field is 0), and the subsystem will receive requests from the SRC through a socket (the value for the contact field is 3). As defined, the subsystem can be successfully started, and the ps command output shows that the subsystem runs under the root user. The subsystem trace on request and subsystem stop normal request can be successfully sent to the subsystem, and the subsystem can reply to the requests.

[21] Use of the srcsrpy routine is discussed in 5.6, “Programming for the SRC in a Daemon” on page 84.

```
1
# odmget -q "subsysname = 'subsys01'" SRCsubsys

SRCsubsys:
        subsysname = "subsys01"
        synonym = ""
        cmdargs = "-l /tmp/subsys01.SRC"
        path = "/usr/lpp/somelpp/bin/subsys01"
        uid = 0
        auditid = 0
        standin = "/dev/null"
        standout = "/dev/null"
        standerr = "/dev/null"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = "daes"

# startsrc -s subsys01
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 6426.

# ps -p 6426 -l
       F S UID   PID  PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
  340401 A   0  6426  3946   0  60 20 a32a   168                -  0:00 subsys01

# traceson -s subsys01
Short tracing logged to /tmp/subsys01.SRC.6426.
0513-091 The request to turn on tracing was completed successfully.

# stopsrc -s subsys01
0513-044 The stop of the subsys01 Subsystem was completed successfully.
```

Figure 40 (Part 1 of 3). Running an SRC Subsystem under Various Users

```
2
# grep subsys01 /etc/passwd
subsys01:!:202:0::/:

# grep system /etc/group
system:!:0:subsys01,root

# chssys -s subsys01 -u 202
0513-077 Subsystem has been changed.

# odmget -q "subsysname = 'subsys01'" SRCsubsys

SRCsubsys:
        subsysname = "subsys01"
        synonym = ""
        cmdargs = "-l /tmp/subsys01.SRC"
        path = "/usr/lpp/somelpp/bin/subsys01"
        uid = 202
        auditid = 0
        standin = "/dev/null"
        standout = "/dev/null"
        standerr = "/dev/null"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = "daes"

# startsrc -s subsys01
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 6444.

# ps -p 6444 -l
      F S UID   PID  PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
  340401 A 202  6444  3946   0  60 20 2322   168                -  0:00 subsys01

# traceson -s subsys01
Short tracing logged to /tmp/subsys01.SRC.6444.
0513-091 The request to turn on tracing was completed successfully.

# stopsrc -s subsys01
0513-044 The stop of the subsys01 Subsystem was completed successfully.
```

Figure 40 (Part 2 of 3). Running an SRC Subsystem under Various Users

```
3
# grep agar /etc/passwd
agar:!:200:1:Eric M. Agar:/u/agar:/bin/ksh

# chssys -s subsys01 -u 200
0513-077 Subsystem has been changed.

# odmget -q "subsysname = 'subsys01'" SRCsubsys

SRCsubsys:
        subsysname = "subsys01"
        synonym = ""
        cmdargs = "-l /tmp/subsys01.SRC"
        path = "/usr/lpp/somelpp/bin/subsys01"
        uid = 200
        auditid = 0
        standin = "/dev/null"
        standout = "/dev/null"
        standerr = "/dev/null"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = "daes"

# startsrc -s subsys01
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 6462.

# ps -p 6462 -l
       F S UID   PID  PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
  340401 A 200  6462  3946   0  60 20 1c31c   168                -  0:00 subsys01

# traceson -s subsys01
0513-056 Timeout waiting for command response.

# stopsrc -s subsys01
0513-056 Timeout waiting for command response
.
# lssrc -s subsys01
Subsystem         Group            PID     Status
 subsys01         daes                     inoperative
```

Figure 40 (Part 3 of 3). Running an SRC Subsystem under Various Users

In Figure 40, part **2**, the subsys01 subsystem is run under a different user. The first grep command shows the definition of the subsys01 user in /etc/passwd. The subsys01 user has a user ID of 202, and belongs to a group with group ID of 0. The second grep command shows the definition of the system group in /etc/group. The system group has group ID 0. The chssys command is used to change the definition of the subsys01 subsystem so that it is run under the subsys01 user. The odmget command shows the change in the subsys01 definition

took effect; the value of the uid field is now 202. As defined, the subsystem can be successfully started, and the ps command output shows the subsystem runs under the subsys01 user. The subsystem trace on and subsystem stop normal requests can be successfully sent to the subsystem, and the subsystem can reply to the requests.

In Figure 40, part 3, the subsys01 subsystem is run under yet another user. The grep command shows the definition of the agar user in /etc/passwd. The agar user has a user ID of 200, and belongs to a group with group ID of 1. Recall from the definition of the system group shown in part 2 that the agar user does not belong to the system group. The chssys command is used to change the definition of the subsys01 subsystem so that it is run under the agar user. The odmget command shows that the change in the subsys01 definition took effect; the value of the uid field is now 200. As defined, the subsystem can be successfully started, and the ps command output shows the subsystem runs under the agar user. However, the subsystem trace on and subsystem stop normal requests do not appear to work. The traceson and stopsrc commands return with an error message, indicating that the commands did not receive replies to the requests. The subsys01 subsystem was able to receive these requests, but when it attempted to send a reply by calling srcsrpy, the necessary socket could not be created. The traceson and stopsrc commands timed out after waiting for a response. The lssrc command shows that the subsystem actually did receive the subsystem stop normal request, and it acted upon the request.

5.4 Subsystem Failure, Respawning, and Notification

Since an SRC-controlled daemon runs in a child process of srcmstr, the srcmstr process receives a SIGCHLD signal when the daemon terminates. When srcmstr receives notification of the termination of a subsystem, the SRC considers the termination abnormal if the srcmstr process had not sent a stop request to the subsystem.

When an abnormal termination is detected, the actions taken by srcmstr depend on the definition of the subsystem. If the subsystem is defined to not permit respawning, srcmstr logs an entry in the AIX error log, and determines if there is an SRC notification method that should be run. The subsystem is not run again until a startsrc command is executed specifying it. SRC notification methods will be discussed later.

If the subsystem is defined to permit respawning, and the subsystem is not respawning too quickly, srcmstr logs an entry in the AIX error log, and respawns the subsystem. If the subsystem is defined to permit respawning, and the subsystem is respawning too quickly, srcmstr logs an entry in the AIX error log, and determines if there is an SRC notification method that should be run. A subsystem is respawning too quickly if it has just been respawned twice within the time specified by waittime in the subsystem's definition.

5.4.1 Subsystem Failure

To illustrate the actions taken by srcmstr when a subsystem terminates abnormally, the shell script in Figure 41 is defined. The shell script, named sleepy_daemon, takes one parameter, the number of seconds to sleep. If no time is specified, sleepy_daemon will sleep for 5 seconds. The shell script prints a message, which includes the PID of the process running it and the number of seconds it will sleep, on its standard output. Then, the shell script puts the process to sleep. Once the process wakes from the sleep, it exits. It uses the number of seconds it slept as its exit value. Obviously, this shell script is not designed to be a useful daemon, but it will illustrate what srcmstr does in the face of abnormal subsystem termination.

```
# cat /u/agar/tests/sleepy_daemon
#!/bin/ksh

if [[ $# -ge 1 ]]; then
        sleep_time=$1
else
        sleep_time=5
fi

print "\nsleepy_daemon: process $$ about to sleep for ${sleep_time} seconds."

sleep ${sleep_time}

print "sleepy_daemon: goodbye cruel world."

exit ${sleep_time}
```

Figure 41. The Source for the sleepy_daemon *Test Script*

In Figure 42 on page 75, sleepy_daemon is defined to the SRC as a subsystem that will accept signal communication. The stop normal and stop forced signals are defined to be SIGTERM. This allows sleepy_daemon to not install signal handlers, because the default disposition for SIGTERM is process termination. Notice the value of the action field is 2, indicating srcmstr will not restart sleepy_daemon if it terminates abnormally. Also, notice that when sleepy_daemon is run by the SRC, its standard output will go to the console, /dev/console. After the subsystem is defined, the swcons command is used to redirect console output to the terminal on which the commands are being entered. This will allow us to observe the output of sleepy_daemon. Next, the startsrc command is used to start sleepy_daemon. The subsystem announces it will sleep for 5 seconds, does so, announces it will terminate, and presumably does that also. The lssrc command shows the subsystem is inoperative. If sleepy_daemon were respawned, we would expect further output, but none occurs.

```
# mkssys -s sleepy_daemon -p /u/agar/tests/sleepy_daemon -u 0 -S -n 15 -f 15
0513-071 The sleepy_daemon Subsystem has been added.

# odmget -q "subsysname = 'sleepy_daemon'" SRCsubsys

SRCsubsys:
        subsysname = "sleepy_daemon"
        synonym = ""
        cmdargs = ""
        path = "/u/agar/tests/sleepy_daemon"
        uid = 0
        auditid = 0
        standin = "/dev/console"
        standout = "/dev/console"
        standerr = "/dev/console"
        action = 2
        multi = 0
        contact = 2
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 15
        sigforce = 15
        display = 1
        waittime = 20
        grpname = ""

# swcons $(tty)

# startsrc -s sleepy_daemon
0513-059 The sleepy_daemon Subsystem has been started. Subsystem PID is 9064.
#
sleepy_daemon: process 9064 about to sleep for 5 seconds.
sleepy_daemon: goodbye cruel world.



# lssrc -s sleepy_daemon
Subsystem         Group            PID     Status
 sleepy_daemon                             inoperative
#
```

Figure 42. Running sleepy_daemon *without Respawning*

It was previously mentioned that when a subsystem that is not respawnable terminates abnormally, srcmstr will place an entry in the AIX error log. Figure 43 on page 76 shows the latest entry in the AIX error log with the SRC label[22]. Look at the Detail Data, at the end of the entry. The SOFTWARE ERROR CODE detail data provides an SRC error value. The possible values are defined in /usr/include/srcerrno.h. The value -9017 means a subsystem terminated abnormally. The ERROR CODE detail data provides the exit status of the subsystem process interpreted as a signed decimal number. The value 1280 indicates that

[22] All AIX error log entries may be viewed with the command: errpt -a | more.

the subsystem exited with 5 as an exit value[23]. The FAILING MODULE detail data provides the name of the subsystem that terminated abnormally; in this case it was sleepy_daemon.

```
---------------------------------------------------------------------------
LABEL:          SRC
IDENTIFIER:     E18E984F

Date/Time:       Fri Apr 14 20:31:49
Sequence Number: 249
Machine Id:      000175951000
Node Id:         it1n13
Class:           S
Type:            PERM
Resource Name:   SRC

Description
SOFTWARE PROGRAM ERROR

Probable Causes
APPLICATION PROGRAM

Failure Causes
SOFTWARE PROGRAM

        Recommended Actions
        PERFORM PROBLEM RECOVERY PROCEDURES

Detail Data
SYMPTOM CODE
           0
SOFTWARE ERROR CODE
       -9017
ERROR CODE
         1280
DETECTING MODULE
'srchevn.c'@line:'272'
FAILING MODULE
sleepy_daemon
---------------------------------------------------------------------------
```

Figure 43. AIX Error Log Entry for sleepy_daemon *(Not Respawnable)*

5.4.2 Subsystem Respawning

In Figure 44 on page 77, the SRC definition of sleepy_daemon is changed to cause srcmstr to attempt to respawn the daemon when abnormal termination is detected. This is done with the chssys command. Notice that the value of the action field has changed to 1, indicating respawning is desired. After the subsystem definition is changed, startsrc is used to start sleepy_daemon. The

[23] The exit status of a process has information encoded in it, such as an exit value or the number of the signal that terminated the process. The exit status can be interpreted with the macros defined in sys/wait.h. The dae_status program, described in B.3, "dae_status(1)" on page 392, interprets exit status values with these macros.

output indicates that the daemon was executed three times. This includes two respawnings. The lack of further output indicates that srcmstr did not continue to respawn the daemon.

```
# chssys -s sleepy_daemon -R
0513-077 Subsystem has been changed.

# odmget -q "subsysname = 'sleepy_daemon'" SRCsubsys

SRCsubsys:
        subsysname = "sleepy_daemon"
        synonym = ""
        cmdargs = ""
        path = "/u/agar/tests/sleepy_daemon"
        uid = 0
        auditid = 0
        standin = "/dev/console"
        standout = "/dev/console"
        standerr = "/dev/console"
        action = 1
        multi = 0
        contact = 2
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 15
        sigforce = 15
        display = 1
        waittime = 20
        grpname = ""

# startsrc -s sleepy_daemon
0513-059 The sleepy_daemon Subsystem has been started. Subsystem PID is 8984.
#
sleepy_daemon: process 8984 about to sleep for 5 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 8988 about to sleep for 5 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 8992 about to sleep for 5 seconds.
sleepy_daemon: goodbye cruel world.




# lssrc -s sleepy_daemon
Subsystem         Group            PID     Status
 sleepy_daemon                             inoperative
```

Figure 44. Running sleepy_daemon *with Respawning*

This attempt to run sleepy_daemon resulted in three entries being placed in the AIX error log, one for each abnormal termination of the subsystem. The last AIX error log entry for sleepy_daemon is shown in Figure 45, part **1**. The SOFTWARE ERROR CODE is -9020; the /usr/include/srcerrno.h file indicates this value means

the retry limit has been exceeded for the subsystem. The previous two AIX error log entries for sleepy_daemon are shown in Figure 45, part **2** and part **3**. In both these entries, the SOFTWARE ERROR CODE is -9035; according to /usr/include/srcerrno.h, this value means the subsystem terminated abnormally, and will be restarted.

1

```
---------------------------------------------------------------------------
LABEL:          SRC
IDENTIFIER:     E18E984F

Date/Time:       Fri Apr 14 20:39:15
Sequence Number: 252
Machine Id:      000175951000
Node Id:         it1n13
Class:           S
Type:            PERM
Resource Name:   SRC

Description
SOFTWARE PROGRAM ERROR

Probable Causes
APPLICATION PROGRAM

Failure Causes
SOFTWARE PROGRAM

        Recommended Actions
        PERFORM PROBLEM RECOVERY PROCEDURES

Detail Data
SYMPTOM CODE
           0
SOFTWARE ERROR CODE
       -9020
ERROR CODE
         1280
DETECTING MODULE
'srchevn.c'@line:'265'
FAILING MODULE
sleepy_daemon
---------------------------------------------------------------------------
```

Figure 45 (Part 1 of 3). AIX Error Log Entry for sleepy_daemon *(Respawnable)*

```
2
---------------------------------------------------------------------------
LABEL:          SRC
IDENTIFIER:     E18E984F

Date/Time:       Fri Apr 14 20:39:09
Sequence Number: 251
Machine Id:      000175951000
Node Id:         it1n13
Class:           S
Type:            PERM
Resource Name:   SRC

Description
SOFTWARE PROGRAM ERROR

Probable Causes
APPLICATION PROGRAM

Failure Causes
SOFTWARE PROGRAM

        Recommended Actions
        PERFORM PROBLEM RECOVERY PROCEDURES

Detail Data
SYMPTOM CODE
        1280
SOFTWARE ERROR CODE
       -9035
ERROR CODE
           0
DETECTING MODULE
'srchevn.c'@line:'166'
FAILING MODULE
sleepy_daemon
---------------------------------------------------------------------------
```

Figure 45 (Part 2 of 3). AIX Error Log Entry for sleepy_daemon *(Respawnable)*

```
3
---------------------------------------------------------------------------
LABEL:          SRC
IDENTIFIER:     E18E984F

Date/Time:       Fri Apr 14 20:39:04
Sequence Number: 250
Machine Id:      000175951000
Node Id:         it1n13
Class:           S
Type:            PERM
Resource Name:   SRC

Description
SOFTWARE PROGRAM ERROR

Probable Causes
APPLICATION PROGRAM

Failure Causes
SOFTWARE PROGRAM

        Recommended Actions
        PERFORM PROBLEM RECOVERY PROCEDURES

Detail Data
SYMPTOM CODE
        1280
SOFTWARE ERROR CODE
       -9035
ERROR CODE
           0
DETECTING MODULE
'srchevn.c'@line:'166'
FAILING MODULE
sleepy_daemon
---------------------------------------------------------------------------
```

Figure 45 (Part 3 of 3). AIX Error Log Entry for `sleepy_daemon` *(Respawnable)*

5.4.3 Subsystem Notification

When a non-respawnable subsystem has terminated abnormally, or a respawnable subsystem has terminated abnormally and reached its respawn limit, `srcmstr` determines if there is an SRC notification method that should be run. SRC notification methods are defined in the `SRCnotify` ODM object class. Figure 46 on page 81 shows the definition of the `SRCnotify` object class. When looking for a notification method to run, `srcmstr` first looks for an `SRCnotify` object whose `notifyname` field contains the name of the subsystem that has abnormally terminated. If such an entry is found, the program whose full path name is in the entry's `notifymethod` field is executed. If no object has a `notifyname` that matches the subsystem name, `srcmstr` looks for an `SRCnotify` object whose `notifyname` field contains the name of the group to which the failed subsystem belongs. If such an entry is found, the program whose full path name is in the entry's `notifymethod` field is executed. If no object has a `notifyname` that matches the group name, no notification method is executed. When a

notification method is executed, it is passed two parameters: the name of the failing subsystem, and the name of the group to which the subsystem belongs.

```
# odmshow SRCnotify
class SRCnotify {
        char notifyname[30];                            /* offset: 0xc ( 12) */
        method notifymethod[256];                       /* offset: 0x2a ( 42) */
        };
/*
        columns:        2
        structsize:     0x12c (300) bytes
        data offset:    0x1bc
        population:     0 objects (0 active, 0 deleted)
*/
```

Figure 46. The Definition of the SRCnotify *ODM Object Class*

Figure 47 on page 82 shows an example of an SRC notification method in use. The shell script sleepy_notify is installed as an SRC notification method for the sleepy_daemon subsystem using the mknotify command. The sleepy_notify script is simple: it uses wall to send a message to every user on the system. The sleepy_daemon subsystem is started with startsrc. The subsystem is respawned twice before srcmstr gives up. When srcmstr gives up, it finds a notification method defined for the sleepy_daemon subsystem, and runs the notification method. The example shows that the wall command in the notification method is executed. Finally, lssrc is run to verify that the subsystem is not running.

```
# cat /u/agar/tests/sleepy_notify
#!/bin/ksh

wall "Subsystem $1 terminated abnormally!"

exit 0

# mknotify -n sleepy_daemon -m /u/agar/tests/sleepy_notify
0513-068 The sleepy_daemon Notify method has been added.

# odmget -q "notifyname = 'sleepy_daemon'" SRCnotify

SRCnotify:
        notifyname = "sleepy_daemon"
        notifymethod = "/u/agar/tests/sleepy_notify"

# startsrc -s sleepy_daemon
0513-059 The sleepy_daemon Subsystem has been started. Subsystem PID is 9880.
#
sleepy_daemon: process 9880 about to sleep for 5 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 9884 about to sleep for 5 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 9888 about to sleep for 5 seconds.
sleepy_daemon: goodbye cruel world.




Broadcast message from UNKNOWN@it1n13.aix.kingston.ibm.com (tty) at 20:52:45
...


Subsystem sleepy_daemon terminated abnormally!

# lssrc -s sleepy_daemon
Subsystem         Group            PID     Status
 sleepy_daemon                             inoperative
```

Figure 47. Using an SRC Notification Method

Figure 48 on page 83 illustrates that a subsystem can be respawned often if the subsystem does not fail three times within the subsystem's `waittime` value. In the figure, `sleepy_daemon` is started with a parameter of 60, causing the subsystem to sleep 60 seconds. So, `sleepy_daemon` abnormally terminates about once every 60 seconds. Recall from Figure 44 on page 77, that the `waittime` value for `sleepy_daemon` is 20 seconds. The `srcmstr` daemon continues to respawn the `sleepy_daemon` subsystem, because the subsystem does not fail three times within 20 seconds. Notice that `lssrc` reports the subsystem as active after it has been respawned several times. Notice that the subsystem notification method is never run. The subsystem is continually respawned until it is stopped with a `stopsrc` command.

```
# startsrc -s sleepy_daemon -a "60"
0513-059 The sleepy_daemon Subsystem has been started. Subsystem PID is 6212.
#
sleepy_daemon: process 6212 about to sleep for 60 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 6216 about to sleep for 60 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 6220 about to sleep for 60 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 6224 about to sleep for 60 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 6228 about to sleep for 60 seconds.
sleepy_daemon: goodbye cruel world.

sleepy_daemon: process 6232 about to sleep for 60 seconds.

# lssrc -s sleepy_daemon
Subsystem         Group            PID     Status
 sleepy_daemon                     6232    active

# stopsrc -s sleepy_daemon
0513-044 The stop of the sleepy_daemon Subsystem was completed successfully.

# lssrc -s sleepy_daemon
Subsystem         Group            PID     Status
 sleepy_daemon                              inoperative
```

Figure 48. Running sleepy_daemon *with a Longer Delay*

5.5 Subsystem Groups

Subsystem groups have been mentioned several times in this chapter. When a subsystem is defined, it can be defined to belong to a group. The group to which a subsystem is to belong can be specified with the -G flag of the mkssys command. When other SRC commands are executed, the -g flag can be used to target all the subsystems in a specified group. All the subsystems in a group can be started, stopped, listed, refreshed, or traced. Finally, when a subsystem fails, if an SRC notification method for the subsystem is not defined, but an SRC notification method is defined for the group to which the subsystem belongs, the group notification method will be executed.

When an SRC subsystem is defined, thought should be given as to how the subsystem might conveniently be defined to be a member of a group. Does the subsystem have relationships to other subsystems that suggest the subsystems should be members of the same group? Would it be convenient for a system administrator to target SRC commands to a certain collection of subsystems all at once? Would defining one SRC notification method for a collection of subsystems make sense? The answers to these questions might suggest how subsystems are best formed into groups.

When subsystems are placed into a group, if there are any limitations as to how the subsystems can behave well as a group, they should be documented. For example, the AIX system defines the subsystems `inetd`, `gated`, `named`, `routed`, `rwhod`, `iptrace`, `timed`, and `snmpd` to all be members of a group named `tcpip`. It seems reasonable for these subsystems to be grouped together, but there is a problem. Both the `gated` and `routed` subsystems handle network routing. A system should not have both of these subsystems running at the same time. If both subsystems are running at the same time, the results are unpredictable. Therefore, the AIX documentation warns that the command `startsrc -g tcpip` should never be run.

When a subsystem is defined on the control workstation of an RS/6000 SP, system partitioning should be considered. If one instance of a daemon should run on a control workstation for each system partition on an RS/6000 SP, an SRC group should be defined for the daemon. Each subsystem in the group should execute the daemon for a particular RS/6000 SP partition. The `-g` flag can be used on the SRC commands to manage all the subsystems running the daemon.

5.6 Programming for the SRC in a Daemon

This section describes programming techniques that may be used in a daemon to support SRC control of the daemon. The techniques described here will be of interest to the daemon writer who has decided not to use the daemon support routines described in Chapter 10, “Daemon Support Routines” on page 245.

The daemon support routines use many of the methods described in this section. If you decide to use the daemon support routines when you write a daemon, you will not need to include the code discussed in this section in the daemon.

The topics discussed in this section are covered in Chapter 26 “System Resource Controller” of *AIX Version 4.1: General Programming Concepts: Writing and Debugging Programs*, SC23-2533. This section expands on the information given in *AIX Version 4.1: General Programming Concepts: Writing and Debugging Programs*, through programming examples.

The following sections are applicable to a subsystem defined to receive SRC requests through signals:

- 5.6.1, “Determining If the Daemon Was Started by the SRC” on page 86
- 5.6.2, “Installing Signal Handlers” on page 90
- 5.6.11, “Compiling and Linking an SRC Daemon” on page 126
- 5.6.12, “Multi-Threaded Daemons and the SRC” on page 126

The following sections are applicable to a subsystem defined to receive SRC requests through a message queue:

- 5.6.1, “Determining If the Daemon Was Started by the SRC” on page 86
- 5.6.2, “Installing Signal Handlers” on page 90
- 5.6.6, “Accessing the SRC Request Message Queue” on page 115
- 5.6.7, “Determining If an SRC Message Queue Request Is Pending” on page 115
- 5.6.8, “Processing an SRC Message Queue Request” on page 115
- 5.6.9, “Problems with SRC Message Queues” on page 121

- 5.6.10, “Processing Subsystem-Defined Requests” on page 125
- 5.6.11, “Compiling and Linking an SRC Daemon” on page 126
- 5.6.12, “Multi-Threaded Daemons and the SRC” on page 126

The following sections are applicable to a subsystem defined to receive SRC requests through a socket:

- 5.6.1, “Determining If the Daemon Was Started by the SRC” on page 86
- 5.6.2, “Installing Signal Handlers” on page 90
- 5.6.3, “Accessing the SRC Request Socket” on page 92
- 5.6.4, “Determining If an SRC Socket Request Is Pending” on page 92
- 5.6.5, “Processing an SRC Socket Request” on page 92
- 5.6.10, “Processing Subsystem-Defined Requests” on page 125
- 5.6.11, “Compiling and Linking an SRC Daemon” on page 126
- 5.6.12, “Multi-Threaded Daemons and the SRC” on page 126

5.6.1 Determining If the Daemon Was Started by the SRC

Section 3.1, “Determine How the Daemon Was Started” on page 19 discusses how the daemon support routines determine whether the daemon was started by the SRC. The AIX getprocs routine is used to obtain information from the process table about the parent process of the daemon process. This information is used to determine if SRC started the daemon or not. The major advantage of this technique is that it makes no assumptions about the method used by the SRC to send requests to the daemon. This is important for a generic set of routines like the daemon support routines.

When a daemon wants to determine if it has been started by the SRC, it is not uncommon for it to make assumptions about how it is defined to the SRC. Several examples show strategies for coding based on definition assumptions.

If a daemon is defined to receive SRC requests through signals, a common strategy is to define the SRC subsystem such that file descriptor 0 is associated with /dev/null. The daemon then determines if file descriptor 0 is associated with a character special file that is not a terminal. If it is, the daemon assumes it has been started by the SRC. If file descriptor 0 is not associated with a character special file, or is associated with a character special file that is a terminal, the daemon assumes it has been started by a shell, and it should migrate itself to another process. Figure 49 on page 87 shows this strategy[24].

[24] This example illustrates how a process can determine if it was started by the SRC. The example is not complete, in that it does not handle all the issues related to daemon initialization. Refer to Chapter 3, “Daemon Initialization” on page 19 for a complete discussion about daemon initialization.

```
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mode.h>

void create_child(void)
{
     pid_t pid;

     if ((pid = fork()) == (pid_t)-1) {
           exit(1);
     } else if (pid != 0) {
           exit(0);
     }

     return;
}

void start_daemon(void)
{
     struct sigaction sa;
     struct stat st;

     sa.sa_handler = SIG_IGN;
     sigemptyset(&sa.sa_mask);
     sa.sa_flags = 0;
     if (sigaction(SIGHUP, &sa, NULL) == -1) {
           exit(1);
     }

     if (fstat(0, &st) < 0) {
           create_child();
     } else {
           if ((st.st_mode & S_IFMT) == S_IFCHR) {
                 if (isatty(0)) {
                       create_child();
                 }
           } else {
                 create_child();
           }
     }

     (void) setsid();

     return;
}
```

Figure 49. Testing File Descriptor 0 for a Non-terminal Character Special File

When the init process starts a daemon directly, file descriptor 0 is often associated with /dev/null. That is not really a problem, because actions taken by a daemon under SRC control using signal communication are appropriate when the daemon is directly started by init.

If a daemon is defined to receive SRC requests through a socket, a common strategy is to use `getsockname` to determine if a socket is associated with file descriptor 0. If a socket is associated with file descriptor 0, the daemon assumes it has been started by the SRC. If file descriptor 0 is not associated with a socket, the daemon assumes it has been started by a shell, and it should migrate itself to another process. Figure 50 on page 89 shows this strategy[25].

[25] This example illustrates how a process can determine if it was started by the SRC. The example is not complete, in that it does not handle all the issues related to daemon initialization. Refer to Chapter 3, "Daemon Initialization" on page 19 for a complete discussion about daemon initialization.

```
#include <unistd.h>
#include <signal.h>
#include <sys/socket.h>
#include <sys/types.h>

#define SRC_FD  13

void create_child(void)
{
      pid_t pid;

      if ((pid = fork()) == (pid_t)-1) {
            exit(1);
      } else if (pid != 0) {
            exit(0);
      }

      return;
}

int start_daemon(void)
{
      int rval, addrsz;
      struct sockaddr srcaddr;
      struct sigaction sa;

      sa.sa_handler = SIG_IGN;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = 0;

      if (sigaction(SIGHUP, &sa, NULL) == -1) {
            exit(1);
      }

      addrsz = sizeof(srcaddr);
      if (getsockname(0, &srcaddr, &addrsz) != -1) {
            rval = dup2(0, SRC_FD);
      } else {
            create_child();
            rval = -1;
      }

      (void) setsid();

      return rval;
}
```

Figure 50. Testing File Descriptor 0 for a Socket

The strategy shown in Figure 50 will not allow the daemon to behave correctly if it is ever started directly by `init`. If started by `init`, file descriptor 0 will not be associated with a socket, the daemon will migrate itself to another process, and `init` will think the daemon has terminated when the parent process exits.

If a daemon using the strategy shown in Figure 50 is started by `inetd`, the daemon will probably not function correctly. The `getsockname` call will succeed,

and the daemon will assume it has been started by SRC, not inetd. The actions the daemon takes under these false assumptions are not likely to be correct.

If a daemon is defined to receive SRC requests through a message queue, it might be tempting to judge whether the daemon was actually started by the SRC based on the results of calling msgget to access the daemon's SRC request message queue. If the message queue could be accessed, the daemon must have been started by the SRC; otherwise, it was not started by the SRC. This strategy would be even less reliable than those discussed for signal and socket communications above. When a subsystem defined to used message queue communication terminates, the SRC does not clean up the daemon's message queue until the subsystem is started again with the SRC. If the daemon was started without the SRC, the daemon's message queue might still exist.

5.6.2 Installing Signal Handlers

If a daemon is defined to receive SRC requests through signals, it should install signal handlers for the SRC stop requests. The example in Figure 51 on page 91 assumes the SRC subsystem definition specifies SIGUSR1 for the stop normal signal and SIGUSR2 for the stop forced signal. After the signal handlers are installed, normal_stop will handle subsystem stop normal requests, and quick_stop will handle subsystem stop forced and subsystem stop cancel requests. The quick_stop routine is written to terminate the daemon immediately. The normal_stop routine just sets a global variable to indicate that the daemon has received the subsystem stop normal request. Some other routine in the daemon will presumably periodically check the value of the global variable to see if the daemon has received a subsystem stop normal request. Both approaches are valid for signal handlers supporting the SRC stop requests.

```
#include <signal.h>

struct sigaction sa;
volatile sig_atomic_t stop_requested = 0;

void normal_stop(int signo)
{
      stop_requested = 1;
      return;
}

void quick_stop(int signo)
{
      /* Cleanup the daemon quickly ... */
      _exit(0);
}
      .
      .
      .
      (void) sigemptyset(&sa.sa_mask);
      (void) sigaddset(&sa.sa_mask, SIGUSR1);
      (void) sigaddset(&sa.sa_mask, SIGUSR2);
      sa.sa_flags = SA_RESTART;

      sa.sa_handler = normal_stop;
      if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            /* Log an error message, including errno. */
            exit(1);
      }

      sa.sa_handler = quick_stop;
      if (sigaction(SIGUSR2, &sa, NULL) == -1) {
            /* Log an error message, including errno. */
            exit(1);
      }

      sa.sa_handler = quick_stop;
      if (sigaction(SIGTERM, &sa, NULL) == -1) {
            /* Log an error message, including errno. */
            exit(1);
      }

      /* Continue, as the daemon ... Look at stop_requested periodically */
      .
      .
      .
```

Figure 51. Installing Signal Handlers to Handle SRC Stop Requests

A daemon defined to receive SRC requests through a message queue or socket should install a signal handler for the SIGTERM signal. The installed signal handler will handle the subsystem stop cancel request.

See Chapter 7, “Signal Handling” on page 141 for all sorts of warnings pertaining to signal handling.

5.6.3 Accessing the SRC Request Socket

A daemon defined as an SRC subsystem that receives SRC requests through a socket can access the socket through file descriptor 0. It is common for a daemon to verify that file descriptor 0 is associated with a socket, and to duplicate file descriptor 0 to another file descriptor. An example is shown in Figure 50 on page 89.

5.6.4 Determining If an SRC Socket Request Is Pending

A daemon defined as an SRC subsystem that receives SRC requests through a socket must regularly determine if an SRC request is pending on the SRC request socket. If SRC requests are not answered within 60 seconds, the requesting process usually terminates with an error. If the daemon spends most of its time waiting for requests from other sockets, consider using the `select` routine to wait for SRC requests.

5.6.5 Processing an SRC Socket Request

Figure 52 on page 93 presents an example of a daemon designed to run under SRC control with socket communication. The daemon will act as a concurrent server that communicates with its clients through sockets. The example illustrates how a daemon can receive SRC requests through a socket, and how it can handle those requests.

```
1
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <netdb.h>
#include <spc.h>

int stop_requested = 0;
int client_count = 0;
int acpt_sockd;
int SRC_sockd;

const char * const config_file = "/etc/des.ex.sock.conf";
char config_buf[80];
int  config_cnt = 0;
int  tracing_on = 0;

void quick_stop(int signo);
static int  prepare_socket(void);
static void do_daemon_loop(void);

int main(int argc, char **argv)
{
      struct sigaction sa;

      SRC_sockd = start_daemon();  /*  Call start_daemon from Figure 50. * /

      sa.sa_handler = quick_stop;
      sigemptyset(&(sa.sa_mask));
      sa.sa_flags = 0;
      if (sigaction(SIGTERM, &sa, NULL) == -1) {
            /* Log an error message, including errno. */
            exit(1);
      }

      if (prepare_socket() == 0)
            do_daemon_loop();

      return 0;
}
```

Figure 52 (Part 1 of 10). Processing SRC Requests from a Socket

```
2
static int prepare_socket(void)
{
      struct sockaddr_in saddr;
      struct servent *servent;
      int reuse_on = 1;

      if ((acpt_sockd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            /* Log message; probably include the errno value. */
            return 1;
      }

      if (setsockopt(acpt_sockd, SOL_SOCKET, SO_REUSEADDR,
                           (char *) &reuse_on, sizeof reuse_on) == -1) {
            /* Log message; probably include the errno value. */
            return 1;
      }

      if ((servent = getservbyname("des.ex.sock", "tcp")) == NULL) {
            /* Log message; probably include the errno value. */
            return 1;
      }

      memset(&saddr, 0, sizeof saddr);
      saddr.sin_family = AF_INET;
      saddr.sin_addr.s_addr = INADDR_ANY;
      saddr.sin_port   = servent->s_port;

      if (bind(acpt_sockd, (struct sockaddr *) &saddr, sizeof saddr) == -1) {
            /* Log message; probably include the errno value. */
            return 1;
      }

      if (listen(acpt_sockd, SOMAXCONN) == -1) {
            /* Log message; probably include the errno value. */
            return 1;
      }

      return 0;
}
```

Figure 52 (Part 2 of 10). Processing SRC Requests from a Socket

```
3
#define MAX(x, y)  ( ((x) > (y)) ? (x) : (y) )

static void create_daemon_child(void);
static void handle_SRC_req(void);
static int  refresh_config(void);

static void do_daemon_loop(void)
{
      int numfds;
      fd_set read_fds;


      (void) refresh_config();

      numfds = MAX(acpt_sockd, SRC_sockd) + 1;

      while (!stop_requested) {

            FD_ZERO(&read_fds);
            FD_SET(acpt_sockd, &read_fds);
            if (SRC_sockd >= 0) {
                  FD_SET(SRC_sockd, &read_fds);
            }

            if (select(numfds, &read_fds, NULL, NULL, NULL) == -1) {
                  if (errno == EINTR) {
                        continue;
                  }
                  /* Log message */
                  break;
            }

            if (FD_ISSET(acpt_sockd, &read_fds)) {
                  create_daemon_child();
                  client_count++;
            }

            if ((SRC_sockd >= 0) && (FD_ISSET(SRC_sockd, &read_fds))) {
                  handle_SRC_req();
            }

      }

      return;
}
```

Figure 52 (Part 3 of 10). Processing SRC Requests from a Socket

```
4
static int refresh_config(void)
{
      int config_fd;
      int read_cnt;


      config_cnt = 0;

      if ((config_fd = open(config_file, O_RDONLY)) == -1) {
            /* Log message; probably include the errno value. */
            return 1;
      }

      if ((read_cnt = read(config_fd, config_buf, sizeof config_buf)) == -1)  {
            /* Log message; probably include the errno value. */
            (void) close(config_fd);
            return 1;
      }

      (void) close(config_fd);

      config_cnt = read_cnt;

      return 0;
}
```

Figure 52 (Part 4 of 10). Processing SRC Requests from a Socket

```
5
void do_daemon_work(int serv_sockd);

static void create_daemon_child(void)
{
      int serv_sockd;
      struct sockaddr_in sock_addr;
      int sock_addr_len;
      pid_t childpid;


      do {
            sock_addr_len = sizeof sock_addr;
            serv_sockd = accept(acpt_sockd,
                  (struct sockaddr *) &sock_addr, &sock_addr_len);

      } while ((serv_sockd == -1) && (errno == EINTR));

      if ((childpid = fork()) == -1) {
            /* Log message; probably include the errno value. */
            close(serv_sockd);
            return;
      }

      if (childpid != 0) {             /* This is the parent process. */
            close(serv_sockd);
            return;
      }

      /* This is the child process. */

      close(acpt_sockd);

      do_daemon_work(serv_sockd);

      close(serv_sockd);
      exit(0);

      return;                          /* Should not reach. */
}

void do_daemon_work(int serv_sockd)
{
      /* Do whatever the daemon is supposed to do, right here. */

      return;
}
```

Figure 52 (Part 5 of 10). Processing SRC Requests from a Socket

```
6
static void SRC_stop(struct srchdr *srchdr, struct subreq *subreq);
static void SRC_trace(struct srchdr *srchdr, struct subreq *subreq);
static void SRC_status(struct srchdr *srchdr, struct subreq *subreq);
static void SRC_refresh(struct srchdr *srchdr, struct subreq *subreq);

static void handle_SRC_req(void)
{
      struct srcreq srcreq;    struct srcrep srcrep;
      struct srchdr *srchdr;   struct subreq *subreq;
      int reqlen;
      struct sockaddr_un sockaddr;
      int sockaddr_len;

      sockaddr_len = sizeof sockaddr;
      reqlen = recvfrom(SRC_sockd, &srcreq, sizeof srcreq , 0,
                     (struct sockaddr *) &sockaddr, &sockaddr_len);

      if (reqlen != sizeof srcreq)
            return;

      srchdr = srcrrqs(&srcreq);
      subreq = &srcreq.subreq;

      if (subreq->action == STOP) {
            SRC_stop(srchdr, subreq);
      } else if (subreq->action == TRACE) {
            SRC_trace(srchdr, subreq);
      } else if (subreq->action == STATUS) {
            SRC_status(srchdr, subreq);
      } else if (subreq->action == REFRESH) {
            SRC_refresh(srchdr, subreq);
      } else if (subreq->action == START) {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
      } else {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_SUBICMD;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
      }

      return;
}
```

Figure 52 (Part 6 of 10). Processing SRC Requests from a Socket

```
7
static void SRC_stop(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;
      struct sigaction sa;


      memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
      srcrep.svrreply.objtype = subreq->object;
      strcpy(srcrep.svrreply.objname, subreq->objname);

      if (subreq->object != SUBSYSTEM) {
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      srcrep.svrreply.rtncode = SRC_OK;
      (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);

      if (subreq->parm1 == NORMAL) {
            stop_requested = 1;
      } else {                      /* Assume FORCED */
            sa.sa_handler = SIG_IGN;
            sigemptyset(&(sa.sa_mask));
            sa.sa_flags = 0;
            (void) sigaction(SIGTERM, &sa, NULL);
            (void) kill(0, SIGTERM);
            exit(0);
      }

      return;
}


static void quick_stop(int signo)
{
      if (getsid((pid_t)0) == getpid()) {
            (void) kill(0, SIGTERM);
      }

      _exit(0);
}
```

Figure 52 (Part 7 of 10). Processing SRC Requests from a Socket

```
8
static void SRC_trace(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;
      int debug_on_off;
      const char * const error_msg =
                        "Subsystem des.ex.sock: setsockopt() failed.\n";


      memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
      srcrep.svrreply.objtype = subreq->object;
      strcpy(srcrep.svrreply.objname, subreq->objname);

      if (subreq->object != SUBSYSTEM) {
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      debug_on_off = (subreq->parm2 == TRACEON);

      if (setsockopt(acpt_sockd, SOL_SOCKET, SO_DEBUG,
                     (char *) &debug_on_off, sizeof debug_on_off) == -1) {
            /* Log message; probably include the errno value. */
            srcrep.svrreply.rtncode = SRC_SUBMSG;
            strcpy(srcrep.svrreply.rtnmsg, error_msg);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      tracing_on = (subreq->parm2 == TRACEON);

      srcrep.svrreply.rtncode = SRC_OK;
      (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);

      return;
}
```

Figure 52 (Part 8 of 10). Processing SRC Requests from a Socket

```
9
static void SRC_refresh(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;
      const char * const er_msg = "Subsystem des.ex.sock failed to read %s.\n";
      const char * const ok_msg = "Subsystem des.ex.sock sucessfully read %s.";


      memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
      srcrep.svrreply.objtype = subreq->object;
      strcpy(srcrep.svrreply.objname, subreq->objname);

      if (subreq->object != SUBSYSTEM) {
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      if (refresh_config() != 0) {
            srcrep.svrreply.rtncode = SRC_SUBMSG;
            sprintf(srcrep.svrreply.rtnmsg, er_msg, config_file);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      sprintf(srcrep.svrreply.rtnmsg, ok_msg, config_file);
      (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, CONTINUED);

      srcrep.svrreply.rtncode = SRC_OK;
      (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);

      return;
}
```

Figure 52 (Part 9 of 10). Processing SRC Requests from a Socket

```
10
static void SRC_status(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;
      struct statrep statrep;
      struct statcode *sp;


      if (subreq->object != SUBSYSTEM) {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      sp = &statrep.statcode[0];

      sp->objtype = 0;
      sp->status  = 0;
      strcpy(sp->objname, "");
      sprintf(sp->objtext, "I've seen %d client connections.", client_count);
      (void) srcsrpy(srchdr, &statrep, sizeof statrep, STATCONTINUED);

      sp->objtype = 0;
      sp->status  = 0;
      strcpy(sp->objname, "");
      strcpy(sp->objtext, tracing_on ? "Tracing is on." : "Tracing is off.");
      (void) srcsrpy(srchdr, &statrep, sizeof statrep, STATCONTINUED);

      (void) srcsrpy(srchdr, &statrep, sizeof(struct srchdr), END);

      return;
}
```

Figure 52 (Part 10 of 10). Processing SRC Requests from a Socket

Figure 52, part **1** shows the `main` routine. The `main` routine uses the code in Figure 50 to determine if the daemon has been started by the SRC, or not. It then installs a signal handler for the `SIGTERM` signal. This signal handler will be invoked whenever the subsystem is targeted with the SRC subsystem stop cancel request. Once the signal handler is installed, the main routine calls `prepare_socket` and `do_daemon_loop`.

In Figure 52, part **2**, the `prepare_socket` routine initializes the socket through which the daemon will establish connections with clients. This socket should not be confused with the socket through which the SRC will make requests of the daemon. The socket through which SRC will make requests of the daemon is a UNIX domain datagram socket that is initialized by `srcmstr` and associated with file descriptor 0 of the daemon process just before the daemon program is executed. This socket was duplicated to another file descriptor by `main`. The socket through which the daemon will accept connection requests from clients is an Internet domain stream socket that is initialized by `prepare_socket`. Throughout Figure 52, the file descriptor of the SRC request socket is referenced

by the variable SRC_sockd, and the file descriptor of the socket though which client connection requests are accepted is referenced by the variable acpt_sockd.

The prepare_socket routine sets up the Internet domain stream socket in a standard fashion. It calls socket, bind, and listen. Notice that before calling bind, it calls setsockopt to set the SO_REUSEADDR socket option. This allows the daemon to be successfully restarted after being stopped by the SRC subsystem stop normal, stop forced, and stop cancel requests. If the SO_REUSEADDR socket option is not set, attempts to start the daemon after it has been stopped might result in the bind routine returning the "address already in use" error.

Figure 52, part **3** shows the do_daemon_loop routine. This routine waits for a request to arrive, handles the request, and then waits for another request to arrive. The routine uses select with no time out value. The select call waits for client connection requests through the socket whose file descriptor is in acpt_sockd, and also for SRC requests through the socket whose file descriptor is SRC_sockd. If the select routine indicates that a connection can be established with a client through the daemon connection socket, create_daemon_child is called to establish the connection and to create a child process to serve the client. If the select routine indicates that data can be read from the SRC request socket, handle_SRC_req is called. The handle_SRC_req routine will process the SRC request and return.

Figure 52, part **5** shows the create_daemon_child routine. This routine calls accept to establish a connection with a client, and creates a child process that will serve the client. The child process calls the do_daemon_work routine, which presumably serves the client as intended by the daemon. The do_daemon_work code is not shown, as its content depends on the service being provided by the daemon.

Figure 52, part **6** shows the handle_SRC_req routine, whose function is to process the next request on the SRC request socket. Before considering how SRC requests are handled in the code, it may be helpful to refer to the diagram in Figure 53. The diagram shows the flow of data involved in making SRC subsystem status requests. The flow is similar for the other SRC requests.

When the system administrator executes the lssrc command, the process running lssrc sends a request, through a socket, to the SRC daemon srcmstr. Some SRC requests can be handled directly by srcmstr, while other requests must be forwarded on to the subsystem targeted by the request. The srcmstr daemon can handle the subsystem short status request. So, if the administrator does not specify the -l flag when executing lssrc, srcmstr sends the reply back to the process running lssrc. The reply is sent through a socket. The subsystem is not involved in handling the request at all. The srcmstr daemon cannot handle the subsystem long status request directly, because the reply is up to the individual subsystem. In this case, the srcmstr daemon forwards the request to the subsystem targeted by the request. The srcmstr daemon is not interested in the reply generated by the subsystem. So, the request sent to the subsystem includes an address indicating where the reply should be sent. The address specified corresponds to a socket on which lssrc is waiting for the reply. The subsystem determines its status, and sends the reply data to the process running the lssrc command. All data is sent through sockets, with the possible exception of the data sent from the srcmstr daemon to the subsystem. The mechanism used depends on the definition of the subsystem, whether socket or message queue communication is specified for the subsystem.

The format of SRC requests sent to subsystems is defined by the

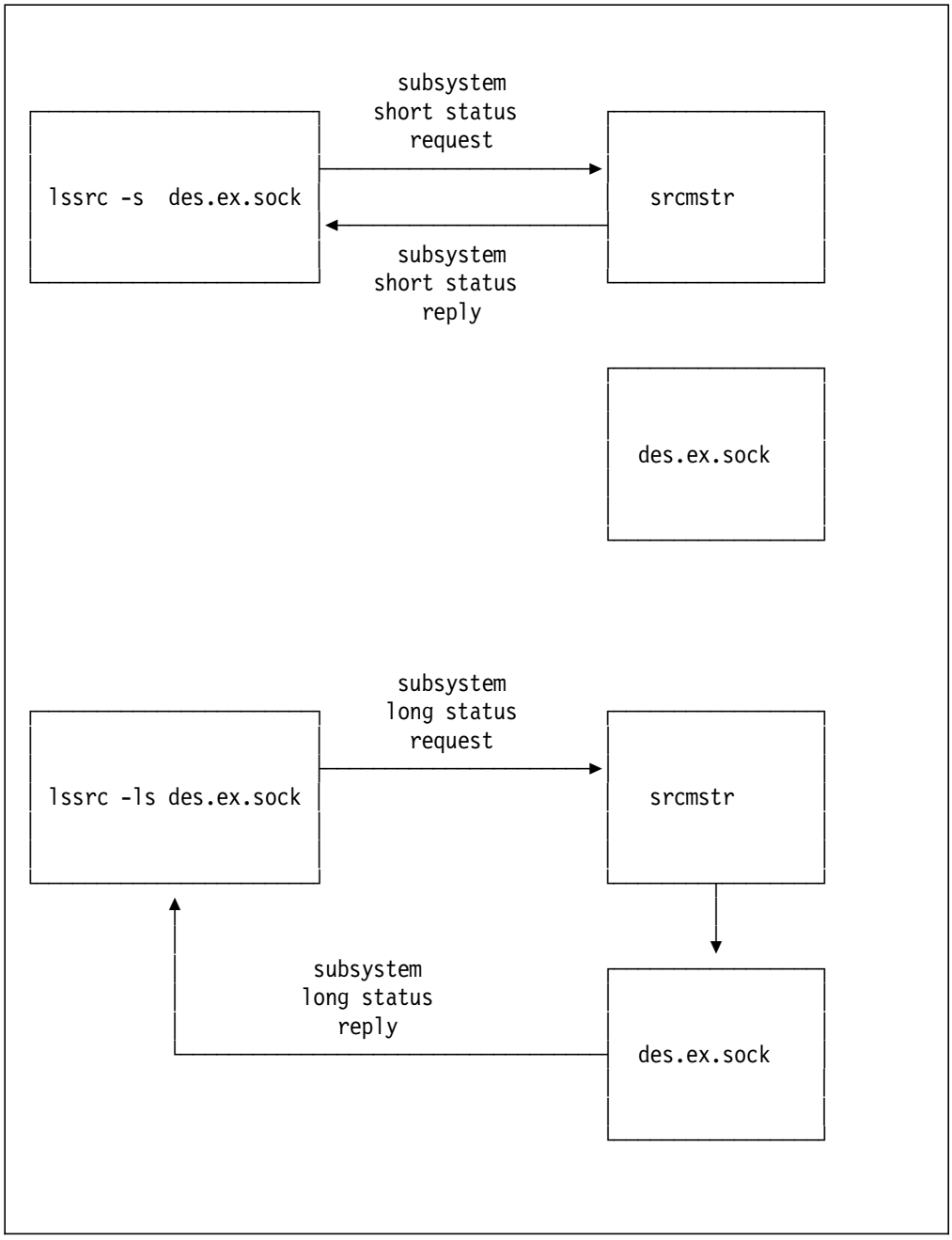


Figure 53. Flow of Data for Subsystem Status Requests

srcreq structure. The structure is defined in spc.h and shown in Figure 54 on page 105. The structure is composed of two other structures, an srchdr structure and an subreq structure[26]. The srchdr structure contains the address of the process to receive the request reply. The subreq structure defines the request being made of the subsystem, and is composed of fields named object, action, parm1, parm2, and objname. The object field indicates whether the request is targeted to the subsystem or to a subserver of the subsystem. If the object

[26] The mtype field is only included in the srcreq structure when SRCBYQUEUE is defined. The SRCBYQUEUE constant should only be defined when the subsystem accepts SRC requests through a message queue.

field has the value of the constant SUBSYSTEM[27], the request is targeted at the subsystem. Any other value identifies the subserver targeted by the request. The action field is set to a value indicating what request is being made. The standard values are represented by the constants START, STOP, TRACE, STATUS, and REFRESH. The parm1 and parm2 fields are request modifiers; their meanings depend on the request being made. Finally, the objname field contains the name of the object targeted by the request.

```
/* src header to be included in all packets that are sent to and
** received from any subsystem
**/
struct srchdr
{
        struct sockaddr_un retaddr;     /* return address for packets */
        short dversion;                 /* SRC packet version */
        short cont;                     /* continuation indicator */
};

/* subsystem request */
struct subreq
{
        short object;           /* object type  one of
                                **    1. SUBSYSTEM
                                **    2. subserver code point as defined
                                **       in subserver object class
                                **/
        short action;           /* action requested of subsystem */
        short parm1;            /* first modifier for subsystem action */
        short parm2;            /* second modifier for subsystem action */
        char objname[SRCNAMESZ];
                                /* name of object that request applies to
                                **      1. subsystem name
                                **      2. subserver object
                                **      3. subserver pid
                                **/

};

/* Subsystem request receipt */
struct srcreq
{
#ifdef SRCBYQUEUE
        long mtype;             /* mtype for message queue subsystems */
#endif
        struct srchdr srchdr;
        struct subreq subreq;
};
```

Figure 54. Definition of the srcreq *Structure*

Referring back to Figure 52, part **6**, the first thing the handle_SRC_req routine does is call the recvfrom routine to read the next request from the SRC request

[27] Unless specified otherwise, all constants discussed in this section are defined in /usr/include/spc.h.

socket. Standard requests from the SRC to a subsystem are the same size, the size of a srcreq structure[28]. So, handle_SRC_req confirms that the size returned by recvfrom is the size of a srcreq structure. The data in the srchdr structure is saved by a routine provided by the SRC for that purpose, srcrrqs. The handle_SRC_req routine looks at the action field in the subreq structure to determine what request is being made, and takes the appropriate action. Other routines in the daemon are called to handle the STOP, TRACE, STATUS and REFRESH requests. If the request being made is START or some invalid request, handle_SRC_req replies with an error code. A START request is only forwarded to a subsystem if it is for a subserver. This subsystem does not support subservers, so a START request is always replied to with an error code. How replies are made to SRC requests is discussed shortly.

Figure 52, part **7** shows the SRC_stop routine, whose function is to process a STOP request. Two arguments are passed to the routine: the pointer returned by srcrrqs, which is srchdr, and a pointer to the subreq structure describing the request being made of the subsystem. The SRC_stop routine examines the value of the object field in the subreq structure. Since this daemon is not supporting subservers, the SRC_stop routine will send a reply with an error code if the object field in the subreq structure is not SUBSYSTEM.

The SRC_stop routine also examines the value of the parm1 field in the subreq structure. For the STOP request, the parm1 field indicates if the stop request is a stop normal request or a stop forced request. The parm1 field value for stop normal is the value of the constant NORMAL, and the value for stop forced is the value of the constant FORCED. This code is used in a daemon that terminates its child processes and exits immediately when it receives the stop forced request. Terminating the child processes conforms with the intended semantics of the SRC stop forced request, since terminating the child processes will terminate the processing for current clients. The code for this daemon sets the stop_requested flag to 1 when it receives the stop normal request. The do_daemon_loop routine, shown in Figure 52, part **3**, checks the stop_requested flag periodically. Child processes are not terminated, since the semantics of the SRC stop normal request is to allow the processing for current clients to complete.

Figure 52, part **7** also shows the quick_stop routine, whose function is to process an SRC stop cancel request. Recall that the main routine installed quick_stop as the signal handler for the SIGTERM signal. When invoked, the quick_stop routine terminates the daemon's child processes and exits immediately. These actions comply with the intended semantics of the SRC stop cancel request.

The child processes are terminated by sending the SIGTERM signal to all processes in the daemon's process group. In the SRC_stop routine, the SIGTERM signal is first ignored so the daemon process does not run the quick_stop routine unnecessarily. Changing the disposition of the SIGTERM signal is not necessary in the quick_stop routine, since the SIGTERM signal is blocked while quick_stop is executed. Since the quick_stop routine will be used as the SIGTERM signal handler for both the daemon process and the child processes, the routine must base its actions on whether or not it is being run by the daemon. The routine

[28] Subsystem-defined requests can be larger. See 5.6.10, "Processing Subsystem-Defined Requests" on page 125 for information about subsystem-defined requests.

can make this determination by determining if the session leader of the process is the process itself.

Most replies to subsystem requests are made using the srcrep structure. This structure is defined in spc.h and shown in Figure 55 on page 108. The structure is composed of two other structures, an srchdr structure and an svrreply structure. The srchdr structure can be ignored for now. The svrreply structure defines the reply being made by the subsystem. The rtncode field should be set to an SRC return code[29]. The objtype field should be set to a value indicating the type of object to which the request was targeted. The objname field should be set to the name of the subsystem or subserver to which the request was targeted. The rtnmsg field can be used to return an error message when the rtncode field is set to SRC_SUBMSG.

[29] The SRC return codes are described in /usr/include/srcerrno.h. A subsystem does not need to include this header explicitly, since it is included by /usr/include/spc.h.

```
/* src header to be included in all packets that are sent to and
** received from any subsystem
**/
struct srchdr
{
        struct sockaddr_un retaddr;     /* return address for packets */
        short dversion;                 /* SRC packet version */
        short cont;                     /* continuation indicator */
};

/* subsystem reply */
struct svrreply
{
        short rtncode;          /* subsystems response to the request
                                ** negative on subsystem error
                                ** or subsystem unique message returned
                                **/
        short objtype;          /* status object type  one of
                                **    1. SUBSYSTEM
                                **    2. subserver code point
                                **    3. error code
                                **/
        char  objtext[65];      /* text discription to be included in
                                ** response of subsystem action
                                **/
        char  objname[SRCNAMESZ];
                                /* name of object (subsystem/subsever) that
                                ** this response belongs to
                                */
        char rtnmsg[256];       /* subsystem unique message */
};

/* Subsystem reply */
struct srcrep
{
        struct srchdr srchdr;
        struct svrreply svrreply;
};
```

Figure 55. Definition of the srcrep *Structure*

When an svrreply structure is filled in, the objtype field is usually set to the value of the object field of the subreq structure that made the request. Similarly, the objname field of the svrreply structure is usually set to the value of the objname field of the subreq structure that made the request.

A reply is made to an SRC request by calling the srcsrpy routine. The arguments usually specified with the call are the value returned by srcrrqs, the address of a filled in srcrep structure, the size of the reply, and a continuation indicator. The continuation indicator allows multiple replies to be sent for an SRC request. This is described shortly. A value of END for the continuation indicator means this is the last reply for the SRC request.

Referring to Figure 52, part **7**, the SRC_stop routine either replies with the SRC_NOSUPPORT return code or the SRC_OK return code. The SRC_NOSUPPORT return code is used if the stop request was not for the subsystem; otherwise, the SRC_OK

return code is used. Take note that the srcsrpy routine should be called to send the reply before the subsystem terminates itself.

Figure 52, part **8** shows the SRC_trace routine, which handles TRACE requests. If the object field of the subreq structure indicates the request is for a subserver, the SRC_NOSUPPORT error code is returned. If the TRACE request is targeted to the subsystem, the SRC_trace routine determines if tracing is to be turned on or off. For the TRACE requests, the parm2 field of the subreq structure indicates if the request is to turn tracing on or off. The constant TRACEON means tracing is to be turned on; the constant TRACEOFF means tracing is to be turned off. This daemon has decided that when tracing is on, the SO_DEBUG socket option will be turned on for the socket accepting client connections, and when tracing is off, the SO_DEBUG socket option will be turned off for the same socket. If the setsockopt routine fails, the SRC_trace routine sends the error message pointed to by error_msg to the process making the trace request. Notice that the text of the message is placed in the rtnmsg field of the svrreply structure, and the rtncode field of the same structure is set to SRC_SUBMSG. The message will probably be displayed on the standard output of the process which made the trace request[30]. If the setsockopt routine succeeds, the SRC_trace routine sets the rtncode field of the svrreply field to SRC_OK, indicating success.

When the parm2 field of the subreq structure for a TRACE request is TRACEON, the parm1 field of the structure indicates if short or long tracing is being requested. The constant SHORTTRACE indicates short tracing is being requested; the constant LONGTRACE indicates long tracing is being requested. The subsystem may or may not distinguish between short and long tracing. The daemon in Figure 52 on page 93 makes no distinction between short and long tracing.

Figure 52, part **9** shows the SRC_refresh routine. The routine handles the REFRESH request. If the object field of the subreq structure indicates the request is for a subserver, the SRC_NOSUPPORT error code is returned. If the REFRESH request is targeted to the subsystem, the SRC_refresh routine calls the refresh_config routine of Figure 52, part **4** to actually refresh the daemon's configuration. If the refresh_config routine reports an error, the SRC_refresh routine sends the error message pointed to by er_msg to the process making the refresh request. If the refresh_config routine returns an indication of success, the SRC_refresh routine sends two replies to the process making the refresh request. Notice that the srcsrpy routine is called twice to send the two replies. On the first call to srcsrpy, the continuation indicator is set to CONTINUED; on the second call to srcsrpy, the continuation indicator is set to END. When a reply is sent with the CONTINUED continuation indicator, the receiving process expects a message in the rtnmsg field of the svrreply structure[31]. The receiving process displays the text of the message on its standard output, and waits for the next reply. The value of the rtncode field of the svrreply structure has no meaning when the CONTINUED continuation indicator is used. The SRC_OK return code is sent with the reply that is sent with the END continuation indicator.

---

[30] A defect in the source code from which the traceson, tracesoff, and refresh commands are built does not allow the program to receive the last 114 bytes of the rtnmsg field. Until this is fixed, subsystems should limit themselves to the first 142 bytes of the rtnmsg field. One of those bytes will be needed for a new-line character and one byte will be needed for the null terminator.

[31] Notice that when the CONTINUED continuation indicator is specified, a message placed in rtnmsg does not end with a new-line character. A new-line character is added by the receiving process to this type of message. Contrast this to the case when the END continuation indicator is specified. Then, a message placed in rtnmsg does end with a new-line character. In this case, the receiving process does not add a new-line character.

The remaining routine to discuss in Figure 52 on page 93 is SRC_status, which responds to STATUS requests. The status of a subsystem is not sent to the requesting process in a srcrep structure. Instead, it is sent in a statrep structure. The statrep structure is defined in spc.h, and shown in Figure 56 on page 110. The statrep structure is composed of a srchdr structure and an array of statcode structures. The definition of the statrep structure indicates only one entry in the array of statcode structures. However, more than one statcode structure may be associated with a reply. Each statcode structure can be thought to represent one line of status output. For each statcode structure returned in response to a status request, the lssrc command executes the printf routine to display output with the following format: the null-terminated string in the objname field, followed by a blank, followed by the null-terminated string in objtext, followed by a blank, followed by a string corresponding to the value in the status field. New-line characters can be placed in the objname and objtext fields to cause more than one line of status output to be generated by one statcode structure.

```
/* src header to be included in all packets that are sent to and
** received from any subsystem
**/
struct srchdr
{
        struct sockaddr_un retaddr;     /* return address for packets */
        short dversion;                 /* SRC packet version */
        short cont;                     /* continuation indicator */
};

/*
**      Subsystem Status Reply
**/
struct statcode
{
        short objtype;          /* status object type  one of
                                **    1. SUBSYSTEM
                                **    2. subserver code point
                                **    3. error code
                                **/
        short status;           /* status code */
        char  objtext[65];      /* text discription to be included in
                                ** printing status for this object
                                **/
        char  objname[SRCNAMESZ];
                                /* name of object (subsystem/subsever) that
                                ** this status belongs to
                                */
};

/* Subsystem status reply */
struct statrep
{
        struct srchdr srchdr;
        struct statcode statcode[1];
};
```

Figure 56. Definition of the statrep *Structure*

Figure 52, part **10** shows the SRC_status routine. The routine handles the STATUS request. If the object field of the subreq structure indicates the request is for a subserver, the SRC_NOSUPPORT error code is returned. If the STATUS request is targeted to the subsystem, the SRC_status routine will reply with the long status of the subsystem. The parm1 field of the subreq structure indicates if short or long status is requested. However, the SRC_status routine does not examine the parm1 field, because a subsystem will never receive a request for subsystem short status. If the subsystem supported subservers, it might need to examine the parm1 field to distinguish between subserver short and long status.

The SRC_status routine returns two lines of status with three calls to srcsrpy.

Before the first call to srcsrpy, the objtext field of a statcode structure is filled in with a message indicating how many client connection requests the daemon has seen. The continuation indicator is specified as STATCONTINUED on the first call to srcsrpy. This is an indication that this reply contains status, and further replies should be expected. Notice that the address and length of the statrep structure are passed as parameters. Before the second call to srcsrpy, the objtext field of a statcode structure is filled in with a message indicating whether or not the daemon is in tracing mode. The continuation indicator is specified as STATCONTINUED on the second call to srcsrpy. Again, this is an indication that this reply contains status, and further replies should be expected.

Finally, a third call to srcsrpy is made. This time the continuation indicator is specified as END. This is an indication that this is the last reply for the status request. No meaningful data is sent with this reply, but the length of the reply must be equal to the size of a srchdr structure.

Figure 57 on page 112 shows a modified version of the SRC_status routine. This version of the routine sends multiple statcode structures in one call to srcsrpy. Enough memory is allocated to hold one srchdr structure and as many statcode structures as are needed. The statcode structures are then filled in before calling srcsrpy. Another call to srcsrpy is still needed to indicate to the requesting process that no more replies should be expected.

```
static void SRC_status(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;
      struct statrep *statrep;
      int statrep_size;
      struct statcode *sp;
      const char * const er_msg = "Insufficient memory to satisfy request.\n";

      if (subreq->object != SUBSYSTEM) {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      statrep_size = sizeof(struct srchdr) + 2 * sizeof(struct statcode);

      if ((statrep = malloc(statrep_size)) == NULL) {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_SUBMSG;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            strcpy(srcrep.svrreply.rtnmsg, er_msg);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      sp = &statrep->statcode[0];
      sp->objtype = 0;
      sp->status  = 0;
      strcpy(sp->objname, "");
      sprintf(sp->objtext, "I've seen %d client connections.", client_count);

      sp++;
      sp->objtype = 0;
      sp->status  = 0;
      strcpy(sp->objname, "");
      strcpy(sp->objtext, tracing_on ? "Tracing is on." : "Tracing is off.");

      (void) srcsrpy(srchdr, statrep, statrep_size, STATCONTINUED);
      (void) srcsrpy(srchdr, statrep, sizeof(struct srchdr), END);
      free(statrep);

      return;
}
```

Figure 57. Sending Multiple statcode *Structures (Assuming No Padding)*

The maximum number of statcode structures that can be sent in one call to srcsrpy is limited by the maximum packet size supported by the SRC. This maximum packet size is defined by the macro SRCPKTMAX. The number of statcode structures that can be placed in an SRC packet of the maximum size can be calculated as shown in Figure 58 on page 113. In AIX 4.1.2, SRCPKTMAX is defined as 8192 bytes, the size of the srchdr structure is 114 bytes, and the size

of the statcode structure is 100 bytes. Therefore, the number of statcode structures that can be placed in the largest SRC packet is 80.

```
(SRCPKTMAX - sizeof(struct srchdr)) / sizeof(struct statcode)
```

Figure 58. Maximum statcode *Structures in an SRC Packet (Assuming No Padding)*

It should be noted that the code in Figure 57 on page 112 and Figure 58 assumes there is no padding between the srchdr structure and the statcode array in the statrep structure. This is true in AIX 4.1.2. Figure 59 on page 114 shows a version of SRC_status that does not make this assumption. Figure 60 on page 114 shows how the number of statcode structures that fit in an SRC packet of the maximum size can be calculated without making the padding assumption.

```
#define STATCODE_PTR(ptr,x) ((((struct statrep *)(ptr))->statcode) + (x))
#define STATCODE_0_OFF      (((char *) STATCODE_PTR(NULL, 0)) - ((char *) NULL))
#define STATUS_SIZE(items)  ((items) * sizeof(struct statcode))
#define STATREP_SIZE(items) (STATCODE_0_OFF + STATUS_SIZE(items))

static void SRC_status(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;
      struct statrep *statrep;
      int statrep_size;
      struct statcode *sp;
      const char * const er_msg = "Insufficient memory to satisfy request.\n";

      if (subreq->object != SUBSYSTEM) {
            /* Fill in srcrep to indicate error; see Figure 57 for the code. */
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      statrep_size = STATREP_SIZE(2);

      if ((statrep = malloc(statrep_size)) == NULL) {
            /* Fill in srcrep to indicate error; see Figure 57 for the code. */
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      sp = &statrep->statcode[0];
      sp->objtype = 0;
      sp->status  = 0;
      strcpy(sp->objname, "");
      sprintf(sp->objtext, "I've seen %d client connections.", client_count);

      sp++;
      sp->objtype = 0;
      sp->status  = 0;
      strcpy(sp->objname, "");
      strcpy(sp->objtext, tracing_on ? "Tracing is on." : "Tracing is off.");

      (void) srcsrpy(srchdr, statrep, statrep_size, STATCONTINUED);
      (void) srcsrpy(srchdr, statrep, sizeof(struct srchdr), END);
      free(statrep);

      return;
}
```

Figure 59. Sending Multiple statcode *Structures (Making No Padding Assumptions)*

```
      ((SRCPKTMAX - sizeof(struct statrep)) / sizeof(struct statcode)) + 1
```

Figure 60. Maximum statcode *Structures in an SRC Packet (Making No Padding Assumptions)*

5.6.6 Accessing the SRC Request Message Queue

A daemon defined as an SRC subsystem that receives SRC requests through a message queue can access the message queue using the msgget routine. When the msgget routine is called to access the message queue, the message key specified in the subsystem definition must be used. An example is shown in Figure 61 on page 116. The example is explained in the following sections.

5.6.7 Determining If an SRC Message Queue Request Is Pending

A daemon defined as an SRC subsystem that receives SRC requests through a message queue must regularly determine if an SRC request is pending on the SRC request message queue. If SRC requests are not answered within 60 seconds, the requesting process usually terminates with an error. If the daemon spends most of its time waiting, using the select routine to wait for requests seems to be the natural choice. An example is shown in Figure 61 on page 116. The example is explained in the following sections.

5.6.8 Processing an SRC Message Queue Request

Processing SRC requests that arrive through a message queue is similar to processing SRC requests that arrive through a socket. Therefore, this section assumes the information in 5.6.5, “Processing an SRC Socket Request” on page 92 has been read and understood.

The two differences between handling SRC requests from a socket and handling them from a message queue are:

- When handling requests from a socket, socket routines like getsockname and recvfrom are used. When handling requests from a message queue, message queue routines like msgget and msgrcv are used.
- The definition of the srcreq structure is changed to include a message type field when handling SRC requests from a message queue. Refer back to Figure 54 on page 105. The mtype field of the srcreq structure is only defined if the SRCBYQUEUE macro is defined. A subsystem receiving SRC requests from a socket must not define that macro. A subsystem receiving SRC requests from a message queue must define that macro before including the spc.h header file. A subsystem receiving requests from a message queue must take into account the extra field in the srcreq structure.

Figure 61 presents an example of a daemon designed to run under SRC control with message queue communication. The example illustrates how a daemon can receive SRC requests through a message queue, and how it can handle those requests.

```
1
#include <unistd.h>

#define  SRCBYQUEUE
#include <spc.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define SUBSYS_DAE_OFFSET  200
#define MSQ_KEY  0xBADF00D
#define MSQ_TYPE 0xBEEF

int stop_requested = 0;
int msqid;

static void signal_stop(int signo);
static void do_daemon_work(void);


int main(int argc, char **argv)
{
      const int msqperm = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP;
      struct msqid_ds msqid_ds;
      struct srcreq srcreq;


      if ((msqid = msgget(MSQ_KEY, msqperm)) == -1) {
            /* Log some error message */
            return 1;
      }

      do_daemon_work();

      return 0;
}
```

Figure 61 (Part 1 of 5). Processing SRC Requests from a Message Queue

```
2
#include <unistd.h>
#include <sys/select.h>
#include <sys/types.h>
#include <time.h>

void handle_SRC_req(void);

static void do_daemon_work(void)
{
      struct sellist {
            int msqids[1];
      };
      struct sellist read_list;
      struct timeval time_out;
      time_t start_time, end_time, current_time;
      const time_t sample_rate = 5 * 60;       /* Sample every 5 minutes. */
      const int nmsq = 1 << 16;
      int rval;


      while (!stop_requested) {

            /* Do the actual work of the daemon here, perhaps. */

            start_time = time(NULL);
            end_time = start_time + sample_rate;

            current_time = start_time;

            while ((current_time < end_time) && (!stop_requested)) {

                  time_out.tv_sec  = end_time - current_time;
                  time_out.tv_usec = 0;
                  read_list.msqids[0] = msqid;
                  rval = select(nmsq, &read_list, NULL, NULL, &time_out);

                  if ((rval == nmsq) && (read_list.msqids[0] == msqid)) {
                        handle_SRC_req();
                  }

                  current_time = time(NULL);
            }

      }

      return;
}
```

Figure 61 (Part 2 of 5). Processing SRC Requests from a Message Queue

```
3
static void SRC_stop(struct srchdr *srchdr, struct subreq *subreq);
static void SRC_status(struct srchdr *srchdr, struct subreq *subreq);

static void handle_SRC_req(void)
{
      struct srcreq srcreq;    struct srcrep srcrep;
      struct srchdr *srchdr;   struct subreq *subreq;
      int reqlen;

      reqlen = msgrcv(msqid, &srcreq, sizeof srcreq - sizeof(long),
                      MSQ_TYPE, IPC_NOWAIT | MSG_NOERROR);

      if (reqlen != sizeof srcreq - sizeof(long))
            return;

      srchdr = srcrrqs((char *)&srcreq + sizeof(long));
      subreq = &srcreq.subreq;

      if (subreq->action == STOP) {
            SRC_stop(srchdr, subreq);
      } else if (subreq->action == STATUS) {
            SRC_status(srchdr, subreq);
      } else if ((subreq->action == TRACE) || (subreq->action == REFRESH) ||
                 (subreq->action == START)) {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
      } else {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_SUBICMD;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
      }

      return;
}
```

Figure 61 (Part 3 of 5). Processing SRC Requests from a Message Queue

```
4
static void SRC_stop(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;


      memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
      srcrep.svrreply.objtype = subreq->object;
      strcpy(srcrep.svrreply.objname, subreq->objname);

      if (subreq->object != SUBSYSTEM) {
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      srcrep.svrreply.rtncode = SRC_OK;
      (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);

      stop_requested = 1;

      return;
}
```

Figure 61 (Part 4 of 5). Processing SRC Requests from a Message Queue

```
5
#include <time.h>


static void SRC_status(struct srchdr *srchdr, struct subreq *subreq)
{
      struct srcrep srcrep;
      struct statrep statrep;
      struct statcode *sp;
      time_t current_time;


      if (subreq->object != SUBSYSTEM) {
            memset(&srcrep.svrreply, 0, sizeof srcrep.svrreply);
            srcrep.svrreply.rtncode = SRC_NOSUPPORT;
            srcrep.svrreply.objtype = subreq->object;
            strcpy(srcrep.svrreply.objname, subreq->objname);
            (void) srcsrpy(srchdr, &srcrep, sizeof srcrep, END);
            return;
      }

      sp = &statrep.statcode[0];

      sp->objtype = 0;
      sp->status  = 0;
      strcpy(sp->objname, "");
      sprintf(sp->objtext, "\nAll is well with process %d at ", getpid());
      current_time = time(NULL);
      strcat(sp->objtext, ctime(&current_time));
      (void) srcsrpy(srchdr, &statrep, sizeof statrep, STATCONTINUED);

      (void) srcsrpy(srchdr, &statrep, sizeof(struct srchdr), END);

      return;
}
```

Figure 61 (Part 5 of 5). Processing SRC Requests from a Message Queue

The first thing to notice about the example in Figure 61 is that the daemon defines the SRCBYQUEUE macro before including the spc.h header file.

Figure 61, part **1** shows the main routine of the daemon. This routine calls msgget to access the message queue through which SRC requests are expected. The message key specified in the first parameter to msgget should be the key that was specified when the daemon was defined as an SRC subsystem. This program refers to the message key with the macro MSQ_KEY. The msgget routine returns an identifier for the message queue. The returned identifier is used in other routines to perform operations on the message queue. Once the message queue has been accessed, the main routine calls do_daemon_work.

Figure 61, part **2** shows the do_daemon_work routine, which will presumably do the work the daemon is designed to do. This routine is structured for a daemon that will sample some data at some regular interval, in this case every 5 minutes. The routine uses select with a time out value. The time out value specified is the time remaining until the expiration of the sampling interval. The select system call will return when the time out value has expired, when the

SRC request message queue has a message that can be read, or when it is interrupted by a signal handler returning. The do_daemon_work routine is coded so the data sample is taken at the proper interval. This may involve several calls to select per interval if select returns before the time out specified has expired. If the select routine indicates data can be read from the message queue, the handle_SRC_req routine is called. When handle_SRC_req returns, select is called again if there is more time to wait before the expiration of the sampling interval, and a stop request has not been received.

Figure 61, part 3 shows the handle_SRC_req routine, whose function is to process the next request on the SRC request message queue. The first thing the handle_SRC_req routine does is call the msgrcv routine to read the next request from the SRC request message queue. The IPC_NOWAIT flag is specified so the process cannot be blocked in the msgrcv routine. The MSG_NOERROR flag is specified so requests that are larger than expected can be received. Standard requests from the SRC to a subsystem are the same size[32]. So, handle_SRC_req confirms that the size returned by msgrcv is the size that is expected. The data in the srchdr structure is saved by a routine called srcrrqs, which is provided by the SRC for that purpose. The handle_SRC_req routine looks at the action field in the subreq structure to determine what request is being made, and takes the appropriate action. Other routines in the daemon are called to handle the STOP and STATUS requests. If the request being made is some invalid request or a request not supported by this daemon, handle_SRC_req replies with an error code.

There are three points in Figure 61, part 3 where the extra mtype field in the srcreq structure is taken into account. First, the size of the buffer passed to the msgrcv routine must not include the size of the mtype field. Second, the size returned by msgrcv does not include the size of the mtype field. Lastly, when srcrrqs is called to save away the srchdr structure, the pointer passed to the routine must be offset by the size of the mtype field. Contrast the code in Figure 61, part 3 with similar code in Figure 52, part 6 that obtains the next SRC request from a socket.

If you compare the code in Figure 61, part 3 through part 5 that sends replies to the requesting process, with the code in Figure 52 on page 93 that sends replies to the requesting process, you will see that the techniques used to send replies to SRC requests are identical for a subsystem receiving requests from a socket and a subsystem receiving requests from a message queue.

5.6.9 Problems with SRC Message Queues

There are some problems with how the SRC daemon, srcmstr, handles message queues. The following problems have been reported:

1. If an SRC subsystem is run under the root user and the subsystem is then stopped and the user for the subsystem is changed to a non-root user, any attempts to start the subsystem will fail until one of the following occurs:
   - The system is rebooted.
   - The user for the subsystem is changed back to root.
   - The message queue created when the subsystem was last run is removed with ipcrm.

[32] Subsystem-defined requests can be larger. See 5.6.10, “Processing Subsystem-Defined Requests” on page 125 for information about subsystem-defined requests.

2. Any user on the system can send a request to an SRC subsystem using message queue communication. The user just needs to know a little about the structures used to send SRC requests to a subsystem, and how to send a request on a message queue. Such a user can write a simple program to cause the subsystem to terminate.

These problems are illustrated using the subsystem whose code was shown in Figure 61. Assuming the executable code for this program is installed in /usr/lpp/somelpp/bin/SRC_msq, the subsystem is defined to the SRC as subsystem des.ex.msq in Figure 62. Notice that the program should be executable by the root user and any user in the system group.

```
# mkssys -s des.ex.msq -p /usr/lpp/somelpp/bin/SRC_msq -u 0 \
         -I 195948557 -m 48879 -G daes
0513-071 The des.ex.msq Subsystem has been added.

# odmget -q "subsysname = 'des.ex.msq'" SRCsubsys

SRCsubsys:
        subsysname = "des.ex.msq"
        synonym = ""
        cmdargs = ""
        path = "/usr/lpp/somelpp/bin/SRC_msq"
        uid = 0
        auditid = 0
        standin = "/dev/console"
        standout = "/dev/console"
        standerr = "/dev/console"
        action = 2
        multi = 0
        contact = 1
        svrkey = 195948557
        svrmtype = 48879
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = "daes"

# ls -l /usr/lpp/somelpp/bin/SRC_msq
-r-xr-x---   1 root  system  2148 Aug 18 13:20 /usr/lpp/somelpp/bin/SRC_msq
```

Figure 62. Defining the des.ex.msq *Subsystem*

In Figure 63 on page 123, the des.ex.msq subsystem is executed under the root user, but then cannot be executed under the agarsys user, who is in the system group, until the message queue left behind by the first execution of des.ex.msq is removed.

```
# ipcs -q
IPC status from /dev/mem as of Sat Aug 19 12:43:17 EDT 1995
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
q      0 0x4107001c -Rrw-rw----     root   printq

# startsrc -s des.ex.msq
0513-059 The des.ex.msq Subsystem has been started. Subsystem PID is 18344.

# stopsrc -s des.ex.msq
0513-044 The stop of the des.ex.msq Subsystem was completed successfully.

# ipcs -q
IPC status from /dev/mem as of Sat Aug 19 12:44:51 EDT 1995
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
q      0 0x4107001c -Rrw-rw----     root   printq
q 323585 0x0badf00d --rw-rw-rw-     root   system

# grep agarsys /etc/passwd
agarsys:*:1825:0::/home/agarsys:/usr/bin/ksh

# chssys -s des.ex.msq -u 1825
0513-077 Subsystem has been changed.

# startsrc -s des.ex.msq
0513-033 The des.ex.msq Subsystem could not be started.
The Subsystem's communication queue could not be established.

# ipcrm -q 323585

# ipcs -q
IPC status from /dev/mem as of Sat Aug 19 12:46:13 EDT 1995
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
q      0 0x4107001c -Rrw-rw----     root   printq

# startsrc -s des.ex.msq
0513-059 The des.ex.msq Subsystem has been started. Subsystem PID is 18368.

# ipcs -q
IPC status from /dev/mem as of Sat Aug 19 12:46:37 EDT 1995
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
q      0 0x4107001c -Rrw-rw----     root   printq
q 327681 0x0badf00d --rw-rw-rw-  agarsys   system
```

Figure 63. Attempts to Run des.ex.msq *with Different Users*

The sequence of commands in Figure 64 on page 124 show that when des.ex.msq is run under the root user, a non-root user that is not in the system group cannot stop des.ex.msq with stopsrc, but can easily write a program, sendstop, to cause the subsystem to stop. In Figure 64 on page 124, the # prompt is for the root user, and the $ prompt is for the user agar.

```
# stopsrc -s des.ex.msq
0513-044 The stop of the des.ex.msq Subsystem was completed successfully.

# chssys -s des.ex.msq -u 0
0513-077 Subsystem has been changed.

# startsrc -s des.ex.msq
0513-059 The des.ex.msq Subsystem has been started. Subsystem PID is 18378.

# ps -fp 18378
     UID   PID  PPID   C    STIME    TTY  TIME CMD
    root 18378  2600   0 12:50:15      -  0:00 /usr/lpp/somelpp/bin/SRC_msq

$ whoami
agar

$ groups
staff

$ lssrc -s des.ex.msq
Subsystem         Group            PID     Status
 des.ex.msq       daes             18378   active

$ stopsrc -s des.ex.msq
ksh: stopsrc: 0403-006 Execute permission denied.

$ lssrc -s des.ex.msq
Subsystem         Group            PID     Status
 des.ex.msq       daes             18378   active

$ ./sendstop 195948557 48879

$ lssrc -s des.ex.msq
Subsystem         Group            PID     Status
 des.ex.msq       daes                     inoperative
```

Figure 64. Stopping des.ex.msq *by an Unauthorized User*

Figure 65 on page 125 shows the source for the program, sendstop, that can be used to stop any subsystem accepting message queue communications from the SRC. The program takes two parameters: the key of the message queue to which a stop request is to be sent, and the type of the message to send to the message queue. This information is easily obtained from the SRCsubsys ODM object describing the subsystem.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/ipc.h>
#include <sys/msg.h>

#define SRCBYQUEUE
#include <spc.h>


int main(int argc, char **argv)
{
      struct srcreq srcreq;
      int msqkey, msqtype, msqid;


      if (argc < 3) {
            fprintf(stderr, "USAGE: %s msq_key msq_type\n", argv[0]);
            return 1;
      }

      msqkey  = atoi(argv[1]);
      msqtype = atoi(argv[2]);

      if ((msqid = msgget(msqkey, 0)) == -1) {
            perror("msgget()");
            return 1;
      }

      memset(&srcreq, '\0', sizeof srcreq);
      srcreq.mtype = msqtype;
      srcreq.subreq.action = STOP;
      srcreq.subreq.object = SUBSYSTEM;

      if (msgsnd(msqid, &srcreq, sizeof srcreq - sizeof(long),
                  IPC_NOWAIT) == -1) {
            perror("msgsend()");
            return 1;
      }

      return 0;
}
```

Figure 65. Source for the sendstop *Program*

5.6.10 Processing Subsystem-Defined Requests

The standard requests defined by the SRC are START, STOP, TRACE, STATUS, and REFRESH. A subsystem may choose to define additional requests. Additional requests are represented in the action field of the subreq structure by values larger than 255; values through 255 are reserved by the SRC. The parm1 and parm2 fields can be used as request modifiers for additional requests, as they are

for the SRC-defined requests. The other fields of the subreq structure should be used by additional requests as they are by the SRC-defined requests.

The daemon support routines support subsystems that would like to handle subsystem-defined requests. They even support request parameters beyond those that fit in the subreq structure. See Chapter 10, “Daemon Support Routines” on page 245 for the details.

If a subsystem defines additional SRC requests, some client must send the requests. Usually this involves writing a new program that can be used to send the request. Such a program can use the srcsrqt routine provided by the SRC library to send the request and receive the reply. See the man page for srcsrqt in *AIX Version 4.1: Technical Reference, Volume 2: Base Operating System and Extensions*, SC23-2615. A model for such a program is discussed in this book in Chapter 12, “New SRC Program Support” on page 303. In most cases it should not be necessary to modify the model; the needed customization can be performed through compile time definitions.

5.6.11 Compiling and Linking an SRC Daemon

A daemon under SRC control which receives SRC requests through a socket should be compiled with the macro _BSD defined to be 43 or 44. Such a daemon should be linked with the libsrc.a, libbsd.a, and libc.a libraries.

There are no special compilation requirements for a daemon under SRC control which receives SRC requests through a message queue. Such a daemon should be linked with the libsrc.a and libc.a libraries.

There are no special compilation or linking requirements for a daemon under SRC control which receives SRC requests through signals.

5.6.12 Multi-Threaded Daemons and the SRC

It is expected that some daemons written by Scalable POWERparallel Systems, RS/6000 Division, will be multi-threaded. Unfortunately, the library routines provided by AIX to support communications between the SRC and SRC-controlled daemons are not thread-safe. Until those routines are made thread-safe, a multi-threaded daemon under SRC control may be limited to signal communications from the SRC.

5.7 Programming an SRC Notification Method

If a daemon running under the control of the SRC uses no other recovery mechanism, it should support the use of the SRC notification mechanism. If there are no default recovery actions that will be shipped to the customer, the documentation of the subsystem should include instructions to the customer on how to install an SRC notification method for the subsystem. These instructions should include information about the dae_msg program, an SRC notification support program documented in Appendix B, “SRC Notification Support Programs man Pages” on page 385.

If there are default recovery actions that will be shipped to the customer, the installation of the subsystem should add an object to the SRCnotify object class, defining an SRC notification method for the subsystem. The notification method installed should be the dae_notify program, an SRC notification support program documented in Appendix B, “SRC Notification Support Programs man Pages” on

page 385. The documentation of the subsystem should include instructions about the user exits provided by the SRC notification support programs.

The SRC notification support programs are explained in Chapter 11, “SRC Notification Support Programs” on page 295.

Chapter 6. Managing a Daemon with the Internet Superserver

The inetd daemon is usually started at system boot time. It reads its configuration file, usually /etc/inetd.conf, to determine what Internet services it supports. Each Internet service corresponds to a TCP/IP or UDP/IP port. The inetd daemon reads the /etc/services file to obtain the mappings from services to ports. The inetd daemon waits for client requests to arrive on the ports corresponding to the services supported by inetd. When a client request arrives on a port, the information from the inetd configuration file is used to determine what should be done. The information may indicate that the inetd daemon itself supports clients of the service represented by the port. In that case, inetd services the client request. More often, clients of the service represented by the port are supported by another daemon. In this case, inetd creates a child process running the appropriate daemon. The inetd configuration file provides the path name of the daemon, the parameters to be passed to the daemon, the user ID under which the daemon process should run, and whether inetd should allow multiple processes to run the daemon at the same time.

A daemon may be placed under inetd control if it has the following characteristics:

- The daemon does not have to start running when the system is booted.
- An instance of the daemon does not have to continuously run to monitor the system.
- It is convenient for an instance of the daemon to be spawned for each client requesting services from the daemon.
- The daemon communicates with clients through the socket network programming interface to UDP/IP or TCP/IP.
- It is acceptable to the daemon that it will not be restarted by inetd if it terminates abnormally.
- When the daemon is running on an AIX system, the limited SRC support provided by inetd is acceptable.

The remainder of this chapter discusses inetd configuration files, and the relationship between inetd and the SRC. The important points made about putting a new daemon under inetd control are:

- A description of the daemon, and the Internet service it provides, must be added to the default inetd configuration file, /etc/inetd.conf.
- A mapping from Internet service to port number must be added to the /etc/services file.
- On an AIX system, a new SRC subserver must be defined. The new subserver must refer to the name of the new Internet service and the port number associated with the service.
- The inetd daemon must be forced to re-read its configuration files. On an AIX system, this can be done with the refresh command. On another type of system, the HUP signal should be sent to the inetd process with the kill command.

6.1 The Configuration and Services Files

Each entry in the inetd configuration file, which is /etc/inetd.conf by default, has the following format:

```
ServiceName SocketType ProtocolName Wait/NoWait UserName ServerPath ServerArgs
```

- The ServiceName field names an Internet service supported by inetd.
- The ProtocolName field indicates the protocol used by the service. Protocols are defined in the /etc/protocols file. The two protocols supported by inetd are udp and tcp.
- The SocketType field indicates the type of socket to be used for the service; the common values are stream, for a stream socket, and dgram, for a datagram socket.
- The Wait/NoWait field contains the value wait or nowait. If wait is specified, only one client of the service is supported at one time. If nowait is specified, multiple clients of the service can be supported concurrently.
- The UserName field specifies the user name under which the daemon supporting the service should run.
- The ServerPath field specifies the full path name of the daemon program that supports clients of the service.
- The ServerArgs field specifies the parameters to be passed to the daemon.

For full documentation on the meaning of the fields and their possible values, refer to the “inetd.conf File Format for TCP/IP” entry in Chapter 2. “File Formats” of *AIX Version 4.1: Files Reference*, SC23-2512.

Each entry of the /etc/services file has the following format:

```
ServiceName PortNumber/ProtocolName Aliases
```

The inetd daemon uses the ServiceName and ProtocolName fields to associate /etc/services entries with inetd configuration entries. Given an entry in the inetd configuration file, the values of the ServiceName and ProtocolName fields are used to find the associated entry in the /etc/services file. The PortNumber field in the /etc/services entry contains the port number associated with the service described in the inetd configuration file. For full documentation on the meaning of the fields and their possible values, refer to the “services File Format for TCP/IP” entry in Chapter 2. “File Formats” of *AIX Version 4.1: Files Reference*, SC23-2512.

Figure 66 on page 131 shows the information stored in /etc/inetd.conf and /etc/services about the telnet service. Based on these entries, you know these statements are true about how the telnet service is supported:

- The inetd daemon waits, through a stream socket, for a connection request to arrive on TCP port 23.
- When a connection request arrives on the port, the inetd daemon establishes a connection with the client and creates a child process.
- The child process is run by the root user.
- File descriptors 0, 1, and 2 of the child process are associated with the stream socket connected to the client.
- The child process executes the program /usr/sbin/telnetd; no parameters are passed to the program.
- The telnetd program services the client.

- The inetd process does not wait for the child process to terminate before waiting for another connection request to arrive on TCP port 23.

```
# grep "^telnet" /etc/inetd.conf
telnet  stream  tcp     nowait  root    /usr/sbin/telnetd      telnetd

# grep "^telnet" /etc/services
telnet          23/tcp
```

Figure 66. Information about the telnet *Service*

6.2 Updating the Configuration and Services Files

When a daemon is placed under inetd control, the inetd configuration and /etc/services files need to be updated with information about the daemon and the service.

In most systems, including AIX 4.1, the inetd configuration and /etc/services files are updated with an editor or script language. After one or both of these files have been updated, the inetd daemon should be told to re-read the files. This can be done by sending the SIGHUP signal to the inetd process. On an AIX system, it can also be done by using the refresh command. See Figure 67 for examples.

```
# vi /etc/inetd.conf

# ps -e | grep inetd
  5512      -  0:05 inetd

# kill -HUP 5512

# vi /etc/services

# refresh -s inetd
0513-095 The request for subsystem refresh was completed successfully.
```

Figure 67. Causing inetd *to Re-read Configuration Files*

Previous versions of AIX stored the information about Internet services in the Object Data Manager (ODM). In these versions of AIX, the /etc/inetd.conf and /etc/services files were just traditional representations of data stored in the ODM. The data was to be modified with the inetserv command; the files should not have been updated directly with an editor. In AIX 4.1, information about Internet services was taken out of the ODM, and there is no longer a need for the inetserv command.

6.3 The SRCsubsvr Object Class

On AIX, the inetd daemon is defined as an SRC subsystem. For information about how SRC subsystems are defined, refer to 5.2, “Subsystem Definition” on page 62. The SRC definition for inetd is shown in Figure 68. Notice that if inetd terminates abnormally, it is not respawned by the SRC (the action value is 2). Also notice that the SRC sends requests to inetd through a socket (the contact value is 3). This means the inetd subsystem could support all the SRC requests. The inetd subsystem does in fact support the subsystem stop, trace, and refresh requests. It also supports several subserver requests.

```
# odmget -q "subsysname = 'inetd'" SRCsubsys

SRCsubsys:
        subsysname = "inetd"
        synonym = ""
        cmdargs = ""
        path = "/usr/sbin/inetd"
        uid = 0
        auditid = 0
        standin = "/dev/console"
        standout = "/dev/console"
        standerr = "/dev/console"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = "tcpip"
```

Figure 68. The SRC Definition of the inetd *Subsystem*

An SRC subsystem can support subservers. The only known subsystem to do so is inetd. A subserver can represent some subset of the function delivered by the subsystem. A subserver might represent a process that the subsystem controls, but that is not required. The inetd subsystem uses subservers to represent Internet services. By defining an SRC subserver object for each Internet service supported by inetd, SRC commands can refer to specific Internet services.

Figure 69 on page 133 shows the definition of the SRCsubsvr object class, as revealed by the odmshow command. The fields have the following meanings:

```
# odmshow SRCsubsvr
class SRCsubsvr {
        char sub_type[30];                              /* offset: 0xc ( 12) */
        char subsysname[30];                            /* offset: 0x2a ( 42) */
        short sub_code;                                 /* offset: 0x48 ( 72) */
        };
/*
        columns:        3
        structsize:     0x4c (76) bytes
        data offset:    0x1e8
        population:     19 objects (19 active, 0 deleted)
*/
```

Figure 69. The Definition of the SRCsubsvr *Object Class*

- The subsysname field specifies the name of the subsystem with which the subserver is associated. Each subserver is associated with a subsystem. When an SRC command is applied to a subserver, a request is sent to the subsystem associated with the subserver.
- The sub_type field specifies the name of the subserver. This name is used on SRC commands to refer to the subserver.
- The sub_code field is an integer that identifies the subserver to its associated subsystem. When an SRC subserver request is sent to a subsystem, the subserver to which the request applies is specified by a sub_code field value.

For inetd subservers, the subsysname field value is "inetd," the sub_type field value is the name of an Internet service as specified in /etc/inetd.conf and /etc/services, and the sub_code field value is the port associated with the Internet service. Figure 70 shows an example. The telnet subserver is associated with the inetd subsystem. Furthermore, the sub_code field value for the telnet subserver is 23, which is the port number associated with the telnet Internet service.

```
# grep "^telnet" /etc/inetd.conf
telnet  stream  tcp     nowait  root    /usr/sbin/telnetd      telnetd

# grep "^telnet" /etc/services
telnet          23/tcp

# odmget -q "sub_type = 'telnet'" SRCsubsvr

SRCsubsvr:
        sub_type = "telnet"
        subsysname = "inetd"
        sub_code = 23
```

Figure 70. The Definition of the telnet *Subserver*

When a daemon is placed under inetd control, a subserver should be defined in SRCsubsvr for the service being provided by the daemon. SRC subserver definitions are made with the mkserver command, changed with the chserver command, and removed with the rmserver command. These commands

manipulate objects in the SRCsubsvr object class and notify the SRC daemon, srcmstr, of the changes. Figure 71 on page 134 shows the mkserver command that would create the telnet subserver definition seen in Figure 70 if the subserver had not already been defined.

```
# mkserver -t telnet -s inetd -c 23
0513-063 Subserver type already exists on file.
```

Figure 71. Using mkserver *to Create a SRC Subserver*

6.4 Using SRC Commands with the inetd Subsystem

This section illustrates how SRC commands can be applied to the inetd subsystem and to specific inetd subservers.

6.4.1 Starting, Stopping, and Listing Status of the Subsystem

Section 5.1, “The SRC Commands” on page 50, gave several examples of using the SRC commands to make requests of subsystems. Many of these examples were applied to the inetd subsystem. The inetd subsystem can be stopped with stopsrc, started with startsrc, queried for its status with lssrc, put into debug mode with traceson, taken out of debug mode with tracesoff, and refreshed with refresh. Figure 72 on page 135 repeats some examples using stopsrc, startsrc, and lssrc. Notice that the output for lssrc -l includes information about the Internet services supported by inetd. Figure 67 on page 131 showed an example using refresh. Examples of inetd debugging mode and the traceson and tracesoff commands are provided in section 6.4.3, “Debugging the Subsystem and Its Subservers” on page 137.

```
# stopsrc -s inetd
0513-044 The stop of the /usr/sbin/inetd Subsystem was completed successfully.

# startsrc -s inetd
0513-059 The inetd Subsystem has been started. Subsystem PID is 5560.

# lssrc -s inetd
Subsystem         Group            PID     Status
 inetd            tcpip            5560    active

# lssrc -ls inetd
Subsystem         Group            PID     Status
 inetd            tcpip            5560    active

Debug         Inactive

Signal        Purpose
 SIGALRM      Establishes socket connections for failed services
 SIGHUP       Rereads configuration database and reconfigures services

 SIGCHLD      Restarts service in case the service dies abnormally

Service       Command                  Arguments                Status
 ttdbserverd  /usr/dt/bin/rpc.ttdbserverd rpc.ttdbserverd 100083   active
 time         internal                                           active
 daytime      internal                                           active
 chargen      internal                                           active
 discard      internal                                           active
 time         internal                                           active
 daytime      internal                                           active
 chargen      internal                                           active
 discard      internal                                           active
 echo         internal                                           active
 pcnfsd       /usr/sbin/rpc.pcnfsd     pcnfsd 150001 1-2         active
 rwalld       /usr/lib/netsvc/rwall/rpc.rwalld rwalld 100008 1          active
 rusersd      /usr/lib/netsvc/rusers/rpc.rusersd rusersd 100002 1-2       active

 rstatd       /usr/sbin/rpc.rstatd     rstatd 100001 1-3         active
 ntalk        /usr/sbin/talkd          talkd                     active
 exec         /usr/sbin/rexecd         rexecd                    active
 login        /usr/sbin/rlogind        rlogind                   active
 telnet       /usr/sbin/telnetd        telnetd                   active
 ftp          /usr/sbin/ftpd           ftpd                      active
```

Figure 72. Starting, Stopping, and Obtaining Status from inetd *with the SRC*

6.4.2 Starting, Stopping, and Listing Status of Subservers

The stopsrc, startsrc, and lssrc commands can be applied to a particular inetd subserver. This is accomplished by using the -t flag to identify a subserver, instead of using the -s flag to identify a subsystem. When the stopsrc command is applied to an inetd subserver, the definition of the corresponding Internet service is commented out in the inetd configuration file, and the inetd daemon is forced to reread its configuration file. This disables the inetd support for the

Internet service. If the stop request made is subserver stop normal (-f is not specified), processes already running to provide the Internet service to clients are not affected, but new requests for the service are rejected. For example, when the commands in Figure 73 are executed, current telnet sessions are not affected, but new telnet sessions cannot be established. If the stop request made is subserver stop forced (-f is specified), processes already running to provide the Internet service to clients are terminated[33], and new requests for the service are rejected.

```
# grep "telnet" /etc/inetd.conf
telnet  stream  tcp     nowait  root    /usr/sbin/telnetd      telnetd

# lssrc -ls inetd | grep telnet
 telnet       /usr/sbin/telnetd        telnetd                  active

# ps -e | grep telnet
  6366      -  1:19 telnetd
  7276      -  0:01 telnetd
  7802      -  0:03 telnetd
  9568      -  0:00 telnetd
 10352      -  0:01 telnetd

# stopsrc -t telnet
0513-127 The telnet subserver was stopped successfully.

# grep "telnet" /etc/inetd.conf
#telnet  stream  tcp     nowait  root    /usr/sbin/telnetd      telnetd

# lssrc -ls inetd | grep telnet

# ps -e | grep telnet
  6366      -  1:19 telnetd
  7276      -  0:01 telnetd
  7802      -  0:03 telnetd
  9568      -  0:00 telnetd
 10352      -  0:01 telnetd

# telnet $(hostname)
Trying...
telnet: connect: Connection refused
telnet> quit
```

Figure 73. Using stopsrc *to Disable* inetd *Support for an Internet Service*

When the startsrc command is applied to an inetd subserver, the definition of the corresponding Internet service is uncommented in the inetd configuration file, and the inetd daemon is forced to reread its configuration file. This re-enables the inetd support for the Internet service. For example, the inetd support for the telnet Internet service is re-enabled as shown in Figure 74 on page 137.

[33] This is the intended behavior, but not the current behavior. Currently, the subserver stop forced request does not terminate processes providing the Internet service to clients, because of a defect in inetd. The defect has been reported.

```
# startsrc -t telnet
0513-124 The telnet subserver has been started.

# grep "telnet" /etc/inetd.conf
telnet  stream  tcp     nowait  root    /usr/sbin/telnetd      telnetd

# lssrc -ls inetd | grep telnet
 telnet       /usr/sbin/telnetd        telnetd                 active

# telnet $(hostname)

AIX Version 4
(C) Copyrights by IBM and by others 1982, 1994.
login:
```

Figure 74. Using startsrc *to Enable* inetd *Support for an Internet Service*

When the lssrc command is applied to a particular inetd subserver, only information about the particular subserver is displayed. The information displayed about the subserver is no different than what is displayed when lssrc -l is applied to the inetd subsystem[34].

6.4.3 Debugging the Subsystem and Its Subservers

The inetd subsystem supports the SRC subsystem trace requests. When the traceson command is applied to the inetd subsystem, the subsystem goes into debugging mode. As discussed in 5.1.5, "The traceson and tracesoff Commands" on page 58, inetd logs messages when in debugging mode. It will also turn on the SO_DEBUG socket option for sockets it creates while in debugging mode. The information traced for a socket with the SO_DEBUG option turned on can be displayed with the trpt command.

Figure 75, part **1** shows an example where telnet socket activity is traced. The first lssrc command shows that the inetd subsystem is not in debugging mode. Then the traceson command is used to tell the inetd subsystem to go into debugging mode. The second lssrc command shows that the inetd subsystem is in debugging mode. Whether by design or accident, inetd has not actually turned on the SO_DEBUG socket option for any of its sockets yet. Once inetd is in debugging mode, the Internet service of interest must be disabled and re-enabled to cause inetd to turn on the SO_DEBUG socket option for the socket used to support the service. This is done with the stopsrc and startsrc commands; the telnet subserver is stopped, then started. Next, the trpt command is executed to follow TCP tracing records. Tracing records are generated when a telnet request is received.

[34] In AIX 3.2.5, when lssrc is applied to an inetd subserver, information is displayed about each process currently running that is supporting a client of the corresponding Internet service. The information includes the PID of the process, and the hostname and IP address on which the client is running. This information disappeared in AIX 4.1. This has been reported as a defect.

```
1
# lssrc -ls inetd | grep "^Debug"
Debug         Inactive

# traceson -s inetd
0513-091 The request to turn on tracing was completed successfully.

# lssrc -ls inetd | grep "^Debug"
Debug         Active

# stopsrc -t telnet
0513-127 The telnet subserver was stopped successfully.

# startsrc -t telnet
0513-124 The telnet subserver has been started.

# trpt -f -s -a

124 ESTABLISHED:user RCVD -> ESTABLISHED
        rcv_nxt 5ceb762d rcv_wnd 4000 snd_una 4934004e snd_nxt 493400cd snd_max
493400cd
        snd_wl1 5ceb762a snd_wl2 4934004e snd_wnd 4000
124 ESTABLISHED:user SEND -> ESTABLISHED
        rcv_nxt 5ceb762d rcv_wnd 4000 snd_una 4934004e snd_nxt 493400cd snd_max
493400cd
        snd_wl1 5ceb762a snd_wl2 4934004e snd_wnd 4000
129 ESTABLISHED:input (src=9.117.10.14,4571,
dst=129.40.93.113,23)5ceb762d@49340
0cd(win=4000)<ACK> -> ESTABLISHED
        rcv_nxt 5ceb762d rcv_wnd 4000 snd_una 493400cd snd_nxt 493400cd snd_max
493400cd
        snd_wl1 5ceb762d snd_wl2 493400cd snd_wnd 4000
129 ESTABLISHED:output (src=129.40.93.113,23,
dst=9.117.10.14,4571)[493400cd..49
3400d0)@5ceb762d(win=4000)<ACK,PUSH> -> ESTABLISHED
        rcv_nxt 5ceb762d rcv_wnd 4000 snd_una 493400cd snd_nxt 493400d0 snd_max
493400d0
        snd_wl1 5ceb762d snd_wl2 493400cd snd_wnd 4000
149 ESTABLISHED:input (src=9.117.10.14,4571,
dst=129.40.93.113,23)5ceb762d@49340
0d0(win=4000)<ACK> -> ESTABLISHED
        rcv_nxt 5ceb762d rcv_wnd 4000 snd_una 493400d0 snd_nxt 493400d0 snd_max
493400d0
        snd_wl1 5ceb762d snd_wl2 493400d0 snd_wnd 4000
```

Figure 75 (Part 1 of 2). Using the inetd *Debug Mode*

```
2
# tracesoff -s inetd
0513-093 The request to turn off tracing was completed successfully.

# lssrc -ls inetd | grep "^Debug"
Debug         Inactive

# stopsrc -t telnet
0513-127 The telnet subserver was stopped successfully.

# startsrc -t telnet
0513-124 The telnet subserver has been started.
```

Figure 75 (Part 2 of 2). Using the inetd *Debug Mode*

Figure 75, part 2 shows the inetd subsystem being taken out of debugging mode. The tracesoff command is applied to the inetd subsystem to get it out of debug mode. Then the telnet subserver is disabled and re-enabled with the stopsrc and startsrc commands. This causes inetd to re-create the socket supporting the telnet service; the re-created socket will not have the SO_DEBUG socket option turned on. It should be noted that telnet sessions that were started when inetd was in debugging mode will continue to generate TCP trace records.

For more information about TCP tracing records see the trpt and netstat man pages in *AIX Version 4.1: Command Reference*, SBOF-1851.

6.5 Programming a Daemon under inetd Control: A Word of Caution

It is not unusual for a daemon under inetd control to make itself a session leader or a process group leader. This might be attempted with code similar to that in Figure 76 on page 140. This approach will work when inetd is not running in debug mode. But, when inetd is running in debug mode, the call to setsid will fail. This is because when inetd is in debug mode, it makes a created process a session leader before issuing exec for the daemon program. If the daemon program executes the code fragment in Figure 76 on page 140, the call to setsid will fail, and the daemon will exit. The setsid function call fails because the process is a process group leader. Similar problems occur when using other functions that attempt to make the process a process group leader.

```
#include <sys/types.h>
#include <unistd.h>

      .
      .
      .

      if (setsid() == (pid_t)-1) {
            exit(1);
      }

      .
      .
      .
```

Figure 76. Improper Method to Make an inetd *Process a Session Leader*

It is recommended that the approach taken in 3.4, “Disassociate the Daemon from Its Controlling Terminal” on page 25 be used. In that section, code is shown that calls setsid, ignores the return code from setsid, and then confirms that the process is a session leader without a controlling terminal.

Chapter 7. Signal Handling

On UNIX systems, dealing with signals is not peculiar to daemons. However, programming a daemon tends to require a more sophisticated use of signals than other types of programs. The improper handling of signals can negatively affect a daemon's reliability. This chapter discusses some considerations to allow for more reliable handling of signals.

This chapter is not a comprehensive tutorial about signal handling. Chapter 10, "Signals" in *Advanced Programming in the UNIX Environment*, W. Richard Stevens, 1992, Addison-Wesley Publishing Company is recommended, as are the XPG4 man pages for the signal-related routines.

7.1 Choosing among Signal Routines

Signal-related routines have undergone much change on UNIX systems. Efforts to improve the reliability of signal handling have introduced additional signal-related routines. The desire to support old programs results in the continued existence of obsolete routines. As a result, there is a large collection of signal-related routines on modern UNIX systems.

A modern UNIX system might have four or more collections of signal-related routines. Which signal-related routines should a daemon writer use?

7.1.1 The `signal` Routine

The first collection is composed of one routine, `signal`. This is the oldest signal-related routine available. It has a simple interface, which makes it tempting to use. However, it has some serious problems. The major problems with the `signal` routine are:

- When an installed signal handler is called, the disposition of the signal for which it is called is changed back to the default. If the signal is delivered to the process again before the signal handler is re-installed, the default action for the signal will take place. This is often process termination.
- There is no mechanism to block the delivery of a signal.

Figure 77 on page 142 shows the source code for an example program, `signal_user`, which uses the `signal` routine to install a signal handler. The `main` routine installs `sig_handler` as the signal handler for the `SIGUSR1` signal, and sits in a loop forever. When a process running this program receives the `SIGUSR1` signal, the `sig_handler` routine executes, doing some busy work. If the `SIGUSR1` signal is delivered to the process again, the process will terminate, because the default disposition of the `SIGUSR1` signal is to terminate the process. This is shown in Figure 78 on page 143.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

#define HOW_BUSY   10000000

char msg_in[]  = "Entered the signal handler.\n";
char msg_out[] = "Leaving the signal handler.\n";

int  busy_work[HOW_BUSY];


void sig_handler (int signo)
{
        int i;


        write(STDOUT_FILENO, msg_in, sizeof msg_in - 1);

        for (i = 0; i < HOW_BUSY; i++) {
                busy_work[i] = i;
        }

        write(STDOUT_FILENO, msg_out, sizeof msg_out - 1);

        return;
}


int main(int argc, char **argv)
{

        if (signal(SIGUSR1, sig_handler) == SIG_ERR) {
                exit(1);
        }

        while (1 == 1) {
                sleep(60);
        }

        return 0;
}
```

Figure 77. Source for the signal_user *Program*

```
$ ./signal_user &
[2]     8982
$
$ kill -USR1 8982
$ Entered the signal handler.

$ kill -USR1 8982
$
[2] + User signal 1            ./signal_user &
```

Figure 78. Running the signal_user *Program*

Most signal handlers installed by the signal routine will re-install the signal handler immediately. This is shown in the source code in Figure 79 on page 144 for the program signal_user2. This makes it less likely that a signal reverting to its default disposition will cause the program to terminate. However, this can still happen. In signal_user2, there is still a period of time in sig_handler in which the signal has its default disposition. In Figure 80 on page 145, signal_user2 is started, and then bombarded with the SIGUSR1 signal. The process running the signal_user2 program continues to run as long as the SIGUSR1 signal is delivered when sig_handler is installed as the SIGUSR1 signal handler. Eventually, the SIGUSR1 signal is delivered when sig_handler is not installed as the SIGUSR1 signal handler, and the process terminates.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

#define HOW_BUSY   10000000

char msg_in[]  = "Entered the signal handler.\n";
char msg_out[] = "Leaving the signal handler.\n";

int  busy_work[HOW_BUSY];


void sig_handler (int signo)
{
        int i;


        (void) signal(SIGUSR1, sig_handler);

        write(STDOUT_FILENO, msg_in, sizeof msg_in - 1);

        for (i = 0; i < HOW_BUSY; i++) {
                busy_work[i] = i;
        }

        write(STDOUT_FILENO, msg_out, sizeof msg_out - 1);

        return;
}


int main(int argc, char **argv)
{

        if (signal(SIGUSR1, sig_handler) == SIG_ERR) {
                exit(1);
        }

        while (1 == 1) {
                sleep(60);
        }

        return 0;
}
```

Figure 79. Source for the signal_user2 *Program*

```
$ ./signal_user2 &
[1]     9564

$ while true; do kill -USR1 9564; done

Entered the signal handler.
Entered the signal handler.
Entered the signal handler.

      .
      .
      .

Entered the signal handler.
Entered the signal handler.
Entered the signal handler.
Entered the signal handler.
Entered the signal handler.
Entered the signal handler.
Entered the signal handler.
Entered the signal handler.
Entered the signal handler.


[1] + User signal 1             ./signal_user2 &
$
```

Figure 80. Running the signal_user2 *Program*

Notice in Figure 80 that while the SIGUSR1 signal handler is executing, another occurrence of the signal can cause the signal handler to run again, interrupting the first execution of the signal handler. This behavior is not normally desirable.

Other problems with the signal routine are:

- The current disposition of a signal cannot be checked without changing its disposition.
- There is no way to control whether interrupted system calls are automatically restarted.

Due to the serious problems associated with the signal routine, it is generally described as unreliable. It is not recommended for use in new programs. The sigaction routine, and its associated routines, are recommended for new programs. See 7.1.4, “The sigaction Routine and Related Routines” on page 146.

7.1.2 The sigset Routine and Related Routines

System V systems provide the sigset routine to install a signal handler. The signal handler remains installed unless the program explicitly changes the disposition of the signal. Since the signal’s disposition is not automatically restored to its default, the most serious reliability problem associated with the signal routine is eliminated. Furthermore, while the signal handler is executing, the signal whose delivery caused the signal handler to execute is blocked from further delivery until the signal handler finishes.

Associated with the sigset routine are the sigignore, sighold, sigrelse, and sigpause routines. The sigignore routine provides an alternative method for setting the disposition of a signal to be ignored. The sighold, sigrelse, and sigpause routines allow the program to control when signals are blocked, that is, temporarily deferred, from delivery.

These routines are an improvement over the signal routine. However, it is recommended that new programs use the sigaction routine and its associated routines. See 7.1.4, “The sigaction Routine and Related Routines.”

7.1.3 The sigvec Routine and Related Routines

BSD systems provide the sigvec routine to install a signal handler. The sigvec routine provides the same improvements over signal that sigset does. In addition, it allows for control over whether restartable interrupted system calls are restarted. See 7.6, “Automatic Restart of Interrupted System Calls” on page 175 for a discussion of restarting interrupted system calls.

Associated with the sigvec routine are the sigblock, sigsetmask, and sigpause routines. The sigblock, sigsetmask, and sigpause routines allow the program to control when signals are blocked, that is, temporarily deferred, from delivery. However, only signals 1 through 31 can be blocked by these routines.

These routines are an improvement over the signal routine. However, it is recommended that new programs use the sigaction routine and its associated routines. See the next section, 7.1.4, “The sigaction Routine and Related Routines.”

7.1.4 The sigaction Routine and Related Routines

The sigaction routine is the POSIX-specified way to install reliable signal handlers. It overcomes the problems associated with the signal routine. A signal handler installed for a signal remains installed until the signal’s disposition is explicitly changed by the program. Furthermore, while the signal handler is executing, the signal whose delivery caused the signal handler to execute is blocked from further delivery until the signal handler finishes. An additional set of signals to be blocked during the execution of the signal handler can be specified when sigaction is used to install a signal handler. The sigaction routine also allows for the specification of flags that control how the signal handler behaves. For example, one flag controls whether restartable interrupted system calls are restarted automatically. The sigaction routine can also be used to check the current disposition of a signal, without changing its disposition.

Associated with the sigaction routine are the sigprocmask, sigpending, and sigsuspend routines. These routines allow the program to control when signals are blocked, that is, temporarily deferred, from delivery. This is done by setting the process signal mask. The process signal mask is a signal set that controls the delivery of signals to the process. Signal sets, in general, are manipulated by the routines sigemptyset, sigfillset, sigaddset, sigdelset, and sigismember. See the man pages for these routines for all the details.

Figure 81 on page 147 shows the source code for an example program, sigaction_user. This program is similar to the signal_user program shown in Figure 77 on page 142, except it uses the sigaction routine to install a signal handler instead of the signal routine. The main routine installs sig_handler as

the signal handler for the SIGUSR1 signal, and sits in a loop forever. When a process running this program receives the SIGUSR1 signal, the sig_handler routine executes, doing some busy work. If the SIGUSR1 signal is continuously delivered to the process, the process will execute the sig_handler signal handler once at a time, and it will never terminate due to the delivery of the SIGUSR1 signal. This is shown in Figure 82 on page 148.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

#define HOW_BUSY   10000000

char msg_in[]  = "Entered the signal handler.\n";
char msg_out[] = "Leaving the signal handler.\n";

int  busy_work[HOW_BUSY];


void sig_handler (int signo)
{
        int i;


        write(STDOUT_FILENO, msg_in, sizeof msg_in - 1);

        for (i = 0; i < HOW_BUSY; i++) {
                busy_work[i] = i;
        }

        write(STDOUT_FILENO, msg_out, sizeof msg_out - 1);

        return;
}


int main(int argc, char **argv)
{
        struct sigaction sa;

        sa.sa_handler = sig_handler;
        (void) sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;

        if (sigaction(SIGUSR1, &sa, NULL) == -1) {
                exit(1);
        }

        while (1 == 1) {
                sleep(60);
        }

        return 0;
}
```

Figure 81. Source for the sigaction_user *Program*

```
$ ./sigaction_user &
[1]     9624

$ while true; do kill -USR1 9624; done
Entered the signal handler.
Leaving the signal handler.
Entered the signal handler.
Leaving the signal handler.

      .
      .
      .

Entered the signal handler.
Leaving the signal handler.
Entered the signal handler.
Leaving the signal handler.
Entered the signal handler.
^C

$ ps -p 9624
   PID    TTY  TIME CMD
  9624  pts/0  30:48 sigaction_user
$
```

Figure 82. Running the sigaction_user *Program*

7.1.5 The Signal Routines and AIX

AIX supports all the signal routines mentioned above. It also supports variations of some of the routines. These routines are in the libc.a, libc_r.a, and libbsd.a libraries. The libbsd.a library holds BSD versions of routines also held in libc.a. The libc.a and libc_r.a libraries hold all the other routines. The routines in libc.a are not thread-safe, while the routines in libc_r.a are thread-safe.

A program desiring the System V functionality of the signal routine should be linked without the libbsd.a library, and with the libc.a or libc_r.a library. The libbsd.a library includes a BSD version of the signal routine. This routine does not have many of the faults of the System V version of signal. The disposition of a signal whose signal handler was installed with the BSD version of signal does not revert to the default when the signal handler is invoked. Furthermore, the signal that caused a signal handler to be invoked is blocked while the signal handler is running.

A program using the System V signal routines sigset, sigignore, sighold, sigrelse, and sigpause should be linked with the libc.a or libc_r.a libraries. Linking with the libbsd.a library will have no effect on these routines.

A program using the BSD signal routines sigvec, sigblock, sigsetmask, and sigpause should be linked with the libbsd.a library. These routines are 4.3BSD compatible. A program that calls sigvec but is not linked with the libbsd.a library will not be using a 4.3BSD compatible version of sigvec. It will be using a version of sigvec compatible with an old AIX version of sigvec. The old AIX

version of sigvec does not support automatically restarting interrupted system calls. The 4.3BSD compatible version of sigvec in libbsd.a allows the caller to specify whether automatically restartable system calls are automatically restarted or not.

A program using the POSIX-compliant signal routines sigaction, sigprocmask, sigpending, sigsuspend, sigemptyset, sigfillset, sigaddset, sigdelset, and sigismember should be linked with the libc.a or libc_r.a library. The functionality of these routines will not be affected if the libbsd.a library is linked with the program.

The standards warn against using different sets of signal-related routines to change the disposition of the same signal in the same process. The standards say the effects are undefined. This problem is mitigated to some degree in AIX, because the two versions of the signal routine, the two versions of the sigvec routine, and the sigset routine are all implemented by calling the sigaction routine. This is fortunate for daemons. The init process uses sigaction to set the disposition of signals. The srcmstr process uses the sigvec routine in libc.a to set signal dispositions. Finally, the inetd process uses the sigvec routine in libbsd.a to set signal dispositions. The daemon support routines described in Chapter 10, "Daemon Support Routines" on page 245 use the sigaction routine to set signal dispositions. This should work correctly whether the daemon was started by init, srcmstr, inetd, or a shell. However, it is still recommended to keep the mixed use of these routines to a minimum.

On AIX, if the disposition of any signal is changed by sigset or sighold, the process runs with the "sigset signal handling style" until it calls one of the exec functions. A child process will inherit the "sigset signal handling style" from its parent process. In a process running with the "sigset signal handling style," a call to longjmp will not restore the process signal mask to the state it was in when setjmp was called. In a process running without "sigset signal handling style," a call to longjmp will restore the process signal mask to the state it was in when setjmp was called.

7.2 Atomic Access to a Variable

It is common for a signal handler to communicate with another signal handler, or with the portion of the program not executed by a signal handler, using variables in the program's global memory. However, a program cannot generally assume that it can atomically access a global variable without being interrupted by the execution of a signal handler. A program can block the signals for which signal handlers have been installed to protect access to global variables (see 7.3, "Signal Handlers and Critical Sections" on page 154). However, there is one data type for which blocking signals is not required.

The ANSI C standard defines a data type called sig_atomic_t[35]. A variable of type sig_atomic_t can be accessed atomically. That is, the value of the variable can be fetched, or a new value can be stored into the variable, without interruption, even by the execution of a signal handler.

For purposes of portability, a program should assume that the only variables that can be accessed atomically are those defined with type sig_atomic_t. The

[35] As does the POSIX and X/OPEN standards.

sig_atomic_t type is defined in signal.h. The sig_atomic_t type is required by ANSI C to be "... the integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts"[36]. The phrase "integral type" means a variable of type sig_atomic_t can hold an integer value. Since the ANSI C standard does not specify the size of the "integral type," a portable program should assume it is the smallest possible integral type, a signed character. Since it is not specified whether the "integral type" is signed or unsigned, a portable program should assume that a variable of type sig_atomic_t cannot hold negative values. ANSI C requires an implementation complying with the standard to define a constant, named SCHAR_MAX in limits.h, that specifies the maximum value of a signed character in the implementation. So, a portable program can assume that a variable of type sig_atomic_t can hold an integer value ranging from 0 to SCHAR_MAX, inclusive. The ANSI C standard also requires that the value of SCHAR_MAX be at least 127. Therefore, a variable of type sig_atomic_t can portably hold a value from 0 to 127, inclusive.

A variable that is modified in signal handling routines and accessed in non-signal handling routines should be defined with the type modifier volatile, letting the compiler know that the value of the variable can change asynchronously.

The standards only guarantee that a signal handler can store a value into a global variable declared to be of type volatile sig_atomic_t. They do not define what will happen if the value of such a variable is fetched in a signal handler. The XPG4 standard states:

> If the signal occurs other than as the result of calling abort(), kill() or raise(), the behaviour is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above[37] or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile sig_atomic_t.

Figure 83 on page 151 shows an example using a variable of type volatile sig_atomic_t. The value of the variable stop_requested is initialized to 0, tested in the main routine, and set to 1 in the signal handler named sig_handler. Because the variable is of type sig_atomic_t, the main routine will never see an intermediate value for stop_requested on any system complying with the ANSI C standard. It might seem reasonable to declare stop_requested as type int, assuming that a reasonable machine would be able to access a variable of that type atomically, but ANSI C does not guarantee that.

[36] The ANSI C Standard, Section 7.7.

[37] These functions are listed in this document in Figure 86 on page 156.

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


volatile sig_atomic_t stop_requested = 0;


void sig_handler (int signo)
{
      stop_requested = 1;
      return;
}


int main(int argc, char **argv)
{
      struct sigaction sa;

      sa.sa_handler = sig_handler;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;

      (void) sigaction(SIGTERM, &sa, NULL);

      while (stop_requested == 0) {
            fputs("Process is still running.\n", stdout);
            sleep(10);
      }
      fputs("Process is terminating.\n", stdout);

      return 0;
}
```

Figure 83. Example of a Typical Use of a `sig_atomic_t` *Variable*

If a global variable of a type other than `sig_atomic_t` is modified within a signal handler and accessed elsewhere, and it is important that the program not see an intermediate value, access to the variable should be synchronized in some manner. Using `sigprocmask` to temporarily block signals while accessing the variable outside of the signal handler is a valid approach.

7.2.1 A Problem with `sig_atomic_t` on AIX

If the ANSI language level of the C compiler is used on AIX to compile the program in Figure 83, the compiler will generate an error message. If the extended language level of the C compiler is used, the program will just generate a warning message. If the program in Figure 83 is in a source file named `atomic.c`, Figure 84 on page 152 shows the results of attempts to compile the program. The first attempt to compile the program specifies that the ANSI level of the compiler should be used (`-qlanglvl=ansi` is specified). The compilation fails with an error. The second attempt to compile the program specifies that the extended level of the compiler should be used (`-qlanglvl=extended` is specified). The compilation succeeds, but a warning message is generated.

```
$ xlc -D_ALL_SOURCE -o atomic atomic.c -qlanglvl=ansi -qhalt=e
"atomic.c", line 7.17: 1506-112 (E) Duplicate type qualifier volatile ignored.
$ echo $?
1
$ ls -l atomic
atomic not found
$
$ xlc -D_ALL_SOURCE -o atomic atomic.c -qlanglvl=extended -qhalt=e
"atomic.c", line 7.17: 1506-112 (W) Duplicate type qualifier volatile ignored.
$ echo $?
0
$ ls -l atomic
-rwxr-xr-x   1 agar     staff       3961 May  2 16:46 atomic
```

Figure 84. AIX Problem with sig_atomic_t *Type*

The problem in AIX with variables declared to be of type volatile sig_atomic_t occurs as a result of how sig_atomic_t is defined in AIX. The details are described in a defect opened against AIX. The text of that defect follows:

SUMMARY

On AIX, in <sys/signal.h>, the type sig_atomic_t is defined as

```
typedef volatile int sig_atomic_t;
```

The inclusion of the type qualifier "volatile" in the definition of sig_atomic_t violates the ANSI C standard. The problem can be fixed by defining sig_atomic_t as:

```
typedef int sig_atomic_t;
```

RATIONALE

In "Section 7.7.1.1 The signal function" of the ANSI C standard, the following paragraph appears:

"If the signal occurs other than as the result of calling the abort or raise function, the behavior is undefined if the signal handler calls any function in the standard library other than the signal function itself . . . or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile sig_atomic_t. . . . "

This paragraph indicates that a program that is to modify a variable of static storage duration within a signal handler should define the variable with the type "volatile sig_atomic_t". A variable "var" should be defined as:

```
static volatile sig_atomic_t var;
```

On an AIX system, because of how sig_atomic_t is defined, this is comparable to:

```
static volatile volatile int var;
```

However this definition is invalid, according to the ANSI C standard. The standard states in “Section 6.5.3 Type qualifiers”: “The same type qualifier shall not appear more than once in the same specifier list or qualifier list, either directly or via one or more typedefs.” So, the definition that is required in section 7.7.1.1, is invalid on an AIX system because of the constraint placed on type qualifiers in section 6.5.3. If the “volatile” type qualifier is removed from the definition of sig_atomic_t in <sys/signal.h> on AIX, the definition that is required by section 7.7.1.1 will be valid on an AIX system.

EXAMPLES

Here is some code in a source file x.c:

```
#include <signal.h>

static volatile sig_atomic_t var;

int main(int argc, char **argv)
{
       return 0;
}
```

Here is an attempt to compile this code failing:

```
$ xlc -D_ANSI_C_SOURCE -o x x.c -qlanglvl=ansi -qhalt=e
"x.c", line 3.17: 1506-112 (E) Duplicate type qualifier volatile ignored.
$ echo $?
1
```

Here is source code in a file x2.c,
that defines sig_atomic_t without volatile:

```
typedef int sig_atomic_t;

static volatile sig_atomic_t var;

int main(int argc, char **argv)
{
        return 0;
}
```

The code can be compiled:

```
$ xlc -D_ANSI_C_SOURCE -o x2 x2.c -qlanglvl=ansi -qhalt=e
$ echo $?
0
```

AIX VERSION

Found on AIX 4.1.2, and AIX 3.2.5.

The response to this defect is that the problem is real, but it will not be fixed now, because it may cause existing code to fail.

7.3 Signal Handlers and Critical Sections

It may not be safe for a signal handler to be called at any time. For example, a program may manipulate global data more complicated than `sig_atomic_t` data within and outside of signal handlers. It may not be safe to allow a signal handler to execute when the program is in the middle of manipulating the global data. A program can control when a signal handler is allowed to execute by manipulating the process signal mask. This can be done with the `sigprocmask` routine. When the program is in a critical section which is manipulating the global data, signal handlers that will manipulate the same data should be blocked from execution.

A signal is said to be *generated* when the event that causes the signal occurs. When action is taken by a process in response to the signal, like terminating or executing a signal handler, the signal is said to be *delivered*. Between the time that a signal is generated and delivered, the signal is said to be *pending*. A signal does not stay pending for long, unless it is *blocked*. A blocked signal that is pending remains pending until the signal is unblocked and delivered, or until the signal disposition is changed such that it is ignored. A signal is blocked and unblocked with the `sigprocmask` routine. A signal can also be blocked when a signal handler is executed. The signals that are blocked when a signal handler is executing are specified when the signal handler is installed. It should be noted that when a signal is generated for a process multiple times while the signal is blocked by the process, it is up to the system implementation to decide whether the signal is delivered multiple times when the signal is unblocked. A portable application should not assume the signal will be delivered once, nor should it assume the signal will be delivered more than once.

Figure 85 on page 155 presents an example. The example assumes that the `main` routine and the signal handlers for the `SIGUSR1` and `SIGUSR2` signals will manipulate a global linked list. Manipulating a linked list typically involves several steps that must be performed without interruption. If the manipulation of a linked list is interrupted, the linked list may become corrupted. The example protects the linked list in two ways. First, when the signal handlers are installed, the `sa_mask` field of the `sigaction` structure is set such that the `SIGUSR1` and `SIGUSR2` signal handlers cannot execute concurrently. When the `SIGUSR1` or `SIGUSR2` signal handler is executing, the `SIGUSR1` and `SIGUSR2` signals will be blocked. Second, before the non-signal handling code manipulates the global linked list, it sets the process signal mask such that the `SIGUSR1` and `SIGUSR2` signals are blocked. Once the linked list has been manipulated, the signal mask is reset to its previous value.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void usr1_handler(int signo)
{
      /* ... Examine or manipulate the global linked list ... */
      return;
}

void usr2_handler(int signo)
{
      /* ... Examine or manipulate the global linked list ... */
      return;
}


int main(int argc, char **argv)
{
      struct sigaction sa;
      sigset_t crit_mask, saved_mask;

      (void) sigemptyset(&crit_mask);
      (void) sigaddset(&crit_mask, SIGUSR1);
      (void) sigaddset(&crit_mask, SIGUSR2);

      sigemptyset(&sa.sa_mask);
      (void) sigaddset(&sa.sa_mask, SIGUSR1);
      (void) sigaddset(&sa.sa_mask, SIGUSR2);
      sa.sa_flags = SA_RESTART;

      sa.sa_handler = usr1_handler;
      (void) sigaction(SIGUSR1, &sa, NULL);
      sa.sa_handler = usr2_handler;
      (void) sigaction(SIGUSR2, &sa, NULL);

      .
      .
      .
      (void) sigprocmask(SIG_BLOCK, &crit_mask, &saved_mask);
      /* ... Manipulate the global linked list ... */
      (void) sigprocmask(SIG_SETMASK, &saved_mask, NULL);
      .
      .
      .

      return 0;
}
```

Figure 85. Protecting Critical Sections

A program may also have to manipulate the process signal mask because of the routines being used by the signal handlers and the non-signal handling code. See 7.4, “Signal Handlers and Signal-Safe Routines” on page 156.

7.4 Signal Handlers and Signal-Safe Routines

It should not be assumed that it is safe to call any routine from a signal handling routine. In fact, most routines are probably unsafe to call from a signal handling routine. Routines tend to be unsafe to call from a signal handling routine because signals can be delivered at any time, unless blocked by the process signal mask, and many routines are not reentrant. For example, a routine may use static data to hold intermediate results. If such a routine is interrupted by the execution of a signal handler that calls the routine again, one or both executions of the routine may not behave as expected. Many routines manipulate global data. If such a routine is interrupted by the execution of a signal handler that calls any other routine that manipulates the same global data, unexpected results may occur.

7.4.1 The Official Signal-Safe Routines

The POSIX 1003.1 and X/OPEN standards define sets of system routines that compliant systems must implement in a reentrant fashion or in such a way that they are not interruptible by the delivery of signals. The routines in the set can safely be called from signal handling routines, without restriction. The set of routines specified by POSIX 1003.1 is shown in Figure 86. The XPG3 standard adds `abort`, `chroot`, `exit`, and `longjmp` to this set. However, those four routines are not listed as safe by the XPG4 standard, so they should probably be avoided. The XPG4 standard adds the `fpathconf` and `raise` routines to the POSIX 1003.1 set of signal-safe routines.

```
access        getegid       rmdir         tcflow
alarm         geteuid       setgid        tcflush
cfgetispeed   getgid        setpgid       tcgetattr
cfgetospeed   getgroups     setsid        tcgetprgp
cfsetispeed   getpgrp       setuid        tcsendbreak
cfsetospeed   getpid        sigaction     tcsetattr
chdir         getppid       sigaddset     tcsetpgrp
chmod         getuid        sigdelset     time
chown         kill          sigemptyset   times
close         link          sigfillset    umask
creat         lseek         sigismember   uname
dup2          mkdir         signal        unlink
dup           mkfifo        sigpending    ustat
execle        open          sigprocmask   utime
execve        pathconf      sigsuspend    wait
_exit         pause         sleep         waitpid
fcntl         pipe          stat          write
fork          read          sysconf
fstat         rename        tcdrain
```

Figure 86. Signal-Safe Routines As Specified by POSIX 1003.1

An application routine can be implemented to be safe with respect to signals, if it is implemented in a reentrant fashion, or such that it is not interruptible by the delivery of a signal. If the application routine is interruptible by signals, it must restrict its calls to system routines to the defined set of signal-safe system routines.

The standards state that if any routine that is unsafe with respect to signals is interrupted by a signal handler routine that then calls any unsafe routine, the results are undefined. The program is not guaranteed to work as expected, and the results may vary from system to system. Any problems that arise are likely to be intermittent, and may terminate the process. A program can protect itself from these types of problems with either of the following approaches:

- Signal handler routines can be implemented such that they only call routines that are safe with respect to signals.
- The program can insure that an unsafe routine is never interrupted by the execution of a signal handling routine. This may involve blocking the delivery of signals, or the use of some other program synchronization.

Notice that the standard I/O routines, and the `malloc` and `free` routines are not in the set of signal-safe routines in Figure 86 on page 156. These routines typically manipulate global data. The `malloc` and `free` routines manipulate the process heap. The standard I/O routines manipulate global buffers. Many routines are unsafe with respect to signal handling because they call `malloc` and `free`. The standard I/O routines call `malloc` and `free`, which is another reason they are unsafe with respect to signals. For examples showing the unsafe nature of these routines with respect to signal handlers, see 7.4.4, “Examples of Using Signal-Unsafe Routines” on page 158.

7.4.2 Other Signal-Safe Routines

The limited number of routines defined to be signal-safe by POSIX and X/OPEN can be frustrating. An actual UNIX implementation probably has many other system routines that can be considered signal-safe. The possibilities are discussed in this section. However, counting on a system routine to be signal-safe that is not defined to be so by a standard can lead to porting difficulties.

Routines that are known to be signal-unsafe are so because of the way they treat data of static duration in user level. Most of the routines defined as signal-safe by POSIX and X/OPEN are typically implemented as system calls. One could reasonably expect that a system would not implement a system call such that it dealt with static user level data in a fashion that would make the system call signal-unsafe. Therefore, it would be reasonable to treat any system routine implemented as a system call on a particular system as signal-safe on that system.

Take, for example, the `select` routine. The `select` routine is defined as part of XPG4 UNIX Conformance. Yet, it is not included in the set of signal-safe routines. On AIX, the `select` routine is a system call. It would be reasonable to treat `select` as a signal-safe function on AIX. However, there is no guarantee that `select` is implemented as a system call on another system claiming XPG4 UNIX Conformance.

A system like AIX includes many system routines that are not included in any standard. An obvious example is the set of socket routines. The `socket`, `bind`, `listen`, `accept`, and `connect` routines are implemented on AIX as system calls. It seems reasonable to treat these routines as signal-safe on AIX.

7.4.3 Long Jumping and Signal-Safe Routines

The siglongjmp routine is not included in the set of signal-safe routines by POSIX or X/OPEN. Yet, siglongjmp is a valid way to terminate the execution of a signal handler. The reason it is not included in the set of signal-safe routines is that the program may execute signal-unsafe routines after the long jump is made. Since the execution of the signal handler may have interrupted a signal-unsafe routine, the results of the execution of another signal-unsafe routine after the long jump is undefined. It is as if the signal handler had interrupted a signal-unsafe routine and then executed another signal-unsafe routine.

It is valid to leave a signal handler with siglongjmp if the program takes the necessary steps to ensure that the signal handler will not interrupt the execution of a signal-unsafe routine. An example of this is given in 7.9, “Doing the Right Thing with Signals” on page 184. It is also valid to leave a signal handler with siglongjmp if the program takes the necessary steps to ensure that once the siglongjmp routine has been called, the program will not execute a signal-unsafe routine.

The use of siglongjmp to leave a signal handler is preferred to the use of longjmp or _longjmp, because of the differences in how these routines treat the process signal mask. The XPG4 standard specifies that the _longjmp routine does not affect the process signal mask. When _longjmp is used to long jump, the process signal mask remains as it was in the signal handler. The XPG4 standard indicates that it is unspecified whether the longjmp call leaves the process signal mask unchanged, or restores the process signal mask to the value it had when setjmp was called. The XPG4 standard requires that siglongjmp restore the process signal mask to the value it had when sigsetjmp was called, if the sigsetjmp call indicated the state of the process signal mask should be saved. That is usually the desired behavior.

7.4.4 Examples of Using Signal-Unsafe Routines

The absence of the standard I/O routines and the memory allocation routines from the set of signal-safe routines cannot be overemphasized. It is often natural to want to call the standard I/O routines in a signal handler, or to free or allocate memory in the signal handler. However, this can lead to an unreliable program, if not done carefully. Several examples are shown to illustrate the unsafe nature of these routines with respect to signal handling.

Figure 87 on page 159 shows the source code for a program called malloc_test. The main routine installs the catch routine as the signal handler for the SIGUSR1 signal. It then executes a loop that calls the malloc routine to allocate memory, and calls the free routine to free the memory. When the process running malloc_test is sent the SIGUSR1 signal, the catch routine will call the malloc routine to allocate memory, and it will call the free routine to free the memory allocated. Since the malloc and free routines are not considered signal-safe, the routines are called in a signal handler, and the delivery of the SIGUSR1 signal can interrupt the execution of the calls to malloc and free in the main routine; this program is not guaranteed to work. Figure 88 on page 160 shows the malloc_test program failing on an AIX 4.1.2 system. The program is started, and the process running the program is sent the SIGUSR1 signal several times. After the signal is sent a few times, the program terminates with a memory fault. It is likely that the global data malloc and free use to keep track of the state of the process heap became corrupted.

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

void catch(int signo)
{
      char *ptr;

      ptr = malloc(2000);

      if (ptr == NULL) {
            _exit(2);
      }

      free(ptr);
      return;
}

int main(int argc, char **argv)
{
      char *ptr;
      struct sigaction sa;

      sa.sa_handler = catch;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = 0;

      if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            exit(3);
      }

      while (1 == 1) {

            ptr = malloc(2000);

            if (ptr == NULL) {
                  exit(1);
            }

            free(ptr);

      }

      return 0;
}
```

Figure 87. Source Code for the `malloc_test` *Program*

```
$ ./malloc_test &
[1]     6728

$ kill -USR1 6728
$ kill -USR1 6728
$ kill -USR1 6728
$ kill -USR1 6728
$ kill -USR1 6728
$ kill -USR1 6728
[1] + Memory fault(coredump)   ./malloc_test &
$
```

Figure 88. Running the malloc_test *Program*

Figure 89 on page 161 shows the source code for a program called print_test. The structure of this program is similar to the malloc_test program of Figure 87 on page 159: the main routine installs a signal handler for SIGUSR1, and the main routine and the signal handler both call a routine that is not signal-safe, in this case the fprintf routine. Since fprintf is not a signal-safe routine, this program is not guaranteed to work. Figure 90 on page 162 shows the print_test program failing on an AIX 4.1.2 system. The program is executed twice. Both times, the program is started, and the process running the program is bombarded with the SIGUSR1 signal. After about five seconds of this bombardment, the process terminates with a memory fault. The first time the print_test program is run, the standard output stream is buffered by default. The second time the program is run, the standard output stream is unbuffered. The program fails regardless of the buffering state of standard output.

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


void catch(int signo)
{
      fprintf(stdout,
              "It's fine handling signal %2$d in process %1$d.\n",
              getpid(), signo);
      return;
}


int main(int argc, char **argv)
{
      struct sigaction sa;


      sa.sa_handler = catch;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;

      if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            exit(1);
      }

      if ((argc > 1) && (strcmp(argv[1], "-u") == 0)) {
            setbuf(stdout, NULL);
      }

      /*
       * Prevent the program from generating output until the first signal
       * arrives.  This gives the user a chance to type in a command to
       * start sending SIGUSR1 signals to the process.
       */
      (void) sigsuspend(0);

      while (1 == 1) {
            fprintf(stdout, "It's fine being in the %2$s "
                            "of process %1$d.\n", getpid(), "mainline");
      }

      return 0;
}
```

Figure 89. Source Code for the print_test *Program*

```
$ ./print_test &
[1]     9396

$ while true; do kill -USR1 9396; done
      .
      .
      .
It's fine being in the mainline of process 9396.
It's fine being in the mainline of process 9396.
It's fiIt's fine handling signal 30 in process 9396.
ne being in the mainline of process 9396.
It's fine being in the mainlIt's fine handling signal 30 in process 9396.
ine of process 9396.
It's fine being in the mainline of process 9396.
It's fine handling signal 30 in process 9396.
It's fine being in the mainline of process 9396.
It's fine being in the mainline of process 9396.
It's fine being in the mainline of process 9396.
It's fine being in the mainline of process 9396.
It's fine being in the It's fine handling signal 30 in process 9396.
It's fine handling signal 30 in process 9396.

[1] + Memory fault(coredump)   ./print_test &


$ ./print_test -u &
[1]     9088

$ while true; do kill -USR1 9088; done
      .
      .
      .
It's fine handling signal 30 in process 9088.
It's fine handling signal 30 in process 9088.
It's fine being in the mainline of process 9088.
It's fine being in the mainline of process 9088.
It's fine being in the mainline of process 9088.
It's fine being in the It's fine handling signal 30 in process 9088.
It's fine handling signal 30 in process 9088.
mainline of process 9088.
It's fine being in the mainline of process 9088.
It's fine being in the It's fine handling signal 30 in process 9088.
It's fine handling signal 30 in process 9088.

[1] + Memory fault(coredump)   ./print_test -u &
```

Figure 90. Running the print_test *Program*

Figure 91 on page 163 shows the source code for a program called print_test2. This program is a modification of print_test. It performs the same function as print_test, but is implemented to take into account the fact that fprintf is not a signal-safe routine. The main routine uses the sigprocmask routine to prevent the delivery of the SIGUSR1 signal while fprintf is executing. As a result, the catch signal handler never interrupts the fprintf routine. Whenever the fprintf routine is called from the catch routine, any previous call to fprintf is already completed. Figure 92 on page 164 shows the results of running the print_test2

program, and bombarding the process running the program with the SIGUSR1 signal. Notice that the lines printed by the main routine and the catch routine are not mixed up, in contrast to the previous examples. The print_test2 program continues to run until terminated with some other signal.

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


void catch(int signo)
{
      fprintf(stdout,
              "It's fine handling signal %2$d in process %1$d.\n",
              getpid(), signo);
      return;
}


int main(int argc, char **argv)
{
      struct sigaction sa;
      sigset_t block_mask, save_mask;


      sa.sa_handler = catch;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;

      if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            exit(1);
      }

      if ((argc > 1) && (strcmp(argv[1], "-u") == 0)) {
            setbuf(stdout, NULL);
      }

      (void) sigsuspend(0);
      (void) sigemptyset(&block_mask);
      (void) sigaddset(&block_mask, SIGUSR1);

      while (1 == 1) {
            (void) sigprocmask(SIG_BLOCK, &block_mask, &save_mask);
            fprintf(stdout, "It's fine being in the %2$s "
                            "of process %1$d.\n", getpid(), "mainline");
            (void) sigprocmask(SIG_SETMASK, &save_mask, NULL);
      }

      return 0;
}
```

Figure 91. Source Code for the print_test2 *Program*

```
$ ./print_test2 &
[1]     10974

$ while true; do kill -USR1 10974; done
      .
      .
      .

It's fine being in the mainline of process 10974.
It's fine handling signal 30 in process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.
It's fine handling signal 30 in process 10974.
It's fine being in the mainline of process 10974.
It's fine being in the mainline of process 10974.

      .
      .
      .
```

Figure 92. Running the print_test2 *Program*

Of course, an alternative to using the standard I/O routines in a signal handler is to use the open, read, write, lseek, and close system calls. These system calls are in the set of signal-safe routines. Figure 93 on page 165 shows the print_test3 program, which uses the write system call instead of the fprintf routine. This program behaves correctly, even though the SIGUSR1 signal is never explicitly blocked. Figure 94 on page 166 shows the results of running print_test3.

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

char main_msg[80];
int  main_msg_len;
char catch_msg[80];
int  catch_msg_len;


void catch(int signo)
{
      write(STDOUT_FILENO, catch_msg, catch_msg_len);
      return;
}

int main(int argc, char **argv)
{
      struct sigaction sa;


      main_msg_len =  sprintf(main_msg,
                        "It's fine being in the %2$s of process %1$d.\n",
                         getpid(), "mainline");

      catch_msg_len = sprintf(catch_msg,
                       "It's fine handling signal %2$d in process %1$d.\n",
                        getpid(), SIGUSR1);

      sa.sa_handler = catch;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;

      if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            exit(1);
      }

      (void) sigsuspend(0);

      while (1 == 1) {
            write(STDOUT_FILENO, main_msg, main_msg_len);
      }

      return 0;
}
```

Figure 93. Source Code for the print_test3 *Routine*

```
$ ./print_test3 &
[1]     10798

$ while true; do kill -USR1 10798; done
      .
      .
      .

It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine handling signal 30 in process 10798.
It's fine handling signal 30 in process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
It's fine being in the mainline of process 10798.
      .
      .
      .
      .
```

Figure 94. Running the print_test3 *Program*

Figure 95 on page 167 shows the source code for a program called print_malloc_test. This program illustrates two points. First, two signal-unsafe routines that may seem unrelated, fprintf and malloc in this case, may interfere with each other when signal handlers are executed. Second, the execution of one signal handler may be interrupted by another signal handler. In this program, the main routine does not call any signal-unsafe routine. The signal handler catch1 calls the fprintf routine, and the signal handler catch2 calls the malloc routine. When catch1 is running, a SIGUSR2 signal may arrive that will interrupt catch1 and cause catch2 to start running. Similarly, when catch2 is running, a SIGUSR1 signal may arrive that will interrupt catch2 and cause catch1 to start running. Since these signal handlers can interrupt each other and they call signal-unsafe routines, this program is not guaranteed to work. Figure 96 on page 168 shows that the program terminates with a memory fault when it is bombarded with the SIGUSR1 and SIGUSR2 signals.

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

void catch1(int signo)
{
      fprintf(stdout, "It's fine handling signal %2$d in process %1$d.\n",
              getpid(), signo);
      return;
}

void catch2(int signo)
{
      char *ptr;
      char buf[] = "Caught SIGUSR2.\n";

      write(STDOUT_FILENO, buf, sizeof buf - 1);
      if ((ptr = malloc(2000)) == NULL)
            _exit(2);
      free(ptr);
      return;
}

int main(int argc, char **argv)
{
      struct sigaction sa;

      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;

      sa.sa_handler = catch1;
      if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            exit(1);
      }

      sa.sa_handler = catch2;
      if (sigaction(SIGUSR2, &sa, NULL) == -1) {
            exit(1);
      }

      while (1 == 1) {
            sleep(60);
      }

      return 0;
}
```

Figure 95. Source Code for print_malloc_test *Program*

```
$ ./print_malloc_test &
[1]     10836

$ while true; do kill -USR1 10836; kill -USR2 10836; done
      .
      .
      .

It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
Caught SIGUSR2.
It's fine handling signal 30 in process 10836.
Caught SIGUSR2.
Caught SIGUSR2.
Caught SIGUSR2.
[1] + Memory fault(coredump)   ./print_malloc_test &
```

Figure 96. Running the print_malloc_test *Program*

Figure 97 on page 169 shows the source code for a program called print_malloc_test2. It is a modification of print_malloc_test. When the signal handlers for SIGUSR1 and SIGUSR2 are installed, the signal mask specified masks off the two signals. As a result, when either the catch1 or catch2 signal handler is running, the SIGUSR1 and SIGUSR2 signals are blocked. Therefore, the signal handlers cannot interrupt each other. This program runs correctly, as illustrated in Figure 98 on page 170.

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

void catch1(int signo)
{
      fprintf(stdout, "It's fine handling signal %2$d in process %1$d.\n",
                  getpid(), signo);
      return;
}

void catch2(int signo)
{
      char *ptr;
      char buf[] = "Caught SIGUSR2.\n";

      write(STDOUT_FILENO, buf, sizeof buf - 1);
      if ((ptr = malloc(2000)) == NULL)
            _exit(2);
      free(ptr);
      return;
}

int main(int argc, char **argv)
{
      struct sigaction sa;

      sigemptyset(&sa.sa_mask);
      sigaddset(&sa.sa_mask, SIGUSR1);
      sigaddset(&sa.sa_mask, SIGUSR2);
      sa.sa_flags = SA_RESTART;

      sa.sa_handler = catch1;
      if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            exit(1);
      }

      sa.sa_handler = catch2;
      if (sigaction(SIGUSR2, &sa, NULL) == -1) {
            exit(1);
      }

      while (1 == 1) sleep(60);

      return 0;
}
```

Figure 97. Source Code for print_malloc_test2 *Program*

```
$ ./print_malloc_test2 &
[1]     9082
$ while true; do kill -USR1 9082; kill -USR2 9082; done
      .
      .
      .

Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.
Caught SIGUSR2.
It's fine handling signal 30 in process 9082.

      .
      .
      .
```

Figure 98. Running the print_malloc_test2 *Program*

7.5 Interrupted System Calls

Certain system calls are considered "slow." A slow system call is one that is waiting on an event that may never occur. For example, if a process has called the read system call on a terminal device, the process may block if no data from the terminal is currently available. There is no guarantee that any data will ever be available to be read from the terminal, so the process may wait forever. A "fast" system call is one that may wait for an event, but will not wait forever. For example, a read from a disk device is usually considered to be a fast system call. A read from a disk is expected to complete in a finite amount of time[38], or at least, report an error in a finite amount of time.

[38] If a system supports mandatory record locking, a read or write call to a region of a file that is locked may be treated as a slow system call. AIX supports mandatory record locking for a file when its group execute permission bit is off and its set-group-ID bit is on. See the fcntl man page.

The execution of a signal handler may interrupt a slow system call. If a process has installed a signal handler for a signal, and then blocks in a “slow” system call, the installed signal handler will be executed if its associated signal is delivered to the process. If the signal handler returns, the slow system call will return, indicating an error[39], and the errno variable will be set to EINTR. The program may react to the delivery of the signal, if the signal handler left some indication that the signal was delivered; the program may restart the interrupted system call by calling it again. For example, in Figure 99, the wait routine is being called to wait for the termination of a child process. Since the wait system call is considered a slow system call, the routine is called again if it returns -1 and the errno variable is set to EINTR.

```
#include <sys/types.h>
#include <sys/wait.h>

      .
      .
      .

      pid_t child_pid;
      int child_status;

      .
      .
      .

      do {
            child_pid = wait(&child_status);
      } while ( (child_pid == (pid_t)-1) && (errno == EINTR) );

      .
      .
      .
```

Figure 99. Dealing with an Interrupted Call to the wait *Routine*

When coding to account for the possible interruption of a slow system call, the documentation of the routine should be referenced to determine how to recover from the interruption. The recovery may not always be as simple as the code in Figure 99 for the wait routine. For example, if a read or write system call is interrupted by a signal, an error indication is not always returned. If some I/O took place before the interruption, the read or write routine must report the number of bytes that were read or written. This may be fewer than the number requested. This condition should not be considered an error by the program. The program should adjust its buffer pointer and count of bytes to be read or written by the number of bytes returned, and call the read or write routine again to continue the I/O. This technique is illustrated in Figure 100 on page 173 and Figure 101 on page 174. Figure 100 on page 173 shows the source code for a routine called writen. The writen routine has the same interface as the write routine, and serves as an interface to the write routine. The writen routine deals

[39] The return value indicating an error depends on the routine. Typical error values are -1 for a routine returning an integral value and NULL for a routine returning a pointer. Certain routines may not always return an error, as discussed in the next paragraph.

with the write routine being interrupted by a signal handler. If -1 is returned by write and errno is EINTR, writen calls write again with the same arguments. If the write routine writes fewer bytes than requested, the writen routine adjusts the buffer pointer, buf_ptr, and the number of bytes to write, todo_size, before calling write again. The readn routine in Figure 101 on page 174 deals with interrupted read calls in much the same way[40].

[40] Even if a program has not installed signal handlers, routines like readn and writen are useful when performing I/O on file types that are defined as possibly reading or writing fewer bytes than requested in non-error situations. Examples of such file types are pipes and stream sockets. Similar routines are suggested in *UNIX Network Programming*, W. Richard Stevens, 1990, Prentice Hall, Englewood Cliffs.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

/*
 *  The writen() routine writes up to "reqsize" bytes to file descriptor
 *  "fd" from buffer "buf".  It returns the number of bytes successfully
 *  written.  A return of -1 indicates an error.
 */

int writen(int fd, char *buf, int reqsize)
{

      int todo_size;
      int actual_size;
      int written_size;
      char *buf_ptr;
      int saved_errno;


      saved_errno = errno;
      actual_size = 0;
      todo_size   = reqsize;
      buf_ptr = buf;

      while (todo_size > 0) {

            if ((written_size = write(fd, buf_ptr, todo_size)) == -1) {
                  if (errno == EINTR) {
                        continue;
                  }
                  return -1;
            }

            if (written_size == 0) {
                  break;
            }

            actual_size += written_size;
            todo_size   -= written_size;
            buf_ptr     += written_size;

      }

      errno = saved_errno;
      return actual_size;
}
```

Figure 100. Source Code for the writen *Routine*

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

/*
 *  The readn() routine reads up to "reqsize" bytes from file descriptor
 *  "fd" into buffer "buf".  It returns the number of bytes successfully
 *  read.  A return of -1 indicates an error.
 */

int readn(int fd, char *buf, int reqsize)
{
      int todo_size;
      int actual_size;
      int read_size;
      char *buf_ptr;
      int saved_errno;


      saved_errno = errno;
      actual_size = 0;
      todo_size   = reqsize;
      buf_ptr = buf;

      while (todo_size > 0) {

            if ((read_size = read(fd, buf_ptr, todo_size)) == -1) {
                  if (errno == EINTR) {
                        continue;
                  }
                  return -1;
            }

            if (read_size == 0) {
                  break;
            }

            actual_size += read_size;
            todo_size   -= read_size;
            buf_ptr     += read_size;

      }

      errno = saved_errno;
      return actual_size;
}
```

Figure 101. Source Code for the `readn` *Routine*

The author of this document has never seen a comprehensive list of slow system calls. However, reads and writes to pipes, terminal devices, and network devices; opens of terminal devices associated with a modem; `pause`; `wait`; certain `ioctl` operations; and IPC routines are mentioned in *Advanced Programming in the UNIX Environment*, W. Richard Stevens, 1992, Addison-Wesley Publishing Company.

It should be noted that many system routines that are not system calls make calls to one or more system calls, and therefore, might be interrupted by a signal handler. The standard I/O routines are good examples. Routines like scanf and printf ultimately call read or write. When read or write are interrupted, the standard I/O routines do not call read or write again. If read or write returns -1, and errno is set to EINTR, the standard I/O routines just return and let the caller deal with it. As will be shown in 7.7, "Standard I/O and Signal Handlers" on page 178, the caller cannot always deal with the problem effectively. Figure 102 lists the routines documented by XPG4 Issue 2 as possibly returning with errno set to EINTR.

```
catclose      fputwc        getwc         scanf
catgets       fputws        getwchar      select
chmod         fread         ioctl         semop
chown         freopen       lchown        sigpause
close         fscanf        lockf         sigsuspend
closedir      fseek         msgrcv        sleep
creat         fstatvfs      msgsend       statvfs
dup2          fsync         open          tcdrain
fchdir        ftruncate     pause         tcsetattr
fchmod        fwrite        poll          tmpfile
fchown        getc          printf        truncate
fclose        getchar       putc          vfprintf
fcntl         getgrent      putchar       vprintf
fflush        getgrgid      putmsg        wait
fgetc         getgrnam      putpmsg       wait3
fgets         getmsg        puts          waitid
fgetwc        getpass       putw          waitpid
fgetws        getpmsg       putwc         write
fopen         getpwnam      putwchar      writev
fprintf       getpwuid      read
fputc         gets          readv
fputs         getw          rewind
```

Figure 102. Routines That May Be Interrupted by a Signal Handler (XPG4, Issue 2)

The list in Figure 102 cannot be considered a comprehensive list of routines that can be interrupted by a signal handler on AIX. Additional routines that are in AIX, but not part of the XPG4 standard, can be interrupted. For example, these socket routines can be interrupted by signal handlers: accept, connect, recv, recvfrom, send, sendmsg, and sendto.

7.6 Automatic Restart of Interrupted System Calls

It can be cumbersome to put routines that can be interrupted by signal handlers into loops checking for the EINTR errno value. To alleviate the problem, some system calls are classified as restartable. This means that some interruptible system calls can be restarted automatically by the system. When a system call is automatically restarted, the system call behaves correctly and the caller does not see any indication that the system call had been interrupted. The system calls that are restartable according to the AIX 4.1 documentation are listed in Figure 103 on page 176.

```
fcntl       lockf       readx       write
flock       read        wait        writev
ioctl       readv       wait3       writevx
ioctlx      readvx      waitpid     writex
```

Figure 103. Restartable System Calls in AIX 4.1

A system routine that is interruptible because it calls an interruptible system call can be considered restartable if the system call is restartable. For example, the standard I/O routines in Figure 104 are restartable because read and write are restartable.

```
fclose      fputs       getchar     puts
fflush      fputwc      gets        putw
fgetc       fputws      getw        putwc
fgets       fread       getwc       putwchar
fgetwc      fscanf      getwchar    rewind
fgetws      fseek       printf      scanf
fprintf     fwrite      putc        vfprintf
fputc       getc        putchar     vprintf
```

Figure 104. Restartable Standard I/O Routines

Whether or not a restartable system call is automatically restarted when interrupted by the execution of a signal handler depends on how the signal handler was installed for the signal that caused the signal handler to execute. The sigaction structure contains a field called sa_flags that is used to modify the effect of the sigaction call. On System V systems, AIX, and systems supporting XPG4 UNIX Conformance, the SA_RESTART flag can be set in sa_flags to specify that when the signal handler being installed interrupts a restartable system call due to the delivery of the signal specified, the system call is to be automatically restarted.

Figure 105 on page 177 shows an example to illustrate the effect of SA_RESTART. The example installs a signal handler for the SIGUSR1 and SIGUSR2 signals. The signal handler installed for both signals is the catch routine. However, SA_RESTART is specified when catch is installed for SIGUSR1, and it is not specified when catch is installed for SIGUSR2. The example then calls readn (see Figure 101 on page 174) to read data from standard input and writen (see Figure 100 on page 173) to write the data to standard output. Data is read and written until the end of the input is reached, or an error occurs. Recall that readn internally calls read, and writen internally calls write. If the read routine returns an error with errno set to EINTR, the readn routine just calls read again. The writen routine provides similar processing for write.

In the program in Figure 105 on page 177, the delivery of the SIGUSR2 signal may cause read and write to return -1, with errno set to EINTR. In this case, the logic in readn and writen that makes the system call again is needed. The delivery of SIGUSR1 will not cause read and write to return -1, with errno set to EINTR. When SIGUSR1 is delivered, if the read or write system call is interrupted without having read or written any data, the system call is automatically restarted; the logic in

readn and writen that checks for EINTR is not needed, but it is harmless, in this case.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

int readn(int fd, char *buf, int reqsize);
int writen(int fd, char *buf, int reqsize);

void catch(int signo)
{
      return;
}

int main(int argc, char **argv)
{
      struct sigaction sa;
      char buf[80];
      int count;


      sa.sa_handler = catch;
      sigemptyset(&sa.sa_mask);

      sa.sa_flags = SA_RESTART;
      (void) sigaction(SIGUSR1, &sa, NULL);

      sa.sa_flags = 0;
      (void) sigaction(SIGUSR2, &sa, NULL);

      while (1 == 1) {

            if ((count = readn(STDIN_FILENO, buf, sizeof buf)) == -1) {
                  break;
            }

            if (count == 0) {
                  break;
            }

            if ((count = writen(STDOUT_FILENO, buf, count)) == -1) {
                  break;
            }

      }

      return 0;
}
```

Figure 105. Installing a Signal Handler with and without the SA_RESTART *Flag*

7.7 Standard I/O and Signal Handlers

Care should be taken when a program uses the standard I/O routines and installs signal handlers. When signal handlers that will return are installed in a program that uses the standard I/O routines, specify that restartable interrupted system calls should be automatically restarted. This section provides an example of why this should be done.

The prod_fputs program shown in Figure 106 on page 179 uses a loop of calls to fputs to write data to standard output. The cons_fgets program shown in Figure 107 on page 180 uses a loop of calls to fgets to read data from standard input. In Figure 108 on page 181, the programs are run with the standard output of prod_fputs piped to the standard input of cons_fgets. The prod_fputs program installs a signal handler for the SIGUSR1 signal. The signal handler is installed such that an interrupted system call will not automatically be restarted. The prod_fputs program writes data in a pattern, and the cons_fgets program verifies it is receiving the same pattern of data. The programs work until the SIGUSR1 signal is sent to prod_fputs. The program output indicates that the signal handler for SIGUSR1 interrupted an attempt by fputs to write bytes containing the value 231. It also reports the fputs routine returned an error, and errno was set to EINTR. The prod_fputs program reacted to this by calling fputs again, which is the proper reaction to a system call being interrupted. The call to fputs that returned an error must have placed data in the standard I/O buffer before returning, because the cons_fgets program reports that an unexpected number of bytes were received with the value 231. If fputs can place bytes into the standard I/O buffer, and not report the number of bytes placed in the buffer when interrupted by the execution of a signal handler, then it cannot be used reliably when interrupted by a signal handler.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <signal.h>

#define FILLBUF  if (val == 255) { val = 1; } else { val++; }    \
                 memset(buf, val, sizeof buf);  *(buf + sizeof buf - 1) = '\0';

void sig_catch(int signo)
{
      char msg[] = "prod: caught SIGUSR1.\n";

      write(STDERR_FILENO, msg, sizeof msg - 1);
      return;
}

int main(int argc, char **argv)
{
      int val = 0, rc;  char buf[2 * PIPE_BUF];  struct sigaction sa;

      sa.sa_handler = SIG_IGN;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = (argc > 1) ? SA_RESTART : 0;
      (void) sigaction(SIGPIPE, &sa, NULL);

      sa.sa_handler = sig_catch;
      (void) sigaction(SIGUSR1, &sa, NULL);

      FILLBUF;

again:
      while ((rc = fputs(buf, stdout)) != EOF) {
            fprintf(stderr, "prod: wrote %ld bytes of %d.\n", rc, val);
            FILLBUF;
      }

      if (ferror(stdout) && errno == EINTR) {
            fprintf(stderr, "prod: EINTR error writing %d.\n", val);
            clearerr(stdout);
            goto again;
      }

      fprintf(stderr, "prod: fputs(): %s\n", strerror(errno));
      exit(1);
}
```

Figure 106. The prod_fputs *Program*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
      char buf[10000];
      char value[2];
      char *look_ptr;
      int  look_cnt;
      int  spn_cnt;
      int  match_cnt;

      *value = '\01';  *(value + 1) = '\0';  match_cnt = 0;

      while (fgets(buf, sizeof buf, stdin) != NULL) {

            look_cnt = strlen(buf);  look_ptr = buf;
            while (look_cnt > 0) {
                  spn_cnt = strspn(look_ptr, value);
                  if (spn_cnt == 0) {
                        fprintf(stderr, "cons: read  %d bytes of %d.\n",
                                match_cnt, *value);
                        if (match_cnt != (2 * PIPE_BUF - 1)) {
                              fprintf(stderr, "cons: data error!\n");
                              exit(1);
                        }
                        if (*value == '\377') {
                              *value = '\01';
                        } else {
                              *value += 1;
                        }
                        match_cnt = 0;
                  } else {
                        match_cnt += spn_cnt;
                        look_ptr += spn_cnt;
                        look_cnt -= spn_cnt;
                  }
            }
      }

      fprintf(stderr, "cons: terminate\n");
      return 0;

}
```

Figure 107. The cons_fgets *Program*

```
$ prod_fputs | cons_fgets &

      .
      .
      .

prod: wrote 65535 bytes of 226.
cons: read  65535 bytes of 226.
prod: wrote 65535 bytes of 227.
cons: read  65535 bytes of 227.
prod: wrote 65535 bytes of 228.
cons: read  65535 bytes of 228.
prod: wrote 65535 bytes of 229.
cons: read  65535 bytes of 229.
$ kill -USR1 7530
prod: wrote 65535 bytes of 230.
cons: read  prod: caught SIGUSR1.
prod: EINTR error writing 231.
65535 bytes of 230.
prod: wrote 65535 bytes of 231.
cons: read  102629 bytes of 231.
cons: data error!
prod: fputs(): Broken pipe
```

Figure 108. Failure of fputs *to Deal with Being Interrupted*

When the prod_fputs program is started with an argument, it installs the signal handler for the SIGUSR1 signal such that restartable interrupted system calls are automatically restarted. This means that when the write call in fputs is interrupted by the execution of the SIGUSR1 signal, the write system call is restarted automatically. The caller of fputs never receives an error indication. Even in the face of constant bombardment of the prod_fputs process with the SIGUSR1 signal, the prod_fputs process continuously sends the expected data pattern to its standard output.

7.8 Protecting errno in a Signal Handler

The XPG4 man page for sigaction points out that routines called by a signal handler may change the value of the global variable errno. It suggests that a signal handler might want to save the value of errno when it starts and restore the value just before it returns.

The code in Figure 109 on page 183 illustrates why saving and restoring the value of errno in a signal handler may be important. The main routine installs a signal handler, refresh, for the SIGHUP signal. The signal handler will open and read a configuration file. The call to open in refresh may fail. If it fails, errno will be set to some value. After installing the signal handler, the main routine creates a child process, and waits for the child process to terminate. Since the program has installed a signal handler that may return, the call to wait is imbedded in a loop that tests for interruption of the wait call. Suppose that while this program is blocked in wait, the SIGHUP signal is delivered to the process. The refresh routine will be executed. When the refresh routine returns, the wait function will return to main with a return value of -1, and errno will be set to EINTR. Suppose that SIGHUP is delivered to the process again, before the process has a chance to

examine the value of errno. The refresh routine will be executed again. If the open call in refresh fails, the value of errno may be changed. For example, if the configuration file does not exist, errno will be set to ENOENT. The refresh routine returns, and the main routine can finally examine the value of errno that presumably was set by wait. But the value that was set by wait was changed when refresh called open. Instead of finding the errno value to be EINTR, the main routine will find it to be ENOENT. The program will mistakenly report that wait returned an errno value of ENOENT.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>

void refresh(int signo)
{
      int fd;

      if ((fd = open("/etc/configfile", O_RDONLY)) == -1)
            return;
      /*   ... Code that reads the configuration file goes here. ...   */
      close(fd);
      return;
}

int main(int argc, char **argv)
{
      struct sigaction sa;
      pid_t child_pid;
      int child_status;

      sa.sa_handler = refresh;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = 0;

      (void) sigaction(SIGHUP, &sa, NULL);

      /*   ... Code that creates a child process goes here.  ...    */

      do {
            child_pid = wait(&child_status);
            /*
             *  Suppose SIGHUP is delivered here; errno may be changed.
             */
      } while ( (child_pid == (pid_t)-1) && (errno == EINTR) );

      if (child_pid == (pid_t)-1) {
            perror("wait(): ");
      }

      return 0;
}
```

Figure 109. Illustration of a Signal Handler Changing the Value of errno

Figure 110 on page 184 shows a modification to the refresh signal handler that saves and restores the value of errno. The value of errno can safely be stored in a variable of type int, because the ANSI C standard specifies errno must have type int.

```
void refresh(int signo)
{
      int fd;
      int saved_errno;


      saved_errno = errno;

      if ((fd = open("/etc/configfile", O_RDONLY)) == -1) {
            errno = saved_errno;
            return;
      }

      /*   ... Code that reads the configuration file goes here. ...   */

      (void) close(fd);
      errno = saved_errno;

      return;
}
```

Figure 110. Illustration of a Signal Handler Protecting the Value of errno

Experimentation indicates that saving and restoring the value of errno in AIX 4.1 is not required. After a signal handler returns, AIX appears to restore the value of errno to the value it had before the signal handler was executed. This does not appear to be required by any standard, however. For portability, it is suggested that a signal handler save and restore errno if its value can be changed by the signal handler.

7.9 Doing the Right Thing with Signals

Often it is difficult to do the right thing when it comes to processing signals. Consider the following scenario: A program catches the SIGTERM signal. When the program catches SIGTERM, it wants to clean up and terminate. In order to clean up correctly, it needs to call some routines that are not signal-safe. Assuming the program is peppered with calls to signal-unsafe routines, calling signal-unsafe routines in the signal handler can be risky, as discussed in 7.4, "Signal Handlers and Signal-Safe Routines" on page 156.

If it is feasible to block the delivery of the SIGTERM signal whenever the program is calling signal-unsafe routines, the SIGTERM signal handler can safely call signal-unsafe routines to clean up and terminate. An outline of this approach is shown in Figure 111 on page 185.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void term_handler(int signo)
{
      /* ... Signal-safe and/or unsafe routines ... */
      exit(0);  /* exit() itself is signal-unsafe.  */
}

int main(int argc, char **argv)
{
      struct sigaction sa;
      sigset_t term_mask, saved_mask;

      (void) sigemptyset(&term_mask);
      (void) sigaddset(&term_mask, SIGTERM);

      sa.sa_handler = term_handler;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;
      (void) sigaction(SIGTERM, &sa, NULL);

      for ( ; ; ) {

            /* ... Signal-safe routines are executed here ...   */

            (void) sigprocmask(SIG_BLOCK, &term_mask, &saved_mask);

            /* ... Signal-unsafe routines are executed here ... */

            (void) sigprocmask(SIG_SETMASK, &saved_mask, NULL);

            /* ... Signal-safe routines are executed here ...   */

            (void) sigprocmask(SIG_BLOCK, &term_mask, &saved_mask);

            /* ... Signal-unsafe routines are executed here ... */

            (void) sigprocmask(SIG_SETMASK, &saved_mask, NULL);

            /* ... Signal-safe routines are executed here ...   */
      }

      /* Never reached. */
      return 0;
}
```

Figure 111. Blocking SIGTERM *When Signal-Unsafe Routines Are Called*

It may be impractical to block the SIGTERM signal every time the program calls a signal-unsafe function. If this is the case, the signal handler could just set a global variable to indicate the signal has been received. The rest of the program would then have to periodically check the global variable. If the global variable indicates the SIGTERM signal has arrived, the program should clean up and terminate. The program must check the value of the global variable at intervals

that will allow for a timely response to the signal. In the case of SIGTERM, this would mean acting on the arrival of the signal within 20 or 30 seconds[41].

Assume that the program includes a call to the select routine, and the program spends most of its time blocked in the select routine waiting for a request from a client. Figure 112 on page 187 illustrates how the program may choose to deal with the SIGTERM signal under these conditions. The program checks the value of the stop_requested flag just before calling the select routine. If the flag has been set to a non-zero value, the SIGTERM signal has been sent to the process; instead of calling the select routine to wait for a new client request, the program cleans up and terminates itself. There is a race condition in Figure 112 on page 187. It is possible that just after the program tests the value of the stop_requested flag, but before it calls the select routine, the SIGTERM signal is delivered to the process running the program. The SIGTERM signal handler sets stop_requested to 1, and returns. Then the select routine is called. The select routine blocks waiting for another client request. The program will not be able to check the value of the stop_requested flag until a client request is received. It is very possible that the program will not process the SIGTERM signal in a timely fashion.

[41] When a system is shutting down, it is common for the system to send the SIGTERM signal to all running processes, wait 20 or 30 seconds, and send SIGKILL to the processes still running. The SIGKILL signal kills the processes immediately.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <sys/select.h>

volatile sig_atomic_t stop_requested = 0;

void term_handler(int signo)
{
      stop_requested = 1;
      return;
}

int main(int argc, char **argv)
{
      struct sigaction sa;
      fd_set read_fds;
      int sock_fd;

      sa.sa_handler = term_handler;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;
      (void) sigaction(SIGTERM, &sa, NULL);

      /* ... Initialize the client request socket, and set up sock_fd ... */

      for ( ; ; ) {
            FD_ZERO(&read_fds);
            FD_SET(sock_fd, &read_fds);

            if (stop_requested) break;             /* SIGTERM delivered */

            /* ... Race condition exists here! ...          */

            if (select(sock_fd+1, &read_fds, NULL, NULL, NULL) == -1) {
                  if (errno == EINTR) continue;
                  /* Unexpected error; log message and get out. */
                  break;
            }

            /* ... Process client request with any routines needed ... */
      }

      /* ... Call signal-safe and/or -unsafe routines to clean up .. */
      return 0;
}
```

Figure 112. Processing a Signal Request Outside of a Signal Handler

Figure 113 on page 188 shows a possible solution to the race condition described previously. The call to select is changed such that it specifies a time out value. If no client requests have been received in 15 seconds, the select routine will return. This allows the program to examine the value of the stop_requested flag at least once every 15 seconds. The problem with this solution is that it is a polling solution. Approximately every 15 seconds, the

process will run. Usually it will find that there is nothing to do but to call select again to wait for something interesting to happen, or for the expiration of the next 15 second interval. If this is the only reason for the process to poll, this solution is not attractive. If there are other reasons for the process to poll, this solution might be considered.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <sys/select.h>
#include <sys/time.h>

volatile sig_atomic_t stop_requested = 0;

void term_handler(int signo)
{
      stop_requested = 1;
      return;
}

int main(int argc, char **argv)
{
      struct sigaction sa;
      fd_set read_fds;
      int sock_fd, rc;
      struct timeval tmout = {15, 0};    /* Fifteen second time out value */

      sa.sa_handler = term_handler;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;
      (void) sigaction(SIGTERM, &sa, NULL);

      /* ... Initialize the client request socket, and set up sock_fd ... */

      for ( ; ; ) {
            FD_ZERO(&read_fds);
            FD_SET(sock_fd, &read_fds);

            if (stop_requested) break;           /* SIGTERM delivered */

            if ((rc = select(sock_fd+1, &read_fds, NULL, NULL, &tmout)) == -1) {
                  if (errno == EINTR) continue;
                  /* Unexpected error; log message and get out. */
                  break;
            }
            if (rc == 0) continue;              /* Timeout */

            /* ... Process client request with any routines needed ... */
      }

      /* ... Call signal-safe and/or -unsafe routines to clean up .. */
      return 0;
}
```

Figure 113. Avoiding a Race Condition through Polling

Another solution to the race condition in Figure 112 on page 187 is presented in Figure 114 on page 189. This solution is much more complicated than that presented in Figure 113, but it does not involve polling. Before calling select, the context of the process is saved by calling sigsetjmp. While sigsetjmp is being called, the SIGTERM signal must be blocked, because the SIGTERM signal handler may now call siglongjmp to long jump to the point of the sigsetjmp call. If siglongjmp were called before or as the process context was being saved by sigsetjmp, the results would be unpredictable. After sigsetjmp saves the context of the process, a global variable, stop_jump, is set to indicate that the process context has been saved, and the SIGTERM signal is unblocked. The program then proceeds to check the value of stop_requested, and, if it is not set, to call the select routine. If the SIGTERM signal is delivered just before the select routine is called, or while the select routine is being called, or while the select routine is blocked, the SIGTERM signal handler will call siglongjmp to long jump to the point where sigsetjmp was called. Then the program will test the stop_requested flag, clean up, and terminate the process. The program sets stop_jump to 0 before processing a client request. It is assumed that this program does not want the SIGTERM signal handler to long jump while a client request is being processed. This depends, of course, on the application. The value of stop_jump can be changed without blocking the SIGTERM signal, since stop_jump is defined to be of type sig_atomic_t.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <sys/select.h>
#include <sys/time.h>
#include <setjmp.h>


volatile sig_atomic_t stop_requested = 0;
volatile sig_atomic_t stop_jump = 0;
sigjmp_buf jmp_env;


void term_handler(int signo)
{
      stop_requested = 1;
      if (stop_jump) {
            siglongjmp(jmp_env, 1);
      }
      return;
}
```

Figure 114 (Part 1 of 2). Avoiding a Race Condition through Long Jumping

```
int main(int argc, char **argv)
{
      struct sigaction sa;
      fd_set read_fds;
      int sock_fd;
      sigset_t block_mask, saved_mask;

      (void) sigemptyset(&block_mask);
      (void) sigaddset(&block_mask, SIGTERM);

      sa.sa_handler = term_handler;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;
      (void) sigaction(SIGTERM, &sa, NULL);

      /* ... Initialize the client request socket, and set up sock_fd ... */

      (void) sigprocmask(SIG_BLOCK, &block_mask, &saved_mask);
      stop_jump = !sigsetjmp(jmp_env, 1);
      (void) sigprocmask(SIG_SETMASK, &saved_mask, NULL);

      while (!stop_requested) {
            FD_ZERO(&read_fds);
            FD_SET(sock_fd, &read_fds);

            if (select(sock_fd+1, &read_fds, NULL, NULL, NULL) == -1) {
                  if (errno == EINTR) continue;
                  stop_jump = 0;
                  /* Unexpected error; log message and get out. */
                  break;
            }

            stop_jump = 0;

            /* ... Process client request with any routines needed ... */

            stop_jump = 1;
      }

      stop_jump = 0;

      /* ... Call signal-safe and/or -unsafe routines to clean up .. */
      return 0;
}
```

Figure 114 (Part 2 of 2). Avoiding a Race Condition through Long Jumping

It is important to realize that the solution presented in Figure 114 on page 189 will only work if the select routine is signal-safe on the system running the program. It should also be noted that for the time period in which it is possible for the SIGTERM signal handler to call siglongjmp, the program does not call any signal-unsafe routine. This allows the program to call signal-unsafe routines with confidence after the SIGTERM signal handler has called siglongjmp.

As shown in Figure 114 on page 189, sigsetjmp and siglongjmp can be useful for eliminating race conditions associated with processing signals. However, they

must be used with care. If a program has more than one signal handler, one signal handler interrupts the execution of another signal handler, and the second signal handler calls siglongjmp, the first signal handler will not complete its execution. This problem may be avoided by not allowing two signal handlers to execute concurrently; use the sa_mask field of the sigaction structure to block signal delivery while a signal handler is running. Another issue to consider seriously is the state of the process signal mask. In Figure 114 on page 189, sigsetjmp is called with an non-zero value for the second parameter. As a result, the sigsetjmp routine saves the state of the process signal mask, and the call to siglongjmp restores the process signal mask to its saved state.

None of the solutions presented thus far is totally satisfying. One solution involves polling, and the rest are difficult to use. This leads to the proposal of the signal pipe, described in the next section.

7.10 The Signal Pipe

In the previous section, several solutions to the problem of processing signals while waiting for client activity with the select system call were considered. Much of the discussion involved avoiding a race condition that could cause a program to miss a signal. The root cause of the race condition is that the method used to check for the delivery of a signal is unrelated to the method used to determine if a request has been sent by a client. The solution considered in this section allows a program to check for signal delivery and client requests using the same method, the select system call. Signals are converted into something that can be detected by select, using a mechanism called the “signal pipe.”

The basic idea of the signal pipe is simple. An unnamed pipe is created for signal handling. When a signal is caught by a signal handler, the handler writes data into the pipe, if data is not already in the pipe. When the program waits for client requests using select, it also determines if the read end of the pipe is ready for reading. Using this technique, race conditions are avoided, polling is avoided, and any number of signals can be handled correctly.

Figure 115 on page 192 shows the skeleton of a program using a signal pipe. At the beginning of the program, a routine named sig_pipe_start is called. That routine is given an array of signal numbers to be handled by the signal pipe. The sig_pipe_start routine creates a pipe, and installs a signal handler for the specified signals. The sig_pipe_start routine returns the file descriptor of the read end of the pipe. The program determines when the pipe can be read, using select along with the file descriptor returned by sig_pipe_start. When invoked, the signal handler installed by sig_pipe_start will ensure some data is in the pipe. Whenever the pipe file descriptor is ready to be read, the program calls sig_pipe_caught for each signal handled by the signal pipe; this allows the program to determine which signals have been delivered. The sig_pipe_caught routine will remove data from the pipe once all delivered signals have been reported to the program. Removing the data from the pipe also resets the ready to be read status to not ready. So, if another signal is not delivered by the time select is called again, select will block the program until a signal is delivered or a client request is received. Once the program is finished with the signal pipe, sig_pipe_stop is called to clean up the signal pipe.

Figure 116 on page 194 shows the signal pipe implementation. Four global variables are used to keep the state of the signal pipe. The sig_pipe_sigs

variable is a signal set that indicates which signals are being handled by the signal pipe. The sig_pipe_caught_sigs variable is a signal set that indicates which signals have been delivered, but not yet reported to the user of the signal pipe. The sig_pipe_caught_cnt variable is a count of the number of signals that have been delivered, but not reported. It is used to determine when it is necessary to read from or write to the pipe. Finally, the sig_pipe_fds variable is used to store the file descriptors of the read and write ends of the pipe.

Note that in the sig_pipe_start routine, both ends of the pipe are made non-blocking. The program should not be blocked while reading or writing the pipe. Writing into the pipe is done in the sig_pipe_catcher signal handler. Blocking there would block the program indefinitely. Reading from the pipe is done in the sig_pipe_caught routine. Blocking there would cause the program to block until a signal was delivered; client requests would be ignored.

Note that the signal pipe implementation blocks signals in order to serialize access to the global variables that are used to reflect the state of the signal pipe. The signal handler is installed in such a manner that whenever it is invoked, all signals handled by the signal pipe are blocked. These signals are also blocked in sig_pipe_caught.

Note that the signal handler only calls signal-safe routines. The user of the signal pipe may freely use routines that are signal-safe and those that are not.

The sig_pipe_cleanup routine restores signal dispositions to the default, SIG_DFL. This may not have been the best choice, since the default is often process termination. An alternative is to specify SIG_IGN.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <sys/select.h>

extern int  sig_pipe_start(int *sigs, int num_sigs);
extern int  sig_pipe_caught(int sig);
extern int  sig_pipe_stop(void);
```

Figure 115 (Part 1 of 2). Using a Signal Pipe

```
int main(int argc, char **argv)
{
    int sigs[] = {SIGUSR1, SIGUSR2, SIGTERM};
    int rc;
    fd_set read_fds;
    int sock_fd, sig_pipe_fd, max_fd;


    sig_pipe_fd = sig_pipe_start(sigs, sizeof sigs / sizeof sigs[0]);
    if (sig_pipe_fd == -1) {
        /* Unexpected error; log message and get out. */
        exit(1);
    }

    /* ... Initialize the client request socket, and set up sock_fd ... */

    for ( ; ; ) {
        FD_ZERO(&read_fds);
        FD_SET(sock_fd, &read_fds);
        FD_SET(sig_pipe_fd, &read_fds);
        max_fd = (sock_fd > sig_pipe_fd) ? sock_fd : sig_pipe_fd;

        do {
            rc = select(max_fd+1, &read_fds, NULL, NULL, NULL);
        } while (rc == -1 && errno == EINTR);

        if (rc == -1) {
            /* Unexpected error; log message and get out. */
            break;
        }

        if (FD_ISSET(sig_pipe_fd, &read_fds)) {
            if (sig_pipe_caught(SIGUSR1)) {
                /* ... Process SIGUSR1 request ... */
            }
            if (sig_pipe_caught(SIGUSR2)) {
                /* ... Process SIGUSR2 request ... */
            }
            if (sig_pipe_caught(SIGTERM)) {
                break;
            }
        }

        if (FD_ISSET(sock_fd, &read_fds)) {
            /* ... Process client request with any routines needed ... */
        }
    }

    (void) sig_pipe_stop();

    /* ... Call signal safe and/or unsafe routines to clean up ... */
    return 0;
}
```

Figure 115 (Part 2 of 2). Using a Signal Pipe

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/types.h>

#define SIG_PIPE_ASSERT(assertion)  if (!(assertion)) { _exit(255); }

static sigset_t sig_pipe_sigs;
static sigset_t sig_pipe_caught_sigs;
static int      sig_pipe_caught_cnt;
static int      sig_pipe_fds[2] = {-1, -1};

extern int  sig_pipe_start(int *sigs, int num_sigs);
extern int  sig_pipe_caught(int sig);
extern int  sig_pipe_stop(void);

static int  sig_pipe_nonbock(int pipe_fd);
static void sig_pipe_catcher(int sig);
static int  sig_pipe_cleanup(void);
```

Figure 116 (Part 1 of 5). Signal Pipe Implementation

```
int sig_pipe_start(int *sigs, int num_sigs)
{
    int i;
    struct sigaction sa;


    if (sig_pipe_fds[0] >= 0) {
        /* Log an error message */
        return -1;
    }

    sigemptyset(&sig_pipe_sigs);

    for (i = 0; i < num_sigs; i++) {
        if (sigaddset(&sig_pipe_sigs, sigs[i]) == -1) {
            /* Log an error message */
            (void) sig_pipe_cleanup();
            return -1;
        }
    }

    sigemptyset(&sig_pipe_caught_sigs);
    sig_pipe_caught_cnt = 0;

    if (pipe(sig_pipe_fds) == -1) {
        /* Log an error message */
        (void) sig_pipe_cleanup();
        return -1;
    }

    if ((sig_pipe_nonbock(sig_pipe_fds[0]) == -1) ||
        (sig_pipe_nonbock(sig_pipe_fds[1]) == -1)) {
        (void) sig_pipe_cleanup();
        return -1;
    }

    sa.sa_handler = sig_pipe_catcher;
    sa.sa_mask    = sig_pipe_sigs;
    sa.sa_flags   = SA_RESTART;

    for (i = 0; i < num_sigs; i++) {
        if (sigaction(sigs[i], &sa, NULL) == -1) {
            /* Log an error message */
            (void) sig_pipe_cleanup();
            return -1;
        }
    }

    return sig_pipe_fds[0];
}
```

Figure 116 (Part 2 of 5). Signal Pipe Implementation

```
static int sig_pipe_nonbock(int pipe_fd)
{
    int file_flags;


    if ((file_flags = fcntl(pipe_fd, F_GETFL, 0)) == -1) {
        /* Log an error message */
        return -1;
    }

    if (fcntl(pipe_fd, F_SETFL, (file_flags | O_NONBLOCK)) == -1) {
        /* Log an error message */
        return -1;
    }

    return 0;
}


static void sig_pipe_catcher(int sig)
{
    int sig_caught;
    int rc;


    sig_caught = sigismember(&sig_pipe_caught_sigs, sig);
    SIG_PIPE_ASSERT(sig_caught >= 0);

    if (!sig_caught) {

        rc = sigaddset(&sig_pipe_caught_sigs, sig);
        SIG_PIPE_ASSERT(rc == 0);

        if (sig_pipe_caught_cnt++ == 0) {
            do {
                rc = write(sig_pipe_fds[1], " ", 1);
            } while (rc == -1 && errno == EINTR);
            SIG_PIPE_ASSERT(rc == 1);
        }

    }

    return;
}
```

Figure 116 (Part 3 of 5). Signal Pipe Implementation

```
int sig_pipe_caught(int sig)
{
    int         sig_caught;
    sigset_t    saved_sigset;
    char        buf[20];
    int         rc;


    if (sig < 1 || sig > SIGMAX) {
        return 0;
    }

    rc = sigprocmask(SIG_BLOCK, &sig_pipe_sigs, &saved_sigset);
    SIG_PIPE_ASSERT(rc == 0);

    sig_caught = sigismember(&sig_pipe_caught_sigs, sig);
    SIG_PIPE_ASSERT(sig_caught >= 0);

    if (sig_caught) {

        rc = sigdelset(&sig_pipe_caught_sigs, sig);
        SIG_PIPE_ASSERT(rc == 0);

        if (--sig_pipe_caught_cnt == 0) {
            do {
                rc = read(sig_pipe_fds[0], &buf, sizeof buf);
            } while (rc > 0 || rc == -1 && errno == EINTR);
            SIG_PIPE_ASSERT(rc == -1 && errno == EAGAIN);
        }

    }

    rc = sigprocmask(SIG_SETMASK, &saved_sigset, NULL);
    SIG_PIPE_ASSERT(rc == 0);

    return sig_caught;
}
```

Figure 116 (Part 4 of 5). Signal Pipe Implementation

```
int  sig_pipe_stop(void)
{
    if (sig_pipe_fds[0] < 0) {
        /* Log an error message */
        return -1;
    }

    return sig_pipe_cleanup();
}

static int sig_pipe_cleanup(void)
{
    int rc = 0;
    int sig;
    int sig_member;
    struct sigaction sa;

    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    for (sig = 1; sig <= SIGMAX; sig++) {
        sig_member = sigismember(&sig_pipe_sigs, sig);
        if (sig_member == -1) {
            rc = -1;
            continue;
        }
        if (sig_member) {
            if (sigaction(sig, &sa, NULL) == -1) {
                /* Log an error message */
                rc = -1;
            }
        }
    }

    sigemptyset(&sig_pipe_sigs);
    sigemptyset(&sig_pipe_caught_sigs);
    sig_pipe_caught_cnt = 0;

    if (close(sig_pipe_fds[1]) == -1) {
        /* Log an error message */
        rc = -1;
    }
    sig_pipe_fds[1] = -1;

    if (close(sig_pipe_fds[0]) == -1) {
        /* Log an error message */
        rc = -1;
    }
    sig_pipe_fds[0] = -1;

    return rc;
}
```

Figure 116 (Part 5 of 5). Signal Pipe Implementation

7.11 Multi-Threaded Programs

POSIX 1003.1c introduces standardized routines for creating multi-threaded processes. AIX 4.1 supports draft 7 of this standard. See Chapter 9, “Parallel Programming” and Chapter 11, “Threads Programming Guidelines” in *AIX Version 4.1: General Programming Concepts: Writing and Debugging Programs* (SC23-2533), for complete information about threads programming in AIX 4.1. This section will briefly discuss signal handling in multi-threaded processes.

A multi-threaded process may have signal handlers like a traditional single-threaded process. A signal handler is considered a resource of the process. A signal may be sent to a specific thread of a process, or to the process itself. When a signal is delivered to a specific thread, the signal is handled by that thread. When a signal is delivered to the process, any thread of the process may be called upon to handle the signal. Each thread of a multi-threaded process has a thread signal mask, instead of the process having a global process signal mask. Therefore, threads may independently block signals. When a signal is delivered to a process, it is handled by a thread that does not have that signal blocked, if there are any. If all the process threads have the signal blocked, the signal remains pending until a thread unblocks the signal.

In addition to the signal handler model for dealing with signals, a multi-threaded process may use a new model based on a routine called `sigwait`. A program using this model typically creates one thread specifically for the purpose of handling signals. All threads block the signals of interest to the process, including the thread which will handle the signals. The thread which will handle the signals loops on a call to `sigwait`. The `sigwait` routine unblocks and waits for specified signals to be delivered to the thread. When one of the identified signals has been delivered to the thread, the `sigwait` routine blocks the signals again, and returns the signal number of the signal that has been delivered. The thread can then process the signal. This is all done without the program having to define a signal handler. The thread can safely process the signal with any thread-safe routine. Since all the `libc.a` routines have been made thread-safe (and placed in `libc_r.a`), the thread can safely use many more routines than a signal handler can. Note that in this model, all signals sent to the process are delivered to the signal handling thread, because only the signal handling thread unblocks the signals.

Figure 117 on page 200 shows the source code for a program called `print_test4`. It is similar in function to the `print_test` program shown in Figure 89 on page 161, the `print_test2` program shown in Figure 91 on page 163, and the `print_test3` program shown in Figure 93 on page 165. The program calls `printf` to print messages in the `main` routine and calls `printf` to print messages in the `signal_routine` routine. The process starts with one thread, the initial thread, running the `main` routine. The `main` routine uses `sigthreadmask` to block the `SIGUSR1` signal. A second thread is then created when `main` calls `pthread_create`. This second thread executes the `signal_routine` routine. The initial thread continues to execute the `main` routine, which prints messages to the standard output forever. The second thread, meanwhile, blocks the `SIGUSR1` signal with `sigthreadmask`; this is actually not necessary, because the second thread inherited its signal mask from the initial thread, which already blocked `SIGUSR1`.

Next, the second thread executes a loop. In the loop, sigwait is called. The signal mask passed to sigwait specifies the SIGUSR1 signal. The sigwait routine unblocks SIGUSR1, and waits until the thread receives the SIGUSR1 signal. When the SIGUSR1 signal is received, sigwait returns, and the signal_routine routine prints a message. You may recall from 7.4, "Signal Handlers and Signal-Safe Routines" on page 156, that there were problems when a signal handler interrupted a printf routine and then executed printf. The problem occurred because the printf routine was not signal-safe. A multi-threaded program is bound with libc_r.a; all the routines in this standard library are thread-safe. Routines, like printf, are designed to work correctly when called concurrently from different threads. The print_test4 routine can be run indefinitely without failure, even while constantly receiving the SIGUSR1 signal.

```
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>


void *signal_routine(void * arg)
{
      sigset_t block_mask;
      int signo;


      (void) sigemptyset(&block_mask);
      (void) sigaddset(&block_mask, SIGUSR1);
      (void) sigthreadmask(SIG_BLOCK, &block_mask, NULL);

      while (1 == 1) {
            (void) sigwait(&block_mask, &signo);

            if (signo != SIGUSR1) {
                  continue;
            }

            fprintf(stdout,
                    "It's fine handling signal %2$d in process %1$d.\n",
                    getpid(), signo);

      }

      return NULL;

}
```

Figure 117 (Part 1 of 2). Source Code for the print_test4 *Program*

```
int main(int argc, char **argv)
{
      sigset_t block_mask;
      pthread_t signal_thread;
      int rc;


      (void) sigemptyset(&block_mask);
      (void) sigaddset(&block_mask, SIGUSR1);
      (void) sigthreadmask(SIG_BLOCK, &block_mask, NULL);

      rc = pthread_create(&signal_thread, NULL, signal_routine, NULL);
      if (rc != 0) {
            fprintf(stderr, "pthread_create() error: %d.\n", rc);
            exit(1);
      }


      while (1 == 1) {

            fprintf(stdout, "It's fine being in the %2$s "
                            "of process %1$d.\n", getpid(), "mainline");

      }

      return 0;
}
```

Figure 117 (Part 2 of 2). Source Code for the print_test4 *Program*

Chapter 8. AIX Paging Space Allocation

A process on an AIX system can run with one of two paging space allocation policies. The paging space allocation policy with which a process runs determines when paging-space blocks are allocated for the pages of the process. It also affects what might happen to the process when the system runs low on free paging-space blocks. This chapter describes the AIX paging space allocation policies, and issues related to those policies. Of particular interest are the actions a process can take to protect itself from termination when the system runs low on free paging-space blocks.

8.1 The Paging Space Allocation Policies Described

A process running on an AIX system runs with either the early paging space allocation policy or the late paging space allocation policy. The late paging space allocation policy is the default policy.

When a process runs with the late paging space allocation policy, a paging-space block is not allocated for a page in the address space of the process unless and until the process modifies the page. If a process allocates a sparse array while running under this policy, paging-space blocks are only allocated for the pages actually modified in the array. No paging-space blocks are allocated for pages that are not modified. This policy may allow more processes to run concurrently on a system with a given amount of paging space. However, it may also allow the system to overcommit its resources. Processes may be allowed to allocate pages for which the system will not be able to provide paging-space blocks.

When a process runs with the early paging space allocation policy, a paging-space block is allocated for a page in the address space of the process when the page is allocated. For example, if the process allocates a large array, enough paging-space blocks are also allocated to allow the array to be paged out. If there are not enough free paging-space blocks to cover the memory being allocated, the allocation routine returns an error. When processes are running on the system with this policy, those processes are guaranteed to be able to utilize the pages successfully allocated to their address spaces. However, the system may require more paging space than if the processes were running with the late paging space allocation policy.

8.2 Actions Taken by AIX When Free Paging Space Is Low

An AIX system includes two configurable thresholds related to low paging space. These are the paging-space warning threshold and the paging-space kill threshold.

When the number of free paging-space blocks on the system falls below the paging-space warning threshold, all user processes are sent the SIGDANGER signal. The default disposition for this signal is to ignore it. A process may install a signal handler for SIGDANGER. This signal handler could release unneeded paging-space blocks, which would increase the number of free paging-space blocks on the system.

If the number of free paging-space blocks on the system falls below the paging-space kill threshold, the kernel will start taking action to free paging-space blocks. It will do this by sending the SIGKILL signal to selected processes. A process cannot change the disposition of the SIGKILL signal. A process receiving the SIGKILL signal will terminate. This will increase the number of free paging-space blocks on the system. The kernel will continue to kill user processes until the number of free paging-space blocks exceeds the paging-space kill threshold. The kernel tries to send the SIGKILL signal to the processes using the most paging-space blocks. Certain processes are exempt from being terminated. These include:

- The init process
- A process that has a SIGDANGER signal handler installed
- A process running with the early paging space allocation policy

A daemon process important to the proper functioning of a system should protect itself from termination when the system is low on paging space. This can be done by installing a signal handler for SIGDANGER, discussed in 8.4, “Installing a SIGDANGER Signal Handler” on page 212, or by running with the early paging space allocation policy, discussed in 8.5, “Running with the Early Paging Space Allocation Policy” on page 215.

8.3 Influencing and Monitoring Free Paging Space

This section discusses how a program can monitor free paging space and affect the availability of paging space.

8.3.1 Monitoring Free Paging Space from a Program

AIX provides the psdanger routine, which allows a program to determine the total number of paging-space blocks on the system, the number of free paging-space blocks on the system, the difference between the number of free paging-space blocks and the paging-space warning threshold, and the difference between the number of free paging-space blocks and the paging-space kill threshold. Figure 118 on page 205 shows the source of a program named freeps that displays information about the paging space on the system. The display_stats routine collects information with four calls to psdanger, and then displays information to standard output. The call to psdanger with the argument of 0 obtains the total number of paging-space blocks on the system. The call with argument -1 obtains the number of free paging-space blocks on the system. The call with argument SIGDANGER obtains the difference between the number of free paging-space blocks and the paging-space warning threshold. The call with argument SIGKILL obtains the difference between the number of free paging-space blocks and the paging-space kill threshold. The values for the paging-space warning threshold and paging-space kill threshold can easily be computed from the data returned by the four calls to psdanger. The main routine calls display_stats the number of times and at the interval specified by the arguments to the program. Figure 119 on page 207 shows some examples of running the freeps program.

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


void display_stats(int header)
{
      int total_psb;         /* Total paging space blocks */
      int free_psb;          /* Free  paging space blocks */
      int free_psb_aw;       /* Free  paging space blocks above warning */
      int free_psb_ak;       /* Free  paging space blocks above kill */


      total_psb   = psdanger(0);
      free_psb    = psdanger(-1);
      free_psb_aw = psdanger(SIGDANGER);
      free_psb_ak = psdanger(SIGKILL);


      if (header) {
            printf("%+9s %+9s %+9s %+9s %+9s %+9s %+9s\n",
                   "tpsb", "fpsb", "%used",
                   "fpsbaw", "psbw", "fpsbak", "psbk");
      }

      printf("%9d %9d %9.2f %9d %9d %9d %9d\n", total_psb, free_psb,
             ((total_psb - free_psb) * 100.0) / total_psb,
             free_psb_aw, free_psb - free_psb_aw,
             free_psb_ak, free_psb - free_psb_ak);

      return;
}
```

Figure 118 (Part 1 of 2). Source Code for the freeps *Program*

```
int main(int argc, char **argv)
{

      int interval, count, i;

      switch (argc) {

            case 3:
                  interval = atoi(argv[1]);
                  count    = atoi(argv[2]);
                  break;

            case 2:
                  interval = atoi(argv[1]);
                  count    = -1;
                  break;

            case 1:
                  interval = 0;
                  count    = 0;
                  break;

            default:
                  fprintf(stderr, "Usage %s [Interval [Count] ]\n",
                          argv[0]);
                  exit(1);

      }

      display_stats(1);

      for (i = 0; (count < 0) || (i < count); (count > 0) ? i++ : 0) {

            sleep(interval);
            display_stats(0);

      }

      return 0;
}
```

Figure 118 (Part 2 of 2). Source Code for the freeps *Program*

```
$ ./freeps
     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk
    65536     63232      3.52     61184      2048     62720       512

$ ./freeps 1 10
     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk
    65536     63232      3.52     61184      2048     62720       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
    65536     63228      3.52     61180      2048     62716       512
$
```

Figure 119. Running the freeps *Program*

8.3.2 Influencing Free Paging Space from a Program

Figure 120 on page 208 shows the source code for a program named piggy; it will be used to demonstrate how a program can influence the amount of free paging space on the system. The program includes a slightly modified version of the display_stats routine introduced in Figure 118 on page 205. The main routine calls display_stats often to illustrate how the actions of main affect the free paging space on the system. The steps taken by the program are:

20,000 pages are allocated.
Each page is modified.
The allocated pages may be disclaimed.
The allocated pages are freed.

What it means to disclaim a page will be explained shortly. The following results were obtained on a system that was running little else besides the piggy program.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <errno.h>
#include <signal.h>
#include <sys/shm.h>

#define PAGESIZE   4096
#define PAGEALLOC  20000
#define BYTEALLOC  (PAGESIZE * PAGEALLOC)


void display_stats(int header, char *header_string)
{
      int total_psb;      /* Total paging space blocks */
      int free_psb;       /* Free  paging space blocks */
      int free_psb_aw;    /* Free  paging space blocks above warning */
      int free_psb_ak;    /* Free  paging space blocks above kill */

      total_psb   = psdanger(0);
      free_psb    = psdanger(-1);
      free_psb_aw = psdanger(SIGDANGER);
      free_psb_ak = psdanger(SIGKILL);

      if (header) {
            printf("\n%+9s %+9s %+9s %+9s %+9s %+9s %+9s %+9s\n",
                   "tpsb", "fpsb", "%used",
                   "fpsbaw", "psbw", "fpsbak", "psbk", header_string);
      }

      printf("%9d %9d %9.2f %9d %9d %9d %9d\n", total_psb, free_psb,
             ((total_psb - free_psb) * 100.0) / total_psb,
             free_psb_aw, free_psb - free_psb_aw,
             free_psb_ak, free_psb - free_psb_ak);

      return;
}
```

Figure 120 (Part 1 of 2). The Source Code for the piggy *Program*

```
int main(int argc, char **argv)
{
      char *space, *ptr;


      display_stats(1, "start");

      if ((space = malloc(BYTEALLOC)) == NULL) {
            perror("malloc() failure: ");
            return 1;
      }
      display_stats(1, "malloc");

      for (ptr = space; ptr < space + BYTEALLOC; ptr += PAGESIZE) {
            *ptr = '\377';
      }
      display_stats(1, "touch");

      if (argc > 1) {
            if (disclaim(space, BYTEALLOC, ZERO_MEM) == -1) {
                  perror("disclaim() failure: ");
                  return 1;
            }
            display_stats(1, "disclaim");
      }

      free(space);
      display_stats(1, "free");

      return 0;
}
```

Figure 120 (Part 2 of 2). The Source Code for the piggy *Program*

Figure 121 on page 210 shows a run of the piggy program. Once the program is started, there are 63,288 free paging-space blocks on the system. After the program allocates 20,000 pages, the number of free paging-space blocks on the system has decreased by just 7 pages. The number of free paging-space blocks does not decrease by 20,000 because the process is running with the late paging space allocation policy. Once the program modifies all of the 20,000 allocated pages, the number of free paging-space blocks decreases by 19,999. Apparently, one of the paging-space blocks is allocated at malloc time. After the 20,000 pages are freed, the number of free paging-space blocks has not changed. Calling free to free allocated memory does not free paging-space blocks.

```
$ ./piggy

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     start
    65536     63288      3.43     61240      2048     62776       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk    malloc
    65536     63281      3.44     61233      2048     62769       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     touch
    65536     43282     33.96     41234      2048     42770       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk      free
    65536     43282     33.96     41234      2048     42770       512
```

Figure 121. A Run of the piggy *Program (Late Allocation, Pages Are Not Disclaimed)*

Figure 122 shows the next run of the piggy program. This time, an argument is passed to the program. This causes the program to call disclaim after touching the allocated pages. The disclaim routine is provided by AIX to allow a process to declare that it no longer needs the contents of a range of memory in the process address space. Any paging-space blocks allocated for the pages in the specified range of memory are freed. Notice that after the piggy program calls disclaim for the 20,000 pages of allocated memory, the number of free paging-space blocks increases by 19,999 blocks.

```
$ ./piggy 1

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     start
    65536     63288      3.43     61240      2048     62776       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk    malloc
    65536     63281      3.44     61233      2048     62769       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     touch
    65536     43282     33.96     41234      2048     42770       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk  disclaim
    65536     63281      3.44     61233      2048     62769       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk      free
    65536     63281      3.44     61233      2048     62769       512
```

Figure 122. A Run of the piggy *Program (Late Allocation, Pages Are Disclaimed)*

If a program allocates large amounts of memory that are only needed temporarily, it may do the system a service if it disclaims the memory, instead of just freeing it. After a memory range has been disclaimed, attempts to fetch data from the region will return zeros. The program can store new values in disclaimed memory. When a value is stored in a disclaimed page, a paging-space block is allocated for the page.

Figure 123 on page 211 shows the piggy program being run with the early paging space allocation policy[42]. Notice that when the program starts with the early paging space allocation policy, the system has 79 fewer free paging-space blocks than when it was started with the late paging space allocation policy. Notice that once the program allocates 20,000 pages of memory, the number of free paging-space blocks decreases by over 20,000 blocks. These paging-space blocks are not available to other processes in the system, even if they are not really used by the piggy program.

```
$ PSALLOC=early ./piggy

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     start
    65536     63209      3.55     61161      2048     62697       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk    malloc
    65536     43193     34.09     41145      2048     42681       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     touch
    65536     43193     34.09     41145      2048     42681       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk      free
    65536     43193     34.09     41145      2048     42681       512
$
```

Figure 123. A Run of the piggy *Program (Early Allocation, Pages Are Not Disclaimed)*

Figure 124 on page 212 shows the piggy program calling disclaim while running with the early paging space allocation policy. Notice that disclaim has no effect when the program is running with this policy.

[42] The process running the program is made to run with the early paging space allocation policy by placing PSALLOC=early in its environment. The PSALLOC environment variable is discussed in 8.5, "Running with the Early Paging Space Allocation Policy" on page 215.

```
$ PSALLOC=early ./piggy 1

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     start
    65536     63209      3.55     61161      2048     62697       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk    malloc
    65536     43193     34.09     41145      2048     42681       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk     touch
    65536     43184     34.11     41136      2048     42672       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk  disclaim
    65536     43193     34.09     41145      2048     42681       512

     tpsb      fpsb     %used    fpsbaw      psbw    fpsbak      psbk      free
    65536     43193     34.09     41145      2048     42681       512
$
```

Figure 124. A Run of the piggy *Program (Early Allocation, Pages Are Disclaimed)*

8.4 Installing a SIGDANGER Signal Handler

As mentioned in 8.2, “Actions Taken by AIX When Free Paging Space Is Low” on page 203, the SIGDANGER signal is sent to all user processes when the number of free paging-space blocks falls below the paging-space warning threshold. There are two reasons a process may want to install a signal handler for the SIGDANGER signal. First, it may want to help the system out by disclaiming unneeded pages, thus potentially freeing paging-space blocks. Secondly, having a signal handler installed for SIGDANGER will make the process immune to being killed if the number of free paging-space blocks falls below the paging-space kill threshold.

Figure 125 on page 213 shows the skeleton of a program that has installed a signal handler for SIGDANGER solely for the purpose of avoiding being killed when the system is low on free paging space. The signal handler does nothing, except return. Notice that the signal handler is installed with the SA_RESTART flag specified. Since the signal handler does not really do anything, having system calls automatically restarted seems helpful. See 7.6, “Automatic Restart of Interrupted System Calls” on page 175 for information about the automatic restarting of interrupted system calls.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>


void catch_danger(int signo)
{
      /* Make the system believe we'll free something, but don't do it! */
      return;
}


int main(int argc, char **argv)
{
      struct sigaction sa;

      sa.sa_handler = catch_danger;
      sigemptyset(&sa.sa_mask);
      sa.sa_flags = SA_RESTART;

      (void) sigaction(SIGDANGER, &sa, NULL);

      /* ... Do what the program does ... */

      return 0;
}
```

Figure 125. Installing a Signal Handler to Avoid Termination Due to Low Paging Space

Figure 126 on page 214 shows the skeleton of a program that will attempt to do something helpful when warned that the system is running low on free paging space. The program may allocate some memory whose starting address and length are recorded in the global variables `not_important` and `not_important_length`. The example assumes the data stored in this memory area is useful, but not essential. If the `SIGDANGER` signal is delivered to the process, the `catch_danger` signal handler will check to see if the memory area has been allocated. If it has, `disclaim` will be called to free any paging-space blocks allocated to the area, and `free` will be called to free the data. The call to `disclaim` may actually help the system. The call to `free` is of no importance to the system, but may be important to the logic of the program[43]. This program probably has some critical sections where the non-essential memory area is allocated, initialized, and used. These critical sections would have to block the `SIGDANGER` signal from being delivered temporarily so the signal handler does not interfere with the program's use of the memory area at a critical time. See 7.3, "Signal Handlers and Critical Sections" on page 154 for details.

[43] The `free` routine is not a signal-safe routine. Since `free` is called in the `SIGDANGER` signal handler, the program in Figure 126 on page 214 may run into trouble if any other routine that is not signal-safe is called outside of the `SIGDANGER` signal handler. See 7.4, "Signal Handlers and Signal-Safe Routines" on page 156 for details.

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/shm.h>


char *not_important = NULL;
int   not_important_length = 0;


void catch_danger(int signo)
{
   if (not_important != NULL) {
         (void) disclaim(not_important, not_important_length, ZERO_MEM);
         free(not_important);
         not_important = NULL;
         not_important_length = 0;
   }

   return;
}


int main(int argc, char **argv)
{
   struct sigaction sa;


   sa.sa_handler = catch_danger;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = SA_RESTART;

   (void) sigaction(SIGDANGER, &sa, NULL);

   /* ... Do what the program does ... */

   return 0;
}
```

Figure 126. Installing a Signal Handler to disclaim *Memory*

8.4.1 Advantages of a SIGDANGER Signal Handler

Installing a signal handler for the SIGDANGER signal to avoid termination when the system runs low on paging space has the following advantages over running a process with the early paging space allocation policy:

- Installing a signal handler for the SIGDANGER signal is straightforward when compared to the details of arranging for a process to run with the early paging space allocation policy.
- Handling the SIGDANGER signal does not involve allocating more paging-space blocks, as running with the early paging space allocation policy may.
- Some programs may not run successfully with the early paging space allocation policy, because too many paging-space blocks are required. For example, programs that allocate large sparse arrays may not be able to allocate the required number of paging-space blocks.

- A daemon with an installed SIGDANGER signal handler will not pass on immunity from termination due to low system paging space to a spawned program. A child process would inherit the installed signal handler, but once the child process called an exec routine, the disposition of the SIGDANGER signal would return to the default. Contrast this with running a daemon with the early paging space allocation policy. Such a daemon can prevent a spawned program from inheriting immunity from termination when system paging space is low, but in a more complicated manner.

  This is an important consideration. The number of processes on a system immune to termination when the system runs low on free paging space should be kept to a minimum. If too many processes inherit this immunity, the system may not be able to terminate enough processes to keep the system running.

8.4.2 Disadvantages of a SIGDANGER Signal Handler

Once a signal handler that returns is installed in a process, the logic of the program being run must account for the possibility of interrupted system calls. Some system calls can be automatically restarted by the system, if the signal handler is installed to allow for this. Other system calls are never automatically restarted. The program must look for an indication that the system call has been interrupted, and when it has, make the system call again. Chapter 7, “Signal Handling” on page 141 discusses the issues pertaining to signal handlers in detail.

8.5 Running with the Early Paging Space Allocation Policy

The paging space allocation policy of a process can change when the process executes one of the exec routines. At that time, the kernel looks for the string “PSALLOC=early” in the process environment. If the string is found, the process will proceed to run with the early paging space allocation policy. If the string is not found (PSALLOC is not in the environment, it does not have a value, or its value is anything other than “early”), the process will proceed to run with the late paging space allocation policy.

If a process running a program should run with a particular paging space allocation policy, the environment must be manipulated before a call to an exec routine. Usually the environment is manipulated if the process should run with the early paging space allocation policy, as the late paging space allocation policy is the default. There are several ways to do this for a daemon. The environment can be manipulated outside of the daemon program, or within the daemon program. Manipulating the environment outside of the program is easier, but manipulating the environment within the program allows for more control over which processes run with which paging space allocation policy. For example, a program could ensure that the process running it is running with the early paging space allocation policy, and any child processes created by the program run with the late paging space allocation policy.

8.5.1 Determining the Paging Space Allocation Policy of a Process

The paging space allocation policy under which a process is running can be determined by looking at the process flags for the process. The sys/proc.h header file defines the process flag values. The process flag that indicates whether a process is running with the early paging space allocation policy is named SPSEARLYALLOC. Its value in AIX 4.1.2 is 0x04000000; the value has changed from release to release. If this flag is set for a process, the process is running with the early paging space allocation policy. If this flag is not set for a process, the process is running with the late paging space allocation policy.

The simplest way to see the process flags for a process is through the ps command. The example in Figure 127 shows two SRC subsystems being started. The subsys01 subsystem forces itself to run with the early paging space allocation policy[44], while the subsys02 subsystem runs with the late paging space allocation policy by default[45]. The grep command output shows that the value of the SPSEARLYALLOC flag on this system is 0x04000000. The ps command is executed using the -p and -l flags to display a long listing for the two processes running the subsystems. The process flags are in the first field of the ps output. It can be seen that the subsys01 process is running with the early paging space allocation policy, and the subsys02 process is running with the late paging space allocation policy.

```
# startsrc -s subsys01
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 8434.

# startsrc -s subsys02
0513-059 The subsys02 Subsystem has been started. Subsystem PID is 8694.

# grep SPSEARLYALLOC /usr/include/sys/proc.h
#define SPSEARLYALLOC   0x04000000      /* allocates paging space early    */

# ps -p 8434,8694 -l
       F S UID  PID PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
 4340401 A   0 8434 3946   0  60 20 15335   164                -  0:00 subsys01
  240001 A   0 8694 3946   0  60 20 18338   104                -  0:00 subsys02
```

Figure 127. Determining the Paging Space Allocation Policy of a Process

8.5.1.1 A Problem with the SPSEARLYALLOC Process Flag

The preceding paragraphs describe how the SPSEARLYALLOC process flag can be used to determine if a process is running with the early paging space allocation policy or not. However, in AIX 4.1.2, and presumably in previous versions of AIX, there is a condition under which a process will be running with the early paging space allocation policy, but the SPSEARLYALLOC flag will not be set. The condition occurs when a process running with the early paging space allocation policy creates a child process, and the child process has not issued exec. The child process inherits use of the early paging space allocation policy from the parent

[44] The subsys01 subsystem can force itself to run with the early paging space allocation policy by using the techniques described in 8.5.4, “Setting PSALLOC in a Daemon” on page 224.

[45] The subsys02 subsystem runs with the late paging space allocation policy if and only if PSALLOC=early is not in the /etc/environment file, and is not set in the subsys02 SRC definition, and is not set by the subsys02 program itself.

process. However, the SPSEARLYALLOC flag is not set in the process flags for the child process. Once the child process calls an exec function, the state of the SPSEARLYALLOC process flag and the paging space allocation policy under which the process is running will be in agreement.

In AIX release 4.1.2 and earlier, a process running with the early paging space allocation policy, but without SPSEARLYALLOC set in its process flags, allocates paging-space blocks when pages are allocated, but is not protected from termination when the system runs low on paging space. This problem has been reported to AIX development. They recognized it as a problem, and fixed it in releases after AIX 4.1.2[46].

Even without a fix for this problem, the SPSEARLYALLOC process flag is still useful, because there is a process flag that indicates if the process has issued exec. This flag is named SEXECED, and has a value of 0x00200000 on AIX 4.1.2. If this flag is set for a process then the process has issued exec, and the state of the SPSEARLYALLOC flag correctly indicates the paging space allocation policy being used by the process. If the SEXECED flag is not set for a process, the SPSEARLYALLOC flag will not be set, the paging space allocation policy under which the process is running cannot be determined from the process flags, and the process is a candidate for termination if the system runs low on paging space (unless the process has installed a signal handler for the SIGDANGER signal).

Figure 128 on page 218 presents some examples. First, the subsys01 and subsys02 subsystems are started with the SRC. A couple of grep commands show the values of the SPSEARLYALLOC and SEXECED flags. The output of the ps command shows that both the subsys01 and subsys02 processes have issued exec. Therefore, the state of the SPSEARLYALLOC flag can be used to determine the paging space allocation policies under which the processes are running. The subsys01 process is running with the early paging space allocation policy. The subsys02 process is running with the late paging space allocation policy. Next, the subsys02 subsystem is stopped with stopsrc, so it can be restarted from the shell command line.

When the subsys02 daemon is started from the shell command line, the PSALLOC=early string is specified with the shell command that starts the daemon. The shell creates a child process, places PSALLOC=early in the environment of the child process, and then calls exec for the subsys02 program in the child process. Thus, the child process is running with the early paging space allocation policy. The process created by the shell recognizes it has been started by a shell. Therefore, it creates a child process of its own, and then it (the parent process) exits[47]. The second child process continues to run as the subsys02 daemon. The output of the next ps command shows that the subsys02 daemon process has not issued exec. Therefore, the output of this command does not indicate which paging space allocation policy is used by the process. All that can be said for certain, based solely on the ps output, is that the process is not protected from termination when the system runs low on paging space if it has not installed a SIGDANGER signal handler. Knowledge of how the subsys02 program is written and how it was started leads to the conclusion that it is running with the early paging space allocation policy.

---

[46] This problem was first recognized by Larry Brenner, and reported to AIX development by him.

[47] For an explanation of why this is done, see 3.3, “Migrate the Daemon to Another Process” on page 25.

```
# startsrc -s subsys01
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 8434.

# startsrc -s subsys02
0513-059 The subsys02 Subsystem has been started. Subsystem PID is 8694.

# grep SPSEARLYALLOC /usr/include/sys/proc.h
#define SPSEARLYALLOC   0x04000000      /* allocates paging space early    */

# grep SEXECED /usr/include/sys/proc.h
#define SEXECED         0x00200000      /* process has exec'd              */

# ps -p 8434,8694 -l
       F S UID  PID PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
 4340401 A   0 8434 3946   0  60 20 15335   164                -  0:00 subsys01
  240001 A   0 8694 3946   0  60 20 18338   104                -  0:00 subsys02

# stopsrc -s subsys02
0513-044 The stop of the subsys02 Subsystem was completed successfully.

# PSALLOC=early /u/agar/daes/subsys02 -l /tmp/subsys02

# ps -el | grep subsys02
  140401 A   0 10268     1   0  60 20 8388   216                -  0:00 subsys02
```

Figure 128. Looking at the SPSEARLYALLOC *and* SEXECED *Flags*

8.5.2 Setting PSALLOC with the SRC

The definition of an SRC subsystem does not allow for the specification of environment variable values. If it did, a subsystem could be defined such that PSALLOC was set to early in the subsystem's environment, causing the subsystem to run with the early paging space allocation policy. The environment with which an SRC subsystem runs can be influenced by the contents of the /etc/environment file and any environment values specified on the startsrc command. Specifying PSALLOC=early in the /etc/environment file would cause all processes on the system to run with the early paging space allocation policy; this is probably not desirable. Figure 129 shows an example of starting a subsystem and using the startsrc command to cause it to run with the early paging space allocation policy. A problem with this approach is that if the user of the startsrc command does not specify the allocation policy, the subsystem will run with the late paging space allocation policy.

```
# startsrc -s subsys03 -e "PSALLOC=early"
0513-059 The subsys03 Subsystem has been started. Subsystem PID is 8198.

# ps -p 8198 -l
       F S UID  PID PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
 4340401 A   0 8198 3946   0  60 20 10250   160                -  0:00 subsys03
```

Figure 129. Using startsrc *to Specify the Early Paging Space Allocation Policy*

An SRC subsystem could be defined such that when it is started through the SRC, a shell script is run. The shell script could set PSALLOC to early in the environment, and then call exec for the daemon program. It is important that the process started by the SRC continue running. It would be incorrect for the shell script to spawn another process to run the daemon program, and then exit. The SRC would interpret the termination of the shell script process as the termination of the subsystem. Figure 130 on page 220 illustrates this approach. The subsys04 subsystem is defined such that the subsys04.init script is run when the subsystem is started. The shell script just sets up PSALLOC in the environment, and calls exec for the daemon program, subsys04. Any parameters passed to the shell script are passed on to the daemon program, through "$@." The output of the ps command shows that the subsys04 daemon program is running in the process started by the SRC, and that process is running with the early paging space allocation policy.

```
# cat /usr/lpp/somelpp/bin/subsys04.init
#!/bin/ksh

export PSALLOC=early
exec /usr/lpp/somelpp/bin/subsys04 "$@"

# mkssys -s subsys04 -u 0 -p /usr/lpp/somelpp/bin/subsys04.init \
         -a "-s /tmp/subsys04" -i /dev/null -o /dev/null -e /dev/null
0513-071 The subsys04 Subsystem has been added.

# odmget -q "subsysname = 'subsys04'" SRCsubsys

SRCsubsys:
        subsysname = "subsys04"
        synonym = ""
        cmdargs = "-s /tmp/subsys04"
        path = "/usr/lpp/somelpp/bin/subsys04.init"
        uid = 0
        auditid = 0
        standin = "/dev/null"
        standout = "/dev/null"
        standerr = "/dev/null"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = ""

# startsrc -s subsys04
0513-059 The subsys04 Subsystem has been started. Subsystem PID is 8292.

# lssrc -s subsys04
Subsystem         Group            PID     Status
 subsys04                          8292    active

# ps -p 8292 -l
       F S UID  PID PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
 4340401 A   0 8292 3946   0  60 20 19219   164                -  0:00 subsys04
```

Figure 130. Using a Shell Script to Add to a Daemon's Environment

When a shell script is run as part of starting a daemon, care should be taken with redirection of I/O. Redirection to or from a terminal device that is not a controlling terminal of a session may cause the process to acquire the terminal as a controlling terminal. See 3.4, "Disassociate the Daemon from Its Controlling Terminal" on page 25 for information about controlling terminals and how to lose one.

A shell script can also be used to help bring up a daemon that is started from the `init` process, or from the `inetd` process.

There is another way to define an SRC subsystem such that it runs with the early paging space allocation policy. This is described in section 8.5.3, “Setting PSALLOC in a System File” on page 221.

8.5.3 Setting PSALLOC in a System File

When a program is executed from a Korn shell command line, environment variable values can be specified before the command name. The specified environment variable values are placed in the environment of the process running the command; the values are not put in the environment of the shell. An example is shown in Figure 131. If a shell does not support this behavior, the env command can be used to add values to a process environment and then exec another program. This is also shown in Figure 131.

```
# cat /tmp/program
echo $TEST1
echo $TEST2

# echo $TEST1

# echo $TEST2

# TEST1=hello TEST2=world /tmp/program
hello
world

# echo $TEST1

# echo $TEST2

# /bin/env TEST1=hello TEST2=world /tmp/program
hello
world

# echo $TEST1

# echo $TEST2

#
```

Figure 131. Using the env *Command to Add to a Program's Environment*

When a system configuration file describes how a daemon is started, the env command can be used to place values in the daemon's environment. This can be used to make a daemon run with the early paging space allocation policy. This approach can be taken with the definition of an SRC subsystem in the ODM, with an entry in the /etc/inittab file, and with an entry in the inetd configuration file.

A subsystem might normally be defined to the SRC with an mkssys command similar to that shown in Figure 132 on page 222. If the subsys05 subsystem should always run with the early paging space allocation policy, the subsystem could be defined as shown in Figure 133 on page 222. Here, the program specified with the -p parameter is /bin/env. The arguments specified with the -a parameter are composed of the specification of the PSALLOC=early string followed

by the path name of the daemon program and the arguments to be passed to that program. The ps command in Figure 133 on page 222 shows that the daemon does run with the early paging space allocation policy.

```
# mkssys -s subsys05 -u 0 -p /usr/lpp/somelpp/subsys05 \
         -a "-l /tmp/subsys05" -i /dev/null -o /dev/null -e /dev/null
```

Figure 132. A Typical Definition of an SRC Subsystem

```
# mkssys -s subsys05 -u 0 -p /bin/env \
         -a "PSALLOC=early /usr/lpp/somelpp/subsys05 -l /tmp/subsys05" \
         -i /dev/null -o /dev/null -e /dev/null

# odmget -q "subsysname = 'subsys05'" SRCsubsys

SRCsubsys:
        subsysname = "subsys05"
        synonym = ""
        cmdargs = "PSALLOC=early /usr/lpp/somelpp/subsys05 -l /tmp/subsys05"
        path = "/bin/env"
        uid = 0
        auditid = 0
        standin = "/dev/null"
        standout = "/dev/null"
        standerr = "/dev/null"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = ""

# startsrc -s subsys05
0513-059 The subsys05 Subsystem has been started. Subsystem PID is 6690.

# ps -p 6690 -l
      F S UID  PID PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
 4340401 A   0 6690 3946   0  60 20 18378   208                -  0:00 subsys05
```

Figure 133. Using the env *Program in an SRC Subsystem Definition*

A daemon to be started by init, without the SRC, might be defined in /etc/inittab in a manner similar to the definition of subsys06 in Figure 134 on page 223. If the subsys06 daemon should always run with the early paging space allocation policy, the daemon could be defined as shown in Figure 135 on page 223. Here, the program specified to run is /bin/env. The arguments specified include the PSALLOC=early string followed by the path name of the daemon program and the arguments to be passed to that program. The ps

command in Figure 135 on page 223 shows that the daemon does run with the early paging space allocation policy.

```
# grep subsys06 /etc/inittab
subsys06:2:respawn:/usr/lpp/somelpp/subsys06 -l /tmp/subsys06
```

Figure 134. A Typical inittab *Entry for a Daemon*

```
# grep subsys06 /etc/inittab
subsys06:2:respawn:/bin/env PSALLOC=early /usr/lpp/somelpp/subsys06 -l /tmp/subsys06

# ps -el | grep subsys06
 4340401 A   0 6752    1   2  61 20 15375   204               -  0:00 subsys06
```

Figure 135. An inittab *Entry Using the* env *Command*

A daemon to be started by inetd might be defined in /etc/inetd.conf in a manner similar to the definition of subsys07 in Figure 136. If the subsys07 daemon should always run with the early paging space allocation policy, the daemon could be defined as shown in Figure 137. Here, the program specified to run is /bin/env. The arguments specified include the PSALLOC=early string followed by the path name of the daemon program and the arguments to be passed to that program. Recall that in the inetd configuration file, the first argument specified must be the name of the program. In Figure 137, that name is env. The ps command in Figure 137 is executed after a client requests the service provided by subsys07; its output shows that the daemon does run with the early paging space allocation policy. Notice that the lssrc output for subsys07 is not particularly useful.

```
# grep subsys07 /etc/inetd.conf
subsys07  stream  tcp  nowait  root  /usr/lpp/somelpp/subsys07  subsys07 -l /tmp/subsys07
```

Figure 136. A Typical inetd.conf *Entry for a Daemon*

```
# grep subsys07 /etc/inetd.conf
subsys07 stream tcp nowait root /bin/env env PSALLOC=early /usr/lpp/somelpp/subsys07 -l /tmp/sub
sys07

# lssrc -ls inetd | grep subsys07
 subsys07     /bin/env                 env PSALLOC=early        active

# ps -el | grep subsys07
 4240001 A   0 8108 5524   0  60 20 1c0fc   196   237878      -  0:00 subsys07
```

Figure 137. An inetd.conf *Entry Using the* env *Command*

8.5.4 Setting PSALLOC in a Daemon

A program may change the paging space allocation policy under which it runs. There are several advantages of doing this in the program, instead of assuming the PSALLOC environment variable is set properly outside of the program. First, the program can ensure it runs with the desired paging space allocation policy, regardless of how it was started. Second, the program can influence the paging space allocation policy used by programs it spawns. Third, the program can compensate for the problem discussed in 8.5.1.1, “A Problem with the SPSEARLYALLOC Process Flag” on page 216.

The subsection titled “Programming Interface” of the section titled “Understanding Paging Space Allocation Policies” in Chapter 7, “Paging Space and Virtual Memory” in *AIX Version 4.1: System Management Guide: Operating System and Devices*, SC23-2525, recommends that a program wanting to run with a specific paging space allocation policy take the following steps:

- Examine the value of the PSALLOC environment variable.
- If the PSALLOC environment variable is not set to a value consistent with the desired paging space allocation policy, set the variable, and call an exec routine specifying the program being run and the arguments passed to the program. This will cause the process to run the program with the desired paging space allocation policy. As the program runs again, it will find the PSALLOC environment variable to be set to a value consistent with the desired paging space allocation policy.
- If the PSALLOC environment variable is set to a value consistent with the desired paging space allocation policy, no action is necessary to change the paging space allocation policy under which the process is running.

These steps will work correctly in many cases. However, the steps assume the process has executed an exec routine before these steps are taken. As mentioned in section 3.3, “Migrate the Daemon to Another Process” on page 25, it is not unusual for a daemon program to create a child process that will not call an exec routine. If a process has not executed an exec routine, the paging space allocation policy under which the process is running is not determined by the value of PSALLOC, but inherited from the parent process. The paging space allocation policy inherited from the parent process may not be consistent with the value of PSALLOC inherited from the parent process. It is possible that the parent process changed the value of the PSALLOC environment variable without calling an exec routine afterward, in order to influence the paging space allocation policy under which its child processes will run after they call exec.

These recommended steps also suffer from the problem described in 8.5.1.1, “A Problem with the SPSEARLYALLOC Process Flag” on page 216. If a process is running with the early paging space allocation policy, and PSALLOC=early is in the environment of the process, and the process then creates a child process, the child process inherits the paging space allocation policy and the PSALLOC environment variable from its parent. Therefore, the child process is running with the early paging space allocation policy, and the value of PSALLOC in its environment is consistent with that policy. If the child process does not exec another program, but runs the inherited program, and then follows the steps previously shown, it will do nothing to influence the paging space allocation policy under which it runs. The problem with doing nothing is that the SPSEARLYALLOC flag will not be set in the process flags. Therefore, the child

process may be a target for termination when the system is low on paging space, even though it runs with the early paging space allocation policy.

A program may influence the paging space allocation policy with which it runs in a manner that overcomes the difficulties discussed above. The key is to consider both the value of the PSALLOC environment variable and the value of the process flags. The steps involved are the following:

1. Get the value of the process flags.
2. If the SEXECED process flag is set, indicating the process has called an exec routine, and the value of the SPSEARLYALLOC flag is consistent with the desired paging space allocation policy, then the process is running with the desired paging space allocation policy. In this case there is nothing to do; skip the remaining steps.
3. If this step is reached, either the process has not called an exec routine, in which case the paging space allocation policy under which the process is running cannot be determined, or the process has called an exec routine and the process is not running with the desired paging space allocation policy. In either case, proceed to the next step.
4. Get the current value of the PSALLOC environment variable.
5. If the value of the PSALLOC environment variable is consistent with the desired paging space allocation policy, and the process flags indicate the process has called an exec routine, something has gone wrong; exit.
6. If the value of the PSALLOC environment variable is not consistent with the desired paging space allocation policy, set the value such that it is consistent with the desired policy.
7. Call an exec routine specifying the program being run and the arguments passed to the program. This will cause the process to run the program with the desired paging space allocation policy. As the program runs again, it will find that the process flags indicate the process has called an exec routine, and the value of the SPSEARLYALLOC flag is consistent with the desired paging space allocation policy. The program will then take no other action with regard to the paging space allocation policy under which the process runs.

Once a program has ensured it is running under the desired paging space allocation policy, it may want to ensure that programs spawned by it run with a certain paging space allocation policy. For example, a daemon may want to run with the early paging space allocation policy, but it may want programs spawned by it to run with the late paging space allocation policy. The program can arrange for this by changing the value of PSALLOC in its environment to some value other than early, after the previously listed steps are executed. If the program then creates a child process that issues exec for another program, the child process will start running with the early paging space allocation policy, but will switch to the late paging space allocation policy when the exec routine is called.

The daemon support routines include routines similar to those shown in Figure 138 on page 226. These routines ensure the calling process is running with the desired paging space allocation policy, and that any programs spawned by the daemon process will run with the late paging space allocation policy.

```
1
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <procinfo.h>
#include <sys/proc.h>

int  check_psalloc_env(int early);
int  set_psalloc_env(int early);
int  get_process_flags(unsigned long *process_flags);
int  check_psalloc_flag(unsigned long pflags, int early);

int set_psalloc(int early, const char *prog_path, char * const argv[])
{
   int rc;
   unsigned long process_flags;

   if ((rc = get_process_flags(&process_flags)) != 0)
         return rc;

   if (((process_flags & SEXECED) != SEXECED) ||
       (!check_psalloc_flag(process_flags, early))) {

         if (check_psalloc_env(early)) {
              if ((process_flags & SEXECED) == SEXECED) {
                   /* Log an error message */
                   return 1;
              }
         } else {
              if ((rc = set_psalloc_env(early)) != 0) {
                   return rc;
              }
         }

         execv(prog_path, argv);

         /* Log message about execv error; include errno value */
         return 1;
   }

   if (!check_psalloc_env(0)) {
         if ((rc = set_psalloc_env(0)) != 0) {
              return rc;
         }
   }

   return 0;
}
```

Figure 138 (Part 1 of 3). Routines to Ensure a Specified Paging Space Allocation Policy

```
2
char early_alloc_string[] = "PSALLOC=early";
char late_alloc_string[]  = "PSALLOC=late";


int check_psalloc_env(int early)
{
   char *envptr;

   envptr = getenv("PSALLOC");

   if ((envptr != NULL) && (strcmp(envptr, "early") == 0)) {
         return (early != 0);
   } else {
         return (early == 0);
   }

}


int set_psalloc_env(int early)
{
   char *envptr;


   if (early) {
         envptr = early_alloc_string;
   } else {
         envptr = late_alloc_string;
   }

   if (putenv(envptr) != 0) {
         /* Log message about putenv error; include errno value */
         return 1;
   }

   if (!check_psalloc_env(early)) {
         /* Log an error message */
         return 1;
   }

   return 0;
}
```

Figure 138 (Part 2 of 3). Routines to Ensure a Specified Paging Space Allocation Policy

```
3
int get_process_flags(unsigned long *process_flags)
{
   struct procsinfo pi;
   pid_t pid;
   pid_t scratch_pid;


   pid = getpid();
   scratch_pid = pid;

   if (getprocs(&pi, sizeof pi, NULL, 0, &scratch_pid, 1) != 1) {
         /* Log message about getprocs error; include errno value */
         return 1;
   }

   if ((pi.pi_state == SNONE) || (pi.pi_state == SIDL) ||
       (pi.pi_state == SZOMB) || (pi.pi_pid != pid)) {
         /* Log an error message */
         return 1;
   }

   *process_flags = pi.pi_flags;

   return 0;
}


int check_psalloc_flag(unsigned long pflags, int early)
{

   if (early) {
         return ((pflags & SPSEARLYALLOC) == SPSEARLYALLOC);
   } else {
         return ((pflags & SPSEARLYALLOC) != SPSEARLYALLOC);
   }

}
```

Figure 138 (Part 3 of 3). Routines to Ensure a Specified Paging Space Allocation Policy

Figure 138, part **1** shows the set_psalloc routine. The routine takes three arguments. The first argument, early, is a flag that indicates whether the early paging space allocation policy is desired. The second argument is the path name of the program being executed, and the third argument is a pointer to the arguments passed to the program when it was invoked. The set_psalloc routine calls the get_process_flags routine to get the process flags for the process. If the SEXECED flag is not set in the process flags, or the SPSEARLYALLOC flag is not set to a value consistent with the desired paging space allocation policy (determined by a call to check_psalloc_flag), the set_psalloc routine will have to set the paging space allocation policy for the process. The routine calls check_psalloc_env to determine if PSALLOC is set to a value consistent with the desired paging space allocation policy. If it is not, the value of PSALLOC is changed by a call to set_psalloc_env. Once PSALLOC has an appropriate value, the execv routine is called to re-execute the program. As the program is re-executed, it will call set_psalloc, which will again call get_process_flags to get

the process flags. This time it should find that the SEXECED flag is set, indicating the process has called an exec routine, and the SPSEARLYALLOC flag is now set to a value consistent with the desired paging space allocation policy. Once the calling process is running with the desired paging space allocation policy, the set_psalloc routine sets the PSALLOC variable to a value consistent with the late paging space allocation policy. This is done with calls to check_psalloc_env and set_psalloc_env. Any programs spawned by this process will run with the late paging space allocation policy.

Figure 138, part **2** shows the check_psalloc_env and set_psalloc_env routines. The check_psalloc_env routine receives an argument, early, that indicates if the early paging space allocation policy is desired. The routine gets the value of the PSALLOC environment variable with a call to getenv. The check_psalloc_env routine then returns 0 if the value of the environment variable is not consistent with the desired paging space allocation policy, or 1 if the value is consistent with the desired policy. The set_psalloc_env routine receives the same argument as check_psalloc_env. The routine calls putenv to set the PSALLOC environment variable to a value consistent with the desired paging space allocation policy. It then calls check_psalloc_env to make sure the environment variable has been set correctly.

Figure 138, part **3** shows the get_process_flags and check_psalloc_flag routines. The get_process_flags routine calls the getprocs routine to get information from the kernel process table about the process executing the routine. The process flags are returned to the caller of get_process_flags. The check_psalloc_flag routine determines if the value of the SPSEARLYALLOC flag in the process flags is consistent with the desired paging space allocation policy.

8.5.5 Advantages of the Early Paging Space Allocation Policy

The advantage of running a process with the early paging space allocation policy to avoid termination when the system is low on paging space is that the disadvantages of installing a signal handler for the SIGDANGER signal are avoided. See 8.4.2, “Disadvantages of a SIGDANGER Signal Handler” on page 215.

8.5.6 Disadvantages of the Early Paging Space Allocation Policy

The disadvantage of running a process with the early paging space allocation policy to avoid termination when the system is low on paging space is that the advantages of installing a signal handler for the SIGDANGER signal are not enjoyed. See 8.4.1, “Advantages of a SIGDANGER Signal Handler” on page 214.

Chapter 9. Ensuring Exclusivity

It is not uncommon for a daemon to have some sort of requirement relating to exclusivity. For example, a daemon program may be designed such that at any time, there should be only one process running the program on a system. On a control workstation associated with a partitioned RS/6000 SP system, it may be necessary to have multiple processes running the same daemon, one for each RS/6000 SP partition.

The SRC, inetd, and init include mechanisms to enforce exclusivity. An SRC subsystem can be defined to allow or not allow multiple instances. The value of the Wait/NoWait field in the inetd configuration file entry for an inetd daemon determines whether multiple instances of the daemon can run at once. An entry for a daemon in /etc/inittab causes one instance of the daemon to be started. These mechanisms are useful and should be used, as appropriate. However, they can fail. The SRC mechanism fails whenever srcmstr is terminated and respawned. The inetd mechanism fails when inetd is terminated and restarted. None of the mechanisms prevent other instances of a daemon from being started from a shell.

9.1 Limitations of System Methods to Ensure Exclusivity

Normally, when a daemon is defined in the inetd configuration file with the value of wait in the Wait/Nowait field, inetd will allow only one instance of the daemon to run at a time. However, whenever the inetd daemon terminates, it loses track of the daemons it had spawned. If another instance of inetd is started, it is possible that more than one instance of a daemon defined with the wait value will be running at the same time. This is illustrated in Figure 139 on page 232. The grep command output shows that wait is specified for the talkd daemon on the sysA system. A talk session is requested from the sysB system to the sysA system. The two ps commands run on sysA show that the talkd daemon has been spawned by the inetd daemon. The inetd daemon is then killed. The next two ps commands show that the inetd daemon is not running on sysA, and the talkd daemon that had been started by inetd has been inherited by the init process. Next, the startsrc command is used to restart the inetd daemon. The next two ps commands show that inetd is now running, and the talkd daemon inherited by the init process continues to run. Next, another talk session is requested, this time from the sysC system. The final ps command shows that two talkd daemons are running at the same time; one talkd daemon had been started from the first inetd process and inherited by init, and the second talkd daemon was started by the new inetd process.

```
[sysA] # grep talkd /etc/inetd.conf
#talk   dgram   udp     wait    root    /usr/sbin/talkd         talkd
ntalk   dgram   udp     wait    root    /usr/sbin/talkd         talkd

[sysB] # talk root@sysA

[sysA] # ps -el | grep talk
  240001 A   0 15174 16142   0  60 20 47e4   160   2377d8      -  0:00 talkd

[sysA] # ps -el | grep inetd
  240001 A   0 16142  3946   0  60 20 57e5   212               -  0:00 inetd

[sysA] # kill -9 16142

[sysA] # ps -el | grep inetd

[sysA] # ps -el | grep talk
  240001 A   0 15174     1   0  60 20 47e4   160   2377d8      -  0:00 talkd

[sysA] # startsrc -s inetd
0513-059 The inetd Subsystem has been started. Subsystem PID is 7522.

[sysA] # ps -el | grep inetd
  240001 A   0  7522  3946  19  69 20 6806   196               -  0:00 inetd

[sysA] # ps -el | grep talk
  240001 A   0 15174     1   0  60 20 47e4   160   2377d8      -  0:00 talkd

[sysC] # talk root@sysA

[sysA] # ps -el | grep talk
  240001 A   0  8812  7522   0  60 20 1a75a   160   23773c      -  0:00 talkd
  240001 A   0 15174     1   0  60 20 47e4   160   2377d8      -  0:00 talkd
```

Figure 139. Starting Two Instances of the talkd *Daemon*

A similar problem is associated with the SRC. When an SRC subsystem is defined, you can specify whether multiple instances of the subsystem should run at the same time. If multiple instances are not allowed for a subsystem, and the SRC is used to attempt to start a second instance, the SRC displays an error message and will not start the second instance of the subsystem. However, like inetd, the SRC daemon, srcmstr, loses track of the daemons it has spawned if it terminates and is restarted. This is illustrated in Figure 140 on page 233. The odmget command shows that the inetd subsystem is defined such that multiple instances should not be allowed to run at the same time (the value of multi is 0). The ps command shows that a process is running the inetd daemon. Next, the startsrc command is issued, in an attempt to start another inetd daemon. The startsrc command displays an error message indicating multiple instances of the inetd subsystem are not allowed. Next, the kill command is used to terminate the srcmstr daemon. The next two ps commands show that the srcmstr daemon was respawned (notice the change in the PID), and the inetd daemon that had been running has now been inherited by the init process. Next, the startsrc command is again used to attempt to start a second inetd process. This time the attempt succeeds. A ps command shows that there are now two inetd processes running on the system.

```
# odmget -q "subsysname = 'inetd'" SRCsubsys

SRCsubsys:
        subsysname = "inetd"
        synonym = ""
        cmdargs = ""
        path = "/usr/sbin/inetd"
        uid = 0
        auditid = 0
        standin = "/dev/console"
        standout = "/dev/console"
        standerr = "/dev/console"
        action = 2
        multi = 0
        contact = 3
        svrkey = 0
        svrmtype = 0
        priority = 20
        signorm = 0
        sigforce = 0
        display = 1
        waittime = 20
        grpname = "tcpip"

# ps -el | grep inetd
  240001 A   0  7522  3946   0  60 20 6806   212              -  0:00 inetd

# startsrc -s inetd
0513-029 The inetd Subsystem is already active.
Multiple instances are not supported.

# ps -el | grep srcmstr
  240001 A   0  3946     1   0  60 20 10110   332              -  0:00 srcmstr

# kill -9 3946

# ps -el | grep srcmstr
  240001 A   0  4016     1   1  60 20 1e13e   220              -  0:00 srcmstr

# ps -el | grep inetd
  240001 A   0  7522     1   0  60 20 6806   212              -  0:00 inetd

# startsrc -s inetd
0513-059 The inetd Subsystem has been started. Subsystem PID is 16318.

# ps -el | grep inetd
  240001 A   0  7522     1   0  60 20 6806   212              -  0:00 inetd
  240001 A   0 16318  4016   8  64 20 1801   196              -  0:00 inetd
```

Figure 140. Starting Two Instances of the inetd *Daemon through the SRC*

Even if the srcmstr or inetd daemons are not terminated and restarted, they cannot prevent another instance of a daemon from being started from the command line. For example, even though the inetd daemon is defined to be an SRC subsystem which does not allow multiple instances, another instance of the

inetd daemon can be started from the command line. This is illustrated in Figure 141 on page 234.

```
$ lssrc -a | grep inetd
 inetd            tcpip            5524    active

$ ps -el | grep inetd
  240001 A   0 5524 3946   0  60 20 12112   212              -  0:00 inetd

$ startsrc -s inetd
0513-029 The inetd Subsystem is already active.
Multiple instances are not supported.

$ /usr/sbin/inetd

$ ps -el | grep inetd
  240001 A   0 5524 3946   0  60 20 12112   212              -  0:00 inetd
   40001 A   0 7142    1   9  64 20 191b9   208              -  0:00 inetd
```

Figure 141. Starting a Second Instance of the inetd *Daemon from the Command Line*

9.2 Ensuring Exclusivity Within a Daemon

A daemon should use the capabilities of the system to help ensure that the exclusivity requirements of the daemon are met. Even with the flaws previously discussed, these capabilities are useful. For example, the advantage of using the capabilities of the SRC in this regard is that the SRC can deliver a message to the originator of the startsrc command. However, a daemon serious about exclusivity should also determine itself if a new instance of the daemon should be permitted. The advantage of the daemon checking for exclusivity problems is that the daemon check can be independent of how the daemon was started. Unlike the SRC, when a daemon detects a problem, it can only log an error message and terminate the process. The originator of the daemon may not see the logged error message.

There are several ways a daemon program might determine if multiple instances of the daemon program are running. Two generic techniques that come to mind are file locking and System V semaphores. A daemon program could use fcntl to attempt to get an advisory lock on a file when the daemon program is started. If a process running the daemon program can get the lock, it can hold the lock for as long as it is running. If a process running the daemon program cannot get the lock, that is an indication that another process is running the daemon; the process without the lock should log an error message, and terminate. What file should be locked? It may be convenient to lock the file holding the daemon program executable[48]. A lock on a file obtained through fcntl is released automatically when the process holding the lock terminates. This is an important feature; if a daemon terminates unexpectedly, it will automatically release the lock that might prevent other processes from running the daemon program.

[48] However, the NFS problem discussed in the next paragraph might be more likely to happen if this file is used.

A problem with using fcntl to ensure daemon exclusivity is that the lock is recognized across systems if the file is in a file system mounted by NFS or exported using NFS. If the file being locked by the daemon is shared across systems using NFS, a daemon running on one system could prevent the daemon from running on another system. This is usually not desirable. The next technique discussed does not suffer from this problem.

A technique similar to file locking is to use a System V semaphore. A daemon program could attempt to lock a semaphore when the daemon program is started. If a process running the daemon program can lock the semaphore, it can hold the lock on the semaphore for as long as it is running. If a process running the daemon program cannot lock the semaphore, that is an indication that another process is running the daemon; the process without the lock should log an error message, and terminate. It is common to generate a semaphore key from a file name. What file name should be used? It may be convenient to use the name of the file holding the daemon program executable. When a semaphore is used properly, a lock on the semaphore will be removed when the process holding the lock terminates. As described in the fcntl discussion, this is an important attribute for daemons using semaphores to enforce exclusivity.

Semaphores are not shared across systems. Even if the file used to generate a semaphore key is shared across systems using NFS, each system will generate its own semaphore based on the key. A daemon running on one system would not prevent the daemon from running on another system.

The daemon support routines allow a daemon to enforce exclusivity requirements through the dae_init_exclusive and dae_init routines. The daemon support routines use System V semaphores to enforce the exclusivity requirements. These routines include code similar to that shown in Figure 142 on page 236. The code in get_excl is executed when a daemon initializes itself. The routine is passed two parameters,excl_path and excl_ID. The values passed through these two parameters are used in the call to ftok that generates the key of the semaphore that will be used by the process executing the routine. Different values for excl_path and excl_ID will generate different keys. These values can be used to control the degree of exclusivity needed. For example, if only one process should be able to run a particular daemon program at a time, all processes running that daemon program should pass identical parameters to get_excl. Each daemon program should use a different value for excl_path, so they do not interfere with each other. If one process per RS/6000 SP system partition should be able to run a specific daemon program, all calls to get_excl from that daemon program might use the same value for excl_path, to represent the daemon program, but different values might be used for excl_ID, to represent the RS/6000 SP partition for which the daemon would run.

```
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>

static key_t sem_key;                    /* Semaphore key                */
static int   sem_ID;                     /* Semaphore ID                 */
static pid_t sem_PID_lock = (pid_t)-1;   /* Process that locked semaphore */


void release_excl(void)
{
   if (getpid() == sem_PID_lock) {
         (void) semctl(sem_ID, 0, IPC_RMID, 0);
   }
   return;
}


int get_excl(const char *excl_path, char excl_ID)
{
   int   sem_perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;
   struct sembuf sem_ops[2] = {{0, 0, IPC_NOWAIT}, {0, 1, SEM_UNDO}};
   int rc;

   if ((sem_key = ftok(excl_path, excl_ID)) == (key_t)-1) {
         return 1;
   }

   if ((sem_ID = semget(sem_key, 1, IPC_CREAT | sem_perms)) == -1) {
         return 1;
   }

   do {
         rc = semop(sem_ID, sem_ops, 2);
   } while ((rc == -1) && (errno == EINTR));

   if (rc == -1) {
         return 1;
   }

   sem_PID_lock = getpid();
   (void) atexit(release_excl);
   return 0;
}
```

Figure 142. Using a Semaphore to Enforce Daemon Exclusivity

Once the key of the semaphore is obtained by ftok, the key is passed to semget to access the semaphore represented by that key. If the semaphore already exists on the system, semget will access the semaphore without changing its value. The semaphore will be created and initialized if it does not already exist in the system, because the IPC_CREAT flag is specified on the call to semget. The

semget routine initializes a newly created semaphore to 0. There is no way to initialize a System V semaphore to 1 in an atomic fashion. This is inconvenient, because the semaphore should be created in the unlocked state, and the traditional value for an unlocked semaphore is 1. To avoid a race condition that would exist if the semaphore's value is set to 1 by a call to semctl after semget creates it, the daemon support code uses the convention that a value of 0 means the semaphore is not locked, and a value of 1 means the semaphore is locked. Adopting this non-traditional convention with System V semaphores to avoid a race condition is suggested in *UNIX Network Programming*, W. Richard Stevens, 1990, Prentice Hall, Englewood Cliffs.

Once the semaphore is accessed, the semop routine is used to attempt to lock the semaphore. This is done with the operations defined in the sem_ops array. The first operation will test to see if the semaphore's value is 0. If the value is not 0, the semop routine will return an error; if the IPC_NOWAIT flag had not been specified with the semaphore operation, the process would have blocked. If the semaphore's value is 0, semop does not return; it performs the next operation in the sem_ops array. This operation changes the value of the semaphore to 1, locking the semaphore. Once the semaphore is locked, semop returns with a return code of 0. Notice that the SEM_UNDO flag is specified in the second semaphore operation. This is very important. If the process terminates without having explicitly unlocked the semaphore, it will automatically be unlocked. The SEM_UNDO flag causes the system to undo the action performed by the semaphore operation if it has not been undone by the time the process terminates. This allows another process to run the daemon program after the first process terminates unexpectedly.

If the semop routine returns an error, the semaphore could not be locked, meaning the exclusivity requirements of the daemon program could not be met by the process; the get_excl routine returns an error, and ultimately, the dae_init routine returns an error. When dae_init returns an error, the process should exit. If the semop routine does not return an error, the process has locked the semaphore, and it may proceed to run the daemon program. In this case, get_excl registers the release_excl routine as an exit handler by calling atexit. If the process terminates by calling exit, the release_excl routine will be called. The release_excl routine will remove the semaphore from the system, which also automatically unlocks it. If the process terminates without having called exit, the lock on the semaphore is released because SEM_UNDO was specified when the semaphore was locked, but the semaphore is not removed from the system. The release_excl routine is not absolutely necessary, but it is a good practice to clean up System V IPC objects whenever possible.

Note that the get_excl routine records the process ID (PID) of the process that obtained a lock on the semaphore, and that the release_excl routine only removes the semaphore if the process which locked the semaphore is executing release_excl. This is a necessary precaution. Suppose the process which called get_excl created a child process. If the child process does not execute another program, but exits while running the program that was inherited from its parent process, the child process will call release_excl. If care was not taken in release_excl, the child process could remove the semaphore that its parent locked. Note that a child process inherits its exit handlers from its parent.

Figure 143 on page 238 shows the code for a program named semtest1 that will be used to illustrate the effectiveness of the get_excl and release_excl routines. The semtest1 program is not a daemon, just a little test program. The program

calls get_excl with the path name of the file holding the program executable, and an ID that is obtained from a program argument. If the program argument is not provided, the ID defaults to 1. Notice that the program is designed to run until it receives the SIGTERM signal. When it receives the SIGTERM signal, the program will return from the main routine. When a program returns from the main routine, exit is automatically called. That is significant to this program, since get_excl installs release_excl as an exit handler. Figure 144 on page 239 shows the results of running the semtest1 program.

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


volatile sig_atomic_t stop_requested = 0;


void catch_term(int signo)
{
   stop_requested = 1;
   return;
}


int main(int argc, char **argv)
{
   struct sigaction sa;
   char excl_ID = 1;


   if (argc > 1) {
         excl_ID = atoi(argv[1]);
   }

   sa.sa_handler = catch_term;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = SA_RESTART;

   (void) sigaction(SIGTERM, &sa, NULL);

   if (get_excl("/u/agar/sem/semtest1", excl_ID) != 0) {
         fprintf(stderr, "%d: Violates exclusivity requirements.\n",
                 getpid());
         return 1;
   }

   while (stop_requested == 0) {
         (void) sigsuspend(0);
   }

   return 0;
}
```

Figure 143. Source for the semtest1 *Program*

```
1
$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:40:48 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system

$ ./semtest1 &
[1]     6320

$ ps -el | grep semtest1
  200001 A 200  6320  8520   0  64 24 92a9    56           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:41:07 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   4100 0x01080269 --ra-ra-r--     agar    staff

$ ./semtest1 &
[2]     8120
$ 8120: Violates exclusivity requirements.

[2] +  Done(1)                 ./semtest1 &

$ ps -el | grep semtest1
  200001 A 200  6320  8520   0  64 24 92a9    56           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:41:56 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   4100 0x01080269 --ra-ra-r--     agar    staff
```

Figure 144 (Part 1 of 4). Examples of Running the semtest1 *Program*

```
 2
$ ./semtest1 2 &
[2]     8128

$ ps -el | grep semtest1
  200001 A 200  6320  8520   0  64 24 92a9    56           pts/1  0:00 semtest1
  200001 A 200  8128  8520   0  64 24 e30e    60           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:42:25 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   4100 0x01080269 --ra-ra-r--     agar    staff
s      5 0x02080269 --ra-ra-r--     agar    staff

$ ./semtest1 1 &
[3]     10952
$ 10952: Violates exclusivity requirements.

[3] +  Done(1)                 ./semtest1 1 &

$ ./semtest1 2 &
[3]     10954
$ 10954: Violates exclusivity requirements.

[3] +  Done(1)                 ./semtest1 2 &

$ ps -el | grep semtest1
  200001 A 200  6320  8520   0  64 24 92a9    56           pts/1  0:00 semtest1
  200001 A 200  8128  8520   0  64 24 e30e    60           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:43:20 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   4100 0x01080269 --ra-ra-r--     agar    staff
s      5 0x02080269 --ra-ra-r--     agar    staff
```

Figure 144 (Part 2 of 4). Examples of Running the semtest1 *Program*

```
3
$ kill -TERM 6320

$ ps -el | grep semtest1
  200001 A 200  8128  8520   0  64 24 e30e    60           pts/1  0:00 semtest1

[1] -  Done                    ./semtest1 &

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:43:47 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s      5 0x02080269 --ra-ra-r--     agar    staff

$ ./semtest1 1 &
[1]     6360

$ ps -el | grep semtest1
  200001 A 200  6360  8520   0  64 24 102b0    60           pts/1  0:00 semtest1
  200001 A 200  8128  8520   0  64 24 e30e    60           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:44:21 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   8196 0x01080269 --ra-ra-r--     agar    staff
s      5 0x02080269 --ra-ra-r--     agar    staff
```

Figure 144 (Part 3 of 4). Examples of Running the semtest1 *Program*

```
4
$ kill -KILL 8128

$ ps -el | grep semtest1
  200001 A 200  6360  8520   0  64 24 102b0    60           pts/1  0:00 semtest1

[2] - Killed                  ./semtest1 2 &

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:45:07 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   8196 0x01080269 --ra-ra-r--     agar    staff
s      5 0x02080269 --ra-ra-r--     agar    staff

$ ./semtest1 2 &
[2]     8166

$ ps -el | grep semtest1
  200001 A 200  6360  8520   0  64 24 102b0    60           pts/1  0:00 semtest1
  200001 A 200  8166  8520   0  64 24 f30f    60           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:45:29 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   8196 0x01080269 --ra-ra-r--     agar    staff
s      5 0x02080269 --ra-ra-r--     agar    staff

$ ./semtest1 2 &
[3]     10990
$ 10990: Violates exclusivity requirements.

[3] +  Done(1)                ./semtest1 2 &
```

Figure 144 (Part 4 of 4). Examples of Running the semtest1 *Program*

Figure 144, part **1** shows that only one instance of semtest1 can be run at one time with the same excl_ID value. First, the ipcs command is used to display the semaphores that are currently defined on the system. Then, semtest1 is started and placed in the background. No arguments are given to semtest1, so the default excl_ID value is passed to the get_excl routine. A ps command shows that the semtest1 program is in fact running. The ipcs command is run again, showing the semaphore that was created by the call to get_excl in semtest1. Notice that the semaphore's key is 0x01080269. This key was generated by the call to ftok in get_excl. The value of the key was generated from the values passed to ftok, /u/agar/sem/semtest1 and 1. The semaphore's identifier is 4100. This identifier is a handle generated by the system for the semaphore. Its value

is of little importance. Next, an attempt is made to run another instance of semtest1 with the default value for excl_ID. This attempt fails, because the get_excl routine in the second process running semtest1 cannot lock the semaphore. A ps command confirms that the original process running semtest1 continues to be the only process running semtest1. The ipcs command confirms that the semaphore created by the first process running semtest1 continues to exist on the system.

Figure 144, part **2** shows that another instance of semtest1 can be running on the system if it uses an excl_ID value not currently used by any other process running semtest1 on the system. A process is started, specifying 2 as the value for excl_ID. A ps command shows that two processes are now running semtest1. An ipcs command shows that another semaphore has been created on the system. The key for this semaphore is slightly different than the key for the previously created semaphore. The different values used for excl_ID account for the difference. Attempts to start more processes running semtest1 with excl_ID values of 1 or 2 fail due to the inability to obtain the semaphore locks.

Figure 144, part **3** shows what happens when a process running semtest1 is terminated with the SIGTERM signal. The ipcs command indicates that the semaphore that had been locked by the terminated process is no longer defined in the system. This happened because the semtest1 process called exit to terminate. This caused the release_excl exit handler to execute. The release_excl routine called semctl to remove the semaphore from the system. Since the process running semtest1, which had specified a value of 1 for excl_ID, was terminated, it is now possible to run another process running semtest1 with the value of 1 for excl_ID. The second ipcs command run in part **3** shows that a semaphore with key 0x01080269 has been recreated. The recreated semaphore has a different semaphore identifier than it had before, which is proper.

Figure 144, part **4** shows what happens when a process running semtest1 is terminated in such a way that the process does not execute the exit routine. In this example, the SIGKILL signal is used to terminate the process. The ipcs command indicates that the semaphore that had been locked by the terminated process is still defined in the system. Even though the semaphore is still defined, it should not be locked. Since the process running semtest1, which had specified a value of 2 for excl_ID, was terminated, it should now be possible to run another process running semtest1 with the value of 2 for excl_ID. The example shows that it is. The second ipcs command run in part **4** shows that an additional semaphore was not created, as expected.

There is a limitation to using the ftok routine to generate semaphore keys. The key generated by ftok for a specific file name and identifier may change if the file referred to by the file name is removed and recreated. This is illustrated in Figure 145 on page 244. Once the semtest1 program is removed and recreated, another semtest1 process can be run with an excl_ID value of 1. So, if a daemon is using a semaphore to enforce its exclusivity requirements, it is recommended that processes running the daemon program should be stopped before a new version of the program is installed.

```
$ ps -el | grep semtest1
  200001 A 200  6360  8520   0  64 24 102b0    60           pts/1  0:00 semtest1
  200001 A 200  8166  8520   0  64 24 f30f    60           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:45:29 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   8196 0x01080269 --ra-ra-r--     agar    staff
s      5 0x02080269 --ra-ra-r--     agar    staff

$ ./semtest1 1 &
[3]     9866
$ 9866: Violates exclusivity requirements.

$ cp semtest1 semtest1.tmp; rm -f semtest1; mv semtest1.tmp semtest1

$ ./semtest1 1 &
[3]     9028

$ ps -el | grep semtest1
  200001 A 200  6360  8520   0  64 24 102b0    60           pts/1  0:00 semtest1
  200001 A 200  8166  8520   0  64 24 f30f    60           pts/1  0:00 semtest1
  200001 A 200  9028  8520   0  64 24 e30e    56           pts/1  0:00 semtest1

$ ipcs -s
IPC status from /dev/mem as of Fri May 19 19:47:38 1995
T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
s   4096 0x4d06004f --ra-ra----     root   system
s      1 0x620588b8 --ra-r--r--     root   system
s      2 0x03141592 --ra-ra----     root   system
s      3 0x0105884d --ra-------     root   system
s   8196 0x01080269 --ra-ra-r--     agar    staff
s      5 0x02080269 --ra-ra-r--     agar    staff
s      6 0x0108026a --ra-ra-r--     agar    staff
```

Figure 145. Failure of Semaphore When File Is Recreated

Chapter 10. Daemon Support Routines

The daemon support routines are a collection of routines that take care of many of the issues discussed in this document. It is hoped that a daemon writer will find that these routines make the job of initializing a daemon, and allowing it to run under the control of init, inetd, or the SRC, easier.

10.1 Introduction to the Daemon Support Routines

Several of the daemon support routines deal with initializing a daemon. The dae_init routine actually initializes the daemon. Other routines, those with the dae_init_ prefix, are designed to be called before dae_init; calling these routines before dae_init will influence how dae_init initializes the daemon. The daemon support routines dealing with daemon initialization are:

- dae_init - This routine implements daemon initialization. It deals with the issues discussed in Chapter 3, “Daemon Initialization” on page 19. It may also deal with the issues of paging space allocation policies, SRC unique initialization, and daemon exclusivity. A parameter to dae_init can limit how the daemon may be started. See A.1, “dae_init(3)” on page 316, the dae_init man page, for details.
- dae_init_SRC_sig - When called before dae_init from a daemon expecting signal communication from the SRC, this routine can affect how the daemon will handle requests from the SRC. This routine also affects how the daemon will handle the SIGTERM signal, whether it is under SRC control or not. See A.2, “dae_init_SRC_sig(3)” on page 321, the dae_init_SRC_sig man page, for details.
- dae_init_SRC_msq - When called before dae_init from a daemon expecting message queue communication from the SRC, this routine can affect how the daemon will handle requests from the SRC. This routine also affects how the daemon will handle the SIGTERM signal, whether the daemon is under SRC control or not. See A.3, “dae_init_SRC_msq(3)” on page 325, the dae_init_SRC_msq man page, for details.
- dae_init_SRC_sock - When called before dae_init from a daemon expecting socket communication from the SRC, this routine can affect how the daemon will handle requests from the SRC. This routine also affects how the daemon will handle the SIGTERM signal, whether the daemon is under SRC control or not. See A.4, “dae_init_SRC_sock(3)” on page 328, the dae_init_SRC_sock man page, for details.
- dae_init_term_sig - When called before dae_init from a daemon never expecting to be controlled by the SRC, this routine can affect how the daemon will handle the SIGTERM signal. See A.5, “dae_init_term_sig(3)” on page 336, the dae_init_term_sig man page, for details.
- dae_init_prevent_zombies - When called before dae_init, this routine will prevent the daemon process from creating long lasting zombie processes. See A.6, “dae_init_prevent_zombies(3)” on page 338, the dae_init_prevent_zombies man page, for details.
- dae_init_lowps - When called before dae_init, this routine will cause the daemon process to be protected from termination when the system is low on paging space. See A.7, “dae_init_lowps(3)” on page 340, the dae_init_lowps man page, for details.

- dae_init_psalloc - When called before dae_init, this routine will cause the daemon process to run with the specified paging space allocation policy. See A.8, “dae_init_psalloc(3)” on page 342, the dae_init_psalloc man page, for details.
- dae_init_exclusive - When called before dae_init, this routine will ensure the daemon is running in accordance with the exclusivity requirements of the daemon. See A.9, “dae_init_exclusive(3)” on page 346, the dae_init_exclusive man page, for details.

When a daemon is running under the control of the SRC, and it expects requests from the SRC to come through a message queue or a socket, the daemon must monitor the message queue or socket. When the message queue or socket can be read, the daemon must read the SRC request, decode the request, perform appropriate actions to satisfy the request, and send results to the location indicated in the request. The daemon support routines provide several routines to handle much of the work associated with SRC requests. These routines are:

- The dae_SRC_req routine reads the next SRC request from a message queue or socket, decodes the request, calls a daemon-specific routine to perform the appropriate actions to satisfy the request, and sends the results to the location indicated in the request. See A.10, “dae_SRC_req(3)” on page 349, the dae_SRC_req man page, for details. The daemon must still monitor the message queue or socket to determine when dae_SRC_req should be called.
- The dae_status_short routine allows a daemon to incorporate the SRC subsystem short status output in its subsystem long status output. See A.11, “dae_status_short(3)” on page 351, the dae_status_short man page, for details.
- The dae_status_puts and dae_margin_puts routines provide interfaces similar to puts for specifying the contents of the daemon′s subsystem long status output. See A.12, “dae_status_puts(3)” on page 354, the dae_status_puts man page, and A.14, “dae_margin_puts(3)” on page 360, the dae_margin_puts man page, for details.
- The dae_status_printf and dae_margin_printf routines provide interfaces similar to printf for specifying the contents of the daemon′s subsystem long status output. See A.13, “dae_status_printf(3)” on page 357, the dae_status_printf man page, and A.15, “dae_margin_printf(3)” on page 364, the dae_margin_printf man page, for details.
- The dae_inform_puts and dae_inform_printf routines provide interfaces similar to puts and printf for specifying informational messages to be displayed on the standard output of a process making a subsystem trace on, trace off, refresh, or long status request. These routines can also be used to display informational messages on the standard output of a process making a subsystem-defined request. See A.16, “dae_inform_puts(3)” on page 368, the dae_inform_puts man page, and A.17, “dae_inform_printf(3)” on page 371, the dae_inform_printf man page, for details.
- The dae_error_puts and dae_error_printf routines provide interfaces similar to puts and printf for specifying error messages to be sent to a process making a subsystem trace on, trace off, refresh, or long status request. These routines can also be used to specify error messages to be sent to a process making a subsystem-defined request. See A.18, “dae_error_puts(3)” on page 374, the dae_error_puts man page, and A.19, “dae_error_printf(3)” on page 377, the dae_error_printf man page, for details.

The daemon support routines include a routine that indicates if the calling process has initialized itself as a daemon with the dae_init routine. See A.20, “dae_process_is_the_daemon(3)” on page 380, the dae_process_is_the_daemon man page, for details.

The daemon support routines include a routine that provides the same interface and function as fopen, but protects the process from obtaining a controlling terminal. See A.21, “dae_fopen(3)” on page 381, the dae_fopen man page, for details.

The header file associated with the daemon support routines, dae.h, defines a macro that will conditionally stop a process until a debugger is attached to the process. The process will be stopped only if the environment includes a specific value for a specific variable. Invoking this macro at the very beginning of the main routine in a daemon can help in debugging daemon initialization. See A.22, “DAE_M_STOP(3)” on page 382, the DAE_M_STOP man page, for details.

10.2 Examples of Using the Daemon Support Routines

This section includes examples illustrating how a daemon may use the daemon support routines.

10.2.1 Running under the inetd Process

Figure 146 on page 248 shows an example of how a daemon designed to run under the control of inetd might use dae_init to initialize itself. The constant DAE_P_INETD is passed indirectly through the first parameter. This will cause dae_init to verify that the daemon has been started by inetd. If the daemon has not been started by inetd, dae_init will return with an error value. The second parameter is specified as the address of an area into which detailed error data may be placed. When dae_init returns, the program compares the return value from dae_init with DAE_E_OK, the return value from the daemon support routines that indicates success. If the return value from dae_init is DAE_E_OK, the program can continue to run as a daemon. If the return value from dae_init is not DAE_E_OK, dae_init could not initialize the daemon properly. In this case, the daemon might want to put a message in the error log which includes information from err_detail; an example is shown in 10.2.10, “Logging Detail Data” on page 285. Since the daemon could not be initialized as a daemon, the program should exit. In this example, the exit value is based on the return value from dae_init.

```
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200


int main(int argc, char **argv)
{
   dae_parent_t parent;
   dae_error_detail_t err_detail;
   int rc;


   parent = DAE_P_INETD;

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   /* Continue, as the daemon ... */

   return 0;
}
```

Figure 146. Initializing an inetd *Daemon with the* dae_init *Routine*

Here is a summary of the state of the daemon when dae_init returns DAE_E_OK in Figure 146:

- The daemon was started by inetd.
- The daemon is ignoring the SIGHUP, SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, and SIGTTOU signals.
- The daemon is running in the process that was started by inetd.
- If the daemon is running on an AIX system, it is running with the paging space allocation policy that it inherited from inetd.
- The daemon process is a session leader without a controlling terminal.
- The daemon process is ignoring the SIGPIPE signal.
- No action has been taken to prevent the daemon from producing zombie processes.
- The disposition for the SIGTERM signal is the default. That is, when the SIGTERM signal is delivered to the process, the process will terminate immediately.
- No signal handler is installed for the SIGDANGER signal. If the daemon is running on an AIX system, and the number of free paging-space blocks on the system falls below the paging-space kill threshold, the daemon process may be terminated by the kernel.
- All file descriptors, except for file descriptors 0, 1, and 2, are closed.
- File descriptors 0, 1, and 2 are associated with the socket through which the daemon is to communicate with its client.
- The current working directory of the daemon process is root, /.

- The file mode creation mask of the daemon process is 0.
- Other processes on the system may or may not be running the daemon program.

Figure 147 shows a modified version of the code in Figure 146 on page 248. The modified code adds a routine, term_daemon, that is meant to clean up the daemon when the daemon is requested to terminate through the SIGTERM signal. A call to dae_init_term_sig was added; as a result, dae_init will install term_daemon as the signal handler for the SIGTERM signal.

```
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200


void term_daemon(int signo)
{
   /* clean up the daemon */

   _exit(0);
}


int main(int argc, char **argv)
{
   dae_parent_t parent;
   dae_error_detail_t err_detail;
   int rc;


   parent = DAE_P_INETD;

   dae_init_term_sig(term_daemon, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   /* Continue, as the daemon ... */

   return 0;
}
```

Figure 147. Initializing an inetd *Daemon with the* dae_init *Routine*

The state of the daemon initialized in Figure 147 is the same as the state of the daemon initialized in Figure 146 on page 248, with the following exception:

- The term_daemon routine has been installed as the signal handler for the SIGTERM signal. Therefore, when the SIGTERM signal is delivered to the process, the term_daemon routine will run, and will clean up and terminate the process.

10.2.2 Running under the SRC: Signal Communication

This section presents some examples of how a daemon to be controlled by SRC through signal communication might use dae_init to initialize itself.

Figure 148 shows the first example. The value used for the first parameter of dae_init is DAE_P_SRC. This will cause dae_init to ensure that the daemon is running under the control of the SRC. If the daemon is not running under the control of the SRC, dae_init will return an error indication. Since the daemon is allowed to run only under SRC control, this daemon assumes it will only be compiled and run on an AIX system.

```
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200


int main(int argc, char **argv)
{
   dae_parent_t parent;
   dae_error_detail_t err_detail;
   int rc;


   parent = DAE_P_SRC;

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   /* Continue, as the daemon ... */

   return 0;
}
```

Figure 148. Initializing an SRC Daemon

Here is a summary of the state of the daemon when dae_init returns DAE_E_OK in Figure 148:

- The daemon was started by the SRC.
- The daemon is ignoring the SIGHUP, SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, and SIGTTOU signals.
- The daemon is running in the process that was started by the SRC.
- The daemon is running with a paging space allocation policy that was determined by the environment set up for the process by the SRC. This environment is essentially the environment defined in /etc/environment.
- The daemon process is a session leader without a controlling terminal.
- The daemon process is ignoring the SIGPIPE signal.
- No action has been taken to prevent the daemon from producing zombie processes.

- The disposition for the SIGTERM signal is the default. That is, when the SIGTERM signal is delivered to the process, the process will terminate immediately. Because dae_init_SRC_sig was not called, the daemon support routines assume that the subsystem stop normal and subsystem stop forced signals defined for the daemon are SIGTERM.
- No signal handler is installed for the SIGDANGER signal. If the number of free paging-space blocks on the system falls below the paging-space kill threshold, the daemon process may be terminated by the kernel.
- All file descriptors, except for file descriptors 0, 1, and 2, are closed.
- File descriptors 0, 1, and 2 are open and associated with the files specified in the SRC definition of the subsystem.
- The current working directory of the daemon process is root, /.
- The file mode creation mask of the daemon process is 0.
- Other processes on the system may or may not be running the daemon program.

Figure 149 on page 252 shows an example of how a daemon designed to run under the control of the SRC might use dae_init to initialize itself if it can be compiled and run on non-AIX systems, and it can be compiled in a debugging mode. The value used for the first parameter of dae_init depends on how the program is compiled. If the program is compiled on an AIX system (_AIX is defined) without debugging code (_DEBUG is not defined), DAE_P_SRC is used. This will cause dae_init to ensure that the daemon is running under the control of the SRC. If the program is compiled on an AIX system with debugging code (_DEBUG is defined), the values DAE_P_SRC and DAE_P_OTHER are used. This will allow the daemon to be started from the SRC, a shell, or init. Running the daemon from the command line may be helpful while debugging. If the program is compiled on a system other than AIX, the SRC is not available. Therefore, DAE_P_OTHER is used to allow the daemon to be started from init or a shell.

```
#include <dae.h>

#if   defined(_AIX) && !defined(_DEBUG)
   #define ALLOWED_PARENT  DAE_P_SRC
#elif defined(_AIX) && defined(_DEBUG)
   #define ALLOWED_PARENT  (DAE_P_SRC | DAE_P_OTHER)
#else
   #define ALLOWED_PARENT  DAE_P_OTHER
#endif

#define SUBSYS_DAE_OFFSET  200


int main(int argc, char **argv)
{
   dae_parent_t parent;
   dae_error_detail_t err_detail;
   int rc;


   parent = ALLOWED_PARENT;

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   /* Continue, as the daemon ... */

   return 0;
}
```

Figure 149. Initializing an SRC Daemon

When the daemon in Figure 149 is started by the SRC, its state once dae_init returns successfully is identical to the state described for the daemon in Figure 148 on page 250. When the daemon in Figure 149 is started by init or a shell, its state differs from the state described for the daemon in Figure 148 on page 250 in the following ways:

- If the parent process of the process that called dae_init was not init, the daemon was migrated to another process. In this case, the process that returned from dae_init was created by the process that called dae_init, and the process that called dae_init no longer exists.
- If the daemon was started by init, and the daemon is running on an AIX system, the daemon is running with a paging space allocation policy that was determined by the environment set up for the process by init. This environment is essentially the environment defined in /etc/environment.

- If the daemon was started by a shell, and the daemon is running on an AIX system, the daemon is running with the paging space allocation policy determined by the environment inherited from the shell[49].
- File descriptors 0, 1, and 2 are open and associated with `/dev/null`.

The example in Figure 150 on page 254 assumes that the daemon definition specifies the `SIGUSR1` signal for the subsystem stop normal request and the `SIGUSR2` signal for the subsystem stop forced request. The example also supplies signal handlers for all three types of subsystem stop requests. The example calls `dae_init_SRC_sig` to arrange for these signal handlers to be installed. The signal handler installed to support the subsystem stop normal request, `normal_stop`, just sets a flag indicating that a stop request has been made. This flag should be checked periodically elsewhere in the program. The signal handler installed to support the subsystem stop forced and subsystem stop cancel requests, `quick_stop`, cleans up and terminates within the signal handler. The approach used for stop signal handlers is up to the daemon.

[49] When the daemon is started from a shell, the process flags may not correctly indicate the paging space allocation policy under which the daemon is running. See 8.5.1, "Determining the Paging Space Allocation Policy of a Process" on page 216 for an explanation. The problem can be avoided by specifying a specific paging space allocation policy in a call to the routine `dae_init_psalloc`. Information about the `dae_init_psalloc` routine can be found in 10.2.7, "Setting the Paging Space Allocation Policy" on page 279.

```
#include <signal.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200

volatile sig_atomic_t stop_requested = 0;


void normal_stop(int signo)
{
   stop_requested = 1;
   return;
}

void quick_stop(int signo)
{
   /* Cleanup the daemon quickly ... */

   _exit(0);
}


int main(int argc, char **argv)
{
   dae_parent_t       parent = DAE_P_SRC;
   dae_req_sig_t      SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;


   SRC_rtns.dae_stop_normal = normal_stop;
   SRC_rtns.dae_stop_forced = quick_stop;
   SRC_rtns.dae_stop_cancel = quick_stop;

   dae_init_SRC_sig(SIGUSR1, SIGUSR2, &SRC_rtns, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   /* Continue, as the daemon ... Look at stop_requested periodically */

   return 0;
}
```

Figure 150. Using dae_init_SRC_sig *to Specify Stop Request Signal Handlers*

10.2.3 Running under the SRC: Message Queue Communication

Figure 151 on page 256 presents an example of a daemon designed to run under SRC control with message queue communication. In the main routine, dae_init_SRC_msq is called to provide the daemon support routines with information about the message queue through which SRC requests are expected and the addresses of routines that will handle SRC requests. The first

parameter to dae_init_SRC_msq points to an integer that will be set to the identifier of the message queue through which SRC requests will be received. The second parameter provides the key of that message queue. The third parameter indicates the type of the SRC request messages expected on the queue. Through the fourth parameter, the daemon provides the addresses of routines that will handle SRC requests. The subsystem stop normal and subsystem stop forced requests will be handled by msq_stop. The subsystem stop cancel request will be handled by signal_stop. The subsystem long status request will be handled by report_status. The daemon indicates that it does not support the subsystem trace requests or the subsystem refresh request, by specifying NULL pointer values for the dae_trace_begin, dae_trace_end, and dae_refresh members of the structure whose address is provided in the fourth parameter. The daemon also indicates that it does not support other subsystem-defined requests, by specifying the NULL pointer value for the dae_other_req member of the structure whose address is provided in the fourth parameter. The fifth parameter is set to 1; this will cause the signal_stop routine to be installed as a signal handler such that if it interrupts a restartable system call, the system call will be restarted automatically.

```
1
#include <signal.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200
#define MSQ_KEY  0xBADF00D
#define MSQ_TYPE 0xBEEF

volatile sig_atomic_t stop_requested = 0;
int msqid;

static void msq_stop(void);
static void signal_stop(int signo);
static void report_status(void);
static void do_daemon_work(void);


int main(int argc, char **argv)
{
   dae_parent_t       parent;
   dae_req_msq_t      SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;


   SRC_rtns.dae_stop_normal = msq_stop;
   SRC_rtns.dae_stop_forced = msq_stop;
   SRC_rtns.dae_stop_cancel = signal_stop;
   SRC_rtns.dae_trace_begin = NULL;
   SRC_rtns.dae_trace_end   = NULL;
   SRC_rtns.dae_refresh     = NULL;
   SRC_rtns.dae_long_status = report_status;
   SRC_rtns.dae_other_req   = NULL;

   dae_init_SRC_msq(&msqid, MSQ_KEY, MSQ_TYPE, &SRC_rtns, 1);

   parent = DAE_P_SRC;

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   do_daemon_work();

   return 0;
}
```

Figure 151 (Part 1 of 3). An SRC Daemon Using Message Queue Communication

```
2
#include <time.h>
#include <dae.h>


static void msq_stop(void)
{
   stop_requested = 1;
   return;
}


static void signal_stop(int signo)
{
   stop_requested = 1;
   return;
}


static void report_status(void)
{
   time_t current_time;

   dae_status_short();

   dae_status_printf("\nAll is well with process %d at ", getpid());

   current_time = time(NULL);
   dae_status_puts(ctime(&current_time));

   return;
}
```

Figure 151 (Part 2 of 3). An SRC Daemon Using Message Queue Communication

```
3
#include <unistd.h>
#include <sys/select.h>
#include <sys/types.h>
#include <time.h>
#include <dae.h>

static void do_daemon_work(void)
{
   struct sellist {
         int msqids[1];
   };
   struct sellist read_list;
   struct timeval time_out;
   time_t start_time, end_time, current_time;
   const time_t sample_rate = 5 * 60;       /* Sample every 5 minutes. */
   const int nmsq = 1 << 16;
   int rval;


   while (!stop_requested) {

         /* Do the actual work of the daemon here, perhaps. */

         start_time = time(NULL);
         end_time = start_time + sample_rate;

         current_time = start_time;

         while ((current_time < end_time) && (!stop_requested)) {

              time_out.tv_sec  = end_time - current_time;
              time_out.tv_usec = 0;
              read_list.msqids[0] = msqid;
              rval = select(nmsq, &read_list, NULL, NULL, &time_out);

              if ((rval == nmsq) && (read_list.msqids[0] == msqid)) {
                   dae_SRC_req();
              }

              current_time = time(NULL);
         }

   }

   return;
}
```

Figure 151 (Part 3 of 3). An SRC Daemon Using Message Queue Communication

In Figure 151 on page 256, dae_init is called to actually initialize the daemon. The value passed through the first parameter, DAE_P_SRC, causes dae_init to succeed only if the daemon is running under SRC control. The dae_init routine will access the message queue with the key provided as the second parameter of dae_init_SRC_msq. The message queue is accessed with a call to msgget. The message queue identifier returned by msgget is, in turn, returned to the caller of dae_init through the pointer provided as the first parameter to dae_init_SRC_msq.

Figure 151, part **2** shows the routines that will handle SRC requests. The msq_stop and signal_stop routines will, when invoked, just set a global variable, stop_requested, to indicate that a stop request has been made. The do_daemon_work routine will periodically look at the setting of stop_requested to determine if the daemon is to be stopped. The report_status routine provides the output for a subsystem long status request by calling daemon support routines that allow for specification of long status output. Calling the dae_status_short routine causes the subsystem short status to be included in the long status output. The report_status routine uses dae_status_printf to specify output in a manner similar to printf, and dae_status_puts to specify output in a manner similar to fputs.

Figure 151, part **3** shows the do_daemon_work routine, which will presumably do the work the daemon is designed to do. This routine is structured for a daemon that will sample some data at some regular interval, in this case every 5 minutes. The routine uses select with a time out value. The time out value specified is the time remaining until the expiration of the sampling interval. The select system call will return when the time out value has expired, the SRC request message queue has a message that can be read, or when it is interrupted by a signal handler returning. The do_daemon_work routine is coded so the data sample is taken at the proper interval. This may involve several calls to select per interval if select returns before the time out specified has expired. If the select routine indicates data can be read from the message queue, the dae_SRC_req routine is called. When dae_SRC_req returns, select is called again if there is more time to wait before the expiration of the sampling interval, and a stop request has not been received.

The dae_SRC_req routine will read a request from the message queue and determine what type of request is being made. If a request-handling routine for the request had been specified on the call to dae_init_SRC_msq, the request-handling routine is called. For example, if the request is a subsystem long status request, the report_status routine is called. Once the report_status routine returns to dae_SRC_req, dae_SRC_req sends a response to the destination specified in the request. If dae_SRC_req reads a request for which the daemon has not provided a request-handling routine, dae_SRC_req sends an error response to the destination specified in the request.

The do_daemon_work routine will return to its caller, main, when the stop_requested variable has been set to a non-zero value. Then, main will terminate the program. The program will be terminated shortly after receiving any of the SRC stop requests. Consider a subsystem stop cancel request. Assume the subsystem stop cancel request is made while the daemon is blocked in the select call. The SIGTERM signal will be delivered to the daemon process. This will cause the execution of the signal_stop signal handler. The signal_stop routine sets stop_requested to 1, and returns. The select system call is not an automatically restarted system call, so it returns with a value of -1 (errno is set to EINTR). The do_daemon_work routine will see that stop_requested is not 0, and it will return to its caller.

Now consider a subsystem stop normal request. Again, assume the stop normal request is made while the daemon is blocked in the select call. A message will be placed in the message queue. The select routine will return, indicating the message queue can be read. The do_daemon_work routine will call dae_SRC_req. The dae_SRC_req routine will read a message from the message queue. It will interpret the request as a subsystem stop normal request. It will then call

msq_stop. The msq_stop routine will set the stop_requested global variable to 1. It will then return to dae_SRC_req, which will return to do_daemon_work. The do_daemon_work routine will see that stop_requested is not 0, and it will return to its caller. A similar sequence of events would have occurred had a subsystem stop forced request been received.

Figure 152 shows how the daemon shown in Figure 151 on page 256 can be controlled with SRC commands. First, the SRC definition of the subsystem is done with mkssys. The example assumes the daemon program exists as /usr/lpp/somelpp/bin/SRC.msq. The subsystem is defined with the name des.ex.msq, in a group named des.ex. The -I and -m flags used with mkssys indicate the SRC should make requests of the subsystem through a message queue. The -I flag argument specifies a message queue key of 195948557, which is the decimal equivalent of the 0xBADF00D constant used in the program. The -m flag argument specifies a message type of 48879, which is the decimal equivalent of the 0xBEEF constant used in the program. The subsystem can be started and stopped, and it can provide long status. Requests to start tracing, end tracing, and refresh the subsystem fail, since the daemon did not provide routines to handle these requests.

```
# mkssys -s des.ex.msq -p /usr/lpp/somelpp/bin/SRC.msq -u 0 \
         -i /dev/null -o /dev/null -e /dev/null -I 195948557 -m 48879 -G des.ex
0513-071 The des.ex.msq Subsystem has been added.

# startsrc -s des.ex.msq
0513-059 The des.ex.msq Subsystem has been started. Subsystem PID is 8522.

# lssrc -s des.ex.msq
Subsystem         Group            PID     Status
 des.ex.msq       des.ex           8522    active

# lssrc -ls des.ex.msq
Subsystem         Group            PID     Status
 des.ex.msq       des.ex           8522    active

 All is well with process 8522 at Mon Apr 24 16:11:58 1995

# refresh -s des.ex.msq
0513-129 The  subsystem does not support the requested action.

# traceson -s des.ex.msq
0513-129 The  subsystem does not support the requested action.

# tracesoff -s des.ex.msq
0513-129 The  subsystem does not support the requested action.

# stopsrc -s des.ex.msq
0513-044 The stop of the des.ex.msq Subsystem was completed successfully.

# lssrc -s des.ex.msq
Subsystem         Group            PID     Status
 des.ex.msq       des.ex                   inoperative
```

Figure 152. Controlling the Message Queue Daemon with SRC

10.2.4 Running under the SRC: Socket Communication

Figure 153 presents an example of a daemon designed to run under SRC control with socket communication. The daemon will act as a concurrent server that communicates with its clients through sockets.

```
1
#include <unistd.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200

int stop_requested = 0;
int client_count = 0;
int acpt_sockd, SRC_sockd = -1;

static void normal_stop(void);
static void forced_stop(void);
static void cancel_stop(int signo);
static void begin_trace(int longtrace);
static void end_trace(void);
static void refresh_config(void);
static void report_status(void);
static int  prepare_socket(void);
static void do_daemon_loop(void);

int main(int argc, char **argv)
{
   dae_parent_t       parent = DAE_P_SRC;
   dae_req_sock_t     SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;

   SRC_rtns.dae_stop_normal = normal_stop;
   SRC_rtns.dae_stop_forced = forced_stop;
   SRC_rtns.dae_stop_cancel = cancel_stop;
   SRC_rtns.dae_trace_begin = begin_trace;
   SRC_rtns.dae_trace_end   = end_trace;
   SRC_rtns.dae_refresh     = refresh_config;
   SRC_rtns.dae_long_status = report_status;
   SRC_rtns.dae_other_req   = NULL;

   dae_init_prevent_zombies(parent, 1);
   dae_init_SRC_sock(&SRC_sockd, &SRC_rtns, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   if (prepare_socket() == 0)
         do_daemon_loop();

   return 0;
}
```

Figure 153 (Part 1 of 7). An SRC Daemon Using Socket Communication

```
2
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

static int prepare_socket(void)
{
   struct sockaddr_in saddr;
   struct servent *servent;
   int reuse_on = 1;


   if ((acpt_sockd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
         /* Log message; probably include the errno value. */
         return 1;
   }

   if (setsockopt(acpt_sockd, SOL_SOCKET, SO_REUSEADDR,
                        (char *) &reuse_on, sizeof reuse_on) == -1) {
         /* Log message; probably include the errno value. */
         return 1;
   }

   if ((servent = getservbyname("des.ex.sock", "tcp")) == NULL) {
         /* Log message; probably include the errno value. */
         return 1;
   }

   memset(&saddr, 0, sizeof saddr);
   saddr.sin_family = AF_INET;
   saddr.sin_addr.s_addr = INADDR_ANY;
   saddr.sin_port   = servent->s_port;

   if (bind(acpt_sockd, (struct sockaddr *) &saddr, sizeof saddr) == -1) {
         /* Log message; probably include the errno value. */
         return 1;
   }

   if (listen(acpt_sockd, SOMAXCONN) == -1) {
         /* Log message; probably include the errno value. */
         return 1;
   }

   return 0;
}
```

Figure 153 (Part 2 of 7). An SRC Daemon Using Socket Communication

```
3
#include <unistd.h>
#include <errno.h>
#include <sys/select.h>
#include <sys/types.h>
#include <dae.h>

#define MAX(x, y)  ( ((x) > (y)) ? (x) : (y) )

static void create_daemon_child(void);

static void do_daemon_loop(void)
{
   int numfds;
   fd_set read_fds;


   (void) refresh_config();

   numfds = MAX(acpt_sockd, SRC_sockd) + 1;

   while (!stop_requested) {

         FD_ZERO(&read_fds);
         FD_SET(acpt_sockd, &read_fds);
         FD_SET(SRC_sockd, &read_fds);

         if (select(numfds, &read_fds, NULL, NULL, NULL) == -1) {
              if (errno == EINTR) {
                   continue;
              }
              /* Log message */
              break;
         }

         if (FD_ISSET(acpt_sockd, &read_fds)) {
              create_daemon_child();
              client_count++;
         }

         if (FD_ISSET(SRC_sockd, &read_fds)) {
              dae_SRC_req();
         }

   }

   return;
}
```

Figure 153 (Part 3 of 7). An SRC Daemon Using Socket Communication

```
4
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void do_daemon_work(int serv_sockd);


static void create_daemon_child(void)
{
   int serv_sockd;
   struct sockaddr_in sock_addr;
   int sock_addr_len;
   pid_t childpid;


   do {
         sock_addr_len = sizeof sock_addr;
         serv_sockd = accept(acpt_sockd,
              (struct sockaddr *) &sock_addr, &sock_addr_len);

   } while ((serv_sockd == -1) && (errno == EINTR));

   if ((childpid = fork()) == -1) {
         /* Log message; probably include the errno value. */
         close(serv_sockd);
         return;
   }

   if (childpid != 0) {             /* This is the parent process. */
         close(serv_sockd);
         return;
   }

   /* This is the child process. */

   close(acpt_sockd);

   do_daemon_work(serv_sockd);

   close(serv_sockd);
   exit(0);

   return;                          /* Should not reach. */
}
```

Figure 153 (Part 4 of 7). An SRC Daemon Using Socket Communication

```
5
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/socketvar.h>

static char config_path[MAXPATHLEN + 1] = "/etc/des.ex.sock.conf";
static char config_buf[80];
static int  config_cnt = 0;
static int  tracing_on = 0;


static void normal_stop(void)
{
   stop_requested = 1;
   return;
}


static void forced_stop(void)
{
   struct sigaction sa;


   sa.sa_handler = SIG_IGN;
   sigemptyset(&(sa.sa_mask));
   sa.sa_flags = 0;

   (void) sigaction(SIGTERM, &sa, NULL);
   (void) kill(0, SIGTERM);

   exit(0);
}

static void cancel_stop(int signo)
{
   if (dae_process_is_the_daemon()) {
         (void) kill(0, SIGTERM);
   }

   _exit(0);
}
```

Figure 153 (Part 5 of 7). An SRC Daemon Using Socket Communication

```
6
static void begin_trace(int longtrace)
{
   int debug_on = 1;
   const char *error_msg =
         "Subsystem des.ex.sock failed to put socket in debug mode.\n";


   if (setsockopt(acpt_sockd, SOL_SOCKET, SO_DEBUG,
                     (char *) &debug_on, sizeof debug_on) == -1) {
         /* Log message; probably include the errno value. */
         dae_error_puts(error_msg);
         return;
   }

   tracing_on = 1;

   return;
}


static void end_trace(void)
{
   int debug_on = 0;
   const char *error_msg =
         "Subsystem des.ex.sock failed to take socket out of debug mode.\n";


   if (setsockopt(acpt_sockd, SOL_SOCKET, SO_DEBUG,
                     (char *) &debug_on, sizeof debug_on) == -1) {
         /* Log message; probably include the errno value. */
         dae_error_puts(error_msg);
         return;
   }

   tracing_on = 0;

   return;
}
```

Figure 153 (Part 6 of 7). An SRC Daemon Using Socket Communication

```
7
static void refresh_config(void)
{
   int config_fd;
   int read_cnt;
   const char *error_msg =
         "Subsystem des.ex.sock failed to read its configuration file.\n";


   config_cnt = 0;

   if ((config_fd = open(config_path, O_RDONLY)) == -1) {
         /* Log message; probably include the errno value. */
         dae_error_puts(error_msg);
         return;
   }

   if ((read_cnt = read(config_fd, config_buf, sizeof config_buf)) == -1) {
         /* Log message; probably include the errno value. */
         (void) close(config_fd);
         dae_error_puts(error_msg);
         return;
   }

   config_cnt = read_cnt;

   (void) close(config_fd);

   return;
}


static void report_status(void)
{

   dae_status_short();

   dae_status_printf("\nI've seen %d client connections.\n", client_count);
   dae_status_puts(tracing_on ? "Tracing is on.\n" : "Tracing is off.\n");
   dae_status_printf("Next refresh will read from %s.\n", config_path);

   return;
}
```

Figure 153 (Part 7 of 7). An SRC Daemon Using Socket Communication

In the main routine, dae_init_SRC_sock is called to provide the daemon support routines with information about the socket through which SRC requests are expected and the addresses of routines that will handle SRC requests. The first parameter to dae_init_SRC_sock points to an integer that holds a file descriptor. When dae_init is executed, the value of this integer is interpreted as the file descriptor with which the daemon would like the SRC socket associated. The value specified in this example, -1, indicates no particular file descriptor is desired. When dae_init returns, the integer's value will be the actual file descriptor with which the SRC socket is associated. Through the second parameter of dae_init_SRC_sock, the daemon provides the addresses of routines

that will handle SRC requests. The subsystem stop normal request will be handled by normal_stop, the subsystem stop forced request by forced_stop, the subsystem stop cancel request by cancel_stop, the subsystem trace on request by begin_trace, the subsystem trace off request by end_trace, the subsystem refresh request by refresh_config, and the subsystem long status request by report_status. The third parameter to dae_init_SRC_sock is set to 1; this will cause the cancel_stop routine to be installed as a signal handler such that if it interrupts a restartable system call, the system call will be restarted automatically.

The main routine in Figure 153 on page 261 calls the dae_init_prevent_zombies routine. This daemon creates child processes, but does not care when they terminate or if they succeeded. The call to dae_init_prevent_zombies will ensure that this daemon does not create long-lasting zombie processes, and allows the daemon-specific code to forget about, for the most part, the child processes once they are created.

In Figure 153 on page 261, dae_init is called to actually initialize the daemon. The value passed through the first parameter, DAE_P_SRC, causes dae_init to succeed only if the daemon is running under SRC control. Once dae_init has verified the daemon has been started by SRC, it will verify that file descriptor 0 is associated with a socket. It will then associate the socket with a different file descriptor, and associate file descriptor 0 with /dev/null. The file descriptor associated with the socket is returned through the pointer specified with the first parameter of dae_init_SRC_sock.

In Figure 153, part **2**, the prepare_socket routine initializes the socket through which the daemon will establish connections with clients. This socket should not be confused with the socket through which the SRC will make requests of the daemon. The socket through which SRC will make requests of the daemon is a UNIX domain datagram socket that is initialized by srcmstr and associated with file descriptor 0 of the daemon process just before the daemon program is executed. The dae_init routine associates the socket with a file descriptor other than 0. The socket through which the daemon will accept connection requests from clients is a Internet domain stream socket that is initialized by the daemon itself. Throughout Figure 153, the SRC request socket's file descriptor is held in the variable SRC_sockd, and the file descriptor of the socket though which client connection requests are accepted is held in the variable acpt_sockd.

The prepare_socket routine sets up the Internet domain stream socket in a standard fashion. It calls socket, bind, and listen. Notice that before calling bind, it calls setsockopt to set the SO_REUSEADDR socket option. This allows the daemon to be successfully restarted after being stopped by the SRC subsystem stop normal, stop forced, and stop cancel requests. If the SO_REUSEADDR socket option is not set, attempts to start the daemon after it had been stopped might result in the bind routine returning the "address already in use" error.

Figure 153, part **3** shows the do_daemon_loop routine. This routine waits for a request to arrive, handles the request, and then waits for another request to arrive. The routine uses select with no time out value. The select call waits for client connection requests through the socket whose file descriptor is in acpt_sockd, and for SRC requests through the socket whose file descriptor is in SRC_sockd. If the select routine indicates a connection can be established with a client through the daemon connection socket, create_daemon_child is called to establish the connection and to create a child process to serve the client. If the

select routine indicates data can be read from the SRC request socket, dae_SRC_req is called. The dae_SRC_req routine will process the SRC request, and return.

The dae_SRC_req routine will read a request from the SRC request socket, and determine what type of request is being made. If a request-handling routine for the request had been specified on the call to dae_init_SRC_sock, the request-handling routine is called. For example, if the request is a subsystem long status request, the report_status routine is called. Once the report_status routine returns to dae_SRC_req, dae_SRC_req sends a response to the destination specified in the request. If dae_SRC_req reads a request for which the daemon has not provided a request-handling routine, dae_SRC_req sends an error response to the destination specified in the request.

Figure 153, part **4** shows the create_daemon_child routine. This routine calls accept to establish a connection with a client, and creates a child process that will serve the client. The child process calls the do_daemon_work routine, which presumably serves the client as intended by the daemon. The do_daemon_work routine is not shown, as its content depends totally on the service being provided by the daemon.

Figure 153, part **5** shows the routines that will handle the SRC stop requests. The normal_stop routine will, when invoked, set a global variable, stop_requested, to indicate a stop request has been made. The do_daemon_loop routine periodically looks at the setting of stop_requested to determine if the daemon is to be stopped. Child processes are not terminated, since the semantics of the SRC stop normal request is to allow the processing for current clients to complete. The forced_stop and cancel_stop routines will, when invoked, immediately terminate the daemon process and any child processes of the daemon process. Terminating the child processes conforms with the intended semantics of the SRC stop forced and stop cancel requests, since terminating the child processes will terminate the processing for current clients. The child processes are terminated by sending the SIGTERM signal to all processes in the daemon's process group. In the forced_stop routine, the SIGTERM signal is first ignored so the daemon process does not run the cancel_stop routine unnecessarily. Changing the disposition of the SIGTERM signal is not necessary in the cancel_stop routine, since the SIGTERM signal is blocked while cancel_stop is executed. Since the cancel_stop routine will be used as the SIGTERM signal handler for both the daemon process and the child processes, the routine must base its actions on whether or not it is being run by the daemon. The routine can make this determination by calling the dae_process_is_the_daemon routine.

Figure 153, part **6** shows the routines that will handle the SRC subsystem trace request. The begin_trace routine is called when a subsystem trace on request is received. The end_trace routine is called when a subsystem trace off request is received. The begin_trace routine will use setsockopt to turn on the SO_DEBUG socket option for the socket through which client connections are made. The socket returned by the accept routine will inherit this option. The end_trace routine will use setsockopt to turn off the SO_DEBUG socket option for the same socket. Both these routines return an error message if the setsockopt routine reports an error. Error messages are returned through the dae_error_puts routine.

Figure 153, part **7** shows the routines that will handle the SRC subsystem refresh and long status requests. The refresh_config routine handles the SRC

subsystem refresh request. It reads the daemon's configuration file. It returns an error message by calling dae_error_puts, if the configuration file cannot be read. Presumably, the contents of the configuration file will influence the behavior of the child processes that are created to handle client requests. The report_status routine provides the output for a subsystem long status request by calling daemon support routines that allow for the specification of long status output. Calling the dae_status_short routine causes the subsystem short status to be included in the long status output. The report_status routine uses dae_status_printf to specify output in a manner similar to printf, and dae_status_puts to specify output in a manner similar to fputs. The routine outputs the number of client connections the daemon has made, indicates whether tracing is turned on or off, and outputs the name of the file that will be read when the daemon receives a refresh request. Notice that the report_status routine ignores the return values from dae_status_short, dae_status_printf, and dae_status_puts. This is reasonable; it is comparable to ignoring the return value from printf.

Figure 154 on page 271 shows how the daemon shown in Figure 153 on page 261 can be controlled with SRC commands. First, the SRC definition of the subsystem is done with mkssys. The example assumes the daemon program exists as /usr/lpp/somelpp/bin/SRC.sock. The subsystem is defined with the name des.ex.sock, in a group named des.ex. Since none of the -I, -m, or -S flags were used with mkssys, the SRC will make requests of the subsystem through a socket. The subsystem can be started, and short status can be obtained. Requests (stop the daemon, provide long status, start tracing, end tracing, and refresh the subsystem) succeed, since the daemon provides routines to handle these requests. As expected from the daemon code, an attempt to refresh the subsystem generates an error message when the configuration file is missing.

```
# mkssys -s des.ex.sock -p /usr/lpp/somelpp/bin/SRC.sock -u 0 \
         -i /dev/null -o /dev/null -e /dev/null -G des.ex
0513-071 The des.ex.sock Subsystem has been added.

# startsrc -s des.ex.sock
0513-059 The des.ex.sock Subsystem has been started. Subsystem PID is 8418.

# lssrc -s des.ex.sock
Subsystem         Group            PID     Status
 des.ex.sock      des.ex           8418    active

# lssrc -ls des.ex.sock
Subsystem         Group            PID     Status
 des.ex.sock      des.ex           8418    active

 I've seen 0 client connections.
 Tracing is off.
 Next refresh will read from /etc/des.ex.sock.conf.

# refresh -s des.ex.sock
0513-095 The request for subsystem refresh was completed successfully.

# rm /etc/des.ex.sock.conf

# refresh -s des.ex.sock
Subsystem des.ex.sock failed to read its configuration file.

# traceson -s des.ex.sock
0513-091 The request to turn on tracing was completed successfully.

# lssrc -ls des.ex.sock
Subsystem         Group            PID     Status
 des.ex.sock      des.ex           8418    active

 I've seen 0 client connections.
 Tracing is on.
 Next refresh will read from /etc/des.ex.sock.conf.

# tracesoff -s des.ex.sock
0513-093 The request to turn off tracing was completed successfully.

# stopsrc -s des.ex.sock
0513-044 The stop of the des.ex.sock Subsystem was completed successfully.
```

Figure 154. Controlling the Socket Daemon with SRC

10.2.5 Running under the SRC: Subsystem-Defined Requests

Section 5.6.10, "Processing Subsystem-Defined Requests" on page 125 discusses how a subsystem may respond to requests delivered through the SRC that are not actually defined by the SRC. Programs can be written to deliver these subsystem-defined requests through the SRC. The meaning of the requests and the data involved must be agreed upon by the program sending the request and the subsystem responding to the request.

10.2.5.1 Processing Subsystem-Defined Requests

The daemon support routines support the ability of a subsystem to respond to requests delivered through the SRC but not defined by the SRC. This is supported through the dae_other_req member of the dae_req_msq and dae_req_sock structures, and the dae_init_SRC_msq, dae_init_SRC_sock, and dae_SRC_req routines. See A.3, "dae_init_SRC_msq(3)" on page 325, A.4, "dae_init_SRC_sock(3)" on page 328, and A.10, "dae_SRC_req(3)" on page 349, the routine man pages, for details.

Figure 155 on page 273 shows a modification to the example presented in Figure 153 on page 261, part **1**. The dae_other_req field of the SRC_rtns structure is now given the address of a routine in the subsystem, other_requests. Whenever dae_SRC_req reads a request from the SRC request socket that is not an SRC-defined request but may be a subsystem-defined request, it will call the other_requests routine. The other_requests routine is expected to process valid subsystem-defined requests, and flag invalid requests.

```
#include <unistd.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200

int stop_requested = 0;
int client_count = 0;
int acpt_sockd, SRC_sockd = -1;

   .
   .
   .
static int  other_requests(short request, short parm1, short parm2,
                           const void *parm3, int parm3_size);
static int  prepare_socket(void);
static void do_daemon_loop(void);

int main(int argc, char **argv)
{
   dae_parent_t       parent = DAE_P_SRC;
   dae_req_sock_t     SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;

   SRC_rtns.dae_stop_normal = normal_stop;
   SRC_rtns.dae_stop_forced = forced_stop;
   SRC_rtns.dae_stop_cancel = cancel_stop;
   SRC_rtns.dae_trace_begin = begin_trace;
   SRC_rtns.dae_trace_end   = end_trace;
   SRC_rtns.dae_refresh     = refresh_config;
   SRC_rtns.dae_long_status = report_status;
   SRC_rtns.dae_other_req   = other_requests;

   dae_init_prevent_zombies(parent, 1);
   dae_init_SRC_sock(&SRC_sockd, &SRC_rtns, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   if (prepare_socket() == 0)
         do_daemon_loop();

   return 0;
}
```

Figure 155. Processing Subsystem-Defined Requests. This is a modification of Figure 153 on page 261, part **1** *. Some function prototypes have been removed due to space limitations.*

The other_requests routine is called with five parameters. The first parameter, request, is the number of the request received by the subsystem. The parameters parm1 and parm2 are two signed short numbers that are parameters for the request. Finally, the parameters parm3 and parm3_size provide access to an additional request parameter of arbitrary data type and size. Clients sending subsystem-defined SRC requests and subsystems receiving such requests must

agree on the request number, the meaning of the parm1 and parm2 parameters, whether additional data is sent via parm3, and the data type and size of such data.

Figure 156 on page 275 shows the source code for the other_requests routine. The subsystem defines three additional SRC requests that it will process:

- The request represented by the constant COLLECTSTATS will cause the subsystem to start collecting some sort of statistics.
- The request represented by the constant NOSTATS will cause the subsystem to stop collecting those statistics.
- The request represented by the constant CONFIGPATH will specify the configuration file to be read when the subsystem receives the SRC refresh request.

```
#define COLLECTSTATS   256
#define NOSTATS        257
#define CONFIGPATH     258


int stat_long = 0;       /* Boolean - extended statistics requested          */
int stat_interval = -1;  /* Statistics interval (seconds); -1 means no stats */


int other_requests(short request, short parm1, short parm2,
                   const void *parm3, int parm3_size)
{

   if (request == COLLECTSTATS) {

         stat_long = parm1;
         stat_interval = (parm2 > 0) ? parm2 : 60;
         dae_inform_printf("Subsystem des.ex.sock will collect statistics "
                           "every %d seconds.\n", stat_interval);
         return 0;

   } else if (request == NOSTATS) {

         stat_long = 0;
         stat_interval = -1;
         dae_inform_puts("Subsystem des.ex.sock will no longer "
                         "collect statistics.\n");
         return 0;

   } else if (request == CONFIGPATH) {

         if (parm3_size > sizeof(config_path)) {
              dae_error_printf("Configuration file path too long!  "
                               "It is limited to %d characters.\n",
                               MAXPATHLEN);
         } else {
              strcpy(config_path, parm3);
              dae_inform_printf("Configuration file path changed to %s.\n",
                                config_path);
         }

         return 0;

   }

   return 1;
}
```

Figure 156. Source Code for the other_requests *Routine*

In Figure 156, the code that handles each request treats the request parameters differently. The parameters parm1, parm2, and parm3 serve no purpose for the NOSTATS request, and are ignored. For the COLLECTSTATS request, the parm1 and parm2 parameters have meaning. The parm1 parameter is a Boolean that indicates whether extended statistics are to be collected. The parm2 parameter is expected to hold the sampling interval for the statistics collection. The parm3

parameter has no meaning for the COLLECTSTATS request, and is ignored. For the CONFIGPATH request, parm1 and parm2 have no meaning, but parm3 is expected to point to a null terminated string containing the path name of the configuration file the subsystem is to read when it receives a refresh request. Also, parm3_size is expected to be set to the number of bytes occupied by the string, including the byte taken by the null terminator.

The other_requests routine returns a value of 0 if it has received a request that is recognized by the subsystem. If it receives a request not recognized by the subsystem, it returns 1. This behavior is expected by the daemon support routines. When a non-zero value is returned, the daemon support routines make an appropriate error reply to the requestor. If the other_requests routine encounters an error processing a recognized subsystem-defined request, it should call dae_error_puts or dae_error_printf. In fact, the routine does so if the path name parameter of the CONFIGPATH request is too long. The routine also uses dae_inform_puts and dae_inform_printf to provide informational messages.

10.2.5.2 Creating Programs to Send Subsystem-Defined Requests

Chapter 12, “New SRC Program Support” on page 303 discusses how to create programs that could be used to send the COLLECTSTATS, NOSTATS, and CONFIGPATH requests to a subsystem.

10.2.5.3 Keeping SRC Request Numbers Unique

Care must be taken when a subsystem defines SRC requests. If two subsystems use the same number to represent different requests, confusion could result. For example, assume a subsystem defines two requests, “check for quorum” and “do not check for quorum.” Two new programs would probably be created to send these requests, say quorumon and quorumoff. Other programs would probably be created to send the COLLECTSTATS and NOSTATS requests, say statson and statsoff. If the request numbers used for “check for quorum” and “do not check for quorum” were the same numbers that the des.ex.sock subsystem used for COLLECTSTATS and NOSTATS, the statson and statsoff programs could be used to turn quorum checking on and off in the subsystem that defined the quorum requests, and the quorumon and quorumoff programs could be used to turn statistics collection on and off in des.ex.sock. This would not be desirable. The desired behavior would be for des.ex.sock to return an error when quorumon and quorumoff are applied to it, and for the other subsystem to return an error when statson and statsoff are applied to it.

The SRC provides no help in managing request numbers for subsystem-defined requests. To avoid collisions in any SRC requests defined by Scalable POWERparallel Systems, RS/6000 Division subsystems, a list of request number assignments will be maintained by Scalable POWERparallel Systems, RS/6000 Division. Request numbers can be assigned to particular subsystems, a whole LPP, or some unit between the two.

10.2.6 Ensuring Exclusivity

The dae_init_exclusive routine can be used to ensure that only one process on the system is running a particular daemon. Figure 157 on page 278 shows a modification to the example presented in Figure 153, part **1**. A call to dae_init_exclusive is now made before the dae_init call. The second parameter passed to dae_init_exclusive specifies the path name of the daemon executable, and the third parameter is specified as 1. When dae_init executes, it will generate a semaphore key by calling ftok, specifying the second and third

dae_init_exclusive parameters as the parameters to ftok. The generated key represents a semaphore that a process must lock in order to run this daemon program. If dae_init cannot lock the semaphore, it will return with an error. As seen in Figure 157 on page 278, if dae_init returns with an error, the program will exit. If dae_init can lock the semaphore, it does not release its lock. The semaphore should remain locked for the remainder of the lifetime of the daemon process. When the daemon process terminates, the semaphore lock will be removed automatically.

```
#include <unistd.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200

int stop_requested = 0;
int client_count = 0;
int acpt_sockd, SRC_sockd = -1;

   .
   .
   .

static int  prepare_socket(void);
static void do_daemon_loop(void);

int main(int argc, char **argv)
{
   dae_parent_t       parent = DAE_P_SRC;
   dae_req_sock_t     SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;

   SRC_rtns.dae_stop_normal = normal_stop;
   SRC_rtns.dae_stop_forced = forced_stop;
   SRC_rtns.dae_stop_cancel = cancel_stop;
   SRC_rtns.dae_trace_begin = begin_trace;
   SRC_rtns.dae_trace_end   = end_trace;
   SRC_rtns.dae_refresh     = refresh_config;
   SRC_rtns.dae_long_status = report_status;
   SRC_rtns.dae_other_req   = NULL;

   dae_init_exclusive(parent, "/usr/lpp/somelpp/bin/SRC.sock", 1);
   dae_init_prevent_zombies(parent, 1);
   dae_init_SRC_sock(&SRC_sockd, &SRC_rtns, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   if (prepare_socket() == 0)
         do_daemon_loop();

   return 0;
}
```

Figure 157. Using dae_init_exclusive*. This is a modification of Figure 153 on page 261, part* **1**. *Some function prototypes have been removed due to space limitations.*

The definition of an SRC subsystem allows the subsystem to be specified as allowing multiple instances or not. The functionality provided by dae_init_exclusive is intended to supplement this SRC functionality, not replace it. See A.9, “dae_init_exclusive(3)” on page 346, the dae_init_exclusive man page, for more details.

For rationale as to why a daemon needs to enforce its exclusivity requirements, see 9.1, “Limitations of System Methods to Ensure Exclusivity” on page 231.

10.2.7 Setting the Paging Space Allocation Policy

The dae_init_psalloc routine can be used to set the paging space allocation policy under which a daemon runs. Figure 158 on page 280 shows a modification to the example presented in Figure 153, part **1**. A call to dae_init_psalloc is now made before the dae_init call. The second parameter passed to dae_init_psalloc specifies the paging space allocation policy desired. The third parameter specifies the path name of the daemon executable, and the fourth parameter specifies the arguments with which the program was invoked. When dae_init executes, it will determine the paging space allocation policy in effect. If the current policy is not the desired policy, dae_init will change the policy. Changing the policy entails setting an environment variable and re-executing the daemon program. See A.8, “dae_init_psalloc(3)” on page 342, the dae_init_psalloc man page, for a discussion of the implications of re-executing the daemon program.

```
#include <unistd.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200

int stop_requested = 0;
int client_count = 0;
int acpt_sockd, SRC_sockd = -1;
   .
   .
   .
static int  prepare_socket(void);
static void do_daemon_loop(void);

int main(int argc, char **argv)
{
   dae_parent_t       parent = DAE_P_SRC;
   dae_req_sock_t     SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;

   SRC_rtns.dae_stop_normal = normal_stop;
   SRC_rtns.dae_stop_forced = forced_stop;
   SRC_rtns.dae_stop_cancel = cancel_stop;
   SRC_rtns.dae_trace_begin = begin_trace;
   SRC_rtns.dae_trace_end   = end_trace;
   SRC_rtns.dae_refresh     = refresh_config;
   SRC_rtns.dae_long_status = report_status;
   SRC_rtns.dae_other_req   = NULL;

   dae_init_psalloc(parent, DAE_A_EARLY, "/usr/lpp/somelpp/bin/SRC.sock",
                    argv);
   dae_init_exclusive(parent, "/usr/lpp/somelpp/bin/SRC.sock", 1);
   dae_init_prevent_zombies(parent, 1);
   dae_init_SRC_sock(&SRC_sockd, &SRC_rtns, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   if (prepare_socket() == 0)
         do_daemon_loop();

   return 0;
}
```

Figure 158. Using dae_init_psalloc. *This is a modification of Figure 153 on page 261, part* **1** . *Some function prototypes have been removed due to space limitations.*

For information about AIX paging space allocation policies, see Chapter 8, “AIX Paging Space Allocation” on page 203.

10.2.8 Protection from Low Paging Space Termination

The dae_init_lowps routine can be used to install a signal handler for the SIGDANGER signal that will protect the daemon from being terminated when the system is running low on paging space. Figure 159 on page 282 shows a modification to the example presented in Figure 153 on page 261, part **1**. A call to dae_init_lowps is now made before the dae_init call. The second parameter specified on the call to dae_init_lowps is NULL; this means a signal handler that just returns will be installed for SIGDANGER. The third parameter specified is 1, requesting that the signal handler be installed so that it will restart restartable interrupted system calls.

For information about why a daemon installs a signal handler for the SIGDANGER signal, see Chapter 8, “AIX Paging Space Allocation” on page 203.

```
#include <unistd.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200

int stop_requested = 0;
int client_count = 0;
int acpt_sockd, SRC_sockd = -1;

   .
   .
   .
static int  prepare_socket(void);
static void do_daemon_loop(void);

int main(int argc, char **argv)
{
   dae_parent_t       parent = DAE_P_SRC;
   dae_req_sock_t     SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;

   SRC_rtns.dae_stop_normal = normal_stop;
   SRC_rtns.dae_stop_forced = forced_stop;
   SRC_rtns.dae_stop_cancel = cancel_stop;
   SRC_rtns.dae_trace_begin = begin_trace;
   SRC_rtns.dae_trace_end   = end_trace;
   SRC_rtns.dae_refresh     = refresh_config;
   SRC_rtns.dae_long_status = report_status;
   SRC_rtns.dae_other_req   = NULL;

   dae_init_lowps(parent, NULL, 1);
   dae_init_exclusive(parent, "/usr/lpp/somelpp/bin/SRC.sock", 1);
   dae_init_prevent_zombies(parent, 1);
   dae_init_SRC_sock(&SRC_sockd, &SRC_rtns, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   if (prepare_socket() == 0)
         do_daemon_loop();

   return 0;
}
```

Figure 159. Using dae_init_lowps. *This is a modification of Figure 153 on page 261, part* **1**. *Some function prototypes have been removed due to space limitations.*

10.2.9 Supporting a Daemon under inetd or the SRC

Each example presented so far has assumed that a daemon is always started by inetd, or it is never started by inetd. While this is usually the case, it is possible to write a daemon that can be started with or without inetd. Consider the program presented in Figure 153 on page 261. That program is designed with the assumption that the daemon is never started by inetd; it establishes its own connections with clients. If the program was modified to run with or without inetd, it would have to know how it was started. When it is not started by inetd, it should establish client connections and create a child process for each connected client. When it is started by inetd, it should not establish client connections nor should it create child processes, because inetd is providing that function.

Figure 160 on page 284 shows a modification to the code presented in Figure 153 on page 261, part **1**, that allows the daemon to function correctly with or without the help of inetd. Notice that the value passed to all routines with the parent parameter specifies the SRC and inetd, with one exception. The call to dae_init_exclusive does not specify inetd for the parent parameter. When a concurrent server is running under the control of inetd, multiple instances are desired. When a concurrent server is establishing client connections itself, it only wants one process running as the “connecting” daemon at a time. This is accomplished by calling dae_init_exclusive with the parent parameter set to DAE_P_SRC. The exclusive “connecting” daemon can create as many child processes as it wants.

```
#include <unistd.h>
#include <dae.h>

#define SUBSYS_DAE_OFFSET  200

int stop_requested = 0;
int client_count = 0;
int acpt_sockd, SRC_sockd = -1;

   .
   .
   .
static int  prepare_socket(void);
static void do_daemon_loop(void);
static void do_daemon_work(int serv_sockd);

int main(int argc, char **argv)
{
   dae_parent_t       parent = DAE_P_SRC | DAE_P_INETD;
   dae_req_sock_t     SRC_rtns;
   dae_error_detail_t err_detail;
   int                rc;

   .
   .
   .

   dae_init_exclusive(DAE_P_SRC, "/usr/lpp/somelpp/bin/SRC.sock", 1);
   dae_init_lowps(parent, NULL, 1);
   dae_init_prevent_zombies(parent, 1);
   dae_init_SRC_sock(&SRC_sockd, &SRC_rtns, 1);

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         /* Log an error message using the information in err_detail. */
         return SUBSYS_DAE_OFFSET + rc;
   }

   if (parent == DAE_P_SRC) {
         if (prepare_socket() == 0)
              do_daemon_loop();
   } else {
         do_daemon_work(0);
   }

   return 0;
}
```

Figure 160. Using dae_init_exclusive *Conditionally. This is a modification of Figure 153 on page 261, part* **1** *. Some function prototypes have been removed due to space limitations.*

Notice that the dae_init routine indicates how the daemon was started by modifying the integer pointed to by the first parameter. The caller of dae_init can then examine that integer to determine if actions should be taken that depend on how the daemon was started. In the example in Figure 160, the main routine looks at the value of the parent variable after dae_init returns to determine what it should do next.

10.2.10 Logging Detail Data

When the dae_init routine detects an error, it does not log an error log entry. If it did, daemons that used the daemon support routines would create similar error log entries that would be difficult to distinguish from each other. So, dae_init returns detailed information about an error it encounters through its err_detail parameter. If a daemon receives an error from dae_init, it should log an entry to the error log that contains the detailed data returned by dae_init. Figure 161 shows a code fragment that logs detailed error data returned from dae_init. The code fragment assumes the subsystem name is subsys01, and passes the subsystem name as the resource parameter to ppslog. The code fragment also assumes an error record template was created with the definition shown in Figure 162 on page 286. The ppslog routine is used within Scalable POWERparallel Systems, RS/6000 Division, to write to the error log and to the system log. Code developed outside Scalable POWERparallel Systems, RS/6000 Division, can use the errlog routine to write to the error log, and the syslog routine to write to the system log. The man page for errlog is in *AIX Version 4.1: Technical Reference, Volume 1: Base Operating System and Extensions*, SC23-2614. The man page for syslog is in *AIX Version 4.1: Technical Reference, Volume 2: Base Operating System and Extensions*, SC23-2615.

```
#include <dae.h>
#include <ppslog.h>
#include <subsys01_er.h>

   .
   .
   .

   const char *lpp_name = "<put your lpp name here>";
   char log_buf[1024];
   int buflen;

   .
   .
   .

   if ((rc = dae_init(&parent, &err_detail)) != DAE_E_OK) {
         sprintf(log_buf, "%s: \"%s\", detected in %s,%s at line %d",
                  err_detail.dae_routine, err_detail.dae_error_string,
                  err_detail.dae_filename, err_detail.dae_fileversion,
                  err_detail.dae_fileline);
         buflen = strlen(log_buf);

         ppslog(PPS_ERROR, ERRID_SUBSYS01_ER, "subsys01", log_buf,
                buflen, __FILE__, __LINE__, "%I%", lpp_name, "%s\n",
                log_buf);

         return SUBSYS_DAE_OFFSET + rc;
   }
   .
   .
   .
```

Figure 161. Logging Detailed Error Data Returned by the dae_init *Routine*

```
+SUBSYS01_ER:
comment = "subsys01"
class = S
Log = True
Report = True
Alert = False
Err_Type = PERM
Err_Desc = 2100
Prob_Causes = 1001
Fail_Causes = 1000
Fail_Actions = 1000
Detail_Data = 80,00A2,ALPHA
Detail_Data = 150,001E,ALPHA
```

Figure 162. Error Record Template for Code. See Figure 161.

10.2.11 Cleaning Up in a Child Process

If a daemon uses the daemon support routines and then creates a child process, the daemon developer should consider what the child process has inherited from its parent process. For example, the child process has inherited signal dispositions. If the signal dispositions of the parent do not suit the child process, the child process might need to change its signal dispositions. An alternative for installed signal handlers is to call `dae_process_is_the_daemon` to determine if the process running the signal handler is the process that initialized itself as a daemon with `dae_init`. Installed signal handlers will be removed if the child process executes another program.

If a daemon is running under the SRC, and receives requests from the SRC through a socket, a child process of the daemon will inherit the SRC socket file descriptor. The child process might want to close this file descriptor. The child process should not send or receive data through the file descriptor. The SRC socket file descriptor will be closed if the child process executes another program.

10.3 Linking with the Daemon Support Routines

This section discusses how to link a daemon with the daemon support routines.

10.3.1 General Comments

On an AIX system, the daemon support routines can be compiled to support or not support sockets. When the routines are compiled with the `_BSD` macro defined, they will support sockets. When the routines are compiled without the `_BSD` macro defined, they will not support sockets. The daemon support routines use sockets for two purposes: to determine if the daemon was started by `inetd`, and to receive requests from the SRC sent through sockets. So, if the daemon support routines are compiled with the `_BSD` macro defined, they will recognize when a daemon is started by `inetd`, and they will support a daemon under the SRC that receives requests from the SRC using sockets. Conversely, if the daemon support routines are compiled without the `_BSD` macro defined, they will not recognize when a daemon is started by `inetd`, and they will not support a daemon under the SRC that receives requests from the SRC using sockets.

If a program is statically linked with the daemon support routines, and the daemon support routines were compiled with the macro _BSD defined, the program should be linked with the libsrc.a, libbsd.a, and libc.a libraries. The libbsd.a library must be specified to the linker before the libc.a library.

If a program is statically linked with the daemon support routines, and the daemon support routines were compiled without the macro _BSD defined, the program should be linked with the libsrc.a and libc.a libraries.

If a program is dynamically linked with the daemon support routines, there are no special requirements for linking the program.

When a program is linked with the libbsd.a library, the behavior of certain system routines will differ from their expected behavior on systems complying with X/OPEN standards. If a program is to be linked with the libbsd.a library solely for the purpose of linking with a version of the daemon support routines that support socket communications from the SRC, the programmer should be aware that some system routines called by the program may exhibit BSD behavior.

10.3.2 Linking With the Diskette Code

Once you follow the instructions in Appendix C, “Loading the Diskette” on page 395, you will have access to the source code for the daemon support routines. To build libdae_bsd.a, a library containing a version of the daemon support routines that will support sockets, follow the instructions in dae/src/libdae_bsd/README. To build libdae.a, a library containing a version of the daemon support routines that will not support sockets, follow the instructions in dae/src/libdae/README.

The following discussion assumes you have built and installed the libraries as instructed in the README files without changing the make files or the install directories. The make files cause these libraries to be built as shared libraries.

Let’s consider how to build two programs, SRC.msq and SRC.sock. The source for the SRC.msq program is shown in Figure 151 on page 256. The program is designed to accept SRC requests through a message queue. The SRC.msq program does not need support for socket communication from the SRC, does not expect to be under the control of inetd, and has no need for any BSD functionality. Therefore, it is linked with the libdae.a library. The source for the SRC.sock program is shown in Figure 153 on page 261. The program is designed to accept SRC requests through a socket. Since the SRC.sock program needs support for socket communication from the SRC, it must be linked with the libdae_bsd.a library.

The SRC.msq and SRC.sock programs can be compiled and linked with the commands shown in Figure 163 on page 288.

```
cc -I/usr/local/include SRC.msq.c -o SRC.msq \
   -L/usr/local/lib -ldae

cc -D_BSD=44 -I/usr/local/include SRC.sock.c -o SRC.sock \
   -L/usr/local/lib -ldae_bsd -lbsd
```

Figure 163. Linking with libdae_bsd.a *and* libdae.a *(without ADE)*

10.3.3 Linking Within the AIX 4.1 Development Environment (ADE)

In Scalable POWERparallel Systems, RS/6000 Division, two libraries of the daemon support routines are built within the ADE[50] environment. The libdae_bsd.a library contains support for sockets. The libdae.a library does not contain support for sockets. If a program requires support for socket communication from the SRC, expects to be under the control of inetd, or needs any BSD functionality, it should be linked with the libdae_bsd.a library; otherwise, it should be linked with the libdae.a library. Both libraries are built such that they can be statically linked with programs calling the daemon support routines. A program linking with the libdae_bsd.a library must also be linked with the libsrc.a, libbsd.a, and libc.a libraries. A program linking with the libdae.a library must also be linked with the libsrc.a and libc.a libraries.

Let's consider how to build two programs, SRC.msq and SRC.sock. The source for the SRC.msq program is shown in Figure 151 on page 256. The program is designed to accept SRC requests through a message queue. The SRC.msq program does not need support for socket communication from the SRC, does not expect to be under the control of inetd, and has no need for any BSD functionality. Therefore, it is linked with the libdae.a library. The source for the SRC.sock program is shown in Figure 153 on page 261. The program is designed to accept SRC requests through a socket. Since the SRC.sock program needs support for socket communication from the SRC, it must be linked with the libdae_bsd.a library.

Figure 164 on page 289 shows an example of a make file for the ADE[51] environment that can be used to build SRC.msq and SRC.sock. Note that the make file variable SRC.sock.o_CFLAGS is set to the value -D_BSD=44. This will cause the SRC.sock.c file to be compiled with the _BSD macro defined to the value 44. This specifies 4.4BSD behavior, and matches how the source files of libdae_bsd.a are compiled. Note that the SRC.sock program will be linked with the libdae_bsd.a, libsrc.a, libbsd.a, and libc.a libraries. External references will be resolved by searching the libraries in the order mentioned. The libc.a library is searched by default, and does not have to be listed explicitly in the SRC.sock_LIBS make file variable. Also, note that the SRC.msq program will be linked with the libdae.a, libsrc.a, and libc.a libraries, based on the value of the SRC.msq_LIBS make file variable.

[50] The AIX 4.1 Development Environment (ADE) is an internal environment for building and packaging AIX.

[51] This section assumes prior knowledge of the AIX 4.1 Development Environment (ADE), particularly the use of make files in that environment.

```
$ pwd
/u/agar/sb_dae4/src/example


$ cat Makefile
PROGRAMS = SRC.msq SRC.sock

SRC.sock.o_CFLAGS = -D_BSD=44

SRC.msq_LIBS  = -ldae     -lsrc
SRC.sock_LIBS = -ldae_bsd -lsrc -lbsd

IDIR=/usr/lpp/somelpp/bin/
ILIST = ${PROGRAMS}

MAKEFILE_DEPENDENCY = Makefile

.include <${RULES_MK}>
```

Figure 164. A make *File for Linking with* libdae_bsd.a *and* libdae.a *in ADE*

Figure 165 on page 290 shows the SRC.msq program being built. The C compiler, cc, is invoked twice, once to create SRC.msq.o by compiling SRC.msq.c, and once to link SRC.msq from SRC.msq.o and the specified library routines. Note that -ldae and -lsrc are specified when SRC.msq is linked. Thus, the libdae.a, libsrc.a, and libc.a libraries are searched, in that order, when external references are resolved.

```
$ pwd
/u/agar/sb_dae4/src/example

$ build SRC.msq
relative path: ./example.
mkdir ../../obj/power/example
cd ../../obj/power/example
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build SRC.msq.o ****\n" >&2 ; LANG=En_US LIBPATH=/proj
ect/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/b
uild/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/r
2_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix3
2/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/lib
/nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc -
F/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -c  -D_
IBMR2 -D_POWER -D_AIX -DNLS -D_NLS -DMSG -D__STR31__ -Daiws -D_POWER_RS -D_POWE
R_PC -D_POWER_RS1 -D_POWER_RS2 -D_POWER_RSC -D_POWER_601 -D_POWER_603 -D_POWER_
604 -D_THREADS -DSPAIX411 -DSPAIX41 -M -O      -I. -I/u/agar/sb_dae4/src/exampl
e -I/project/sprel2.4/build/r2_4t6d6/src/example  -I/u/agar/sb_dae4/export/powe
r/usr/include -I/project/sprel2.4/build/r2_4t6d6/export/power/usr/include -I/u/
agar/sb_dae4/ode_tools/power/usr/lpp/xlc/usr/include -I/project/sprel2.4/build/
r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/include ../../../src/example/SRC.msq.c
 || ( rc=$?; /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/rm -f SRC
.msq.u ; exit $rc )
**** Using version cset 3.0.0.0 of cc to build SRC.msq.o ****

"../../../src/example/SRC.msq.c", line 8.10: 1506-112 (W) Duplicate type qualif
ier volatile ignored.
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build SRC.msq ****\n" >&2 ; LANG=En_US LIBPATH=/projec
t/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/bui
ld/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/r2_
4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix32/
prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/lib/n
ls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc -F/
project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -o SRC.ms
q.X -s         -L/u/agar/sb_dae4/export/power/usr/lib -L/u/agar/sb_dae4/export/
power/usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/export/power/usr/lib -L/pr
oject/sprel2.4/build/r2_4t6d6/export/power/usr/ccs/lib  -L/u/agar/sb_dae4/ode_t
ools/power/usr/lpp/xlc/usr/lib -L/u/agar/sb_dae4/ode_tools/power/usr/lpp/xlc/us
r/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/li
b -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/ccs/lib  -
L. SRC.msq.o -m       -ldae     -lsrc     >libdepend.mk.out
**** Using version cset 3.0.0.0 of cc to build SRC.msq ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/mv SRC.msq.X SRC.msq
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/md -rm -all    .
```

Figure 165. Building the SRC.msq *Program*

Figure 166 on page 291 shows the SRC.sock program being built. The C compiler, cc, is invoked twice, once to create SRC.sock.o by compiling SRC.sock.c, and once to link SRC.sock from SRC.sock.o and the specified library routines. Note that -D_BSD=44 is specified when SRC.sock.c is compiled. Note that -ldae_bsd, -lsrc, and -lbsd are specified when SRC.sock is linked. Thus, the

libdae_bsd.a, libsrc.a, libbsd.a, and libc.a libraries are searched, in that order, when external references are resolved.

```
$ pwd
/u/agar/sb_dae4/src/example

$ build SRC.sock
relative path: ./example.
cd ../../obj/power/example
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build SRC.sock.o ****\n" >&2 ; LANG=En_US LIBPATH=/pro
ject/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/
build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/
r2_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix
32/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/li
b/nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc
-F/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -c  -D
_IBMR2 -D_POWER -D_AIX -DNLS -D_NLS -DMSG -D__STR31__ -Daiws -D_POWER_RS -D_POW
ER_PC -D_POWER_RS1 -D_POWER_RS2 -D_POWER_RSC -D_POWER_601 -D_POWER_603 -D_POWER
_604 -D_THREADS -DSPAIX411 -DSPAIX41 -M -O  -D_BSD=44    -I. -I/u/agar/sb_dae4/
src/example -I/project/sprel2.4/build/r2_4t6d6/src/example  -I/u/agar/sb_dae4/e
xport/power/usr/include -I/project/sprel2.4/build/r2_4t6d6/export/power/usr/inc
lude -I/u/agar/sb_dae4/ode_tools/power/usr/lpp/xlc/usr/include -I/project/sprel
2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/include ../../../src/example
/SRC.sock.c || ( rc=$?; /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bi
n/rm -f SRC.sock.u ; exit $rc )
**** Using version cset 3.0.0.0 of cc to build SRC.sock.o ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build SRC.sock ****\n" >&2 ; LANG=En_US LIBPATH=/proje
ct/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/bu
ild/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/r2
_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix32
/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/lib/
nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc -F
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -o SRC.s
ock.X -s         -L/u/agar/sb_dae4/export/power/usr/lib -L/u/agar/sb_dae4/expor
t/power/usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/export/power/usr/lib -L/
project/sprel2.4/build/r2_4t6d6/export/power/usr/ccs/lib  -L/u/agar/sb_dae4/ode
_tools/power/usr/lpp/xlc/usr/lib -L/u/agar/sb_dae4/ode_tools/power/usr/lpp/xlc/
usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/
lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/ccs/lib
 -L. SRC.sock.o -m       -ldae_bsd -lsrc -lbsd     >libdepend.mk.out
**** Using version cset 3.0.0.0 of cc to build SRC.sock ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/mv SRC.sock.X SRC.sock
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/md -rm -all    .
```

Figure 166. Building the SRC.sock *Program*

10.4 Porting the Daemon Support Routines

The daemon support routines have been tested on AIX systems. However, they have been written with portability to other UNIX systems in mind. The strategy used is described in 2.2, “Writing Programs for Portability” on page 14. Code similar to that shown in Figure 6 on page 15, Figure 7 on page 16, and Figure 8 on page 17 occurs in a daemon support routine header file, `dae_std.h`, to organize the support for portability. Currently supported systems are AIX with BSD compatibility, AIX without BSD compatibility, systems meeting the requirements specified for XPG4 X/OPEN UNIX conformance, systems meeting the requirements specified for XPG4 BASE conformance that also support POSIX job control, and systems meeting the requirements specified for XPG3 conformance that also support POSIX job control. If the daemon support routines are actually ported to a non-AIX system, changes may be needed to the daemon support routines.

10.5 Multi-Threaded Daemons and the Daemon Support Routines

It is expected that some daemons written by Scalable POWERparallel Systems, RS/6000 Division, will be multi-threaded. Unfortunately, the library routines provided by AIX to support communications between the SRC and SRC-controlled daemons are not thread-safe. Until those routines are made thread-safe, the daemon support routines may not be able to support multi-threaded daemons.

10.6 Man Pages for the Daemon Support Routines

The following man pages are provided for the daemon support routines:

Chapter 11. SRC Notification Support Programs

This chapter provides justifications for using a standard set of notification programs. It then describes the set of programs developed within Scalable POWERparallel Systems, RS/6000 Division, and how those programs can be used.

11.1 Introduction to the SRC Notification Support Programs

The SRC notification support programs are provided to help subsystems with their notification methods. One program provided, `dae_notify`, is a generic notification method. When `dae_notify` is executed because it is installed as the SRC notification method of a subsystem that has just failed, it takes the following steps:

- It looks for an entry in the AIX error log that describes the abnormal termination of the failing subsystem.
- It looks for a "stage 1" user exit program associated with the failing subsystem, or with the group to which the failing subsystem belongs. If such a program is found, it is executed.
- It looks for a subsystem-supplied notification program associated with the failing subsystem, or with the group to which the failing subsystem belongs. If such a program is found, it is executed.
- It looks for a "stage 2" user exit program associated with the failing subsystem, or with the group to which the failing subsystem belongs. If such a program is found, it is executed.

The user exit programs and the subsystem-supplied notification program are executed with the following information provided as arguments:

- The failing subsystem's name
- The name of the group to which the failing subsystem belongs
- The sequence number of the AIX error log entry that describes the abnormal termination of the failing subsystem
- The exit status of the failing subsystem
- The current value of the notification method return value

It is assumed that the user exit programs will typically be used to communicate the failure of the subsystem to someone. A program provided, `dae_msg`, may be executed from the user exit programs. The `dae_msg` program may receive as arguments the name of the failing subsystem, the name of the group to which the failing subsystem belongs, the sequence number of the AIX error log entry describing the abnormal termination of the failing subsystem, and the exit status of the failing subsystem. It can generate short or long messages describing the reason for the abnormal termination. Whether a short or long message is generated is determined by flags specified when `dae_msg` is executed. The short message is a textual interpretation of the subsystem exit status. The long message includes the short message, and the text from the AIX error log.

11.2 Examples of Using the SRC Notification Support Programs

Note: Most of this chapter assumes that the SRC notification support programs being used are installed in the /usr/lpp/ssp/bin directory. Refer to 11.3, "Building and Shipping the SRC Notification Support Programs" on page 299 for information about the install directory of the SRC notification support programs.

The first example presented is for a subsystem, subsys01, that includes a notification program supplied by Scalable POWERparallel Systems, RS/6000 Division. The example assumes the supplied notification program is installed in the directory /usr/lpp/somelpp/bin. When the subsys01 subsystem is installed, an object in the SRCnotify object class should be created. Figure 167 shows the mknotify command that would create such an object. Note that the program specified for the notify method is dae_notify in /usr/lpp/ssp/bin. The dae_notify program is supplied with the name of the directory to be searched for the supplied subsys01 notification program through the argument of the -s flag. The argument of the -u flag specifies the name of the directory to be searched for user exit programs for the subsys01 subsystem. The output from the odmget command is consistent with the arguments that were used with the mknotify command.

```
# mknotify -n subsys01 \
     -m "/usr/lpp/ssp/bin/dae_notify -s /usr/lpp/somelpp/bin -u /usr/local/bin"

# odmget -q "notifyname = 'subsys01'" SRCnotify

SRCnotify:
        notifyname = "subsys01"
        notifymethod = "/usr/lpp/ssp/bin/dae_notify -s /usr/lpp/somelpp/bin -u /usr/local/bin"

# ls -1 /usr/local/bin/subsys01.*
/usr/local/bin/subsys01.userexit1

# ls -1 /usr/lpp/somelpp/bin/subsys01.*
/usr/lpp/somelpp/bin/subsys01.sysaction

# cat /usr/local/bin/subsys01.userexit1
#!/bin/ksh

/usr/lpp/ssp/bin/dae_msg -l "$1" "$3" "$4" | /usr/bin/mail -s "$1 failed!" root

exit "$5"

# cat /usr/lpp/somelpp/bin/subsys01.sysaction
#!/bin/ksh

# Takes some subsystem specific recovery actions that are not shown here.

exit "$5"
```

Figure 167. Defining a Notification Method Using the dae_notify *Program*

The output of the first ls command in Figure 167 shows that the /usr/local/bin directory contains the subsys01.userexit1 program, the "phase 1" user exit program presumably created by the system administrator. The output of the

second ls command in Figure 167 shows that the /usr/lpp/somelpp/bin directory contains the subsys01.sysaction program, the notification program supplied by Scalable POWERparallel Systems, RS/6000 Division, for subsys01. Looking at the contents of the user exit program reveals it is a Korn shell script that sends a long message generated by dae_msg to the root user through mail. The supplied notification program is a Korn shell script that takes some subsystem unique actions, which are not shown in the example. Notice that both programs exit with the notification method return value supplied in the fifth argument.

Figure 168 shows the subsys01 subsystem being started with the startsrc command, and being killed by sending the SIGUSR1 signal to the program. Apparently, the subsys01 subsystem did not have a signal handler installed for the SIGUSR1 signal, so it was terminated when the signal was received. The lssrc commands show that subsystem was running before receiving the signal, and not running after it received the signal.

```
# startsrc -s subsys01
0513-059 The subsys01 Subsystem has been started. Subsystem PID is 8542.

# lssrc -s subsys01
Subsystem         Group            PID     Status
 subsys01         daes             8542    active

# kill -USR1 8542

# lssrc -s subsys01
Subsystem         Group            PID     Status
 subsys01         daes                     inoperative
```

Figure 168. Forcing the subsys01 *Subsystem to Terminate Abnormally*

When the subsys01 subsystem was killed by the SIGUSR1 signal, the srcmstr daemon should have respawned the subsystem, or executed the subsystem's notification method. This example assumes that the subsys01 subsystem was defined not to be respawned. Therefore, the srcmstr daemon should have run the notification method for subsys01. The notification method should have been run with the command:

```
/usr/lpp/ssp/bin/dae_notify -s /usr/lpp/somelpp/bin \
                            -u /usr/local/bin subsys01 daes
```

This assumes the subsys01 subsystem is defined to be in the daes group. The dae_notify program should have found the subsys01.userexit1 program in /usr/local/bin, and executed it. That program should have executed dae_msg to generate a message and executed mail to deliver the message to the root user. Next, the dae_notify program should have found the subsys01.sysaction program in /usr/lpp/somelpp/bin, and executed that program. Finally, the dae_notify program should have searched /usr/local/bin for the program subsys01.userexit2. Having failed to find it, dae_notify should have looked for daes.userexit2 in the same directory. Since no "phase 2" user exit programs should have been found for the subsystem, dae_notify should have continued with its processing, and terminated. If all this happened, there should be a mail message waiting for the root user. Figure 169 on page 298 shows the message waiting for the root user.

```
From root Sat May 13 17:53:51 1995
Received: by it1n13.aix.kingston.ibm.com (AIX 4.1/UCB 5.64/4.03)
          id AA07294; Sat, 13 May 1995 17:53:51 -0400
Date: Sat, 13 May 1995 17:53:51 -0400
From: root
Message-Id: <9505132153.AA07294@it1n13.aix.kingston.ibm.com>
To: root
Subject: subsys01 failed!

Subsystem "subsys01" terminated abnormally; subsystem terminated by signal 30.

AIX error log entry for "subsys01":
---------------------------------------------------------------------------
LABEL:          SRC
IDENTIFIER:     E18E984F

Date/Time:       Sat May 13 17:53:49
Sequence Number: 324
Machine Id:      000175951000
Node Id:         it1n13
Class:           S
Type:            PERM
Resource Name:   SRC

Description
SOFTWARE PROGRAM ERROR

Probable Causes
APPLICATION PROGRAM

Failure Causes
SOFTWARE PROGRAM

        Recommended Actions
        PERFORM PROBLEM RECOVERY PROCEDURES

Detail Data
SYMPTOM CODE
           0
SOFTWARE ERROR CODE
       -9017
ERROR CODE
          30
DETECTING MODULE
'srchevn.c'@line:'272'
FAILING MODULE
subsys01
```

Figure 169. Mail Message Delivered to root *Concerning the* subsys01 *Subsystem*

The second example presented is for a subsystem, subsys02, that does not include a notification program supplied by Scalable POWERparallel Systems, RS/6000 Division. When this subsystem is installed, no notification method is installed for it. If the system administrator wants some notification when the subsystem terminates abnormally, an object can be defined for the subsystem in

the SRCnotify object class. The notification method specified can use the dae_msg program to generate a message. See Figure 170 on page 299.

```
# mknotify -n subsys02 -m "/usr/local/bin/subsys_fail"

# odmget -q "notifyname = 'subsys02'" SRCnotify

SRCnotify:
        notifyname = "subsys02"
        notifymethod = "/usr/local/bin/subsys_fail"

# cat /usr/local/bin/subsys_fail
#!/bin/ksh

/usr/lpp/ssp/bin/dae_msg -l "$1" | /usr/bin/mail -s "$1 failed!" root

exit 0
```

Figure 170. Defining a Notification Method Not Using the dae_notify *Program*

11.3 Building and Shipping the SRC Notification Support Programs

This section describes how to build the SRC notification support programs. The programs may be built from the supplied diskette. A description of how the programs are built in an active development organization is also included.

11.3.1 Building the Diskette Code

Once you follow the instructions in Appendix C, "Loading the Diskette" on page 395, you will have access to the source code for the SRC notification support programs. To build these programs, follow the instructions in dae/src/SRC_notify/README.

11.3.2 Building Within the AIX 4.1 Development Environment (ADE)

Most of this chapter assumes that the SRC notification support programs are installed in the directory /usr/lpp/ssp/bin. The assumption is that these programs will be shipped on RS/6000 SP machines as part of the ssp.basic package some time in the future. This would be convenient for any subsystems that solely run on RS/6000 SP machines. Subsystems that may run on other AIX systems that wish to take advantage of the SRC notification support programs should ship the programs themselves. This section discusses how that can be done. This section assumes prior knowledge of the AIX 4.1 Development Environment[52] (ADE) and LPP packaging.

In the build trees of the AIX 4.1 Development Environment (ADE) in Scalable POWERparallel Systems, RS/6000 Division, the source for the SRC notification support programs is placed in the directory src/dae/SRC_notify. This directory includes a make file that contains the rules for building the SRC notification

[52] The AIX 4.1 Development Environment (ADE) is an internal environment for building and packaging AIX.

support programs. An LPP wishing to build and ship the SRC notification support programs should take the following steps:

- Create a directory in the build tree in which the SRC notification support programs are to be built for the LPP. Do not plan on building anything other than the SRC notification support programs in this directory.
- Create a make file in the directory created in the above step. This make file should be very simple. It would usually contain an assignment to the variable IDIR. The assigned value should be the directory in which the SRC notification support programs are to be installed. The programs can be installed in different directories, although that is not usual. Individual directories can be specified by assigning values to the variables dae_notify_IDIR, dae_finderr_IDIR, dae_msg_IDIR, dae_status_IDIR, and dae_date_IDIR.

  If you want to have the SRC notification support programs dependent on the make file, include an assignment to the MAKEFILE_DEPENDENCY variable in the make file. The value assigned to the variable should be the name of the make file.

  At the end of the make file, you should include the make file ${MAKETOP}/dae/SRC_notify/Makefile. That make file knows how to build the SRC notification support programs. The make file created by the LPP should not include the ${RULES_MK} file. That file is included by ${MAKETOP}/dae/SRC_notify/Makefile.
- Update the make file in the parent directory of the directory created above. Change the make file such that the newly created directory is listed in the value assigned to the SUBDIRS variable. This will cause make to visit the newly created directory during a build.
- Add entries to the appropriate inslist file, also referred to as a [fileset].il file, describing how the SRC notification support programs are to be installed. Entries should be added for dae_notify, dae_finderr, dae_msg, dae_status, and dae_date. The type specified should be F or f, depending on where the programs are being installed. The uid specified should be 2, for the bin user. The gid specified should be 2, for the bin group. The recommended value for mode is 550.

Let's consider an example. Assume an LPP known as somelpp wishes to ship the SRC notification support programs in the directory /usr/lpp/somelpp/bin. A directory is created in the build tree. The new directory is src/somelpp/SRC_notify. Figure 171 shows the make file placed in src/somelpp/SRC_notify. The make file in src/somelpp should be changed such that SRC_notify is included in the list of directories assigned to the SUBDIRS variable. The inslist file for the fileset in the somelpp LPP that will ship the SRC notification support programs is modified to include entries for these programs.

```
IDIR = /usr/lpp/somelpp/bin/
MAKEFILE_DEPENDENCY = Makefile

.include <${MAKETOP}/dae/SRC_notify/Makefile>
```

Figure 171. Example of a make *File for the SRC Notification Support Programs*

Figure 172 on page 301 shows what these new entries would look like. Notice that the type of each file is F, since the programs are being installed under /usr.

```
F 2 2 550     /usr/lpp/somelpp/bin/dae_notify
F 2 2 550     /usr/lpp/somelpp/bin/dae_finderr
F 2 2 550     /usr/lpp/somelpp/bin/dae_msg
F 2 2 550     /usr/lpp/somelpp/bin/dae_status
F 2 2 550     /usr/lpp/somelpp/bin/dae_date
```

Figure 172. Example of inslist *Entries for the SRC Notification Support Programs*

11.4 Man Pages for the SRC Notification Support Programs

The following man pages are provided for the SRC notification support programs:

- B.1, “dae_notify(1)” on page 386
- B.2, “dae_msg(1)” on page 390
- B.3, “dae_status(1)” on page 392

Chapter 12. New SRC Program Support

Section 5.6.10, “Processing Subsystem-Defined Requests” on page 125 discusses how a subsystem may respond to requests delivered through the SRC that are not actually defined by the SRC. Section 10.2.5, “Running under the SRC: Subsystem-Defined Requests” on page 271 describes how the daemon support routines support this capability. When a new request is defined, a program must be created to send the request to a subsystem. It turns out that a model program can be provided that can be compiled to create a program that sends a specific subsystem-defined request and has the look and feel of the standard SRC commands.

12.1 The New SRC Program Model

The model program provided is in a file named dae_SRCmodel.c. It is based on the single source file that is used to create the traceson, tracesoff, and refresh SRC commands. This program can be customized as required when it is compiled. How the program is customized is described as follows:

- When the model program is compiled, it will include a header file with the name dae_newSRCreqs.h. You should create that header file. In the header file you should define a preprocessor macro for the subsystem-defined request to be supported by the command being created. The preprocessor macro should define a symbolic name for the request number to be sent to a subsystem. The request number must be a value from 256 to 32767, inclusive. When the model program is compiled, the compiler must be directed to search the directory holding dae_newSRCreqs.h for header files.
- When the model program is compiled, the DAE_CMD macro must be equated to the macro in dae_newSRCreqs.h that defines the request to be sent to a subsystem by the command being created. This can be done with an option of the compiler.
- If the program to be created should support the -l flag, the DAE_LONG macro should be defined to a non-zero value. This can be done with an option of the compiler.
- If the program to be created should support the -a flag, the DAE_ARG macro should be defined to DAE_ARG_NUMBER or DAE_ARG_STRING. The value indicates whether the argument associated with the -a flag should be interpreted as a number or as a string. This can be done with an option of the compiler.

A program generated from the model program will support the standard -h, -s, -g, and -p flags of the standard SRC commands. The -h flag can be used to specify a host; the -s flag can be used to specify a subsystem name; the -g flag can be used to specify a group name; and the -p flag can be used to specify the PID of the targeted subsystem. If the DAE_LONG macro is defined to a non-zero value when the program is created, the program will also support the -l flag. The -l flag can be used to specify the long version of a request. If the DAE_ARG macro is defined when the program is created, the program will support the -a flag. The -a flag can be used to supply an argument unique to the request being made by the command.

When a program generated from the model program sends a request to a subsystem, the value of the parm1 field of the subreq structure passed to the subsystem depends on whether the generated program supports the -l flag and whether the -l flag was specified when the program was invoked. If the

generated program supports the -l flag, the parm1 field of the subreq structure will indicate whether or not the -l flag was specified when the program was invoked; parm1 will be 0 when the -l flag is not specified, and 1 when it is specified. A program generated from the model program which does not support the -l flag will always set parm1 of the subreq structure to 0.

When a program generated from the model program sends a request to a subsystem, the value of the parm2 field of the subreq structure passed to the subsystem depends on whether the generated program supports the -a flag, whether it expects the -a argument to be a number, and whether the -a flag was specified when the program was invoked. If the generated program supports the -a flag and the generated program expects the -a argument to be a number, and the -a flag was specified when the program was invoked, the -a argument will be converted into a signed short integer and placed in the parm2 field of the subreq structure. If the generated program does not support the -a flag, the generated program expects the -a argument to be a string, or the -a flag is not specified when the program is invoked, the parm2 field of the subreq structure will be set to 0.

When the generated program interprets the -a argument as a number, possible values are limited to those that can be held in a signed short integer. On AIX, this is a number ranging from -32768 to 32767. The generated program will convert the -a argument to a signed short integer using the sscanf routine with the format string "%hi." This format string will interpret numbers starting with 0x or 0X as hexadecimal numbers, other numbers starting with 0 as octal numbers, and other numbers as decimal numbers. Error checking is not done, and overflow is not detected. If error checking or overflow is important, the program can be generated to expect a string for the -a argument, and the subsystem which receives the string can do the error detection and conversion.

A program generated from the model program that supports the -a flag and expects the -a argument to be a string sends the string to the target subsystem. The length of the string is limited to 1,961 bytes. Strings longer than the limit are silently truncated.

12.2 New SRC Program Examples

Section 10.2.5, "Running under the SRC: Subsystem-Defined Requests" on page 271 provides an example of a subsystem supporting three subsystem-defined requests; one request causes the subsystem to start collecting some sort of statistics, another request causes the subsystem to stop collecting those statistics, and the last request causes the subsystem to read a specified configuration file when it is sent a refresh request. The request to start collecting statistics involves a numeric parameter, an interval in seconds, and an option to collect extended statistics. The request to stop collecting statistics does not require any parameters. The request to read a specified configuration file involves a string holding a path name as a parameter. The dae_SRCmodel.c model program can be used to create programs that can be used by a system administrator to send these requests.

Three programs will be created. The statson program will be used to tell a subsystem to start collecting statistics, the statsoff program will be used to tell a subsystem to stop collecting statistics, and the chconfig program will be used to tell a subsystem to read a certain configuration file when doing a subsystem refresh.

A header file named dae_newSRCreqs.h must be created. It is shown in Figure 173 on page 305. It defines three preprocessor macros, COLLECTSTATS, NOSTATS, and CONFIGPATH. These macros provide symbolic names for the new subsystem-defined requests. The numeric values associated with these symbolic names are identical to the request values that the example subsystem in section 10.2.5, “Running under the SRC: Subsystem-Defined Requests” on page 271 will handle. Once the header file is created, the programs can be generated. The programs are created by compiling the dae_SRCmodel.c model program. When the dae_SRCmodel.c program is compiled, it includes the dae_newSRCreqs.h header file.

```
$ cat dae_newSRCreqs.h
#ifndef _DAE_NEWSRCREQS_H
#define _DAE_NEWSRCREQS_H

#define COLLECTSTATS            256
#define NOSTATS                 257
#define CONFIGPATH              258

#endif
```

Figure 173. Sample dae_newSRCreqs.h *Header File*

When statson is compiled, the DAE_CMD macro should be equated to the COLLECTSTATS macro. As a result, when the statson program sends a request to a subsystem, the action field of the subreq structure will be set to 256, which represents the request to start collecting statistics. The statson program should be compiled with the DAE_LONG macro defined to have a non-zero value. As a result, the statson program will support the -l flag. The program will use the parm1 field of the subreq structure sent to the subsystem to indicate whether the -l flag was specified when the program was invoked. The statson program should also be compiled with the DAE_ARG macro defined to have the value DAE_ARG_NUMBER. As a result, the statson program will support the -a flag. The argument supplied with the -a flag will be converted to a signed short number and placed in the parm2 field of the subreq structure sent to the subsystem. Referring to Figure 156 on page 275, the receiving subsystem will treat this argument as the number of seconds in a statistics gathering interval.

When statsoff is compiled, the DAE_CMD macro should be equated to the NOSTATS macro. As a result, when the statsoff program sends a request to a subsystem, the action field of the subreq structure will be set to 257, which represents the request to stop collecting statistics. Neither the DAE_LONG nor the DAE_ARG macro should be defined when the statsoff program is compiled. As a result, the program will not support the -l flag nor the -a flag, and the parm1 and parm2 fields of the subreq structure sent to a subsystem by the program will always be set to zeros.

When chconfig is compiled, the DAE_CMD macro should be equated to the CONFIGPATH macro. As a result, when the chconfig program sends a request to a subsystem, the action field of the subreq structure will be set to 258, which represents the request to change the configuration file to be read when refreshing the subsystem. The chconfig program should not be compiled with the DAE_LONG macro defined. As a result, the chconfig program will not support the -l flag. The program will always set the parm1 field of the subreq structure sent to the subsystem to zero. The chconfig program should be compiled with

the DAE_ARG macro defined to have the value DAE_ARG_STRING. As a result, the chconfig program will support the -a flag. The argument supplied with the -a flag will be sent to the subsystem as a null-terminated string. Referring to Figure 156 on page 275, the receiving subsystem will receive this string through the parm3 pointer in the other_requests routine, and the string will be treated as the path name of the configuration file to read when the subsystem is refreshed.

The statson, statsoff, and chconfig programs will support the -h, -s, -g, and -p flags.

Figure 174 shows some examples of the statson, statsoff, and chconfig programs in use. First, the programs are targeted toward a subsystem, des.ex.sock, that supports the COLLECTSTATS, NOSTATS, and CONFIGPATH requests with the code shown in Figure 155 on page 273 and Figure 156 on page 275. The statson command targeted to des.ex.sock causes that subsystem to start collecting its statistics every 120 seconds. The statsoff command targeted to des.ex.sock causes that subsystem to stop collecting statistics. The chconfig command targeted to des.ex.sock causes that subsystem to read the configuration file /tmp/alt.config.file when the subsystem refreshes itself. The messages displayed by statson, statsoff, and chconfig when directed toward the des.ex.sock subsystem were generated by the subsystem, when its other_requests routine called dae_inform_printf or dae_inform_puts. When the statson, statsoff, and chconfig commands are targeted to a subsystem that does not support the COLLECTSTATS, NOSTATS, and CONFIGPATH requests, such as inetd, an error message is displayed.

```
# statson -s des.ex.sock -a "120"
Subsystem des.ex.sock will collect statistics every 120 seconds.

# statsoff -s des.ex.sock
Subsystem des.ex.sock will no longer collect statistics.

# chconfig -s des.ex.sock -a "/tmp/alt.config.file"
Configuration file path changed to /tmp/alt.config.file.

# statson -s inetd -a "120"
0513-087 The /usr/sbin/inetd Subsystem has received a request that
it does not understand and could not service.
Contact System Administration.

# statsoff -s inetd
0513-087 The /usr/sbin/inetd Subsystem has received a request that
it does not understand and could not service.
Contact System Administration.

# chconfig -s inetd -a "/tmp/alt.config.file"
0513-087 The /usr/sbin/inetd Subsystem has received a request that
it does not understand and could not service.
Contact System Administration.
```

Figure 174. New SRC Programs in Use

12.3 Building Programs Using the Model

This section describes the compilation and library reference used in building programs, using the model described in section 12.1, "The New SRC Program Model" on page 303.

12.3.1 Building Outside the AIX 4.1 Development Environment (ADE)

To build programs with the new SRC program model outside of ADE, create the header file dae_newSRCreqs.h in the current directory as described in Section 12.1, "The New SRC Program Model" on page 303, copy the dae_SRCmodel.c file to the current directory, and execute the compiler.

- Use the compiler -D flag to define the DAE_CMD, DAE_LONG, and DAE_ARG macros as appropriate.
- Use the compiler -I flag to direct the compiler to search for header files from the current directory.
- Use the compiler -o flag to specify the name of the executable that is to be created.
- Use the compiler -l flag to link the program with the libsrc.a and libodm.a libraries.
- Specify dae_SRCmodel.c as the source file.

Figure 175 shows the three example programs discussed earlier being built outside of the ADE environment.

```
$ cc -DDAE_CMD=COLLECTSTATS -DDAE_LONG -DDAE_ARG=DAE_ARG_NUMBER -I. \
      -o statson dae_SRCmodel.c -lsrc -lodm

$ cc -DDAE_CMD=NOSTATS -I. \
      -o statsoff dae_SRCmodel.c -lsrc -lodm

$ cc -DDAE_CMD=CONFIGPATH -DDAE_ARG=DAE_ARG_STRING -I. \
      -o chconfig dae_SRCmodel.c -lsrc -lodm
```

Figure 175. Compiling Programs Based on dae_SRCmodel.c *(without the ADE)*

Once you follow the instructions in Appendix C, "Loading the Diskette" on page 395, you will have access to the dae_SRCmodel.c file in the directory dae/src/SRC_cmd_model.

12.3.2 Building Inside the AIX 4.1 Development Environment (ADE)

This section assumes prior knowledge of the AIX 4.1 Development Environment (ADE), particularly the use of make files in that environment[53].

To build some programs with the new SRC program model within the ADE of Scalable POWERparallel Systems, RS/6000 Division, you must create a directory in the source tree where those programs, and only those programs, are to be built. Within the directory, create the header file dae_newSRCreqs.h as described previously, and create a make file. This make file will be referred to in the following sections as the program unique make file.

[53] The AIX 4.1 Development Environment (ADE) is an internal environment for building and packaging AIX.

The program unique make file will only include information unique to the commands being built. Information common to building any program from dae_SRCmodel.c is included in the make file src/dae/SRC_cmd_model/Makefile. This make file will be referred to as the program model make file. Most make files in the ADE must include the file identified by the variable RULES_MK. The program unique make file should not include this file. Instead, the program unique make file should include the program model make file. The program model make file includes the file identified by the variable RULES_MK and contains rules for building a program based on dae_SRCmodel.c.

Before including the program model make file, the program unique make file should assign values to certain variables. The values assigned are used to customize the creation of programs from the dae_SRCmodel.c source file. The following variables should be assigned values in the program unique make file:

- PROGRAMS - Assign to this variable a list of the names of the programs to be built. All these programs will be built from dae_SRCmodel.c.

  If only one program is specified in this list, DAE_CMD, DAE_LONG, and DAE_ARG can be defined without qualification.

  If multiple programs are specified, specific values for DAE_CMD, DAE_LONG, and DAE_ARG can be specified for each program by qualifying the variable names with the name of the .o file associated with the program. For example, the values for foo can be specified by defining foo.o_DAE_CMD, foo.o_DAE_LONG, and foo.o_DAE_ARG.
- DAE_CMD - Assign to this variable the name of a symbolic constant representing the command number. The symbolic constant should be defined in the header file dae_newSRCreqs.h.
- DAE_LONG - If the built program is to support the -l flag, assign to this variable a value other than 0. If the built program is not to support the -l flag, either assign to DAE_LONG a value of 0, or do not define DAE_LONG.
- DAE_ARG - If the built program is to support the -a flag, assign to this variable the value DAE_ARG_NUMBER or DAE_ARG_STRING, depending on whether the -a argument should be sent to the target subsystem as a number or a string. If the built program is not to support the -a flag, either assign to DAE_ARG a value of DAE_ARG_NONE, or do not define DAE_ARG.
- IDIR - Specify the directory into which the built commands are to be installed.

Figure 176 on page 309 shows an example of a program unique make file for the programs discussed earlier in this chapter, statson, statsoff, and chconfig. The PROGRAMS variable is assigned the names of the three programs to be built. The IDIR variable is assigned the path name of the directory into which these programs are to be installed. Note that the path name assigned to the IDIR variable must end with a slash (/). For statson.o, the DAE_CMD variable is assigned the value COLLECTSTATS, the DAE_LONG variable is assigned a non-zero value, and the DAE_ARG variable is assigned DAE_ARG_NUMBER. Therefore, after the statson program is built, it will support the COLLECTSTATS request, the -l flag, and the -a flag with a numeric argument. The variable assignments specific to statsoff.o will cause the statsoff program to support the NOSTATS request, and to not support the -l or -a flags. The variable assignments specific to chconfig.o will cause the chconfig program to support the CONFIGPATH request and the -a flag with a string argument, but not the -l flag.

```
$ cat Makefile

PROGRAMS = statson statsoff chconfig

IDIR = /usr/lpp/somelpp/bin/

statson.o_DAE_CMD  = COLLECTSTATS
statson.o_DAE_LONG = 1
statson.o_DAE_ARG  = DAE_ARG_NUMBER

statsoff.o_DAE_CMD = NOSTATS

chconfig.o_DAE_CMD  = CONFIGPATH
chconfig.o_DAE_ARG  = DAE_ARG_STRING

MAKEFILE_DEPENDENCY = Makefile

.include <${MAKETOP}/dae/SRC_cmd_model/Makefile>
```

Figure 176. A make *File to Build Programs Based on* dae_SRCmodel.c *(With ADE)*

Figure 177 on page 310 shows the results of building the statson, statsoff, and chconfig programs in the ADE. The results of the builds shown in the example depend on the make file in Figure 176 and the header file in Figure 173 on page 305 being in the same directory in which the build commands are run. Notice the following:

- As each command is built, a symbolic link is created to dae_SRCmodel.c. This is because the suffix rules in the ADE environment expect to be able to build a file statson.o from a source file named statson.c, not dae_SRCmodel.c.
- When the commands are built, DAE_CMD, DAE_LONG, and DAE_ARG are defined to have values consistent with the value assignments in the program unique make file.
- The compiler flag -I. allows the compiler to find the dae_newSRCreqs.h header file in the current directory.
- When the commands are linked, they are linked with the libsrc.a and libodm.a libraries. These libraries are specified in the program model make file.
- When the install phase is run, the commands are installed in the directory specified in the program unique make file by the value of the IDIR variable.

```
$ build statson
relative path: ./example/new_cmds.
mkdir ../../../obj/power/example/new_cmds
cd ../../../obj/power/example/new_cmds
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/rm -f statson.c
/bin/ln -s ../../../../src/reldaemons/SRC_cmd_model/dae_SRCmodel.c statson.c
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build statson.o ****\n" >&2 ; LANG=En_US LIBPATH=/proj
ect/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/b
uild/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/r
2_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix3
2/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/lib
/nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc -
F/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -c  -D_
IBMR2 -D_POWER -D_AIX -DNLS -D_NLS -DMSG -D__STR31__ -Daiws -D_POWER_RS -D_POWE
R_PC -D_POWER_RS1 -D_POWER_RS2 -D_POWER_RSC -D_POWER_601 -D_POWER_603 -D_POWER_
604 -D_THREADS -DSPAIX411 -DSPAIX41 -M -O  -DDAE_CMD=COLLECTSTATS  -DDAE_LONG=1
  -DDAE_ARG=DAE_ARG_NUMBER     -I. -I/u/agar/sb_dae4/src/example/new_cmds -I/pr
oject/sprel2.4/build/r2_4t6d6/src/example/new_cmds -I../../reldaemons/SRC_cmd_m
odel -I/u/agar/sb_dae4/src/reldaemons/SRC_cmd_model -I/project/sprel2.4/build/r
2_4t6d6/src/reldaemons/SRC_cmd_model -I/u/agar/sb_dae4/export/power/usr/include
 -I/project/sprel2.4/build/r2_4t6d6/export/power/usr/include -I/u/agar/sb_dae4/
ode_tools/power/usr/lpp/xlc/usr/include -I/project/sprel2.4/build/r2_4t6d6/ode_
tools/power/usr/lpp/xlc/usr/include statson.c || ( rc=$?; /project/sprel2.4/bui
ld/r2_4t6d6/ode_tools/power/usr/bin/rm -f statson.u ; exit $rc )
**** Using version cset 3.0.0.0 of cc to build statson.o ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build statson ****\n" >&2 ; LANG=En_US LIBPATH=/projec
t/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/bui
ld/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/r2_
4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix32/
prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/lib/n
ls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc -F/
project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -o statso
n.X -s         -L/u/agar/sb_dae4/export/power/usr/lib -L/u/agar/sb_dae4/export/
power/usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/export/power/usr/lib -L/pr
oject/sprel2.4/build/r2_4t6d6/export/power/usr/ccs/lib  -L/u/agar/sb_dae4/ode_t
ools/power/usr/lpp/xlc/usr/lib -L/u/agar/sb_dae4/ode_tools/power/usr/lpp/xlc/us
r/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/li
b -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/ccs/lib  -
L. statson.o -m       -lsrc -lodm     >libdepend.mk.out
**** Using version cset 3.0.0.0 of cc to build statson ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/mv statson.X statson
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/md -rm -all    .
```

Figure 177 (Part 1 of 4). Compiling Programs Based on dae_SRCmodel.c *(With the ADE)*

```
$ build statsoff
relative path: ./example/new_cmds.
cd ../../../obj/power/example/new_cmds
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/rm -f statsoff.c
/bin/ln -s ../../../../src/reldaemons/SRC_cmd_model/dae_SRCmodel.c statsoff.c
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build statsoff.o ****\n" >&2 ; LANG=En_US LIBPATH=/pro
ject/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/
build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/
r2_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix
32/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/li
b/nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc
-F/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -c  -D
_IBMR2 -D_POWER -D_AIX -DNLS -D_NLS -DMSG -D__STR31__ -Daiws -D_POWER_RS -D_POW
ER_PC -D_POWER_RS1 -D_POWER_RS2 -D_POWER_RSC -D_POWER_601 -D_POWER_603 -D_POWER
_604 -D_THREADS -DSPAIX411 -DSPAIX41 -M -O  -DDAE_CMD=NOSTATS  -DDAE_LONG=0  -D
DAE_ARG=DAE_ARG_NONE     -I. -I/u/agar/sb_dae4/src/example/new_cmds -I/project/
sprel2.4/build/r2_4t6d6/src/example/new_cmds -I../../reldaemons/SRC_cmd_model -
I/u/agar/sb_dae4/src/reldaemons/SRC_cmd_model -I/project/sprel2.4/build/r2_4t6d
6/src/reldaemons/SRC_cmd_model -I/u/agar/sb_dae4/export/power/usr/include -I/pr
oject/sprel2.4/build/r2_4t6d6/export/power/usr/include -I/u/agar/sb_dae4/ode_to
ols/power/usr/lpp/xlc/usr/include -I/project/sprel2.4/build/r2_4t6d6/ode_tools/
power/usr/lpp/xlc/usr/include statsoff.c || ( rc=$?; /project/sprel2.4/build/r2
_4t6d6/ode_tools/power/usr/bin/rm -f statsoff.u ; exit $rc )
**** Using version cset 3.0.0.0 of cc to build statsoff.o ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build statsoff ****\n" >&2 ; LANG=En_US LIBPATH=/proje
ct/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/bu
ild/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/r2
_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix32
/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/lib/
nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc -F
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -o stats
off.X -s         -L/u/agar/sb_dae4/export/power/usr/lib -L/u/agar/sb_dae4/expor
t/power/usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/export/power/usr/lib -L/
project/sprel2.4/build/r2_4t6d6/export/power/usr/ccs/lib  -L/u/agar/sb_dae4/ode
_tools/power/usr/lpp/xlc/usr/lib -L/u/agar/sb_dae4/ode_tools/power/usr/lpp/xlc/
usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/
lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/ccs/lib
 -L. statsoff.o -m       -lsrc -lodm     >libdepend.mk.out
**** Using version cset 3.0.0.0 of cc to build statsoff ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/mv statsoff.X statsoff
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/md -rm -all    .
```

Figure 177 (Part 2 of 4). Compiling Programs Based on dae_SRCmodel.c *(With the ADE)*

```
$ build chconfig
relative path: ./example/new_cmds.
cd ../../../obj/power/example/new_cmds
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/rm -f chconfig.c
/bin/ln -s ../../../../src/reldaemons/SRC_cmd_model/dae_SRCmodel.c chconfig.c
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build chconfig.o ****\n" >&2 ; LANG=En_US LIBPATH=/pro
ject/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/
build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/
r2_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix
32/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/li
b/nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc
-F/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -c  -D
_IBMR2 -D_POWER -D_AIX -DNLS -D_NLS -DMSG -D__STR31__ -Daiws -D_POWER_RS -D_POW
ER_PC -D_POWER_RS1 -D_POWER_RS2 -D_POWER_RSC -D_POWER_601 -D_POWER_603 -D_POWER
_604 -D_THREADS -DSPAIX411 -DSPAIX41 -M -O  -DDAE_CMD=CONFIGPATH  -DDAE_LONG=O
 -DDAE_ARG=DAE_ARG_STRING     -I. -I/u/agar/sb_dae4/src/example/new_cmds -I/pro
ject/sprel2.4/build/r2_4t6d6/src/example/new_cmds -I../../reldaemons/SRC_cmd_mo
del -I/u/agar/sb_dae4/src/reldaemons/SRC_cmd_model -I/project/sprel2.4/build/r2
_4t6d6/src/reldaemons/SRC_cmd_model -I/u/agar/sb_dae4/export/power/usr/include
-I/project/sprel2.4/build/r2_4t6d6/export/power/usr/include -I/u/agar/sb_dae4/o
de_tools/power/usr/lpp/xlc/usr/include -I/project/sprel2.4/build/r2_4t6d6/ode_t
ools/power/usr/lpp/xlc/usr/include chconfig.c || ( rc=$?; /project/sprel2.4/bui
ld/r2_4t6d6/ode_tools/power/usr/bin/rm -f chconfig.u ; exit $rc )
**** Using version cset 3.0.0.0 of cc to build chconfig.o ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/echo "**** Using versi
on cset 3.0.0.0 of cc to build chconfig ****\n" >&2 ; LANG=En_US LIBPATH=/proje
ct/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib:/project/sprel2.4/bu
ild/r2_4t6d6/ode_tools/power/usr/lpp/xlC/lib NLSPATH=/project/sprel2.4/build/r2
_4t6d6/ode_tools/power/usr/lib/nls/msg/%L/%N:/afs/aix.kingston.ibm.com/rs_aix32
/prod/apps/cmvc/2.2.0.1/usr/lpp/cmvc/msg/%L/%N:/usr/lib/nls/msg/%L/%N:/usr/lib/
nls/msg/prime/%N /project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/cc -F
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/etc/xlC.cfg  -qhalt=E -o chcon
fig.X -s         -L/u/agar/sb_dae4/export/power/usr/lib -L/u/agar/sb_dae4/expor
t/power/usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/export/power/usr/lib -L/
project/sprel2.4/build/r2_4t6d6/export/power/usr/ccs/lib  -L/u/agar/sb_dae4/ode
_tools/power/usr/lpp/xlc/usr/lib -L/u/agar/sb_dae4/ode_tools/power/usr/lpp/xlc/
usr/ccs/lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/
lib -L/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/lpp/xlc/usr/ccs/lib
 -L. chconfig.o -m       -lsrc -lodm     >libdepend.mk.out
**** Using version cset 3.0.0.0 of cc to build chconfig ****

/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/mv chconfig.X chconfig
/project/sprel2.4/build/r2_4t6d6/ode_tools/power/usr/bin/md -rm -all    .
```

Figure 177 (Part 3 of 4). Compiling Programs Based on dae_SRCmodel.c *(With the ADE)*

```
$ build install_all
relative path: ./example/new_cmds.
cd ../../../obj/power/example/new_cmds
Creating optlevel ...
Shipping link  /usr/lpp/somelpp/bin/statson ...
Shipping link  /usr/lpp/somelpp/bin/statsoff ...
Shipping link  /usr/lpp/somelpp/bin/chconfig ...
Shipping link  /usr/lpp/optlevel/optlevel ...
```

Figure 177 (Part 4 of 4). Compiling Programs Based on dae_SRCmodel.c *(With the ADE)*

A final note should be made about building in the ADE. As mentioned above, a new directory should be created in which only programs based on dae_SRCmodel.c are built. The make file in the parent directory of this new directory should be changed so the new directory is visited during a build. The change required to the parent directory's make file is to list the name of the new directory in the list of directory names assigned to the SUBDIRS make file variable.

12.4 Packaging Programs Built from the Model

This section assumes prior knowledge of LPP packaging.

As with any program to be packaged and shipped, a program built with the new SRC program model must have an entry in an inslist file, also referred to as a [fileset].il file, describing how the program is to be installed. The recommended values for the fields in the inslist entry depend on what the program does.

If the program affects how a subsystem runs, or it provides information that would not be appropriate to divulge to any user, the following recommendations are made. The type field value should be F or f, depending on where the program is being installed. The uid field value should be 0, signifying the root user. The gid field value should be 0, signifying the system group. The recommended value for the mode field is 550. These values would only allow the root user or a user in the system group to successfully run the command.

If the program does not affect the behavior of the subsystem and supplies information appropriate to divulge to any user, the following recommendations are made: The type field value should be FT or ft depending on where the program is being installed. The uid field value should be 0, signifying the root user. The gid field value should be 0, signifying the system group. The recommended value for the mode field is 2555. These values would make the program a set-group-ID program to the system group, and would allow any user to run the program. A program that sends requests to the SRC and that is to be executable by any user must be a set-group-ID program, because a program must be running with an effective user ID of 0 or an effective group ID of 0 to be able to send requests to the SRC daemon.

12.5 Keeping SRC Request Numbers Unique

Care must be taken when a subsystem defines SRC requests. If two subsystems use the same number to represent different requests, confusion could result. For example, assume a subsystem defines two requests, “check for quorum” and “do not check for quorum.” Two new programs would probably be created to send these requests, `quorumon` and `quorumoff`. If the request numbers used for “check for quorum” and “do not check for quorum” were the same numbers that the `des.ex.sock` subsystem used for `COLLECTSTATS` and `NOSTATS`, the `statson` and `statsoff` programs could be used to turn quorum checking on and off in the subsystem that defined the quorum requests, and the `quorumon` and `quorumoff` programs could be used to turn statistics collection on and off in `des.ex.sock`. This would not be desirable. The desired behavior would be that `des.ex.sock` returns an error when `quorumon` and `quorumoff` are applied to it, and the other subsystem returns an error when `statson` and `statsoff` are applied to it.

The SRC provides no help in managing request numbers for subsystem-defined requests. To avoid collisions in any SRC requests defined by subsystems shipped by Scalable POWERparallel Systems, RS/6000 Division, a list of request number assignments will be maintained. Request numbers can be assigned to particular subsystems, a whole LPP, or some unit between the two. Once a range of numbers is assigned to some unit, it should be managed with macros in a `dae_newSRCreqs.h` file.

Appendix A. Daemon Support Routines man Pages

This appendix provides detailed program instructions for the use of the Daemon Support Routines. The level of detail is consistent with that commonly provided in UNIX systems man pages. The appendix provides reference for the user of the routines; this would usually be a daemon writer interested in using the routines to accomplish the functions explained in this redbook.

A.1 dae_init(3)

NAME

dae_init - Initializes a daemon.

SYNOPSIS

```
#include <dae.h>

int dae_init(
     dae_parent_t *dae_parent,
     dae_error_detail_t *dae_err_detail
);
```

The dae.h file includes these definitions:

```
#define DAE_P_SRC       0x0001
#define DAE_P_INETD     0x0002
#define DAE_P_OTHER     0x0004

typedef int dae_parent_t;


struct dae_error_detail {
     char dae_routine[20];
     char dae_error_string[80];
     char dae_filename[20];
     char dae_fileversion[10];
     long dae_fileline;
};

typedef struct dae_error_detail dae_error_detail_t;
```

PARAMETERS

dae_parent

This is an input/output parameter. On input, it specifies how the daemon may be started. The dae_init routine determines how the daemon was started, and determines if that method is allowed by the value of this parameter. On input, this parameter is formed by using one or more of these constants as operands in an OR operation:

- DAE_P_SRC: The daemon is allowed to be started by the SRC.
- DAE_P_INETD: The daemon is allowed to be started by inetd.
- DAE_P_OTHER: The daemon is allowed to be started in some other manner; for example by init or a shell.

The dae_init routine changes this parameter to be only one of these values. The value indicates how the daemon was actually started.

This pointer must not have the NULL value.

dae_err_detail

If this pointer is not NULL, and dae_init returns an error indication, the area pointed to is updated to provide detailed data about the error. The data provided is suitable for logging, but the dae_init routine does not log errors. Logging is the responsibility of the calling routine. The area pointed to may be updated even if dae_init indicates success. In this case, the information should be ignored.

If this pointer is NULL, detailed data about errors cannot be returned.

DESCRIPTION

The dae_init routine initializes a daemon. The steps taken by dae_init to initialize the daemon are determined by the routine's parameters, how the daemon was started, and whether certain routines were called before dae_init. The routines that can affect the steps taken by dae_init all have names with the prefix "dae_init_." The man page for each of these routines describes how the routine affects the steps taken by dae_init.

The initialization steps taken by dae_init are:

- Determine how the daemon was started.

  First, dae_init determines if the daemon was started by the Internet Superserver (inetd). If file descriptors 0, 1, and 2 are associated with the same internet domain socket, dae_init considers the daemon to have been started by inetd.

  If the daemon was started by inetd, but the dae_parent parameter does not allow this, dae_init returns an error indication.

  If the daemon was not started by inetd, dae_init determines if the daemon was started by the System Resource Controller (SRC). If the current parent process of the daemon is srcmstr, dae_init considers the daemon to have been started by the SRC. If the daemon was started by the SRC, but the dae_parent parameter does not allow this, dae_init returns an error indication.

  If the daemon was not started by inetd nor the SRC, but the dae_parent parameter does not allow this, dae_init returns an error indication.

  The dae_parent parameter is updated to indicate how the daemon was started.

- Ignore terminal-generated signals.

  The following terminal-generated signals are ignored: SIGHUP, SIGINT, SIGQUIT, SIGTSTP, SIGTTIN, and SIGTTOU.

- Migrate the daemon to another process, if appropriate.

  If the daemon was not started by inetd, nor by the SRC, and its current parent process is not init, the daemon is migrated to another process. That is, dae_init calls fork, the parent process terminates, and the child process continues as the daemon. This is necessary to allow a shell to continue processing input after it starts a daemon in the foreground.

- Change the paging space allocation policy under which the daemon process runs.

  The paging space allocation policy of the daemon process might be changed if the dae_init_psalloc routine has been called. See A.8,

"dae_init_psalloc(3)" on page 342, the dae_init_psalloc man page, for details on how the paging space allocation policy is changed.

- Disassociate the daemon from its controlling terminal.

  The dae_init routine will take the steps necessary to make the daemon process a session leader without a controlling terminal.

- Deal with non-terminal generated signals.

  The dae_init routine will ignore the SIGPIPE signal. This will protect the daemon process from termination in the event that a pipe or socket, written to by the daemon, does not have a reader.

  If dae_init_prevent_zombies has been called before dae_init, the dae_init routine may prevent the daemon from generating a buildup of zombie processes. On many systems, including AIX, this simply means dae_init will ignore the SIGCHLD signal. Many systems will not generate zombie processes from terminated child processes if the parent process is ignoring the SIGCHLD signal. On a system without those semantics, the dae_init routine will install a signal handler for SIGCHLD that will clean up generated zombie processes.

  The dae_init routine will set the disposition of the SIGTERM signal. The disposition set depends on whether dae_init_term_sig, dae_init_SRC_sig, dae_init_SRC_msq, or dae_init_SRC_sock have been called. If none of those routines have been called, the disposition of SIGTERM is set to the default, which is to terminate the process on delivery of the signal. If one of the routines was called, but the address of a signal handler for SIGTERM was not provided, the disposition of SIGTERM is set to the default. If one of the routines was called, and the address of a signal handler for SIGTERM was provided, the specified signal handler is installed for the SIGTERM signal.

  If the dae_init_lowps routine has been called, dae_init may install a signal handler for the SIGDANGER signal. This will protect the daemon process from termination when the system is low on paging space. See A.7, "dae_init_lowps(3)" on page 340 for details.

- Close unnecessary file descriptors.

  If the daemon was started by the SRC or by inetd, all file descriptors beyond file descriptor 2 are closed. Otherwise, all file descriptors are closed.

- Ensure the standard file descriptors are open.

  Any of the standard file descriptors (0, 1, or 2) that are not associated with an open file, are opened on /dev/null.

- Change the current working directory.

  The current working directory of the daemon process is changed to root, /.

- Set the process file mode creation mask to 0.

- Perform SRC-unique initialization, if appropriate.

  If the daemon was started by the SRC, some SRC-unique initialization is performed. The initialization performed depends on whether any of the following routines have been called: dae_init_SRC_sig, dae_init_SRC_msq, and dae_init_SRC_sock.

  If the dae_init_SRC_sig routine has been called, the disposition of the signals specified to handle the SRC subsystem stop normal requests and subsystem stop forced requests are set as specified by the parameters of dae_init_SRC_sig. See A.2, "dae_init_SRC_sig(3)" on page 321 for details.

If the dae_init_SRC_msq routine has been called, dae_init accesses the message queue specified by the parameters of dae_init_SRC_msq. If the message queue cannot be accessed, dae_init returns an error indication. If the message queue can be accessed, the integer pointed to by the dae_init_SRC_msq parameter dae_msqid is set to the message queue identifier of the aforementioned message queue. See A.3, “dae_init_SRC_msq(3)” on page 325 for more details.

If the dae_init_SRC_sock routine has been called, dae_init expects file descriptor 0 to be associated with a UNIX domain socket through which srcmstr will send requests to the daemon. If file descriptor 0 is not associated with a UNIX domain socket, dae_init returns an error indication. If file descriptor 0 is associated with a UNIX domain socket, file descriptor 0 is duplicated to another descriptor, and file descriptor 0 is opened against /dev/null. The file descriptor to which the UNIX domain socket is duplicated may have been specified on the call to dae_init_SRC_sock. The integer pointed to by the dae_init_SRC_sock parameter dae_SRC_sockd is set to the file descriptor associated with the socket. See A.4, “dae_init_SRC_sock(3)” on page 328 for more details.

- Ensure the daemon is not already running, if appropriate.

  If the dae_init_exclusive routine has been called, dae_init may use the values specified on the dae_init_exclusive call to ensure that the daemon is not already running. If it is already running, dae_init returns with an error indication. See A.9, “dae_init_exclusive(3)” on page 346 for more details.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

DAE_E_OK	Process successfully initialized as a daemon.
DAE_E_NOTAGAIN	The dae_init routine cannot be called more than once.
DAE_E_PINVALID	The value of the dae_parent parameter is invalid.
DAE_E_PWRONG	Could not verify daemon’s parent is one of those allowed by the dae_parent parameter.
DAE_E_PERROR	Error trying to validate parent.
DAE_E_CHILD	Could not create child process that would become daemon process.
DAE_E_SESSION	Could not make daemon process a session leader.
DAE_E_SIGNAL	Could not change the disposition of a signal.
DAE_E_CLOSE	Could not determine how many file descriptors to close.
DAE_E_DEVNULL	Could not open /dev/null on a standard file descriptor.
DAE_E_CHDIR	Could not change working directory.
DAE_E_SRCPREP	Failure attempting to prepare for SRC communications.
DAE_E_NOPSALLOC	The system does not support changing paging space allocation.
DAE_E_AINVALID	Invalid paging space allocation policy specified.
DAE_E_SETPSALLOC	Failure attempting to set the desired paging space allocation policy.
DAE_E_EXCLINVALID	Invalid parameters were passed to dae_init_exclusive.

DAE_E_EXCLERROR — Unexpected error enforcing exclusivity requirements of the daemon.

DAE_E_EXCLBUSY — Allowing the process to run as a daemon would violate the exclusivity requirements of the daemon program.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

If this routine is called by a program linked with the libdae.a library, it will not check to determine if the program had been started by inetd. Furthermore, if the program had called dae_init_SRC_sock, the dae_init routine will fail and return DAE_E_SRCPREP.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.2, “dae_init_SRC_sig(3)” on page 321
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.5, “dae_init_term_sig(3)” on page 336
A.6, “dae_init_prevent_zombies(3)” on page 338
A.7, “dae_init_lowps(3)” on page 340
A.8, “dae_init_psalloc(3)” on page 342
A.9, “dae_init_exclusive(3)” on page 346

A.2 dae_init_SRC_sig(3)

NAME

dae_init_SRC_sig - Causes dae_init to initialize the daemon under the assumption that SRC will make requests of the daemon using signals.

SYNOPSIS

```
#include <dae.h>

void dae_init_SRC_sig(
     int dae_sig_stop_normal,
     int dae_sig_stop_forced,
     const dae_req_sig_t *dae_SRC_rtns,
     int dae_restart
);
```

The dae.h file includes these definitions:

```
struct dae_req_sig {
        void (*dae_stop_normal)(int dae_signo);
        void (*dae_stop_forced)(int dae_signo);
        void (*dae_stop_cancel)(int dae_signo);
};

typedef struct dae_req_sig dae_req_sig_t;
```

PARAMETERS

dae_sig_stop_normal

Specifies the signal that this daemon expects to receive when the SRC makes the subsystem stop normal request.

If the value of this parameter is 0, the dae_init routine takes no action to support the SRC subsystem stop normal request. In this case, the dae_SRC_rtns->dae_stop_normal parameter value is expected to be NULL.

If the value of this parameter is SIGTERM, the SRC subsystem stop normal request will be handled in the same manner as the subsystem stop cancel request. In this case, the dae_SRC_rtns->dae_stop_normal parameter is expected to have the same value as the dae_SRC_rtns->dae_stop_cancel parameter.

If the value of this parameter is not 0 nor SIGTERM, the action taken depends on the value of the dae_SRC_rtns->dae_stop_normal parameter. If dae_SRC_rtns->dae_stop_normal is not NULL, the routine to which it points is installed as the signal handler for the signal specified by dae_sig_stop_normal. If dae_SRC_rtns->dae_stop_normal is NULL, a signal handler that only calls _exit(0) is installed for the signal specified by dae_sig_stop_normal.

dae_sig_stop_forced

Specifies the signal that this daemon expects to receive when the SRC makes the subsystem stop forced request.

If the value of this parameter is 0, the dae_init routine takes no action to support the SRC subsystem stop forced request. In this case, the dae_SRC_rtns->dae_stop_forced parameter value is expected to be NULL.

If the value of this parameter is SIGTERM, the SRC subsystem stop forced request will be handled in the same manner as the subsystem stop cancel request. In this case, the dae_SRC_rtns->dae_stop_forced parameter is expected to have the same value as the dae_SRC_rtns->dae_stop_cancel parameter.

If the value of this parameter is not 0 nor SIGTERM, the action taken depends on the value of the dae_SRC_rtns->dae_stop_forced parameter. If dae_SRC_rtns->dae_stop_forced is not NULL, the routine to which it points is installed as the signal handler for the signal specified by dae_sig_stop_forced. If dae_SRC_rtns->dae_stop_forced is NULL, a signal handler that only calls _exit(0) is installed for the signal specified by dae_sig_stop_forced.

dae_SRC_rtns:

Specifies the address of a dae_req_sig structure. The structure contains the members dae_stop_normal, dae_stop_forced, and dae_stop_cancel. The meanings of these members are described in the following paragraphs. Specifying a value of NULL for dae_SRC_rtns is semantically equivalent to specifying a pointer to a structure whose members are all NULL.

dae_stop_normal

Specifies the address of a routine that is to be installed as the signal handler for the signal specified in the dae_sig_stop_normal parameter. The daemon will receive the dae_sig_stop_normal signal when the SRC makes the subsystem stop normal request of the daemon. When called, the routine may terminate the daemon, but that is not required. The routine is just required to put the daemon in normal stop mode; it may return without terminating the daemon. When in normal stop mode, the daemon should not accept new requests for work. It should complete all current requests, release resources when current requests are completed, and then terminate itself. The subsystem stop normal request usually originates from the stopsrc command.

Like all signal handlers, the routine specified for dae_stop_normal should accept one parameter, an integer identifying the signal that has been caught. The routine should return nothing.

dae_stop_forced

Specifies the address of a routine that is to be installed as the signal handler for the signal specified in the dae_sig_stop_forced parameter. The daemon will receive the dae_sig_stop_forced signal when the SRC makes the subsystem stop forced request of the daemon. When called, the routine may terminate the daemon, but that is not required. The routine is just required to put the daemon in forced stop mode; it may return without terminating the daemon. When in forced stop mode, the daemon should terminate quickly, without completing current requests. It should release resources and terminate itself as quickly as possible. The subsystem stop forced request usually originates from the stopsrc -f command.

Like all signal handlers, the routine specified for dae_stop_forced should accept one parameter, an integer identifying the signal that has been caught. The routine should return nothing.

dae_stop_cancel

Specifies the address of a routine that is to be installed as the signal handler for the SIGTERM signal. The daemon will receive the SIGTERM signal when the SRC makes the subsystem stop cancel request of the daemon. When called, the routine may terminate the daemon, but that is not required. When called, the routine is just required to put the daemon in cancel stop mode; it may return without terminating the daemon. When in cancel stop mode, the daemon should terminate quickly, without completing current requests. It should release resources and terminate itself as quickly as possible. The subsystem stop cancel request usually originates from the stopsrc -c command.

If a daemon in cancel stop mode has not terminated after a subsystem dependent number of seconds, the SRC terminates the daemon by sending it the SIGKILL signal.

Like all signal handlers, the routine specified for dae_stop_cancel should accept one parameter, an integer identifying the signal that has been caught. The routine should return nothing.

If a value of NULL is specified for dae_stop_cancel, the disposition of the SIGTERM signal is set to the default. This will cause the daemon to terminate immediately when it receives the SIGTERM signal.

dae_restart

Specifies if the signal handlers installed will cause interrupted system calls to be automatically restarted. If the value of this parameter is 0, and a system call is interrupted by the execution of a stop normal, stop forced, or stop cancel signal handler, the interrupted system call will not be restarted. If the value of this parameter is not 0, restartable system calls interrupted by the execution of a stop normal, stop forced, or stop cancel signal handler will be restarted.

DESCRIPTION

A daemon calls the dae_init_SRC_sig routine when it is defined to the SRC as accepting requests through signals. Through this routine's parameters, the calling daemon informs the daemon support routines which signals the SRC will send to the daemon to make the subsystem stop normal and subsystem stop forced requests, and which routines will handle the subsystem stop normal, subsystem stop forced, and subsystem stop cancel requests. The subsystem stop cancel request is always made through the SIGTERM signal.

The actions specified by the call to dae_init_SRC_sig are taken when the daemon calls the dae_init routine. The signal handlers specified for the subsystem stop normal and subsystem stop forced requests are only installed if dae_init determines that the daemon has been started by the SRC. The disposition specified for the SIGTERM signal is implemented whether or not the daemon has been started by the SRC.

The signal handler for the dae_sig_stop_normal signal will be installed in such a manner that the dae_sig_stop_normal and dae_sig_stop_forced signals are blocked when the signal handler is executing.

The signal handler for the dae_sig_stop_forced signal will be installed in such a manner that the dae_sig_stop_normal and dae_sig_stop_forced signals are blocked when the signal handler is executing.

The signal handler for the SIGTERM signal will be installed such that the dae_sig_stop_normal, dae_sig_stop_forced, and SIGTERM signals are blocked when the signal handler is executing.

NOTES

If, after calling dae_init_SRC_sig and dae_init, the daemon process changes the disposition of the dae_sig_stop_normal, dae_sig_stop_forced, or SIGTERM signals, the behavior described on this man page might not occur.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

This routine has no return value. The dae_init routine will return an error indication if any error is encountered initializing the daemon.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.5, “dae_init_term_sig(3)” on page 336

A.3 dae_init_SRC_msq(3)

NAME

dae_init_SRC_msq - Causes dae_init to initialize the daemon under the assumption that SRC will make requests of the daemon using a message queue.

SYNOPSIS

```
#include <dae.h>

void dae_init_SRC_msq(
     int *dae_msqid,
     int dae_msqkey,
     int dae_msqtype,
     const dae_req_msq_t *dae_SRC_rtns,
     int dae_restart
);
```

The dae.h file includes these definitions:

```
struct dae_req_msq {
        void (*dae_stop_normal)(void);
        void (*dae_stop_forced)(void);
        void (*dae_stop_cancel)(int dae_signo);
        void (*dae_trace_begin)(int dae_long);
        void (*dae_trace_end)  (void);
        void (*dae_refresh)    (void);
        void (*dae_long_status)(void);
        int  (*dae_other_req)(short dae_request,
                     short dae_parm1, short dae_parm2,
                     const void *dae_parm3, int dae_parm3_size);
};

typedef struct dae_req_msq dae_req_msq_t;
```

PARAMETERS

dae_msqid
: If this pointer is not NULL, the message queue identifier of the message queue through which the SRC will make requests of this daemon will be returned through this pointer. The message queue identifier is not returned by this routine, but it will be returned when dae_init is invoked.

  If this pointer is NULL, no value is returned through the pointer.

dae_msqkey
: Specifies the message queue key of the message queue through which SRC requests are expected.

dae_msqtype
: Specifies the expected type of the messages carrying SRC requests.

dae_SRC_rtns
: Specifies the address of a dae_req_msq structure. The structure contains the members dae_stop_normal, dae_stop_forced, dae_stop_cancel, dae_trace_begin, dae_trace_end, dae_refresh, dae_long_status, and dae_other_req. The meanings of these members are described in the dae_init_SRC_sock man page. Specifying a value of NULL for dae_SRC_rtns is semantically equivalent to specifying a pointer to a structure whose members are all NULL.

dae_restart
: Specifies if the signal handler installed will cause interrupted system calls to be automatically restarted. If the value of this parameter is 0, and a system call is interrupted by the execution of the stop cancel signal handler, the interrupted system call will not be restarted. If the value of this parameter is not 0, restartable system calls interrupted by the execution of the stop cancel signal handler will be restarted.

DESCRIPTION

A daemon calls the dae_init_SRC_msq routine when it is defined to the SRC as accepting requests through a message queue. Through this routine's parameters, the calling daemon informs the daemon support routines which message queue will be used by the SRC to send requests to the daemon, and which routines will handle those requests. The subsystem stop cancel request is always made through the SIGTERM signal.

The actions specified by the call to dae_init_SRC_msq are taken when the daemon calls the dae_init routine. These actions are only taken if dae_init determines the daemon has been started by the SRC. However, the disposition specified for the SIGTERM signal is implemented whether or not the daemon has been started by the SRC.

A daemon accepting requests from the SRC through a message queue must call dae_SRC_req in a timely fashion when a request is waiting on the queue. A daemon may determine if a request is waiting on the message queue by calling the select routine. The dae_SRC_req routine receives a request from the queue, decodes the request, and may call one of the routines pointed to by the parameters of the dae_init_SRC_msq routine. For more information, see A.10, "dae_SRC_req(3)" on page 349.

NOTES

If, after calling dae_init_SRC_msq and dae_init, the daemon process changes the disposition of the SIGTERM signal, the behavior described on this man page might not occur.

EXAMPLES

See 10.2, "Examples of Using the Daemon Support Routines" on page 247.

RETURN VALUES

This routine has no return value. The dae_init routine will return an error indication if any error is encountered initializing the daemon.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.2, “dae_init_SRC_sig(3)” on page 321
A.4, “dae_init_SRC_sock(3)” on page 328
A.5, “dae_init_term_sig(3)” on page 336
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.4 dae_init_SRC_sock(3)

NAME

dae_init_SRC_sock - Causes dae_init to initialize the daemon under the assumption that SRC will make requests of the daemon using a UNIX domain socket.

SYNOPSIS

```
#include <dae.h>

void dae_init_SRC_sock(
     int *dae_SRC_sockd,
     const dae_req_sock_t *dae_SRC_rtns,
     int dae_restart
);
```

The dae.h file includes these definitions:

```
struct dae_req_sock {
        void (*dae_stop_normal)(void);
        void (*dae_stop_forced)(void);
        void (*dae_stop_cancel)(int dae_signo);
        void (*dae_trace_begin)(int dae_long);
        void (*dae_trace_end)  (void);
        void (*dae_refresh)    (void);
        void (*dae_long_status)(void);
        int  (*dae_other_req)(short dae_request,
                    short dae_parm1, short dae_parm2,
                    const void *dae_parm3, int dae_parm3_size);
};

typedef struct dae_req_sock dae_req_sock_t;
```

PARAMETERS

dae_SRC_sockd

If this pointer is not NULL, it is an input/output parameter.

As an input parameter, the value of the integer pointed to by this parameter specifies the file descriptor the daemon would like associated with the socket through which SRC requests will arrive. File descriptors 0, 1, and 2 may not be specified. A negative file descriptor indicates that the daemon process does not have a preference for which file descriptor is associated with the socket.

As an output parameter, the file descriptor of the UNIX domain socket through which the SRC will make requests of this daemon is returned through this pointer. The file descriptor is not returned by this routine, but it will be returned when dae_init is invoked.

dae_SRC_rtns: Specifies the address of a dae_req_sock structure. The structure contains the members dae_stop_normal, dae_stop_forced, dae_stop_cancel, dae_trace_begin, dae_trace_end, dae_refresh, dae_long_status, and dae_other_req. The meanings of these members are described in the following paragraphs. Specifying a value of NULL for dae_SRC_rtns is semantically equivalent to specifying a pointer to a structure whose members are all NULL.

dae_stop_normal Specifies the address of a routine that is to be called when the daemon receives the subsystem stop normal request from the SRC. When called, the routine may terminate the daemon, but that is not required. When called, the routine is just required to put the daemon in normal stop mode; it may return without terminating the daemon. When in normal stop mode, the daemon should not accept new requests for work. It should complete all current requests, release resources when current requests are completed, and then terminate itself. The subsystem stop normal request usually originates from the stopsrc command.

The routine specified for dae_stop_normal must expect no parameters.

The routine may not return informational messages with dae_inform_printf or dae_inform_puts, nor may it return error messages with dae_error_printf or dae_error_puts.

If a value of NULL is specified for dae_stop_normal, the daemon support routines will execute exit(0) when a subsystem stop normal request is received from the SRC.

dae_stop_forced Specifies the address of a routine that is to be called when the daemon receives the subsystem stop forced request from the SRC. When called, the routine may terminate the daemon, but that is not required. When called, the routine is just required to put the daemon in forced stop mode; it may return without terminating the daemon. When in forced stop mode, the daemon should terminate quickly, without completing current requests. It should release resources and terminate itself as quickly as possible. The subsystem stop forced request usually originates from the stopsrc -f command.

The routine specified for dae_stop_forced must expect no parameters.

The routine may not return informational messages with dae_inform_printf or dae_inform_puts, nor may it return error messages with dae_error_printf or dae_error_puts.

If a value of NULL is specified for dae_stop_forced, the daemon support routines will execute exit(0) when a subsystem stop forced request is received from the SRC.

dae_stop_cancel

Specifies the address of a routine that is to be installed as the signal handler for the SIGTERM signal. The daemon will receive the SIGTERM signal when the SRC makes the subsystem stop cancel request of the daemon. When called, the routine may terminate the daemon, but that is not required. When called, the routine is just required to put the daemon in cancel stop mode; it may return without terminating the daemon. When in cancel stop mode, the daemon should terminate quickly, without completing current requests. It should release resources and terminate itself as quickly as possible. The subsystem stop cancel request usually originates from the stopsrc -c command.

If a daemon in cancel stop mode has not terminated after a subsystem-dependent number of seconds, the SRC terminates the daemon by sending it the SIGKILL signal.

Like all signal handlers, the routine specified for dae_stop_cancel should accept one parameter, an integer identifying the signal that has been caught. The routine should return nothing.

If a value of NULL is specified for dae_stop_cancel, the disposition of the SIGTERM signal is set to the default. This will cause the daemon to terminate immediately when it receives the SIGTERM signal.

dae_trace_begin

Specifies the address of a routine that is to be called when the daemon receives the subsystem trace on request from the SRC. When called, the routine is expected to put the daemon in tracing mode, and return. The meaning of tracing mode is subsystem-dependent. The subsystem trace on request usually originates from the traceson command.

The routine specified for dae_trace_begin should accept one parameter, an integer indicating if long tracing mode is requested. If the integer's value is non-zero, the daemon should enter long tracing mode. If the integer's value is zero, the daemon should enter short tracing mode. The differences between short and long tracing modes are subsystem-dependent.

If the routine detects an error putting the daemon in tracing mode, the routine may call dae_error_printf or dae_error_puts to return an error message. The error message will be displayed on the standard output of the process that sent the subsystem trace on request. If the sending process is executing the traceson command, it will exit with an error code. An attempt to put the daemon in tracing mode when it is already in tracing mode should not be considered an error.

If the routine encounters no errors putting the daemon in tracing mode, it should not call dae_error_printf nor dae_error_puts. If the routine returns without calling either of those error routines, the process that sent the subsystem trace on request will receive an indication of success.

The routine may cause informational messages to be displayed on the standard output of the process that sent the subsystem trace on request by calling the dae_inform_printf and dae_inform_puts routines.

A value of NULL for dae_trace_begin indicates the daemon does not support the SRC subsystem trace on request.

dae_trace_end

Specifies the address of a routine that is to be called when the daemon receives the subsystem trace off request from the SRC. When called, the routine is expected to take the daemon out of tracing mode, and return. The meaning of tracing mode is subsystem-dependent. The subsystem trace off request usually originates from the tracesoff command.

The routine specified for dae_trace_end should expect no parameters.

If the routine detects an error taking the daemon out of tracing mode, the routine may call dae_error_printf or dae_error_puts to return an error message. The error message will be displayed on the standard output of the process that sent the subsystem trace off request. If the sending process is executing the tracesoff command, it will exit with an error code. An attempt to take the daemon out of tracing mode when it is not in tracing mode should not be considered an error.

If the routine encounters no errors taking the daemon out of tracing mode, it should not call dae_error_printf nor dae_error_puts. If the routine returns without calling either of those error routines, the process that sent the subsystem trace off request will receive an indication of success.

The routine may cause informational messages to be displayed on the standard output of the process that sent the subsystem trace off request by calling the dae_inform_printf and dae_inform_puts routines.

A value of NULL for dae_trace_end indicates the daemon does not support the SRC subsystem trace off request.

dae_refresh

Specifies the address of a routine that is to be called when the daemon receives the subsystem refresh request from the SRC. When called, the routine is expected to refresh the daemon, and return. The meaning of refreshing the daemon is subsystem-dependent. If a daemon has a configuration file, it is common for the daemon to re-read the configuration file when receiving the subsystem refresh request. The subsystem refresh request usually originates from the refresh command.

The routine specified for dae_refresh should expect no parameters.

If the routine detects an error refreshing the daemon, the routine may call dae_error_printf or dae_error_puts to return an error message. The error message will be displayed on the standard output of the process that sent the subsystem refresh request. If the sending process is executing the refresh command, it will exit with an error code.

If the routine encounters no errors refreshing the daemon, it should not call dae_error_printf nor dae_error_puts. If the routine returns without calling either of those error routines, the process that sent the subsystem refresh request will receive an indication of success.

The routine may cause informational messages to be displayed on the standard output of the process that sent the subsystem refresh request by calling the dae_inform_printf and dae_inform_puts routines.

A value of NULL for dae_refresh indicates the daemon does not support the SRC subsystem refresh request.

dae_long_status

Specifies the address of a routine that is to be called when the daemon receives the subsystem long status request from the SRC. When called, the routine is expected to provide the long status of the daemon, and return. The status provided is subsystem-dependent. The dae_status_short, dae_status_puts, dae_status_printf, dae_margin_puts, and dae_margin_printf routines can be called to provide the status. See the man pages for those routines for details, as noted in "RELATED INFORMATION." The status is displayed on the standard output of the process which sent the subsystem long status request. This request usually originates from the lssrc command.

The routine specified for dae_long_status should expect no parameters.

If the routine detects an error reporting the status of the daemon, the routine may call dae_error_printf or dae_error_puts to return an error message. The error message will be displayed on the standard output of the process that sent the subsystem long status request. If the sending process is executing the lssrc command, it will exit with an error code.

If the routine encounters no errors reporting the status of the daemon, it should not call dae_error_printf nor dae_error_puts. If the routine returns without calling either of those error routines, the process that sent the subsystem long status request will receive an indication of success.

The routine may cause informational messages to be displayed on the standard output of the process that sent the subsystem long status request by calling the dae_inform_printf and dae_inform_puts routines. These informational messages may be intermixed with the returned status.

A value of NULL for dae_long_status indicates the daemon does not support the SRC subsystem long status request.

dae_other_req

Specifies the address of a routine that is to be called when the daemon receives a subsystem request other than the standard requests discussed in the preceding paragraphs. When called, the routine is expected to process the request, and return. The meaning of the requests handled by this routine are subsystem-dependent. The subsystem requests processed by this routine usually originate from commands not provided with the SRC. See Chapter 12, “New SRC Program Support” on page 303 for information about how these commands can be created.

The routine specified for dae_other_req should expect five parameters. The first parameter, dae_request, identifies the specific request being made of the subsystem. The number representing a request must be agreed upon between the subsystem and the command generating the request. See 10.2.5, “Running under the SRC: Subsystem-Defined Requests” on page 271 for details. The dae_parm1, dae_parm2, and dae_parm3 parameters are request parameters that may modify the meaning of the request. The values of these parameters may or may not be significant for a particular request. The dae_parm1 and dae_parm2 parameters can hold signed short integer values. The dae_parm3 parameter can point to any arbitrary data type. The dae_parm3_size parameter indicates the size of the data pointed to by the dae_parm3 parameter.

If the subsystem does not support the request identified by dae_request, the routine should return a value of 1. If the subsystem does support the request identified by dae_request, the routine should process the request and return 0.

If the routine detects an error while processing the request, the routine may call dae_error_printf or dae_error_puts to return an error message. The error message will be displayed on the standard output of the process that sent the request.

If the routine encounters no errors processing the request, it should not call dae_error_printf nor dae_error_puts. If the routine returns without calling either of those error routines, the process that sent the request will receive an indication of success.

The routine may cause informational messages to be displayed on the standard output of the process that sent the request by calling the dae_inform_printf and dae_inform_puts routines.

A value of NULL for dae_other_req indicates the daemon does not support additional subsystem requests.

dae_restart

Specifies if the signal handler installed will cause interrupted system calls to be automatically restarted. If the value of this parameter is 0, and a system call is interrupted by the execution of the stop cancel signal handler, the interrupted system call will not be restarted. If the value of this parameter is not 0, restartable system calls interrupted by the execution of the stop cancel signal handler will be restarted.

DESCRIPTION

A daemon calls the dae_init_SRC_sock routine when it is defined to the SRC as accepting requests through a UNIX domain socket. Through this routine′s parameters, the calling daemon informs the daemon support routines which routines will handle SRC requests sent to the daemon. The subsystem stop cancel request is always made through the SIGTERM signal.

The actions specified by the call to dae_init_SRC_sock are taken when the daemon calls the dae_init routine. These actions are only taken if dae_init determines the daemon has been started by the SRC. However, the disposition specified for the SIGTERM signal is implemented whether or not the daemon has been started by the SRC.

A daemon accepting requests from the SRC through a socket must call dae_SRC_req in a timely fashion when a request is waiting on the socket. A daemon may determine if a request is waiting on the socket by calling the select routine. The dae_SRC_req routine receives a request from the socket, decodes the request, and may call one of the routines pointed to by the parameters of the dae_init_SRC_sock routine. For more information, see A.10, “dae_SRC_req(3)” on page 349.

NOTES

If, after calling dae_init_SRC_sock and dae_init, the daemon process changes the disposition of the SIGTERM signal, the behavior described on this man page might not occur.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

This routine has no return value. The dae_init routine will return an error indication if any error is encountered initializing the daemon.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

If this routine is called by a program linked with the libdae.a library, the dae_init routine will fail and return DAE_E_SRCPREP.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.2, “dae_init_SRC_sig(3)” on page 321
A.3, “dae_init_SRC_msq(3)” on page 325
A.5, “dae_init_term_sig(3)” on page 336
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.5 dae_init_term_sig(3)

NAME

dae_init_term_sig - Causes dae_init to initialize the disposition of the SIGTERM signal.

SYNOPSIS

```
#include <dae.h>

void dae_init_term_sig(
     void (*dae_term_rtn)(int dae_signo),
     int dae_restart
);
```

PARAMETERS

dae_term_rtn

Specifies the address of a routine that is to be installed as the signal handler for the SIGTERM signal. It is common for a daemon to be sent the SIGTERM signal when it should be stopped.

Like all signal handlers, the routine specified for dae_term_rtn should accept one parameter, an integer identifying the signal that has been caught. The routine should return nothing.

If a value of NULL is specified for dae_term_rtn, the disposition of the SIGTERM signal is set to the default. This will cause the daemon to terminate immediately when it receives the SIGTERM signal.

dae_restart

Specifies if the signal handler installed will cause interrupted system calls to be automatically restarted. If the value of this parameter is 0, and a system call is interrupted by the execution of the signal handler, the interrupted system call will not be restarted. If the value of this parameter is not 0, restartable system calls interrupted by the execution of the signal handler will be restarted.

DESCRIPTION

A daemon calls the dae_init_term_sig routine when it will never be started by the SRC, but it wants to set the disposition of the SIGTERM signal. A daemon that will always be started by inetd might use this routine. An alternative to using this routine is for the daemon to change the disposition of the SIGTERM signal itself by using the sigaction routine.

The disposition changes for the SIGTERM signal requested by the call to dae_init_term_sig are made when the daemon calls the dae_init routine.

NOTES

If, after calling dae_init_term_sig and dae_init, the daemon process changes the disposition of the SIGTERM signal, the behavior described on this man page might not occur.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

This routine has no return value. The `dae_init` routine will return an error indication if any error is encountered initializing the daemon.

LOCATION

This routine resides in the `libdae_bsd.a` and `libdae.a` libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.2, “dae_init_SRC_sig(3)” on page 321
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328

A.6 dae_init_prevent_zombies(3)

NAME

dae_init_prevent_zombies - Causes dae_init to prevent the daemon process from creating zombie processes.

SYNOPSIS

```
#include <dae.h>

void dae_init_prevent_zombies(
     dae_parent_t dae_parent,
     int dae_restart
);
```

PARAMETERS

dae_parent

Specifies the conditions under which zombies will be prevented. Zombies will be prevented only when the daemon was started in a manner specified by this parameter. The value of this parameter is formed by using one or more of these constants as operands in an OR operation:

- DAE_P_SRC: If the daemon was started by the SRC, zombies will be prevented.
- DAE_P_INETD: If the daemon was started by inetd, zombies will be prevented.
- DAE_P_OTHER: If the daemon was not started by inetd nor by SRC, zombies will be prevented.

dae_restart

Specifies if the signal handler installed will cause interrupted system calls to be automatically restarted. If the value of this parameter is 0, and a system call is interrupted by the execution of the signal handler, the interrupted system call will not be restarted. If the value of this parameter is not 0, restartable system calls interrupted by the execution of the signal handler will be restarted.

DESCRIPTION

If a daemon will create child processes, but is not interested in knowing when the child processes terminate and is not interested in the exit values of the child processes, it may call dae_init_prevent_zombies to ensure its child processes do not become long lasting zombie processes. By default, a terminated process remains a zombie process until its parent process has collected its exit status. On many systems, including AIX, ignoring the SIGCHLD signal will prevent a process' terminated child processes from becoming zombie processes. On systems that support this semantic, a call to dae_init_prevent_zombies will cause dae_init to ignore the SIGCHLD signal. On systems that do not support this semantic, the dae_init_prevent_zombies routine will cause dae_init to install a signal handler for SIGCHLD that will cleanup zombie processes by calling waitpid.

NOTES

If, after calling dae_init_prevent_zombies and dae_init, the daemon process changes the disposition of the SIGCHLD signal, the behavior described on this man page might not occur.

This routine may cause the installation of a signal handler that returns. As a result, system calls in the daemon process may be interrupted.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

This routine has no return value. The dae_init routine will return an error indication if any error is encountered initializing the daemon.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page section in this appendix is related:

A.1, “dae_init(3)” on page 316

A.7 dae_init_lowps(3)

NAME

dae_init_lowps - Causes dae_init to install a signal handler for the SIGDANGER signal. This will protect the daemon process from termination when the system is low on paging space.

SYNOPSIS

```
#include <dae.h>

void dae_init_lowps(
     dae_parent_t dae_parent,
     void (*dae_lowps)(int dae_signo),
     int dae_restart
);
```

PARAMETERS

dae_parent

Specifies the conditions under which the SIGDANGER signal handler will be installed. The SIGDANGER signal handler will be installed only when the daemon was started in a manner specified by this parameter. The value of this parameter is formed by using one or more of these constants as operands in an OR operation:

- DAE_P_SRC: If the daemon was started by the SRC, the SIGDANGER signal handler should be installed.
- DAE_P_INETD: If the daemon was started by inetd, the SIGDANGER signal handler should be installed.
- DAE_P_OTHER: If the daemon was not started by inetd nor by SRC, the SIGDANGER signal handler should be installed.

dae_lowps

Specifies the address of a routine that is to be installed as the signal handler for the SIGDANGER signal, or specifies a value of NULL. If the value of dae_lowps is not NULL, the specified signal handler is installed for SIGDANGER. If the value of dae_lowps is NULL, a signal handler that does nothing but return is installed for the SIGDANGER signal.

dae_restart

Specifies if the signal handler installed will cause interrupted system calls to be automatically restarted. If the value of this parameter is 0, and a system call is interrupted by the execution of the signal handler, the interrupted system call will not be restarted. If the value of this parameter is not 0, restartable system calls interrupted by the execution of the signal handler will be restarted.

DESCRIPTION

A daemon calls the dae_init_lowps routine to install a signal handler for the SIGDANGER signal. A daemon might want to install such a signal handler to disclaim a region of memory when the system is low on free paging space, or to protect itself from termination if the system starts killing processes when the level of free paging space becomes critical.

The specified signal handler will not actually be installed until the daemon process calls the dae_init routine.

On an AIX system, when the number of free paging-space blocks falls below a configurable warning level, the kernel sends the SIGDANGER signal to all processes running on the system. By default, the SIGDANGER signal is ignored. Processes may choose to install a signal handler for SIGDANGER that will disclaim regions of memory whose contents are not needed. This may increase the number of free paging-space blocks available to the system. If the number of free paging-space blocks falls below a configurable kill level, the kernel starts sending SIGKILL signals to selected processes until the number of free paging-space blocks exceeds the kill level. A process that has a SIGDANGER signal handler installed will not be killed in low paging space situations.

NOTES

If, after calling dae_init_lowps and dae_init, the daemon process changes the disposition of the SIGDANGER signal, the behavior described on this man page might not occur.

This routine causes the installation of a signal handler that returns. As a result, system calls in the daemon process may be interrupted.

A daemon may protect itself from being terminated when the system is low on free paging-space blocks by either calling dae_init_psalloc to force the daemon to run with the early paging space allocation policy, or calling dae_init_lowps to install a signal handler for the SIGDANGER signal. A daemon may call both routines if it wants to specify the paging space allocation policy with which it runs, and it wants to install a signal handler for the SIGDANGER signal.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

This routine has no return value. The dae_init routine will return an error indication if any error is encountered initializing the daemon.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

PORTABILITY

On a non-AIX system, calling this routine has no effect.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.8, “dae_init_psalloc(3)” on page 342

A.8 dae_init_psalloc(3)

NAME

dae_init_psalloc - Causes dae_init to set the paging space allocation policy of the daemon process as specified by the parameters.

SYNOPSIS

```
#include <dae.h>

void dae_init_psalloc(
     dae_parent_t dae_parent,
     dae_psalloc_t dae_ps_policy,
     const char *dae_prog_path,
     char * const dae_argv[]
);
```

The dae.h file includes these definitions:

```
enum dae_psalloc {
                   DAE_A_DONT_CARE = 0,
                   DAE_A_LATE_TRY,
                   DAE_A_EARLY_TRY,
                   DAE_A_LATE,
                   DAE_A_EARLY,
                   DAE_A_INVALID
                  };

typedef enum dae_psalloc dae_psalloc_t;
```

PARAMETERS

dae_parent

Specifies the conditions under which the paging space allocation policy will be set, as described by the dae_ps_policy parameter. The paging space allocation policy will be set only when the daemon was started in a manner specified by this parameter. The value of this parameter is formed by using one or more of these constants as operands in an OR operation:

- DAE_P_SRC: If the daemon was started by the SRC, the paging space allocation policy will be set as described by the dae_ps_policy parameter.
- DAE_P_INETD: If the daemon was started by inetd, the paging space allocation policy will be set as described by the dae_ps_policy parameter.
- DAE_P_OTHER: If the daemon was not started by inetd nor by SRC, the paging space allocation policy will be set as described by the dae_ps_policy parameter.

dae_ps_policy — Specifies the paging space allocation policy desired. The possible values are:

- DAE_A_DONT_CARE: Do not change the paging space allocation policy of the process.
- DAE_A_LATE_TRY: Attempt to make the process run with the late paging space allocation policy; do not verify the results.
- DAE_A_EARLY_TRY: Attempt to make the process run with the early paging space allocation policy; do not verify the results.
- DAE_A_LATE: Make the process run with the late paging space allocation policy; verify the results.
- DAE_A_EARLY: Make the process run with the early paging space allocation policy; verify the results.

dae_prog_path — Specifies the full path name of the program being executed as the daemon. If the dae_ps_policy parameter is set to DAE_A_LATE_TRY, DAE_A_EARLY_TRY, DAE_A_LATE, or DAE_A_EARLY, the dae_init routine may re-execute this program.

dae_argv — Specifies the arguments with which the daemon program was started. If the dae_ps_policy parameter is set to DAE_A_LATE_TRY, DAE_A_EARLY_TRY, DAE_A_LATE, or DAE_A_EARLY, the dae_init routine may re-execute this program.

DESCRIPTION

On AIX systems, processes run with the late paging space allocation policy in effect by default. The policy with which a process runs can be changed using the PSALLOC environment variable. When exec is issued for a program, the value of the PSALLOC environment variable is examined by the kernel. If the variable has the value early, the program runs with the early paging space allocation policy in effect; otherwise, it runs with the late paging space allocation policy in effect.

A daemon may call the dae_init_psalloc routine if it wants to run with a specific paging space allocation policy. The paging space allocation policy is not changed until the dae_init routine is called. The dae_init routine ensures that the daemon process runs with the desired paging space allocation policy. It also arranges for child processes of the daemon process to run with the late paging space allocation policy.

If dae_init_psalloc had been called to request a specific paging space allocation policy for the daemon process, the dae_init routine will follow these steps:

1. Retrieve the process flags of the process. If the process flags indicate the process is already running with the desired paging space allocation policy, proceed to step 4.
2. If early paging space allocation is desired, put PSALLOC=early in the environment of the process, if it is not already there. If late paging space allocation is desired, put PSALLOC=late in the environment of the process, if the environment does not already contain a value for PSALLOC that is consistent with the late paging space allocation policy.
3. Re-execute the daemon program. The path name used is that provided by the dae_prog_path parameter. The arguments used are those provided through the dae_argv parameter. This is the only mechanism available to change the paging space allocation policy used by a running process. The re-executed daemon program will call dae_init again. In step 1, dae_init

should find that the process flags indicate the process is running with the desired paging space allocation policy. Therefore, step 1 should take the branch to step 4.

4. If the current setting of PSALLOC in the environment of the process is not consistent with the late paging space allocation policy, put PSALLOC=late in the environment of the process. Child processes of the daemon process will inherit this value in their environments. They will then run with the late paging space allocation policy in effect after they exec a program.

It is very significant that dae_init may issue exec again for the daemon program to change the daemon process′ paging space allocation policy. The following points should be kept in mind:

- The value of the dae_prog_path parameter must be correct. If it is not correct, exec will not function for the program correctly.
- The arguments of the program, anchored by dae_argv, must not have been changed by the daemon program; otherwise, the program will get different arguments when exec is re-issued.
- All the actions taken by the daemon program from the beginning of the main routine to the call to exec in dae_init must be repeatable and harmless when repeated.

NOTES

A daemon may protect itself from being terminated when the system is low on free paging-space blocks by either calling dae_init_psalloc to force the daemon to run with the early paging space allocation policy, or calling dae_init_lowps to install a signal handler for the SIGDANGER signal. A daemon may call both routines if it wants to specify the paging space allocation policy with which it runs, and it wants to install a signal handler for the SIGDANGER signal.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

This routine has no return value. The dae_init routine will return an error indication if any error is encountered when initializing the daemon.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

PORTABILITY

On a non-AIX system, this routine should only be called with the dae_ps_policy parameter set to DAE_A_DONT_CARE, DAE_A_LATE_TRY, or DAE_A_EARLY_TRY. Such a call would have no effect on the calling program. If this routine is called with any other value for the dae_ps_policy parameter, the call to dae_init will fail and return the error value DAE_E_NOPSALLOC.

An AIX 3.2 system may or may not support the early paging space allocation policy. If DAE_A_EARLY is specified on a call to dae_init_psalloc on an AIX 3.2 system that does not support the early paging space allocation policy, dae_init will fail. This problem can be avoided by specifying DAE_A_EARLY_TRY on calls to dae_init_psalloc on AIX 3.2 systems.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.7, “dae_init_lowps(3)” on page 340

A.9 dae_init_exclusive(3)

NAME

dae_init_exclusive - Causes dae_init to enforce the exclusivity requirements of the daemon.

SYNOPSIS

```
#include <dae.h>

void dae_init_exclusive(
     dae_parent_t dae_parent,
     const char *dae_excl_path,
     char dae_excl_ID
);
```

PARAMETERS

dae_parent
: Specifies the conditions under which exclusivity should be enforced. The exclusivity described by the remaining parameters will only be enforced when the daemon is started in a manner specified by this parameter. The value of this parameter is formed by using one or more of these constants as operands in an OR operation:

  - DAE_P_SRC: If the daemon was started by the SRC, exclusivity will be enforced as specified by the other parameters.
  - DAE_P_INETD: If the daemon was started by inetd, exclusivity will be enforced as specified by the other parameters.
  - DAE_P_OTHER: If the daemon was not started by inetd nor by SRC, exclusivity will be enforced as specified by the other parameters.

dae_excl_path
: Points to a path name that is used in conjunction with the dae_excl_ID parameter to enforce the exclusivity requirements of the daemon. This is commonly the full path name of the daemon program.

dae_excl_ID
: Specifies a project ID that is used in conjunction with the dae_excl_path parameter to enforce the exclusivity requirements of the daemon. This must be a value from 1 to 255.

DESCRIPTION

It is not uncommon for a daemon to have some sort of requirement relating to exclusivity. For example, a daemon program may be designed such that only one process should ever be running the program on a system. On a control workstation associated with a partitioned RS/6000 SP system, it may be necessary to have multiple processes running the same daemon, one and only one for each RS/6000 SP partition. The dae_init_exclusive routine can be used to instruct dae_init to enforce the desired exclusivity.

The exclusivity requirements are enforced using System V semaphores. The daemon support routines generate a semaphore key using the ftok routine, and the values of dae_excl_path and dae_excl_ID. In order for the dae_init routine to succeed, the semaphore must be successfully locked by the daemon process. The semaphore remains locked for the lifetime of the daemon process. If the

semaphore cannot be locked, allowing the process to run as the daemon would violate the exclusivity requirements of the daemon. Therefore, dae_init will return an error indication to the daemon.

If only one process should run the daemon program on the system at any one time, the daemon program should call dae_init_exclusive with constant values for dae_excl_path and dae_excl_ID. The combined values should be unlikely to be used by some other program. Convenient values might be the full path name of the daemon program, and 1.

If one process per RS/6000 SP partition should run the daemon program on the system at any one time, the daemon program should call dae_init_exclusive with a constant value for dae_excl_path that is unique to the daemon program, and a value for dae_excl_ID that depends on the RS/6000 SP partition to be supported by the calling process. Convenient values might be the full path name of the daemon program for dae_excl_path, and the partition number for dae_excl_ID.

The value used for dae_excl_path must be the path of an existing file. It should also be the path of a file that is not often recreated or replaced. Two calls to ftok may return different keys if the file referred to was recreated between the calls.

The SRC, inetd, and init have mechanisms to allow only one instance of a particular daemon to run at a time on a system. These mechanisms are useful and should be used, as appropriate. However, these mechanisms can fail. The SRC mechanism fails whenever srcmstr is terminated and respawned. The inetd mechanism fails when inetd is terminated and respawned. None of the mechanisms prevent other instances of the daemon being started from a shell. The dae_init_exclusive routine allows a daemon to enforce exclusivity rules in the event the other mechanisms fail.

NOTES

When the daemon terminates, the semaphore lock will be automatically released. If the daemon terminates by calling the exit routine, the semaphore will also be removed from the system. If the daemon terminates without calling the exit routine, the semaphore will remain in the system, but it will not be locked.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

This routine has no return value. The dae_init routine will return an error indication if any error is encountered initializing the daemon.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page section in this appendix is related:

A.1, “dae_init(3)” on page 316

A.10 dae_SRC_req(3)

NAME

dae_SRC_req - Processes the next request on the SRC message queue or socket.

SYNOPSIS

```
#include <dae.h>

void dae_SRC_req(void);
```

PARAMETERS

This routine takes no parameters.

DESCRIPTION

The dae_SRC_req routine should be called by a daemon when it detects that the message queue or socket through which SRC requests arrive contains an SRC request. The daemon can determine this using the select routine. The dae_SRC_req routine reads the next request from the SRC message queue or socket in a non-blocking fashion. If the message queue or socket does not contain an SRC request, the dae_SRC_req routine returns immediately.

If an SRC request is successfully read, the type of the request is determined, and dae_SRC_req determines if the daemon has identified a routine to handle the request. The daemon would have identified routines to handle SRC requests by calling dae_init_SRC_msq or dae_init_SRC_sock. If the daemon has not identified a routine to handle the request, or the request is not a valid SRC request, or the SRC request is not a subsystem request, dae_SRC_req replies to the process making the SRC request with an appropriate error code. If the daemon has identified a routine to handle the request, that routine is called. If the request handling routine calls dae_error_puts or dae_error_printf before returning to dae_SRC_req, the request will be considered to have resulted in an error. If the routine returns to dae_SRC_req without having called dae_error_puts or dae_error_printf, the request is considered to have been successful. In either case, dae_SRC_req makes the appropriate reply. The interfaces of and semantics associated with the request handling routines are described in A.3, “dae_init_SRC_msq(3)” on page 325 and A.4, “dae_init_SRC_sock(3)” on page 328.

EXAMPLES

See 10.2, “Examples of Using the Daemon Support Routines” on page 247.

RETURN VALUES

There is no return value.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.11 dae_status_short(3)

NAME

dae_status_short - Includes subsystem short status in subsystem long status output.

SYNOPSIS

```
#include <dae.h>

int dae_status_short(void);
```

PARAMETERS

This routine takes no parameters.

DESCRIPTION

When the dae_status_short routine is called from a routine supporting the SRC subsystem long status request, the subsystem short status of the calling subsystem is included in the subsystem long status output.

The subsystem short status output starts a new line of output, if necessary.

The dae_status_short routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem long status request until the routine called by dae_SRC_req returns. The routine dae_SRC_req calls to process the SRC subsystem long status request is specified as the dae_long_status member of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq routine or the dae_init_SRC_sock routine.

NOTES

The subsystem short status output is obtained from the SRC daemon, srcmstr. If srcmstr cannot respond to the request, the dae_status_short routine may take 60 seconds to complete.

The output generated by dae_status_short is not necessarily sent immediately to the process making the subsystem long status request. The subsystem long status output is buffered within the daemon support routines. Any buffered output will be sent to the process making the subsystem long status request when the routine supporting the SRC subsystem long status request returns.

EXAMPLES

Assume the routine designated to handle the subsystem long status request for subsystem des.ex.sock is report_status, as shown here:

```
static void report_status(void)
{
     dae_status_short();

     dae_status_printf("\nI've seen %d client connections.\n",
                       client_count);
     dae_status_puts(tracing_on ? "Tracing is on.\n"
                                : "Tracing is off.\n");

     return;
}
```

The output generated by a subsystem long status request made for des.ex.sock might appear as follows:

```
# lssrc -ls des.ex.sock
Subsystem         Group            PID     Status
 des.ex.sock      des.ex           8418    active

 I've seen 0 client connections.
 Tracing is off.
```

The first two lines of output from the lssrc command are created by the call to dae_status_short in report_status.

RETURN VALUES

If the routine is successful, it returns 0. If the routine is not successful, it returns -1. The error conditions detectable by the routine are:

- The daemon is not currently processing a subsystem long status request.
- Memory could not be obtained to receive the subsystem short status output.

A successful return code is not a guarantee that the subsystem short status output is received by the process making the subsystem long status request.

The routine does not change errno.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.12 dae_status_puts(3)

NAME

dae_status_puts - Provides an interface similar to puts for specifying the subsystem long status output.

SYNOPSIS

```
#include <dae.h>

int dae_status_puts(const char *dae_str);
```

PARAMETERS

dae_str
: Pointer to a null terminated string that is placed in the body of the subsystem long status output.

DESCRIPTION

When the dae_status_puts routine is called from a routine supporting the SRC subsystem long status request, the string pointed to by the dae_str parameter is placed in the body of the current line of subsystem long status output.

The dae_status_puts routine behaves like puts as much as possible. Known differences between puts and dae_status_puts are:

- The puts routine displays its output on the standard output of the calling process. The output of dae_status_puts is sent to the process making the SRC subsystem long status request. That process usually displays the output on its standard output.
- The puts routine always appends a newline character at the end of the string. The dae_status_puts routine does not do so. In this respect, the dae_status_puts routine is similar to fputs.
- The dae_status_puts routine cannot place more than 64 characters in the body area of a line. Therefore, the routine may insert line breaks where they are not specified in the string. If the string contains % characters, fewer then 64 characters may fit in the body area of a line.

The dae_status_puts routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem long status request until the routine called by dae_SRC_req returns. The routine dae_SRC_req calls to process the SRC subsystem long status request is specified as the dae_long_status member of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq routine or the dae_init_SRC_sock routine.

NOTES

The SRC imposes a structure on the output of the lssrc command that limits the format of subsystem long status output. A goal of the daemon support routines that support the generation of subsystem long status output is to generalize the format as much as possible. This is accomplished using the model of the subsystem long status output format described in the next paragraph.

Each line of the subsystem long status output is viewed as being composed of a margin, a separator column, and a body area. The width of the margin can be changed dynamically; by default, it is zero characters wide; it can be up to 29 characters wide. The width of the separator column is constant; it is one character wide. The width of the body area depends on the data placed there, up to a maximum of 64 characters.

The dae_margin_puts and dae_margin_printf routines can be used to define the width of the margin and to place data in the margin. If too many characters are specified to be placed in the margin, the data wraps to the margin of the next line of output. Once the dae_margin_puts or dae_margin_printf routine changes the margin width, the change is in effect until another call to one of these routines changes the margin width again.

The separator column is always blank.

The dae_status_puts and dae_status_printf routines can be used to put data in the body area. If too many characters are specified to be placed in the body area, the data wraps to the body area of the next line of output.

The output generated by dae_status_puts is not necessarily sent immediately to the process making the subsystem long status request. The subsystem long status output is buffered within the daemon support routines. Any buffered output will be sent to the process making the subsystem long status request when the routine supporting the SRC subsystem long status request returns.

EXAMPLES

Assume the routine designated to handle the subsystem long status request for subsystem des.ex.sock is report_status, as shown here:

```
static void report_status(void)
{
     dae_status_short();

     dae_status_printf("\nI've seen %d client connections.\n",
                       client_count);
     dae_status_puts(tracing_on ? "Tracing is on.\n"
                                : "Tracing is off.\n");

     return;
}
```

The output generated by a subsystem long status request made for des.ex.sock might appear as follows:

```
# lssrc -ls des.ex.sock
Subsystem         Group            PID     Status
 des.ex.sock      des.ex           8418    active

 I've seen 0 client connections.
 Tracing is off.
```

The line of output stating, "Tracing is off." was created by the call to dae_status_puts in report_status. The blank before the output created by

dae_status_puts is the separator column. Since neither dae_margin_puts nor dae_margin_printf was called, the margin is 0 bytes wide.

RETURN VALUES

If the routine is successful, it returns the string length of the string pointed to by the dae_str parameter. If the routine is not successful, it returns EOF. The error condition detectable by the routine follows:

- The daemon is not currently processing a subsystem long status request.

A successful return code is not a guarantee that the output is received by the process making the subsystem long status request.

The routine does not change errno.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.13 dae_status_printf(3)

NAME

dae_status_printf - Provides an interface similar to printf for specifying the subsystem long status output.

SYNOPSIS

```
#include <dae.h>

int dae_status_printf(const char *dae_fmt, ...);
```

PARAMETERS

dae_fmt
: Pointer to a null terminated string that specifies the format of data to be placed in the body of the subsystem long status output. See the printf man page for information about the format string.

...
: Arguments formatted under the control of the dae_fmt parameter. See the printf man page for information about these arguments.

DESCRIPTION

When the dae_status_printf routine is called from a routine supporting the SRC subsystem long status request, the string pointed to by the dae_fmt parameter and the arguments that follow are used to create an output string that is placed in the body of the current line of subsystem long status output.

The dae_status_printf routine behaves like printf as much as possible. Known differences between printf and dae_status_printf are:

- The printf routine displays its output on the standard output of the calling process. The output of dae_status_printf is sent to the process making the SRC subsystem long status request. That process usually displays the output on its standard output.
- The dae_status_printf routine cannot place more than 64 characters in the body area of a line. So, the routine may insert line breaks where they are not specified in the output string. If the output string contains % characters, fewer then 64 characters may fit in the body area of a line.

The dae_status_printf routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem long status request until the routine called by dae_SRC_req returns. The routine dae_SRC_req calls to process the SRC subsystem long status request is specified as the dae_long_status member of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq routine or the dae_init_SRC_sock routine.

NOTES

The SRC imposes a structure on the output of the lssrc command that limits the format of subsystem long status output. A goal of the daemon support routines that support the generation of subsystem long status output is to generalize the format as much as possible. This is accomplished using the model of the subsystem long status output format described in the next paragraph.

Each line of the subsystem long status output is viewed as being composed of a margin, a separator column, and a body area. The width of the margin can be changed dynamically; by default, it is zero characters wide, but it can be up to 29 characters wide. The width of the separator column is constant; it is one character wide. The width of the body area depends on the data placed there, up to a maximum of 64 characters.

The dae_margin_puts and dae_margin_printf routines can be used to define the width of the margin and to place data in the margin. If too many characters are specified to be placed in the margin, the data wraps to the margin of the next line of output. Once the dae_margin_puts or dae_margin_printf routine changes the margin width, the change is in effect until another call to one of these routines changes the margin width again.

The separator column is always blank.

The dae_status_puts and dae_status_printf routines can be used to put data in the body area. If too many characters are specified to be placed in the body area, the data wraps to the body area of the next line of output.

The output generated by dae_status_printf is not necessarily sent immediately to the process making the subsystem long status request. The subsystem long status output is buffered within the daemon support routines. Any buffered output will be sent to the process making the subsystem long status request when the routine supporting the SRC subsystem long status request returns.

If a single call to this routine generates over 4095 bytes of output, an internal buffer will overflow. The effects of such an overflow are unpredictable.

EXAMPLES

Assume the routine designated to handle the subsystem long status request for subsystem des.ex.sock is report_status, as shown here.

```
static void report_status(void)
{
     dae_status_short();

     dae_status_printf("\nI've seen %d client connections.\n",
                       client_count);
     dae_status_puts(tracing_on ? "Tracing is on.\n"
                                : "Tracing is off.\n");

     return;
}
```

The output generated by a subsystem long status request made for des.ex.sock might appear as follows:

```
# lssrc -ls des.ex.sock
Subsystem         Group            PID     Status
 des.ex.sock      des.ex           8418    active

 I've seen 0 client connections.
 Tracing is off.
```

The line of output stating, "I've seen 0 client connections." was created by the call to dae_status_printf in report_status. The blank before the output created by dae_status_printf is the separator column. Since neither dae_margin_puts nor dae_margin_printf was called, the margin is 0 bytes wide.

RETURN VALUES

If the routine is successful, it returns the string length of the output string generated from the format string and the arguments. If the routine is not successful, it returns -1. The error condition detectable by the routine follows:

- The daemon is not currently processing a subsystem long status request.

A successful return code is not a guarantee that the output is received by the process making the subsystem long status request.

The routine does not change errno.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, "Linking with the Daemon Support Routines" on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, "dae_init(3)" on page 316
A.3, "dae_init_SRC_msq(3)" on page 325
A.4, "dae_init_SRC_sock(3)" on page 328
A.11, "dae_status_short(3)" on page 351
A.12, "dae_status_puts(3)" on page 354
A.14, "dae_margin_puts(3)" on page 360
A.15, "dae_margin_printf(3)" on page 364
A.16, "dae_inform_puts(3)" on page 368
A.17, "dae_inform_printf(3)" on page 371
A.18, "dae_error_puts(3)" on page 374
A.19, "dae_error_printf(3)" on page 377

A.14 dae_margin_puts(3)

NAME

dae_margin_puts - Provides an interface similar to puts for specifying the subsystem long status margin output.

SYNOPSIS

```
#include <dae.h>

int dae_margin_puts(int dae_width, const char *dae_str);
```

PARAMETERS

dae_width
: Specifies the new width of the margin, in characters. If the value is negative, the margin width is not changed. If the value is not negative, the margin width is changed to the specified value. The maximum width of the margin is 29 characters. A value larger than this maximum is silently adjusted to the maximum.

dae_str
: Pointer to a null terminated string that is placed in the margin of the subsystem long status output.

DESCRIPTION

When the dae_margin_puts routine is called from a routine supporting the SRC subsystem long status request, the width of the margin is adjusted as specified by the dae_width parameter, and the string pointed to by the dae_str parameter is placed in the margin of the subsystem long status output. If no data has been written to the body area of the current line, the string specified by dae_str is placed in the margin of the current line. If data has been written to the body area of the current line, the next line becomes the current line, and the string specified by dae_str is placed in the margin of the new current line.

The dae_margin_puts routine behaves like puts as much as possible. Known differences between puts and dae_margin_puts are:

- The puts routine displays its output on the standard output of the calling process. The output of dae_margin_puts is sent to the process making the SRC subsystem long status request. That process usually displays the output on its standard output.
- The puts routine always appends a newline character at the end of the string. The dae_margin_puts routine does not do so. In this respect, the dae_margin_puts routine is similar to fputs.
- The dae_margin_puts routine cannot place more characters in the margin of a line than are allowed by the margin width. Therefore, the routine may insert line breaks where they are not specified in the string.
- The dae_margin_puts routine cannot process the literal % character correctly. If a literal % character is specified for output, the dae_margin_puts routine will fail.

The dae_margin_puts routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem long status request until the routine called by dae_SRC_req returns. The routine dae_SRC_req calls to process the SRC

subsystem long status request is specified as the dae_long_status member of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq routine or the dae_init_SRC_sock routine.

NOTES

The SRC imposes a structure on the output of the lssrc command that limits the format of subsystem long status output. A goal of the daemon support routines that support the generation of subsystem long status output is to generalize the format as much as possible. This is accomplished using the model of the subsystem long status output format described in the next paragraph.

Each line of the subsystem long status output is viewed as being composed of a margin, a separator column, and a body area. The width of the margin can be changed dynamically; by default, it is zero characters wide, but it can be up to 29 characters wide. The width of the separator column is constant; it is one character wide. The width of the body area depends on the data placed there, up to a maximum of 64 characters.

The dae_margin_puts and dae_margin_printf routines can be used to define the width of the margin and to place data in the margin. If too many characters are specified to be placed in the margin, the data wraps to the margin of the next line of output. Once the dae_margin_puts or dae_margin_printf routine changes the margin width, the change is in effect until another call to one of these routines changes the margin width again.

The separator column is always blank.

The dae_status_puts and dae_status_printf routines can be used to put data in the body area. If too many characters are specified to be placed in the body area, the data wraps to the body area of the next line of output.

The output generated by dae_margin_puts is not necessarily sent immediately to the process making the subsystem long status request. The subsystem long status output is buffered within the daemon support routines. Any buffered output will be sent to the process making the subsystem long status request when the routine supporting the SRC subsystem long status request returns.

EXAMPLES

Assume the routine designated to handle the subsystem long status request for subsystem des.ex.sock is get_status, as shown here:

```
#if defined(_BSD)
     #define GETPGRP()   getpgrp((pid_t)0)
#else
     #define GETPGRP()   getpgrp()
#endif

void get_status(void)
{
     char **envp;


     dae_margin_puts(strlen("Subsystem:"), "Subsystem:");
     dae_status_puts("des.ex.sock\n\n");
```

```
     dae_margin_puts(-1, "PID:");
     dae_status_printf("%d\n", getpid());

     dae_margin_puts(-1, "PPID:");
     dae_status_printf("%d\n", getppid());

     dae_margin_puts(-1, "PGID:");
     dae_status_printf("%d\n", GETPGRP());

     dae_margin_puts(-1, "SID:");
     dae_status_printf("%d\n", getsid(getpid()));

     dae_margin_puts(0, "\n");

     envp = environ;

     for (envp = environ; (*envp != NULL); envp++) {
          dae_status_printf("%s\n", *envp);
     }

     return;
}
```

The output generated by a subsystem long status request made for des.ex.sock might appear as follows:

```
Subsystem: des.ex.sock

PID:       6582
PPID:      3934
PGID:      6582
SID:       6582

 TERM=dumb
 AUTHSTATE=compat
 SHELL=/bin/ksh
 HOME=/
 USER=root
 PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin
 TZ=EST5EDT
 LANG=C
 LOCPATH=/usr/lib/nls/loc
 NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
 LC__FASTMSG=true
 ODMDIR=/etc/objrepos
 LOGNAME=root
 LOGIN=root
```

RETURN VALUES

If the routine is successful, it returns the string length of the string pointed to by the dae_str parameter. If the routine is not successful, it returns EOF. The error conditions detectable by the routine follow:

- The daemon is not currently processing a subsystem long status request.
- A literal % character was specified for output.
- The current margin width is 0 characters, and a character other than newline was specified for output.

A successful return code is not a guarantee that the output is received by the process making the subsystem long status request.

The routine does not change errno.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.15 dae_margin_printf(3)

NAME

dae_margin_printf - Provides an interface similar to printf for specifying the subsystem long status margin output.

SYNOPSIS

```
#include <dae.h>

int dae_margin_printf(int dae_width, const char *dae_fmt, ...);
```

PARAMETERS

dae_width
: Specifies the new width of the margin, in characters. If the value is negative, the margin width is not changed. If the value is not negative, the margin width is changed to the specified value. The maximum width of the margin is 29 characters. A value larger than this maximum is silently adjusted to the maximum.

dae_fmt
: Pointer to a null terminated string that specifies the format of data to be placed in the margin of the subsystem long status output. See the printf man page for information about the format string.

...
: Arguments formatted under the control of the dae_fmt parameter. See the printf man page for information about these arguments.

DESCRIPTION

When the dae_margin_printf routine is called from a routine supporting the SRC subsystem long status request, the string pointed to by the dae_fmt parameter and the arguments that follow are used to create an output string that is placed in the margin of the subsystem long status output. If no data has been written to the body area of the current line, the output string is placed in the margin of the current line. If data has been written to the body area of the current line, the next line becomes the current line, and the output string is placed in the margin of the new current line.

The dae_margin_printf routine behaves like printf as much as possible. Known differences between printf and dae_margin_printf are:

- The printf routine displays its output on the standard output of the calling process. The output of dae_margin_printf is sent to the process making the SRC subsystem long status request. That process usually displays the output on its standard output.
- The dae_margin_printf routine cannot place more characters in the margin of a line than are allowed by the margin width. Therefore, the routine may insert line breaks where they are not specified in the string.
- The dae_margin_printf routine cannot process the literal % character correctly. If a literal % character is specified for output, the dae_margin_printf routine will fail.

The dae_margin_printf routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem long status request until the routine called

by dae_SRC_req returns. The routine dae_SRC_req calls to process the SRC subsystem long status request is specified as the dae_long_status member of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq routine or the dae_init_SRC_sock routine.

NOTES

The SRC imposes a structure on the output of the lssrc command that limits the format of subsystem long status output. A goal of the daemon support routines that support the generation of subsystem long status output is to generalize the format as much as possible. This is accomplished using the model of the subsystem long status output format described in the next paragraph.

Each line of the subsystem long status output is viewed as being composed of a margin, a separator column, and a body area. The width of the margin can be changed dynamically; by default, it is zero characters wide, but it can be up to 29 characters wide. The width of the separator column is constant; it is one character wide. The width of the body area depends on the data placed there, up to a maximum of 64 characters.

The dae_margin_puts and dae_margin_printf routines can be used to define the width of the margin and to place data in the margin. If too many characters are specified to be placed in the margin, the data wraps to the margin of the next line of output. Once the dae_margin_puts or dae_margin_printf routine changes the margin width, the change is in effect until another call to one of these routines changes the margin width again.

The separator column is always blank.

The dae_status_puts and dae_status_printf routines can be used to put data in the body area. If too many characters are specified to be placed in the body area, the data wraps to the body area of the next line of output.

The output generated by dae_margin_printf is not necessarily sent immediately to the process making the subsystem long status request. The subsystem long status output is buffered within the daemon support routines. Any buffered output will be sent to the process making the subsystem long status request when the routine supporting the SRC subsystem long status request returns.

If a single call to this routine generates over 4095 bytes of output, an internal buffer will overflow. The effects of such an overflow are unpredictable.

EXAMPLES

Assume the routine designated to handle the subsystem long status request for subsystem des.ex.sock is get_status, as shown here:

```
#if defined(_BSD)
     #define GETPGRP()   getpgrp((pid_t)0)
#else
     #define GETPGRP()   getpgrp()
#endif

void get_status(void)
{
     const char *subsys_str = "des.ex.sock";
```

```
    const int   subsys_len = strlen(subsys_str);
    const char *PID_str    = "  PID:";
    const int   PID_len    = strlen(PID_str);
    char **envp;

    dae_margin_printf(subsys_len + PID_len, "%s%s",
                      subsys_str, PID_str);
    dae_status_printf("%d\n", getpid());

    dae_margin_printf(-1, "%*s", subsys_len + PID_len, "PPID:");
    dae_status_printf("%d\n", getppid());

    dae_margin_printf(-1, "%*s", subsys_len + PID_len, "PGID:");
    dae_status_printf("%d\n", GETPGRP());

    dae_margin_printf(-1, "%*s", subsys_len + PID_len, "SID:");
    dae_status_printf("%d\n", getsid(getpid()));

    dae_margin_puts(0, "\n");

    envp = environ;
    for (envp = environ; (*envp != NULL); envp++) {
        dae_status_printf("%s\n", *envp);
    }

    return;
}
```

The output generated by a subsystem long status request made for des.ex.sock might appear as follows:

```
des.ex.sock  PID: 6582
            PPID: 3934
            PGID: 6582
             SID: 6582

 TERM=dumb
 AUTHSTATE=compat
 SHELL=/bin/ksh
 HOME=/
 USER=root
 PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin
 TZ=EST5EDT
 LANG=C
 LOCPATH=/usr/lib/nls/loc
 NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
 LC__FASTMSG=true
 ODMDIR=/etc/objrepos
 LOGNAME=root
 LOGIN=root
```

RETURN VALUES

If the routine is successful, it returns the string length of the output string generated from the format string and the arguments. If the routine is not successful, it returns -1. The error conditions detectable by the routine follow:

- The daemon is not currently processing a subsystem long status request.
- A literal % character was specified for output.
- The current margin width is 0 characters, and a character other than newline was specified for output.

A successful return code is not a guarantee that the output is received by the process making the subsystem long status request.

The routine does not change errno.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.16 dae_inform_puts(3)

NAME

dae_inform_puts - Provides an interface similar to puts for displaying information messages while processing SRC requests.

SYNOPSIS

```
#include <dae.h>

int dae_inform_puts(const char *dae_str);
```

PARAMETERS

dae_str
: Pointer to a null terminated string that is sent to the standard output of the process making the SRC request.

DESCRIPTION

When the dae_inform_puts routine is called from a routine supporting the SRC subsystem trace on, trace off, refresh, or long status request, or from a routine supporting subsystem-defined SRC requests, the string pointed to by the dae_str parameter is sent to the standard output of the process making the SRC request.

The dae_inform_puts routine behaves like puts as much as possible. Known differences between puts and dae_inform_puts are:

- The puts routine displays its output on the standard output of the calling process. The output of dae_inform_puts is sent to the standard output of the process making the SRC request.
- The puts routine always appends a newline character at the end of the string. The dae_inform_puts routine appends a newline character at the end of the string if a newline character does not end the string, and the string is the last informative message sent by the routine handling the SRC request.
- The dae_inform_puts routine cannot place more than 141 characters on a line. So, the routine may insert line breaks where they are not specified in the string.

The dae_inform_puts routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem trace on request, the SRC subsystem trace off request, the SRC subsystem refresh request, the SRC subsystem long status request, or a subsystem-defined SRC request until the routine called by dae_SRC_req returns. The routines dae_SRC_req would call to process the previously mentioned requests would have been specified by the dae_trace_begin, dae_trace_end, dae_refresh, dae_long_status, and dae_other_req members of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq or dae_init_SRC_sock routine.

NOTES

When the calling routine is supporting the subsystem long status request, the output generated by dae_inform_puts and dae_inform_printf may be intermixed

with the output generated by dae_status_puts, dae_status_printf, dae_margin_puts, and dae_margin_printf.

The output generated by dae_inform_puts is not necessarily sent immediately to the process making the SRC request. The information messages are buffered within the daemon support routines. Any buffered output will be sent to the process making the SRC request when the routine supporting the request returns.

The dae_inform_puts routine could support lines up to 255 characters in length, if the traceson, tracesoff, and refresh commands were changed to accept a full SRC reply packet.

EXAMPLES

Assume the routine designated to handle the subsystem trace on request for subsystem des.ex.sock is start_trace, as shown here:

```
void start_trace(int longtrace)
{
     if (longtrace) {
          current_tracing = 2;
          dae_inform_puts("Long");
     } else {
          current_tracing = 1;
          dae_inform_puts("Short");
     }

     dae_inform_puts(" tracing for dae.ex.sock logged to ");
     dae_inform_puts(trace_path);
     dae_inform_puts(".\n");

     return;
}
```

The start_trace routine assumes two global variables, current_tracing and trace_path. The trace_path variable is assumed to contain the path name of a file to which trace records are written. The current_tracing variable is assumed to take one of three possible values: 0 indicates tracing is off, 1 indicates short tracing is on, and 2 indicates long tracing is on.

The output generated by a subsystem trace on request made for des.ex.sock might appear as follows:

```
# traceson -ls des.ex.sock
Long tracing for des.ex.sock logged to /tmp/des.ex.sock.8804.
0513-091 The request to turn on tracing was completed successfully.
```

The first line of output is created by the calls to dae_inform_puts in start_trace. The second line of output is displayed by the traceson command when it is successful.

RETURN VALUES

If the routine is successful, it returns the string length of the string pointed to by the dae_str parameter. If the routine is not successful, it returns EOF. The error condition detectable by the routine follows:

- The daemon is not currently processing a subsystem trace on, trace off, refresh, or long status request, or a subsystem-defined request.

A successful return code is not a guarantee that the output is received by the process making the SRC request.

The routine does not change errno.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.17 dae_inform_printf(3)

NAME

dae_inform_printf - Provides an interface similar to printf for displaying information messages while processing SRC requests.

SYNOPSIS

```
#include <dae.h>

int dae_inform_printf(const char *dae_fmt, ...);
```

PARAMETERS

dae_fmt
: Pointer to a null terminated string that specifies the format of data to be sent to the standard output of the process making the SRC request. See the printf man page in *AIX Version 4.1: Technical Reference* for information about the format string.

...
: Arguments formatted under the control of the dae_fmt parameter. See the printf man page in *AIX Version 4.1: Technical Reference* for information about these arguments.

DESCRIPTION

When the dae_inform_printf routine is called from a routine supporting the SRC subsystem trace on, trace off, refresh, or long status request, or from a routine supporting subsystem-defined SRC requests, the string pointed to by the dae_fmt parameter and the arguments that follow are used to create an output string that is sent to the standard output of the process making the SRC request.

The dae_inform_printf routine behaves like printf as much as possible. Known differences between printf and dae_inform_printf are:

- The printf routine displays its output on the standard output of the calling process. The output of dae_inform_printf is sent to the standard output of the process making the SRC request.
- The dae_inform_printf routine cannot place more than 141 characters on a line. Therefore, the routine may insert line breaks where they are not specified in the output string.

The dae_inform_printf routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem trace on request, the SRC subsystem trace off request, the SRC subsystem refresh request, the SRC subsystem long status request, or a subsystem-defined SRC request until the routine called by dae_SRC_req returns. The routines that dae_SRC_req would call to process the previously mentioned requests would have been specified by the dae_trace_begin, dae_trace_end, dae_refresh, dae_long_status, and dae_other_req members of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq or dae_init_SRC_sock routine.

NOTES

When the calling routine is supporting the subsystem long status request, the output generated by dae_inform_puts and dae_inform_printf may be intermixed with the output generated by dae_status_puts, dae_status_printf, dae_margin_puts, and dae_margin_printf.

The output generated by dae_inform_printf is not necessarily sent immediately to the process making the SRC request. The information messages are buffered within the daemon support routines. Any buffered output will be sent to the process making the SRC request when the routine supporting the request returns.

The dae_inform_printf routine could support lines up to 255 characters in length, if the traceson, tracesoff, and refresh commands were changed to accept a full SRC reply packet.

If a single call to this routine generates over 4095 bytes of output, an internal buffer will overflow. The effects of such an overflow are unpredictable.

EXAMPLES

Assume the routine designated to handle the subsystem trace on request for subsystem des.ex.sock is start_trace, as shown here:

```
void start_trace(int longtrace)
{
     if (longtrace) {
          current_tracing = 2;
     } else {
          current_tracing = 1;
     }

     dae_inform_printf("%s tracing for des.ex.sock logged to %s.\n",
          (longtrace) ? "Long" : "Short", trace_path);

     return;
}
```

The start_trace routine assumes two global variables, current_tracing and trace_path. The trace_path variable is assumed to contain the path name of a file to which trace records are written. The current_tracing variable is assumed to take one of three possible values: 0 indicates tracing is off, 1 indicates short tracing is on, and 2 indicates long tracing is on.

The output generated by a subsystem trace on request made for des.ex.sock might appear as follows:

```
# traceson -ls des.ex.sock
Long tracing for des.ex.sock logged to /tmp/des.ex.sock.8804.
0513-091 The request to turn on tracing was completed successfully.
```

The first line of output is created by the call to dae_inform_printf in start_trace. The second line of output is displayed by the traceson command when it is successful.

RETURN VALUES

If the routine is successful, it returns the string length of the output string generated from the format string and the arguments. If the routine is not successful, it returns -1. The error condition detectable by the routine follows:

- The daemon is not currently processing a subsystem trace on, trace off, refresh, or long status request, or a subsystem-defined request.

A successful return code is not a guarantee that the output is received by the process making the SRC request.

The routine does not change `errno`.

LOCATION

This routine resides in the `libdae_bsd.a` and `libdae.a` libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.18, “dae_error_puts(3)” on page 374
A.19, “dae_error_printf(3)” on page 377

A.18 dae_error_puts(3)

NAME

dae_error_puts - Provides an interface similar to puts for displaying error messages while processing SRC requests.

SYNOPSIS

```
#include <dae.h>

int dae_error_puts(const char *dae_str);
```

PARAMETERS

dae_str
: Pointer to a null terminated string that is sent as an error message to the process making the SRC request.

DESCRIPTION

When the dae_error_puts routine is called from a routine supporting the SRC subsystem trace on, trace off, refresh, or long status request, or from a routine supporting subsystem-defined SRC requests, the string pointed to by the dae_str parameter is sent as an error message to the process making the SRC request. If the process making the SRC request is running the traceson, tracesoff, refresh, or lssrc command, or a command generated as described in Chapter 12, “New SRC Program Support” on page 303, it will display the error message on its standard output and exit with a bad return code.

The dae_error_puts routine behaves like puts as much as possible. Known differences between puts and dae_error_puts are:

- The puts routine displays its output on the standard output of the calling process. The output of dae_error_puts is sent to the process making the SRC request. That process will likely display the error message on its standard output.
- The puts routine always appends a newline character at the end of the string. The dae_error_puts routine appends a newline character at the end of the string if a newline character does not end the string, and the string is the last error message sent by the routine handling the SRC request.
- The dae_error_puts routine cannot be used to send more than 140 characters to the process making the SRC request. Excess characters will be truncated.

The dae_error_puts routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem trace on request, the SRC subsystem trace off request, the SRC subsystem refresh request, the SRC subsystem long status request, or a subsystem-defined SRC request until the routine called by dae_SRC_req returns. The routines dae_SRC_req would call to process the previously mentioned requests would have been specified by the dae_trace_begin, dae_trace_end, dae_refresh, dae_long_status, and dae_other_req members of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq or dae_init_SRC_sock routine.

NOTES

The output generated by dae_error_puts is not necessarily sent immediately to the process making the SRC request. The error messages are buffered within the daemon support routines. Any buffered output will be sent to the process making the SRC request when the routine supporting the request returns.

The dae_error_puts routine could support lines up to 254 characters in length, if the traceson, tracesoff, and refresh commands were changed to accept a full SRC reply packet.

The traceson command, the tracesoff command, the refresh command, and commands generated as described in Chapter 12, "New SRC Program Support" on page 303, do not exit with a bad return code when they are invoked with the -g flag, even when this routine is used to display an error message.

EXAMPLES

Assume the routine designated to handle the subsystem trace on request for subsystem des.ex.sock is start_trace, as shown here:

```
void start_trace(int longtrace)
{
     int debug_on = 1;

     if (setsockopt(acpt_sockd, SOL_SOCKET, SO_DEBUG,
                    (char *) &debug_on, sizeof debug_on) == -1) {

          /* Log something */

          dae_error_puts("Subsystem des.ex.sock: "
                         "setsockopt() error: \"");
          dae_error_puts(strerror(errno));
          dae_error_puts("\".\n");
     }

     return;
}
```

The start_trace routine will set the SO_DEBUG option for one of the daemon's sockets when receiving the subsystem trace on request. If an error is returned by setsockopt, start_trace will call dae_error_puts to return an error message to the process that sent the request. If no error is reported by setsockopt, start_trace will not call dae_error_puts or dae_error_printf; therefore, the process which sent the request will get an indication of success.

The output generated by a subsystem trace on request made for des.ex.sock might appear as follows if the request fails because the variable acpt_sockd does not represent a valid file descriptor:

```
# traceson -ls des.ex.sock
Subsystem des.ex.sock: setsockopt() error: "Bad file number".
```

RETURN VALUES

If the routine is successful, it returns the string length of the string pointed to by the dae_str parameter. If the routine is not successful, it returns EOF. The error condition detectable by the routine follows:

- The daemon is not currently processing a subsystem trace on, trace off, refresh, or long status request, or a subsystem-defined request.

A successful return code is not a guarantee that the output is received by the process making the SRC request.

The routine does not change errno.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.19, “dae_error_printf(3)” on page 377

A.19 dae_error_printf(3)

NAME

dae_error_printf - Provides an interface similar to printf for displaying error messages while processing SRC requests.

SYNOPSIS

```
#include <dae.h>

int dae_error_printf(const char *dae_fmt, ...);
```

PARAMETERS

dae_fmt — Pointer to a null terminated string that specifies the format of data to be sent as an error message to the process making the SRC request. See the printf man page for information about the format string.

... — Arguments formatted under the control of the dae_fmt parameter. See the printf man page for information about these arguments.

DESCRIPTION

When the dae_error_printf routine is called from a routine supporting the SRC subsystem trace on, trace off, refresh, or long status request, or from a routine supporting subsystem-defined SRC requests, the string pointed to by the dae_fmt parameter and the arguments that follow are used to create an output string that is sent to the process making the SRC request as an error message. If the process making the SRC request is running the traceson, tracesoff, refresh, or lssrc command, or a command generated as described in Chapter 12, “New SRC Program Support” on page 303, it will display the error message on its standard output and exit with a bad return code.

The dae_error_printf routine behaves like printf as much as possible. Known differences between printf and dae_error_printf are:

- The printf routine displays its output on the standard output of the calling process. The output of dae_error_printf is sent to the process making the SRC request. That process will likely display the error message on its standard output.
- The dae_error_printf routine cannot be used to send more than 140 characters to the process making the SRC request. Excess characters will be truncated.

The dae_error_printf routine may be called from the time dae_SRC_req calls a routine to process the SRC subsystem trace on request, the SRC subsystem trace off request, the SRC subsystem refresh request, the SRC subsystem long status request, or a subsystem-defined SRC request, until the routine called by dae_SRC_req returns. The routines that dae_SRC_req would call to process the previously mentioned requests would have been specified by the dae_trace_begin, dae_trace_end, dae_refresh, dae_long_status, and dae_other_req members of the dae_SRC_rtns parameter in a call to the dae_init_SRC_msq or dae_init_SRC_sock routine.

NOTES

The output generated by dae_error_printf is not necessarily sent immediately to the process making the SRC request. The error messages are buffered within the daemon support routines. Any buffered output will be sent to the process making the SRC request when the routine supporting the request returns.

The dae_error_printf routine could support lines up to 255 characters in length, if the traceson, tracesoff, and refresh commands were changed to accept a full SRC reply packet.

The traceson command, the tracesoff command, the refresh command, and commands generated as described in Chapter 12, "New SRC Program Support" on page 303, do not exit with a bad return code when they are invoked with the -g flag, even when this routine is used to display an error message.

If a single call to this routine generates over 4095 bytes of output, an internal buffer will overflow. The effects of such an overflow are unpredictable.

EXAMPLES

Assume the routine designated to handle the subsystem trace on request for subsystem des.ex.sock is start_trace, as shown here:

```
void start_trace(int longtrace)
{
     int debug_on = 1;

     if (setsockopt(acpt_sockd, SOL_SOCKET, SO_DEBUG,
                    (char *) &debug_on, sizeof debug_on) == -1) {

          /* Log something */

          dae_error_printf("Subsystem des.ex.sock: "
                  "setsockopt() error, \"%s\".\n",
                  strerror(errno));
     }

     return;
}
```

The start_trace routine will set the SO_DEBUG option for one of the daemon's sockets when receiving the subsystem trace on request. If an error is returned by setsockopt, start_trace will call dae_error_printf to return an error message to the process that sent the request. If no error is reported by setsockopt, start_trace will not call dae_error_puts or dae_error_printf; therefore, the process which sent the request will get an indication of success.

The output generated by a subsystem trace on request made for des.ex.sock might appear as follows if the request failed because the variable acpt_sockd does not represent a valid file descriptor.

```
# traceson -ls des.ex.sock
Subsystem des.ex.sock: setsockopt() error: "Bad file number".
```

RETURN VALUES

If the routine is successful, it returns the string length of the output string generated from the format string and the arguments. If the routine is not successful, it returns -1. The error condition detectable by the routine follows:

- The daemon is not currently processing a subsystem trace on, trace off, refresh, or long status request, or a subsystem-defined request.

A successful return code is not a guarantee that the output is received by the process making the SRC request.

The routine does not change `errno`.

LOCATION

This routine resides in the `libdae_bsd.a` and `libdae.a` libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page sections in this appendix are related:

A.1, “dae_init(3)” on page 316
A.3, “dae_init_SRC_msq(3)” on page 325
A.4, “dae_init_SRC_sock(3)” on page 328
A.11, “dae_status_short(3)” on page 351
A.12, “dae_status_puts(3)” on page 354
A.13, “dae_status_printf(3)” on page 357
A.14, “dae_margin_puts(3)” on page 360
A.15, “dae_margin_printf(3)” on page 364
A.16, “dae_inform_puts(3)” on page 368
A.17, “dae_inform_printf(3)” on page 371
A.18, “dae_error_puts(3)” on page 374

A.20 dae_process_is_the_daemon(3)

NAME

dae_process_is_the_daemon - Indicates if the calling process is the process that has successfully called dae_init to initialize itself as a daemon.

SYNOPSIS

```
#include <dae.h>

int dae_process_is_the_daemon(void);
```

PARAMETERS

This routine takes no parameters.

DESCRIPTION

The dae_process_is_the_daemon routine can be called by a process to see if it is the process that has successfully called dae_init to initialize itself as a daemon. The routine may be useful in signal handlers. If a daemon process creates a child process, the child process inherits the signal handlers installed by the daemon process. The signal handler may need to perform differently when run in a child process of a daemon than in the daemon process itself.

RETURN VALUES

If the calling process has initialized itself by calling dae_init, the routine returns 1. Otherwise, it returns 0.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

RELATED INFORMATION

The following man page section in this appendix is related:

A.1, “dae_init(3)” on page 316

A.21 dae_fopen(3)

NAME

dae_fopen - Provides an interface similar to fopen, and prevents the daemon from acquiring a controlling terminal as a result of calling open.

SYNOPSIS

```
#include <dae.h>

FILE *dae_fopen(const char *dae_filename, const char *dae_mode);
```

PARAMETERS

dae_filename
: The path name of the file to be opened.

dae_mode
: This parameter controls whether the file is opened for reading, writing, or both. See the fopen man page in *AIX Version 4.1: Technical Reference* for details.

DESCRIPTION

The dae_fopen routine is designed to perform the same function as the standard I/O routine fopen. The only difference is the calling process will not acquire a controlling terminal as a result of opening a terminal device through dae_fopen. If a process that is a session leader without a controlling terminal calls fopen to open a terminal device that is not already a controlling terminal for a session, the process will obtain the terminal device as its controlling terminal. This is usually undesirable for a daemon process; thus, the dae_fopen routine is provided as a safe alternative to fopen.

RETURN VALUES

If an error occurs, NULL is returned. Otherwise, a pointer to an open stream is returned.

LOCATION

This routine resides in the libdae_bsd.a and libdae.a libraries.

For information about determining with which library a program should be linked, and other libraries with which the program must be linked, refer to 10.3, “Linking with the Daemon Support Routines” on page 286.

A.22 DAE_M_STOP(3)

NAME

DAE_M_STOP - A macro that allows a daemon to be debugged from a certain point.

SYNOPSIS

```
#include <dae.h>
```

PARAMETERS

This macro takes no parameters.

DESCRIPTION

The DAE_M_STOP macro will stop the current process if the environment contains a variable DAE_DEBUG with a value of stop. The process will continue to run when sent a SIGCONT signal. The macro is intended to be placed at the very beginning of the main routine of a daemon. It allows a daemon to be stopped as soon as its execution begins, if debugging is required. After the process is started with a command like:

```
startsrc -s subsys -e "DAE_DEBUG=stop"
```

a debugger can be attached to the process, and execution of the process will start just after the macro invocation. The dbx debugger can be attached to the daemon with the command:

```
dbx -a <PID of daemon>.
```

If dbx is to be terminated without terminating the daemon, use the dbx detach command.

EXAMPLES

```
#include <unistd.h>
#include <dae.h>


int main(int argc, char **argv)
{
     /* Process stops here if DAE_DEBUG=stop is in environment */
     DAE_M_STOP;

     /* Do whatever main is supposed to do ... */

     return 0;
}
```

RETURN VALUES

No value is returned.

LOCATION

This macro is defined in the dae.h header file.

Appendix B. SRC Notification Support Programs man Pages

This appendix provides detailed program instructions for the use of the SRC Notification Support Programs. The level of detail is consistent with that commonly provided in UNIX systems man pages. The appendix provides reference for the user of the programs, or for a daemon writer interested in writing an interface to the programs.

B.1 dae_notify(1)

NAME

dae_notify - Generic SRC notification method.

SYNOPSIS

```
dae_notify [-s syspath] [-u usrpath] subsys_name [group_name]
```

ARGUMENTS

-s syspath — The syspath argument specifies the directory to be searched for a subsystem-supplied notification program for the subsystem identified by the subsys_name argument and for the group identified by the group_name argument. If the -s argument is not specified, the directory in which you installed dae_notify is searched.

-u usrpath — The usrpath argument specifies the directory to be searched for user exit programs for the subsystem identified by the subsys_name argument and for the group identified by the group_name argument. If the -u argument is not specified, the same directory is searched for user exit programs as is searched for the subsystem-supplied notification program.

subsys_name — This argument specifies the name of the subsystem that has failed. This argument is provided automatically by the SRC.

group_name — This argument specifies the name of the group to which the failed subsystem belongs. This argument is provided automatically by the SRC. If the subsystem is not defined to be a member of a group, this argument is not provided.

DESCRIPTION

The dae_notify program is designed as a generic SRC notification method. In order for a subsystem to use this generic SRC notification method, an object specifying dae_notify as the notification method must be defined for the subsystem, or the group to which the subsystem belongs, in the SRCnotify object class. The notification method field of the object placed in the SRCnotify object class should include the -s and -u flags, and their arguments, if needed. The subsys_name and group_name arguments should not be included in the notification method field of the object placed in the SRCnotify object class. These arguments are supplied by the srcmstr daemon when the notification method is invoked.

When the dae_notify program is invoked, it takes the following steps:

1. It determines the directory that will be searched for subsystem-supplied notification programs. If the -s flag was specified in the notification method field of the SRCnotify object, the directory to search is specified by the -s argument, syspath. Otherwise, the directory to search defaults to the one in which dae_notify is installed.
2. It determines the directory that will be searched for user exit programs. If the -u flag was specified in the notification method field of the SRCnotify

object, the directory to search is specified by the -u argument, usrpath. If the -u flag was not specified, but the -s flag was specified in the notification method field of the SRCnotify object, the directory to search is specified by the -s argument, syspath. Otherwise, the directory to search defaults to the one in which dae_notify is installed.

3. It looks for an entry in the AIX error log that describes the abnormal termination of the failing subsystem. Included in this entry is the exit status of the subsystem. The program obtains the sequence number of the found AIX error log entry, and the exit status of the subsystem.
4. It looks for a "stage 1" user exit program associated with the failing subsystem, or the group to which the failing subsystem belongs. The directory to be searched was determined in Step 2. First, the program looks in the directory for an executable file named <subsys_name>.userexit1. If this file is found, it is executed, and dae_notify proceeds to the next step. If the file is not found, and the group_name argument was provided, the program looks for an executable file named <group_name>.userexit1. If this file is found, it is executed.
5. It looks for a subsystem-supplied notification program associated with the failing subsystem, or the group to which the failing subsystem belongs. The directory to be searched was determined in Step 1. First, the program looks in the directory for an executable file named <subsys_name>.sysaction. If this file is found, it is executed, and dae_notify proceeds to the next step. If the file is not found, and the group_name argument was provided, the program looks for an executable file named <group_name>.sysaction. If this file is found, it is executed.
6. It looks for a "stage 2" user exit program associated with the failing subsystem, or the group to which the failing subsystem belongs. The directory to be searched was determined in Step 2. First, the program looks in the directory for an executable file named <subsys_name>.userexit2. If this file is found, it is executed, and dae_notify proceeds to the next step. If the file is not found, and the group_name argument was provided, the program looks for an executable file named <group_name>.userexit2. If this file is found, it is executed.
7. It exits.

The arguments passed to the two user exit programs and the subsystem-supplied notification program are identical. These programs are supplied with the following arguments:

subsys_name
: The name of the subsystem that has failed.

group_name
: The name of the group to which the failed subsystem belongs. If the subsystem is not defined to be a member of a group, this argument is a null string.

log_seqno
: The sequence number of the AIX error log entry which describes the abnormal termination of the subsystem. This argument can be used when executing dae_msg or errpt. If the AIX error log entry could not be found, this argument is a null string.

subsys_status
: The exit status of the failing subsystem. This argument can be used when executing dae_msg or dae_status. If the AIX error log entry describing the abnormal termination of the subsystem could not be found, this argument is a null string.

notify_rc

The current value of the notification method return value. This is initialized to 0 when the dae_notify notification method is started. The values returned by the user exit programs and the subsystem-supplied notification program update this value. When dae_notify exits, the value of the notification method return value is used as the exit value.

If a user exit program or the subsystem-supplied notification program does not detect any errors, it should exit with the value supplied by the notify_rc argument. If an error is detected, it can exit with a different value.

If a subsystem-supplied notification program's behavior will depend on the value of the notify_rc argument, the user documentation should describe how the behavior depends on the value. A system administrator would use this documentation when coding the "stage 1" user exit program. The exit value from the "stage 1" user exit program can then be used to influence the behavior of the subsystem-supplied notification program.

If a subsystem-supplied notification program may terminate with an exit value other than the value received through notify_rc, the user documentation should document the possible exit values. A system administrator would use this documentation when coding the "stage 2" user exit program. The value of the notify_rc argument received by the "stage 2" user exit program can then be used to influence its behavior.

It is assumed that the user exit programs will typically be used to communicate the failure of the subsystem to someone. A program is provided, dae_msg, that may be called from the user exit programs. The dae_msg program may be given as arguments the name of the failing subsystem, the name of the group to which the failing subsystem belongs, the sequence number of the AIX error log entry describing the abnormal termination of the failing subsystem, and the exit status of the failing subsystem. It can generate a short or long message describing why the subsystem terminated. Whether a short or long message is generated is determined by the flags specified when dae_msg is executed. The short message is a textual interpretation of the subsystem exit status. The long message includes the short message, and the text from the AIX error log.

EXAMPLES

Refer to 11.2, "Examples of Using the SRC Notification Support Programs" on page 296.

EXIT STATUS

0 - Ran to completion successfully.

>0 - An error occurred.

LOCATION

Refer to 11.3, "Building and Shipping the SRC Notification Support Programs" on page 299 for information about the install directory of the SRC notification support programs.

RELATED INFORMATION

The following man page sections in this appendix are related:

B.2, “dae_msg(1)” on page 390
B.3, “dae_status(1)” on page 392

B.2 dae_msg(1)

NAME

dae_msg - Display a short or long message describing why the subsystem terminated.

SYNOPSIS

```
dae_msg [-s|-l] subsys_name [group_name] [log_seqno subsys_status]
```

ARGUMENTS

-s	Generate a short message. This is the default if neither -s nor -l is specified.
-l	Generate a long message.
subsys_name	The name of the subsystem that has failed.
group_name	The name of the group to which the failed subsystem belongs. This argument is optional.
log_seqno	The sequence number of the AIX error log entry which describes the abnormal termination of the subsystem. If the AIX error log entry could not be found, this argument should be a null string. If the AIX error log entry has not been searched for yet, this argument should not be specified.
subsys_status	The exit status of the failed subsystem. If the AIX error log entry describing the abnormal termination of the subsystem could not be found, this argument should be a null string. If the AIX error log entry has not been searched for yet, this argument should not be specified.

DESCRIPTION

The dae_msg program generates a short or long message describing why a failed subsystem terminated. Whether a short or long message is generated is determined by the flags specified when dae_msg is executed. The short message is a textual interpretation of the subsystem exit status. The long message includes the short message and the text from the AIX error log.

The message generated is written to standard output. This can be piped to another program to deliver the message to the appropriate individual.

NOTES

The examples in this man page assume that the SRC notification support programs are installed in the /usr/lpp/ssp/bin directory. Refer to 11.3, “Building and Shipping the SRC Notification Support Programs” on page 299 for information about the install directory of the SRC notification support programs.

EXAMPLES

The following command can be used in a user exit program called by dae_notify to send a long message to the root user describing why the failed subsystem terminated:

```
/usr/lpp/ssp/bin/dae_msg -l "$1" "$2" "$3" "$4" | \
   /usr/bin/mail -s "$1 failed!" root
```

If the dae_notify generic SRC notification program is not used as the notification method for a subsystem, the subsystem's notification method can send a long message to the root user describing why the failed subsystem terminated with the following command:

```
/usr/lpp/ssp/bin/dae_msg -l "$@" | \
   /usr/bin/mail -s "$1 failed!" root
```

Refer to 11.2, "Examples of Using the SRC Notification Support Programs" on page 296.

EXIT STATUS

0 - Ran to completion successfully.

>0 - An error occurred.

LOCATION

Refer to 11.3, "Building and Shipping the SRC Notification Support Programs" on page 299 for information about the install directory of the SRC notification support programs.

RELATED INFORMATION

The following man page sections in this appendix are related:

B.1, "dae_notify(1)" on page 386
B.3, "dae_status(1)" on page 392

B.3 dae_status(1)

NAME

dae_status - Interpret a process status value.

SYNOPSIS

```
dae_status {-e|-k|-s|-c} subsys_status
```

ARGUMENTS

-e	Determine if the subsystem exited; if it did, display the exit value.
-k	Determine if the subsystem terminated due to the receipt of a signal; if it did, display the number of the signal that caused the subsystem to terminate.
-s	Determine if the subsystem was stopped by the receipt of a signal; if it was, display the number of the signal that stopped the subsystem.
-c	Determine if the subsystem generated a core file.
subsys_status	The status value of the failed subsystem, represented as a decimal, hexadecimal, or octal number.

DESCRIPTION

The dae_status program is given the status value of a failed subsystem, and interprets it. This program is used by the dae_msg program.

If the -e flag is specified, dae_status will determine if the subsystem called _exit to terminate. If it did, dae_status returns a value of 0, and writes to standard output the value specified by the subsystem on the call to _exit. This number is printed as an unsigned decimal number. If the status value indicates the subsystem did not call _exit to terminate, dae_status returns a value of 1. Recall that when a C program calls exit or returns from the main routine, the program calls _exit.

If the -k flag is specified, dae_status will determine if the subsystem terminated due to the receipt of a signal. If it did, dae_status returns a value of 0, and writes to standard output the number of the signal that terminated the subsystem. This number is printed as an unsigned decimal number. If the status value indicates the subsystem did not terminate due to the receipt of a signal, dae_status returns a value of 1.

If the -s flag is specified, dae_status will determine if the subsystem stopped due to the receipt of a signal. If it did, dae_status returns a value of 0, and writes to standard output the number of the signal that stopped the subsystem. This number is printed as an unsigned decimal number. If the status value indicates the subsystem did not stop due to the receipt of a signal, dae_status returns a value of 1. It is not expected that an SRC notification method will ever be executed when a subsystem is stopped. The -s flag is included in dae_status to allow the program to interpret any legitimate status value.

If the -c flag is specified, dae_status will determine if the subsystem generated a core file. If it did, dae_status returns a value of 0. If the status value indicates the subsystem did not generate a core file, dae_status returns a value of 1.

NOTES

The examples in this man page assume that the SRC notification support programs are installed in the /usr/lpp/ssp/bin directory. Refer to 11.3, "Building and Shipping the SRC Notification Support Programs" on page 299 for information about the install directory of the SRC notification support programs.

EXAMPLES

The dae_msg Korn shell script includes code similar to the following to interpret the subsystem status value:

```
pathname=/usr/lpp/ssp/bin
subsystem=$1
subsys_status=$4

if   value=$(${pathname}/dae_status -e ${subsys_status}); then

     print -n "Subsystem \"${subsystem}\" terminated abnormally;"
     print -n " subsystem exited with value ${value}.\n"

elif value=$(${pathname}/dae_status -k ${subsys_status}); then

     print -n "Subsystem \"${subsystem}\" terminated abnormally;"
     print -n " subsystem terminated by signal ${value}.\n"
     if ${pathname}/dae_status -c ${subsys_status}; then
          print "Core file created."
     fi

elif value=$(${pathname}/dae_status -s ${subsys_status}); then

     print "Subsystem \"${subsystem}\" stopped by signal ${value}."

else

     print -n "Subsystem \"${subsystem}\" terminated abnormally;"
     print -n " cause unknown."
fi
```

EXIT STATUS

See the DESCRIPTION section.

LOCATION

Refer to 11.3, "Building and Shipping the SRC Notification Support Programs" on page 299 for information about the install directory of the SRC notification support programs.

RELATED INFORMATION

The following man page sections in this appendix are related:

B.1, “dae_notify(1)” on page 386
B.2, “dae_msg(1)” on page 390

Appendix C. Loading the Diskette

A diskette is included with this redbook. The diskette includes:

- The text of all the figures in the redbook.
- The source code for the daemon support routines described in Chapter 10, “Daemon Support Routines” on page 245.
- The source code for the programs described in Chapter 11, “SRC Notification Support Programs” on page 295.
- The source code for a program that can be used as a model to build commands that send subsystem-defined requests to SRC-controlled daemons. The use of this program is described in Chapter 12, “New SRC Program Support” on page 303.

To load the contents of the diskette onto your system, follow these steps:

1. Insert the diskette into a diskette drive; the /dev/rfd0 drive, for example.
2. From a shell prompt, change the current working directory to the directory where you want the diskette contents loaded. For example:

```
$ cd /u/myuserid/software
```

3. List the contents of the diskette to verify the correct diskette is in the diskette drive. For example:

```
$ tar -tf /dev/rfd0

dae/
dae/figures/
dae/figures/figure_001
dae/figures/figure_002
dae/figures/figure_003
dae/figures/figure_004
   .
   .
   .
dae/figures/README
dae/src/
dae/src/SRC_notify/
dae/src/SRC_notify/makefile
dae/src/SRC_notify/README
dae/src/SRC_notify/dae_date.c
dae/src/SRC_notify/dae_status.c
dae/src/SRC_notify/dae_msg.sh
dae/src/SRC_notify/dae_finderr.sh
dae/src/SRC_notify/dae_notify.sh
dae/src/SRC_cmd_model/
dae/src/SRC_cmd_model/dae_SRCmodel.c
dae/src/SRC_cmd_model/README
dae/src/libdae_bsd/
dae/src/libdae_bsd/makefile
dae/src/libdae_bsd/make_lib
dae/src/libdae_bsd/README
dae/src/libdae/
dae/src/libdae/dae_error.c
dae/src/libdae/dae_SRCout.c
dae/src/libdae/dae_global.h
dae/src/libdae/dae_psalloc.c
dae/src/libdae/dae.h
dae/src/libdae/dae_SRC.c
dae/src/libdae/dae_err.h
dae/src/libdae/dae_excl.c
dae/src/libdae/dae_init.c
dae/src/libdae/dae_misc.c
dae/src/libdae/dae_std.h
dae/src/libdae/dae_getprocs.c
dae/src/libdae/makefile
dae/src/libdae/daeshr.exp
dae/src/libdae/make_lib
dae/src/libdae/README
dae/src/README
dae/README
```

4. Load the contents of the diskette into the current working directory. In the following example, the -o flag causes the extracted files to be assigned the user and group identifier of the user running the tar command:

```
$ tar -xvof /dev/rfd0

x dae
x dae/figures
x dae/figures/figure_001, 60 bytes, 1 tape blocks
x dae/figures/figure_002, 725 bytes, 2 tape blocks
x dae/figures/figure_003, 229 bytes, 1 tape blocks
x dae/figures/figure_004, 1547 bytes, 4 tape blocks
    .
    .
    .
x dae/figures/README, 172 bytes, 1 tape blocks
x dae/src
x dae/src/SRC_notify
x dae/src/SRC_notify/makefile, 3521 bytes, 7 tape blocks
x dae/src/SRC_notify/README, 490 bytes, 1 tape blocks
x dae/src/SRC_notify/dae_date.c, 3713 bytes, 8 tape blocks
x dae/src/SRC_notify/dae_status.c, 6883 bytes, 14 tape blocks
x dae/src/SRC_notify/dae_msg.sh, 6927 bytes, 14 tape blocks
x dae/src/SRC_notify/dae_finderr.sh, 15756 bytes, 31 tape blocks
x dae/src/SRC_notify/dae_notify.sh, 6806 bytes, 14 tape blocks
x dae/src/SRC_cmd_model
x dae/src/SRC_cmd_model/dae_SRCmodel.c, 8284 bytes, 17 tape blocks
x dae/src/SRC_cmd_model/README, 407 bytes, 1 tape blocks
x dae/src/libdae_bsd
x dae/src/libdae_bsd/makefile, 1672 bytes, 4 tape blocks
x dae/src/libdae_bsd/make_lib, 438 bytes, 1 tape blocks
x dae/src/libdae_bsd/README, 666 bytes, 2 tape blocks
x dae/src/libdae
x dae/src/libdae/dae_error.c, 9022 bytes, 18 tape blocks
x dae/src/libdae/dae_SRCout.c, 70362 bytes, 138 tape blocks
x dae/src/libdae/dae_global.h, 13507 bytes, 27 tape blocks
x dae/src/libdae/dae_psalloc.c, 18989 bytes, 38 tape blocks
x dae/src/libdae/dae.h, 11566 bytes, 23 tape blocks
x dae/src/libdae/dae_SRC.c, 42698 bytes, 84 tape blocks
x dae/src/libdae/dae_err.h, 5251 bytes, 11 tape blocks
x dae/src/libdae/dae_excl.c, 8501 bytes, 17 tape blocks
x dae/src/libdae/dae_init.c, 61277 bytes, 120 tape blocks
x dae/src/libdae/dae_misc.c, 8887 bytes, 18 tape blocks
x dae/src/libdae/dae_std.h, 13776 bytes, 27 tape blocks
x dae/src/libdae/dae_getprocs.c, 10199 bytes, 20 tape blocks
x dae/src/libdae/makefile, 1577 bytes, 4 tape blocks
x dae/src/libdae/daeshr.exp, 369 bytes, 1 tape blocks
x dae/src/libdae/make_lib, 438 bytes, 1 tape blocks
x dae/src/libdae/README, 649 bytes, 2 tape blocks
x dae/src/README, 517 bytes, 2 tape blocks
x dae/README, 364 bytes, 1 tape blocks
```

5. A subdirectory, dae, has been created in the current working directory. The contents of the diskette have been copied into that subdirectory.

Appendix D. Special Notices

This publication is intended to help administrators, program designers, and daemon writers who need to originate or understand daemon logic. The information in this publication is not intended as the specification of any programming interfaces that are provided by AIX (5765-393). See the PUBLICATIONS section of the IBM Programming Announcement for AIX (5765-393) for more information about what publications are considered to be product documentation.

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 program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

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 the 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, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

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

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX	IBM
RS/6000	RS/6000 SP

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Appendix E. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

UNIX Network Programming, W. Richard Stevens, 1990, Prentice Hall, Englewood Cliffs

Advanced Programming in the UNIX Environment, W. Richard Stevens, 1992, Addison-Wesley Publishing Company

AIX Version 4.1: Files Reference, SC23-2512

AIX Version 4.1: General Programming Concepts: Writing and Debugging Programs, SC23-2533

AIX Version 4.1: Command Reference, SBOF-1851

AIX Version 4.1: System Management Guide: Operating System and Devices, SC23-2525

AIX Version 4.1: Technical Reference, Volumes 1: Base Operating System and Extensions, SC23-2614

AIX Version 4.1: Technical Reference, Volume 2: Base Operating System and Extensions, SC23-2615

X/Open CAE Specification, System Interfaces and Headers Issue 4, Version 2 , X/Open Company Ltd.

E.1 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RISC System/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RISC System/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
Application Development Redbooks Collection	SBOF-7290	SK2T-8037
Personal Systems Redbooks Collection (available soon)	SBOF-7250	SK2T-8042

How To Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at URL http://www.redbooks.ibm.com.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** — to order hardcopies in United States
- **GOPHER link to the Internet** - type GOPHER.WTSCPOK.ITSO.IBM.COM
- **Tools disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get lists of redbooks:

  ```
  TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
  ```

  To register for information on workshops, residencies, and redbooks:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  ```

  For a list of product area specialists in the ITSO:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
  ```

- **Redbooks Home Page on the World Wide Web**

  http://w3.itso.ibm.com/redbooks

- **IBM Direct Publications Catalog on the World Wide Web**

  http://www.elink.ibmlink.ibm.com/pbl/pbl

  IBM employees may obtain LIST3820s of redbooks from this page.

- **ITSO4USA category on INEWS**
- **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**

  With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** (Do not send credit card information over the Internet) — send orders to:

	IBMMAIL	**Internet**
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** — send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** — send orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1) 415 855 43 29 (Outside USA)** — ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **Direct Services** - send note to softwareshop@vnet.ibm.com
- **On the World Wide Web**

Redbooks Home Page	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Internet Listserver**

  With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank).

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

- **Please put me on the mailing list for updated versions of the IBM Redbook Catalog.**

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

- Invoice to customer number

- Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

DO NOT SEND CREDIT CARD INFORMATION OVER THE INTERNET.

List of Abbreviations

ADE	AIX 4.1 Development Environment (internal environment)
AIX	advanced interactive executive (IBM′s flavor of UNIX)
ANSI	American National Standards Institute
ASCII	American National Standard Code for Information Interchange
AT&T	American Telephone & Telegraph Company
BSD	Berkeley software distribution (UC at Berkeley, UNIX)
C	UNIX system-programming language
CD-ROM	(optically read) compact disk - read only memory
FTP	file transfer program
FTP	file transfer protocol
I/O	input/output
IBM	International Business Machines Corporation
Internet	A worldwide network of TCP/IP-based networks
IP	internet protocol (ISO)
IP	internetwork protocol (OSI)
IPC	inter-processor communication
ITSO	International Technical Support Organization
LPP	licensed program product
NFS	network file system (USA, Sun Microsystems Inc)
ODM	object data manager (AIX)
PID	process identifier/identification
POSIX	portable operating system interface for computer environments, an IEEE operating system standard, closely related to the UNIX system (software writing)
PSSP	IBM Parallel System Support Programs for AIX
PTF	program temporary fix
RISC	reduced instruction set computer/cycles
RS/6000 SP	IBM RS/6000 Scalable POWERparallel Systems
SRC	system resource controller
TCP	transmission control protocol (USA, DoD)
TCP/IP	Transmission Control Protocol/Internet Protocol (USA, DoD, ARPANET; TCP=layer 4, IP=layer 3, UNIX-ish/Ethernet-based system-interconnect protocol)
TTY	teletypewriter
UDP	user datagram protocol (USA, DoD)
UDP/IP	User Datagram Protocol/Internet Protocol (USA, DoD, ARPANET; TCP=layer 4, IP=layer 3,
UID	user identification
UNIX	an operating system developed at Bell Laboratories (trademark of UNIX System Laboratories, licensed exclusively by X/Open Company, Ltd.)
URL	Universal Resource Locator
X	hexadecimal (as in 5Fx)
X/OPEN	a group (Bull, DEC, Ericsson, IBM, ICL, Olivetti, Philips, Siemens-Nixdorf, Unisys) to create a standard interface between software packages and UNIX (portability)

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Printed in U.S.A.