Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL-00

PEXEC
Reference Manual

Draft

by

William Gropp
Ewing Lusk
Mathematics and Computer Science Division
Argonne National Laboratory

January 19, 1996

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

Abstract

Writing cooperative applications (processes) that communicate with each other is a difficult task. One major problem is the interaction between logically unrelated parts of a collective application, such as attaching to new tasks and communicating with existing ones. The PEXEC package uses event-driven techniques, along with data encapsulation and function composition ideas, to provide a simple and robust framework in which to write cooperative applications.

1 Introduction

PEXEC is a system for writing lightweight, fault-tolerant, client/server programs. It is organized around an event-driven model, and provides for highly modular definition of different services to be provided by a program. This document contains detailed documentation on the routines that are part of the PEXEC implementation. These include the basic event-driven driver routines, as well as a variety of convenience routines for creating and using network sockets and child processes.

As an alternate to this manual, the reader should consider using the script `pxman`; this is a script that uses `xman` to provide a X11 Window System interface to the data in this manual.

This manual is organized into four sections. Section 2 describes the design and basic functionality. Section 3 discusses how to write handlers. Section 6 shows example programs, and Section 7 provides manual pages for all of the routines.

2 Overview

In a cooperative application, requests for services (e.g., start a new process, send data to user, answer question) come from a variety of sources. When a request arrives, it should be handled quickly. The usual method for handling this situation is to use an event-driven system, where the code looks something like

```
while (1) {
    wait_for_an_event;
    process_that_event;
    }
```

It is important that the `wait_for_an_event` code not consume CPU time polling for events. This can be difficult to accomplish if several different services each wants to implement its own `wait_for_an_event` routine. For example, you might get something like

```
while (1) {
    if (poll_for_an_event_1) process_event_1;
    if (poll_for_an_event_2) process_event_2;
    }
```

This is undesirable.

Some systems attempt to deal with this by allowing "other" systems to inform them of events to watch for and routines to call for these events. For example, in the X11 Window System, the call `XtAppAddInput` allows an X11 application to specify a file descriptor and routine to call for activity (read, write, or exception) on that file descriptor. Such a facility is useful, but does not allow two systems to cooperate. For example, if another package Y also provided a similar interface, for example, `YAddAppInput`, there would be no way to combine X11 and Y, since each wanted the other to specify the what events to watch for. Systems striving for maximum portability have little choice, since there is no completely system independent way to specify events. PEXEC is limited to Unix, however, where there is a standard way to specify events. In particular, events (having to do with interaction with other processes) in Unix fall primarily into two classes:

- Interaction with other processes, devices, and files is primarily through file descriptors, and the `select` call provides an efficient, non-blocking way to detect activity, and

- Interaction with child processes requires, in addition, handling of `SIGCHLD` signals.

This suggests that the event loop be simply

```
while (1) {
     select( ... );
     process_events;
     }
```

Each application service (or *applet*) needs to specify the file descriptors that it is using and the way in which events on these file descriptors are handled.

By constructing an application to be event driven, with `select` being used as the inner most call, the application can respond quickly to a wide variety of possible events, including local and remote network connections, and local processes.

By making the `select` loop the outer loop, each applet is on a common footing with all other applets. No particular applet is the "controlling" application. All that an applet needs to do is to export the file descriptors that it uses and let the `select` loop know how to handle them. These routines are called *event handlers* or just *handlers*

Composing event handlers In the simplest situation, all of the activity on a file descriptor is intended for a single event handler. For example, the handler may read data as a command and then perform an action.

In a more complex situation, the activity on a file descriptor is processed by a finite state machine; that is, a well-defined sequence of operations where the choice of the next operation to use depends on the current operation and the event (usually the data read). For example, consider passing message data in a secure environment, where one routine reads the message header or envelope, and another (using a different decryption routine) reads the message data. In this system, the finite state machine oscillates between a handler that can read the message envelopes and one that can read the message data. Other examples include having handlers that merely record the fact of the activity; these can be useful in debugging or performance monitoring.

To meet this need, with each file descriptor (fd) is associated a list (or *chain*) of handlers that are called when there is activity on that fd. This technique is sometimes called function composition.

Fault tolerance One of the events that can be detected is an error or end-of-file. Each of these events has a separate handler (this simplifies writing the handlers for the case of no error).

An important event is the *absence* of an event within a certain amount of time. For example, a message or data may be expected within five minutes; if the message does not arrive, that is an event to be handled (as an error). For this situation, for each file descriptor there is an optional timeout; the error handler is called if the timeout is reached.

Composing read and write functions In processing the data the moves through the file descriptors, it is useful to be able to call special functions to process the data. For example, the data may be encrypted or packaged in some way. In PEXEC, the functions to read and write from a file dscriptor may be specified by the user; PEXEC provides simple versions that are just the regular Unix `read` and `write`.

In some cases, the processing of the data may need to be handled by several separate modules. For example, a debugging module may monitor the amount of data moved, while an encryption module might encrypt and decrypt all data. In order to allow these to function independently, the read and write functions can be composed, just as the handlers can.

This approach also provides a convienient way to convert code that is based on blocking operations into the more robust, event driven code used by PEXEC. This is described in Section 5.

Summary The basic idea in the PEXEC package is to lift the `select` loop that many applications use out of the applications so that different "applets" can share the same loop in a modular and efficient way. This requires providing a common interface to specify what happens when the `select` calls finds activity on a file descriptor; this is done by specifying the handler functions. Fault tolerance is provided by having separate functions provided for regular data, end-of-file, and other errors (note that no particular *policy* for handling faults is dictated, just a *mechanism*).

The routines for performing these functions in PEXEC are organized into four classes: The basic routines provide the `select` loop and the code to manage the information needed by the `select` loop. The handler management routines create and free the event handlers. The file descriptor management functions handle associating handlers with a file descriptor and installing it in the set of fds known to the `select` loop. The handler service functions provide some simple handlers that can be used in applications.

Because this approach is based on processing events on file descriptors (fds), and since creating fds for interprocess communication is common yet tricky task, the PEXEC package contains a number of routines that may be used to create fds. Note that these routines are *only* creating fds, and can be used independently from the rest of the PEXEC package. These routines are described under file descriptor service functions. One other way in which file descriptors are created is when a new process is created with `stdin`, `stdout`, and `stderr` file descriptors. Functions to handle creating this process (and handling the exit of that process) are in the process management section.

Finally, there is a collection of routines that make use of the PEXEC package to provide some useful functions, such as tracing all activity and logging the behavior of the package. These are in the miscellaneous section.

These sections are summarized below:

Basic Routines These routines create and manage the PEXEC context, and provide the basic `select` loop.

Handler Management These routines create and free handlers for file descriptors.

Handler Service Functions These routines provide useful handler functions for common operations.

File Descriptor Management These routines install and free handlers on specific file descriptors.

File Descriptor Service Functions These routines provide useful routines for acquiring file descriptors for common operations; in particular, a number of routines for interprocess communication over sockets are provided here.

Process Management These routines provide a simple process management capability.

Miscellaneous These routines provide functions such as debugging, logging, and tracing through use of persistent handlers.

3 Writing Handlers

Handlers are really very simple functions. The most common handler reads some data from a file descriptor (fd) with the `read` routine and then processes the data.

3.1 Stateless Handlers

For example, a simple handler that reads on an fd and writes what has been read to standard out is very simple and is presented below:

```
#define MAX_BUF 128
int PX_Hfcn_to_stdout( ctx, conn, handler, in_out )
PX_Ctx     *ctx;
```

```
PX_Conn    *conn;
PX_Handler handler;
int        in_out;
{
int n, m;
char buf[MAX_BUF];

n = read( conn->fd, buf, MAX_BUF );
if (n > 0) {
    m = write( 1, buf, n );
    if (m != n)
return -1;
    }
else
    return PX_Hfcn_do_eof( ctx, conn, handler, n );

return 0;
}
```

The things to note here are:

- The file descriptor to use is `conn->fd`.
- Errors in reading are handled by `PX_Hfcn_do_eof`, a handler service function.
- Care is taken on the write to check for errors (though the error isn't handled very well).

Rules for handlers:

1. A handler must never block.
2. A handler should not take much time. If it might, the task that it is performing should be performed by another process (thread, if available and appropriate).
3. Handlers must check for errors
4. A handler for reading needs to check for end-of-file. [1]

Robustness Issues The example above is simple but is not as robust as it should be. For example, the `read` might be interrupted by a signal, which will cause it to return with the error `EINTR` (interrupted system call). This is a (usually) benign error, and many codes simply use something like

```
do {
    n = read( ... );
    } while (n == -1 && errno == EINTR);
```

The same applies to the `write` statement.

Another more subtle problem is that the `write` could write some but not all of the data. This requires that the handler be prepared to save some of the data for later writing (and this requires a handler for the affected write file descriptor).

[1] Unix System V users will find this very difficult because end-of-file and no-data-available are indistinguishable. Fortunately, POSIX specifies reasonable behavior and most System V systems support POSIX as well.

3.2 Handlers with state

In many cases, the handler needs to keep some state. For example, a handler that reads input and then processes a command once a newline character is seen needs to save the data until a newline is actually read. This may require multiple `read` calls, so the data needs to be saved by the handler. It can do this in the `extra_data` field which is saved in the `handler` structure and was initialized by the `PX_Handler_simple_create` routine. For example, this code is used to read a command from the specified fd; the data is stored in a small buffer

```
#define MAX_BUF 128
typedef struct {
    int  curlen;
    char buf[MAX_BUF+1];
    } CommandBuf;
int ReadCommand( ctx, conn, handler, in_out )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_out;
{
int        n;
CommandBuf *cbuf = (CommandBuf *)handler->extra_data;

/* Read a new command */
if (cbuf->curlen == MAX_BUF) {
    cbuf->buf[MAX_BUF] = 0;
    fprintf( stderr, "Discarding command %s\n", cbuf->buf );
    cbuf->curlen = 0;
    }

n = read( conn->fd, cbuf->buf+cbuf->curlen, MAX_BUF-cbuf->curlen );
if (n <= 0)
    return PX_Hfcn_do_eof( ctx, conn, handler, n );

/* Actually read something */
cbuf->curlen += n;
cbuf->buf[cbuf->curlen] = 0;

/* Check for newline; process if found,
   otherwise return to PX_Gselect */
...
return 0;
}
```

The `extra_data` field is initialized with

```
void *InitReadCommand( ctx )
PX_Ctx *ctx;
{
CommandBuf *b = (CommandBuf *)malloc( sizeof(CommandBuf) );
b->curlen = 0;
return (void *)b;
}
```

The `destroy` function for this handler can free the `CommandBuf` structure:

```
int FreeCommand( handler )
```

```
PX_Handler handler;
{
CommandBuf *b = (CommandBuf *)handler->extra_data;
if (b) free(b);
return 0;
}
```

3.3 The info function

Each handler has three functions that it may call. Two of these have been discussed above: the function for data and the function for freeing the handler. The third, the `info` function, provides a way for a handler to tell something about itself. This is useful for providing debugging or performance monitoring functions, for example.

The exact nature of the `info` function has not yet been defined.

4 Other approaches and design rationale

PEXEC makes use of a event-driven programming style that makes it possible to write responsive programs. There are other approaches; this section discusses some of them and explains the choices that were made in PEXEC.

4.1 Threads

One way to achieve responsiveness is to use separate *threads*, as long as no system call (like `read` or `write`) in a thread blocks all threads in the process. This approach can be used in C or C++; in newer languages like JAVA, it is the only way to achieve reponsiveness.

An advantage of the thread approach is that the operating system takes care of insuring that `read` and `write` (and other) operations don't block all of the activities. This makes it somewhat easier to write applications, but does require more support from the operating system (not all systems provide the kind of thread support that this needs). In addition, a thread provides more functionality than is needed here, and that usually has a performance penalty. In fact, some systems that were originally built with many separate threads are now looking for ways to reduce the number of threads.

PEXEC is itself thread-safe, so that it can be used in a threaded environment. In addition, the `PX_Write_nb` and `PX_Read_nb` routines provide a way to write more conventional (straight line) code. One can view PEXEC in this form as providing a simple by highly efficient multi-threaded system, with the additional advantage that the the threads can interact in structured way.

4.2 Message-passing

Message-passing systems, particularly ones with dynamic process management, also appear to provide the functionality offered by PEXEC. Unfortunately, most message-passing environment lack two critical features: message-passing operations (particularly `send`) can block and there is no way to wait for either a message or input from a file, terminal, or socket without consuming CPU cycles in a spin loop. Where nonblocking operations are supported, it usually isn't possible to wait for a combination of messages and other operations (such as input from standard input). Finally, most message-passing systems are single-user; PEXEC can support multiple users with a single process.

5 Writing nonblocking applications

In many applications, the most natural way to write the code is as a series of blocking `read` and `write` operations. For example, when exchanging data with another process, the most natural code would wait until a complete message had been received, and then write back the entire data requested. This code might look something like this:

```
nread = 0;
while (nread < sizeof(msg)) {
    n = read( fd, buf + nread, sizeof(msg)-nread );
    ... check for errors ...
    nread += n;
    }
... form answer in wbuf
nsent = 0;
while (nsent < sizeof(smsg)) {
    n = write( fd, buf + nsent, sizeof(smsg)-nsent );
    ... check for errors ...
    nsent += n;
    }
... do some more work (update tables, etc)
```

Unfortunately, this code could cause a program to loop forever in one of the read or write loops if for some reason the operation can not be completed. Of course, this won't happen in the usual cases, but PEXEC is designed to let programs work even in the most hostile cases. To do that, we need to avoid ever writing code that contains potentially infinite loops like that above.

What we need to do is to start the read but then (if we don't get enough data) return to the `select` loop with a handler that will complete our computation as soon as all of the data becomes available. This can be done with a handler, but because it is a common operation, PEXEC provides special support for code that reads or writes data of fixed length and then continues with a computation. In the example above, we create two functions: **(this section is still being worked on)**

```
int fcn1( PX_Conn *conn, void *buf, int n, void *extra_data )
{
/* buf is full (if n > 0; check for error otherwise) */
... form answer in wbuf
return PX_Write_nb( conn, wbuf, sizeof(smsg), fcn2, (void *)0 );
}
int fcn2( PX_Conn *conn, void *buf, int n, void *extra_data )
{
/* buf is the data sent; n < 0 for error */
... do some more work (update tables, etc)
}
```

and then replace the above code with

```
return PX_Read_nb( conn, buf, sizeof(msg), fcn1, (void *)0 );
```

This lets us organize our algorithm as a sequence of blocking steps, yet still allow the `select` loop to insure that the code that is executed after each `read` or `write` operation is activated only when the I/O operation is complete.

notes: still need to decide on error handler and destroy handler, particularly for the read/write buffer.

6 Example applications

This section provides a few complete examples of programs that use PEXEC. All of the source code may be found in the `pexec/examples` directory.

6.1 Hello world

The first application is a distributed "hello world" program. The server waits for connections on a local socket and echoes whatever it reads there to standard output. The client simply opens the local socket, writes `hello world` to the socket, and exits.

Note that the client uses only the file descriptor service routines.

First, the client (`helloc.c`:

```
#include "pexec.h"

int main( argc, argv )
int argc;
char **argv;
{
    int fd;

    fd = PX_FD_Create_local_conn( "/tmp/hellod", 1, (int *)0 );
    write( fd, "Hello World\n", 12 );
    return 0;
}
```

There is very little to note here other than that the PEXEC service routines can be used independently of the `select` loop.

Next, the server (`hellod.c`:

```
#include "pexec.h"
#include <stdio.h>
#include <errno.h>

#define STDIN 0

int GetConnection ANSI_ARGS((PX_Ctx *, PX_Conn *, PX_Handler, int )),
    UnlinkLocal ANSI_ARGS((PX_Ctx *, PX_Conn *, PX_Handler, int ));

int main( argc, argv )
int  argc;
char **argv;
{
    PX_Ctx     ctx;
    PX_Handler exit_on_close, err_fd, to_stdout;
    PX_Handler unlinklocal, getconnection;
    int        fd;
    void       *sock_info;

    PX_Ctx_init( &ctx );

    /* Echo stdin to stdout; closeing stdin causes program to exit */
    exit_on_close = PX_Handler_simple_create( PX_Hfcn_exit_on_close,
                                              (void *)0 );
    err_fd        = PX_Handler_simple_create( PX_Hfcn_error_fd, (void *)0 );
    to_stdout     = PX_Handler_simple_create( PX_Hfcn_to_stdout, (void *)0 );

    PX_FD_add( STDIN,  PX_READ, to_stdout, err_fd, exit_on_close,
          (struct timeval *)0, &ctx );

    /* Add a local socket */
    fd = PX_FD_Create_local_socket( "/tmp/hellod", 1, &sock_info );
    if (fd < 0) {
        fprintf( stderr, "Error creating local socket\n" );
        }
    else {
        getconnection = PX_Handler_create( GetConnection, sock_info,
```

```
                                          (int (*)())0,
                                         (void *)0, UnlinkLocal );
        unlinklocal   = PX_Handler_simple_create( UnlinkLocal, (void *)0 );
        PX_FD_add( fd, PX_READ, getconnection, unlinklocal, unlinklocal,
                  (struct timeval *)0, &ctx );
        PX_Handler_free( unlinklocal );
        PX_Handler_free( getconnection );
        }

    /* gselect is basically an infinite loop */
    if (PX_Gselect( &ctx, (void *)0 )) {
        perror( "Gselect failed" );
        return 1;
        }
    return 0;
}

/* This is called when a listen allows an accept */
int GetConnection( ctx, conn, handler, in_mask )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_mask;
{
    int fd;
    PX_Handler err_fd, close_fd, fd_to_set;

    fd = PX_FD_Accept_connection( conn->fd, handler->extra_data );
    if (fd < 0) {
        fprintf( stderr, "accept errno = %d\n", errno );
        perror( "Accept failed" );
        return fd;
        }

    close_fd      = PX_Handler_simple_create( PX_Hfcn_close_fd, (void *)0 );
    fd_to_set     = PX_Handler_simple_create( PX_Hfcn_to_stdout, (void *)0 );
    err_fd        = PX_Handler_simple_create( PX_Hfcn_error_fd, (void *)0 );
    PX_FD_add( fd, PX_READ, fd_to_set, err_fd, close_fd, (struct timeval *)0,
              ctx );
    return 0;
}

int UnlinkLocal( ctx, conn, handler, err )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        err;
{
    unlink( "/tmp/hellod" );
    return 0;
}
```

This handles input from three different sources. The first is from standard input; primarily, this allows you to terminate the program cleanly by closing its `stdin`. The second source is the named socket `/tmp/hellod`; this allows other processes (on the same machine) to request a connection.

The third source is any fd added when a connection is requested. Note that there may be many simultaneously active connections.

6.2 Network Hello World

This is a version of the hello world program, but for networks.

First, the client (`nhelloc.c`:

```
#include "pexec.h"

int main( argc, argv )
int argc;
char **argv;
{
    int fd;

    fd = PX_FD_Create_network_conn( "godzilla.mcs.anl.gov", 3117, 1,
                                    (int *)0 );
    if (fd < 0) {
        perror( "Could not open connection" );
        return 1;
        }
    write( fd, "Hello World\n", 12 );
    return 0;
}
```

There is very little to note here other than that the PEXEC service routines can be used independently of the `select` loop. This is using a predetermined host/port pair; these could of course be passed in as arguments or read from standard input. This assumes that the server is running on host `godzilla.mcs.anl.gov`.

Next, the server (`nhellod.c`:

```
#include "pexec.h"
#include <stdio.h>
#include <errno.h>

#define STDIN 0

int GetConnection ANSI_ARGS((PX_Ctx *, PX_Conn *, PX_Handler, int ));

int main( argc, argv )
int  argc;
char **argv;
{
    PX_Ctx     ctx;
    PX_Handler exit_on_close, err_fd, to_stdout;
    PX_Handler getconnection;
    int        fd;
    void       *sock_info;

    PX_Ctx_init( &ctx );

    /* Echo stdin to stdout; closeing stdin causes program to exit */
    exit_on_close = PX_Handler_simple_create( PX_Hfcn_exit_on_close,
                                              (void *)0 );
    err_fd        = PX_Handler_simple_create( PX_Hfcn_error_fd, (void *)0 );
    to_stdout     = PX_Handler_simple_create( PX_Hfcn_to_stdout, (void *)0 );
```

```
    PX_FD_add( STDIN,  PX_READ, to_stdout, err_fd, exit_on_close,
          (struct timeval *)0, &ctx );

    /* Add a network socket */
    fd = PX_FD_Create_network_socket( (char *)0, 3117, 1, &sock_info );
    if (fd < 0) {
        fprintf( stderr, "Error creating network socket\n" );
        }
    else {
        getconnection = PX_Handler_simple_create( GetConnection, sock_info );
        PX_FD_add( fd, PX_READ, getconnection, (void *)0, (void *)0,
                  (struct timeval *)0, &ctx );
        PX_Handler_free( getconnection );
        }

    /* gselect is basically an infinite loop */
    if (PX_Gselect( &ctx, (void *)0 )) {
        perror( "Gselect failed" );
        return 1;
        }
    return 0;
}

/* This is called when a listen allows an accept */
int GetConnection( ctx, conn, handler, in_mask )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_mask;
{
    int fd;
    PX_Handler err_fd, close_fd, fd_to_set;

    fd = PX_FD_Accept_connection( conn->fd, handler->extra_data );
    if (fd < 0) {
        fprintf( stderr, "accept errno = %d\n", errno );
        perror( "Accept failed" );
        return fd;
        }

    close_fd      = PX_Handler_simple_create( PX_Hfcn_close_fd, (void *)0 );
    fd_to_set     = PX_Handler_simple_create( PX_Hfcn_to_stdout, (void *)0 );
    err_fd        = PX_Handler_simple_create( PX_Hfcn_error_fd, (void *)0 );
    PX_FD_add( fd, PX_READ, fd_to_set, err_fd, close_fd, (struct timeval *)0,
              ctx );
    return 0;
}
```

This is much like the local version, except that instead of a named socket, a hostname/port number pair are used to specify the socket. Note also that the routine `PX_FD_Create_network_socket` uses a null pointer as the name of the local machine; this allows the routine to determine the correct machine name where-ever it is run.

6.3 A two-way client/server application

Here is a simple client/server application. This makes use of a sample command interpreter in `examples/scommand.c` which is not part of the PEXEC library (but is included as source code).

First, the client:

```
/*
 * This program is a simple connect-to-local socket client.
 */

#include "pexec.h"
#include <stdio.h>

#define STDIN 0

int main( argc, argv )
int  argc;
char **argv;
{
PX_Ctx ctx;
int    fd;
PX_Handler close_fd, fd_to_set, err_fd, exit_on_close, fd_to_out;

PX_Ctx_init( &ctx );

fd = PX_FD_Create_local_conn( "/tmp/pexec", 1, (int *)0 );
if (fd < 0) {
    fprintf( stderr, "Could not open pexec port\n" );
    return 1;
    }

exit_on_close = PX_Handler_simple_create( PX_Hfcn_exit_on_close, (void *)0 );

/* Write to socket */
fd_to_set     = PX_Handler_simple_create( PX_Hfcn_fd_to_set, (void *)0 );
PX_FdAddToSet( fd_to_set, fd );

err_fd        = PX_Handler_simple_create( PX_Hfcn_error_fd, (void *)0 );

/* Setup an initial connection to stdin/out */
PX_FD_add( STDIN,  PX_READ, fd_to_set, err_fd, exit_on_close,
          (struct timeval *)0, &ctx );

/* If there is data on the socket, read it and write to OUR stdout */
fd_to_out    = PX_Handler_simple_create( PX_Hfcn_to_stdout, (void *)0 );
PX_FD_add( fd, PX_READ, fd_to_out, err_fd, (PX_Handler)0,
         (struct timeval *)0, &ctx );

/* gselect is basically an infinite loop */
if (PX_Gselect( &ctx, (void *)0 )) {
    perror( "Gselect failed" );
    return 1;
    }
return 0;
}
```

Now, the server:

```
#include "pexec.h"
#include <stdio.h>
#include <errno.h>

#define STDIN 0

int GetConnection ANSI_ARGS((PX_Ctx *, PX_Conn *, PX_Handler, int )),
    UnlinkLocal ANSI_ARGS((PX_Ctx *, PX_Conn *, PX_Handler, int ));
extern int ReadCommand ANSI_ARGS(( PX_Ctx *, PX_Conn *, PX_Handler, int));
extern void *InitReadCommand( );

int main( argc, argv )
int  argc;
char **argv;
{
PX_Ctx     ctx;
PX_Handler exit_on_close, err_fd;
PX_Handler read_command;
PX_Handler unlinklocal, getconnection;
int        fd;
void       *sock_info;

PX_Ctx_init( &ctx );
PX_Monitor_setup( &ctx );

exit_on_close = PX_Handler_simple_create( PX_Hfcn_exit_on_close, (void *)0 );
err_fd        = PX_Handler_simple_create( PX_Hfcn_error_fd, (void *)0 );

/* Add a command interpreter to stdin */
read_command = PX_Handler_simple_create( ReadCommand, InitReadCommand(&ctx) );
PX_FD_add( STDIN,  PX_READ, read_command, err_fd, exit_on_close,
          (struct timeval *)0, &ctx );

/* Add a local socket */
fd = PX_FD_Create_local_socket( "/tmp/pexec", 1, &sock_info );
if (fd < 0) {
    fprintf( stderr, "Error creating local socket\n" );
    }
else {
    getconnection = PX_Handler_create( GetConnection, sock_info, (int (*)())0,
                                (void *)0, UnlinkLocal );
    unlinklocal   = PX_Handler_simple_create( UnlinkLocal, (void *)0 );
    PX_FD_add( fd, PX_READ, getconnection, unlinklocal, unlinklocal,
          (struct timeval *)0, &ctx );
    PX_Handler_free( unlinklocal );
    PX_Handler_free( getconnection );
    }

/* gselect is basically an infinite loop */
if (PX_Gselect( &ctx, (void *)0 )) {
    perror( "Gselect failed" );
    return 1;
    }
```

```
return 0;
}

/* This is called when a listen allows an accept */
int GetConnection( ctx, conn, handler, in_mask )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_mask;
{
int fd;
PX_Handler err_fd, close_fd, fd_to_set;

fd = PX_FD_Accept_connection( conn->fd, handler->extra_data );
if (fd < 0) {
    fprintf( stderr, "accept errno = %d\n", errno );
    perror( "Accept failed" );
    return fd;
    }

close_fd      = PX_Handler_simple_create( PX_Hfcn_close_fd, (void *)0 );
fd_to_set     = PX_Handler_simple_create( PX_Hfcn_fd_to_set, (void *)0 );
err_fd        = PX_Handler_simple_create( PX_Hfcn_error_fd, (void *)0 );
PX_FD_add( fd, PX_READ, fd_to_set, err_fd, close_fd, (struct timeval *)0,
           ctx );
PX_FdAddToSet( fd_to_set, 1 );
PX_FdAddToSet( fd_to_set, 2 );
return 0;
}

int UnlinkLocal( ctx, conn, handler, err )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        err;
{
unlink( "/tmp/pexec" );
return 0;
}
```

7 PEXEC routines

7.1 Basic Routines

These routines create and manage the PEXEC context, and provide the basic `select` loop.

PX_Ctx_init **PX_Ctx_init**

PX_Ctx_init — Initializes the pexec context

Input Parameter

ctx Pointer to structure to initialize

Returns

0 on success.

Synopsis

```
int PX_Ctx_init( ctx )
PX_Ctx *ctx;
```

Location

`src/fds.c`

PX_Ctx_end **PX_Ctx_end**

PX_Ctx_end — Ends all connections on a PEXEC context

Input Parameter

ctx	Pointer to PEXEC structure

Returns

0 on success.

See Also

PX_Hfcn_close_on_exit

Synopsis

```
int PX_Ctx_end( ctx )
PX_Ctx *ctx;
```

Location

`src/fds.c`

PX_Ctx_go **PX_Ctx_go**

PX_Ctx_go — Marks a PEXEC context for looping

Input Parameter

ctx	PEXEC context

Notes

This routine sets a field that is checked by `PX_Gselect` each time `PX_Gselect` would call `select`. By using the routine `PX_Ctx_stop`, a routine (such as a handler) can cause `PX_Gselect` to exit before the next call to `select`.

Synopsis

```
int PX_Ctx_go( ctx )
PX_Ctx *ctx;
```

Location

`src/fds.c`

PX_Ctx_stop **PX_Ctx_stop**

PX_Ctx_stop — Marks a PEXEC context for no looping

Input Parameter

ctx PEXEC context

Notes

This routine sets a field that is checked by `PX_Gselect` each time `PX_Gselect` would call `select`. By using this, a routine (such as a handler) can cause `PX_Gselect` to exit before the next call to `select`.

See Also

PX_Ctx_go

Synopsis

```
int PX_Ctx_stop( ctx )
PX_Ctx *ctx;
```

Location

`src/fds.c`

PX_Ctx_close_all_conn **PX_Ctx_close_all_conn**

PX_Ctx_close_all_conn — Close all of the connections/fds

Input Parameter

ctx Pexec context

Synopsis

```
int PX_Ctx_close_all_conn( ctx )
PX_Ctx *ctx;
```

Location

`src/fds.c`

PX_Gselect **PX_Gselect**

PX_Gselect — Processes activities on specified file descriptors

Input Parameters

ctx	Context containing FDs to act on
gtimeout	global timeout for this routine. If null, there is no time out. Timeouts are *absolute*, not relative. That is, this value should give the time in the future when the timeout should occur, not the increment in the time.

Returns

0 on success; else -1 with error in `errno`. That value corresponds to the system error that caused the problem.

Return value on timeout (caused by `gtimeout`) is 0.

See Also

PX_Ctx_init, PX_Ctx_end, PX_Ctx_stop

Synopsis

```
int PX_Gselect( ctx, gtimeout )
PX_Ctx *ctx;
struct timeval *gtimeout;
```

Location

`src/gselect.c`

7.2 Handler Management

These routines create and free handlers for file descriptors.

PX_Handler_create **PX_Handler_create**

PX_Handler_create — Creates a handler structure

Input Parameters

fcn	Handler function to call.
extra_data	pointer to extra state needed by `fcn`.
info	Function to call to give information on handler. May be null.
info_data	Pointer to extra state for `info`
destroy	Function to call when freeing handler. May be null.

Returns

New handler, or null if unable to allocate space.

Notes

The function `PX_Handler_simple_create` may be used for handlers without special info or destroy routines.

Synopsis

```
PX_Handler PX_Handler_create( fcn, extra_data, info, info_data, destroy )
int (*fcn) ANSI_ARGS(( PX_Ctx *, PX_Conn *, PX_Handler, int ));
void *extra_data;
int (*info) ANSI_ARGS(( PX_Handler, char **));
void *info_data;
int (*destroy) ANSI_ARGS(( PX_Handler ));
```

Location

`src/fds.c`

PX_Handler_simple_create **PX_Handler_simple_create**

PX_Handler_simple_create — Creates a handler structure

Input Parameters

fcn	Handler function to call.
extra_data	pointer to extra state needed by `fcn`.

Returns

New handler, or null if unable to allocate space.

Notes

This is a simple form of `PX_Handler_create` that has the same effect as

```
    PX_Handler_create( fcn, extra_data,
       (int (*)())0, (void *)0, (int (*)())0 );
```

Synopsis

```
PX_Handler PX_Handler_simple_create( fcn, extra_data )
int (*fcn) ANSI_ARGS(( PX_Ctx *, PX_Conn *, PX_Handler, int ));
void *extra_data;
```

Location

```
src/fds.c
```

PX_Handler_free **PX_Handler_free**

PX_Handler_free — Frees the storage used by a handler

Input Parameter

handler Handler to free

Notes

A reference counting mechanism allows a handler to be used by multiple FDs. The handler is only removed when the reference count reaches zero.

However, extreme care must be exercised in using this feature. In most cases, it is better to create a new handler for each use, since the position of a handler in a chain of handlers on a single FD may differ from FD to FD.

Synopsis

```
int PX_Handler_free( handler )
PX_Handler handler;
```

Location

```
src/fds.c
```

7.3 Handler Service Functions

These routines provide useful handler functions for common operations.

PX_Hfcn_close_fd **PX_Hfcn_close_fd**

PX_Hfcn_close_fd — Closes an fd

Input Parameters

ctx Pexec context
conn Connection
handler This handler
in_out In/out mask. Ignored.

Synopsis

```
int PX_Hfcn_close_fd( ctx, conn, handler, err )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        err;
```

Location

src/handlers.c

PX_Hfcn_do_eof **PX_Hfcn_do_eof**

PX_Hfcn_do_eof — A special handler for EOF on read

Input Parameters

ctx	Pexec context
conn	Connection
handler	This handler
n	value returned by read (less than or equal to zero).

Notes

This is a special handler for end-of-file and error indications from a read. A typical use from within another handler is

```
  n = read( ... );
  if (n <= 0)
      return PX_Hfcn_do_eof( ctx, conn, handler, n );
```

This handler is normally *not* used as an file descriptors handler.

Synopsis

```
int PX_Hfcn_do_eof( ctx, conn, handler, n )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        n;
```

Location

src/handlers.c

PX_Hfcn_error_fd **PX_Hfcn_error_fd**

PX_Hfcn_error_fd — Closes an fd on error

Input Parameters

ctx Pexec context
conn Connection
handler This handler
in_out In/out mask. Ignored.

Synopsis

```
int PX_Hfcn_error_fd( ctx, conn, handler, err )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        err;
```

Location

`src/handlers.c`

PX_Hfcn_exit_on_close **PX_Hfcn_exit_on_close**

PX_Hfcn_exit_on_close — Closes all fds and exit PX_gselec

Input Parameters

ctx Pexec context
conn Connection
handler This handler
in_out In/out mask. Ignored.

See Also

PX_Ctx_end

Synopsis

```
int PX_Hfcn_exit_on_close( ctx, conn, handler, err )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        err;
```

Location

`src/handlers.c`

PX_Hfcn_fd_to_set **PX_Hfcn_fd_to_set**

PX_Hfcn_fd_to_set — Sends data to a set of file descriptors

Input Parameters

ctx	Pexec context
conn	Connection
handler	This handler
in_out	In/out mask. Ignored.

Notes

The set of file descriptors to which data is sent is set with the routine `PX_FdAddToSet`.

Synopsis

```
int PX_Hfcn_fd_to_set( ctx, conn, handler, in_out )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_out;
```

Location

`src/handlers.c`

PX_FdAddToSet **PX_FdAddToSet**

PX_FdAddToSet — Adds a file descriptor to the set of fds that a handler will write to

Input Parameters

handler	Handler created with `PX_Handler_create` and function `PX_Hfcn_fd_to_set`
fd	fd to write to

Notes

Up to 20 fds may be specified in a set.

Synopsis

```
int PX_FdAddToSet( handler, fd )
PX_Handler handler;
int        fd;
```

Location

`src/handlers.c`

PX_Hfcn_handshake **PX_Hfcn_handshake**

PX_Hfcn_handshake — Handler for handshake code

Notes

This implements a simple handshake on a connection. This should be combined with a reasonable timeout to handle failure of the handshake.

See Also

PX_Create_handshake

Synopsis

```
int PX_Hfcn_handshake( ctx, conn, handler, in_out )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_out;
```

Location

`src/rshstart.c`

PX_Create_handshake **PX_Create_handshake**

PX_Create_handshake — Create a handshake structure

Input Parameter

str	String to use as the handshake

Returns

Pointer to `extra_data` to give when creating handler with `PX_Hfcn_handshake`. A typical use is

```
PX_Handler_simple_create( PX_Hfcn_handshake,
                          PX_Create_handshake( "My_String" ) );
```

Notes

The routine at the other end should write the string to the other side of the pipe/socket that this handler is connected to.

Synopsis

```
void *PX_Create_handshake( str )
char *str;
```

Location

src/rshstart.c

PX_Hfcn_to_stderr **PX_Hfcn_to_stderr**

PX_Hfcn_to_stderr — Sends data to standard error

Input Parameters

ctx Pexec context
conn Connection
handler This handler
in_out In/out mask. Ignored.

Synopsis

```
int PX_Hfcn_to_stderr( ctx, conn, handler, in_out )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_out;
```

Location

src/handlers.c

PX_Hfcn_to_stdout **PX_Hfcn_to_stdout**

PX_Hfcn_to_stdout — Sends data to standard output

Input Parameters

ctx Pexec context
conn Connection
handler This handler
in_out In/out mask. Ignored.

Synopsis

```
int PX_Hfcn_to_stdout( ctx, conn, handler, in_out )
PX_Ctx     *ctx;
PX_Conn    *conn;
PX_Handler handler;
int        in_out;
```

Location

src/handlers.c

PX_Hdestroy_extras PX_Hdestroy_extras

PX_Hdestroy_extras — Frees any space in the handler extra data fields

Input Parameter

handler Handler to free space from

Notes

This routine uses the `free` call to free any data in the `extra_data` or `info_data` fields. Most handlers should call this routine (it is safe even if no data is ever allocated).

Synopsis

```
int PX_Hdestroy_extras( handler )
PX_Handler handler;
```

Location

`src/fds.c`

PX_Hinfo_name PX_Hinfo_name

PX_Hinfo_name — Returns the name of a handler

Input Parameter

handler Handler to get information on

Output Parameter

name Pointer to a string of data

Notes

The name returned by this routine was set when the routine was installed as the handler

```
PX_Handler_create( ..., PX_Hinfo_name, "foobar", ... );
```

The string holding the name should be in permanent rather than stack storage, and the destroy routine `PX_Hdestroy_extras` should *not* be used.

This needs to be improved; this routine is a placeholder for the moment.

Synopsis

```
int PX_Hinfo_name( handler, ptr_p )
PX_Handler handler;
char       **ptr_p;
```

Location

`src/fds.c`

7.4 File Descriptor Management

These routines install and free handlers and read/write operations on specific file descriptors.

PX_FD_add **PX_FD_add**

PX_FD_add — Add a file descriptor to the fds known by a pexec context

Input Parameters

fd	File descriptor to add
in_out_mask	indicates whether the file descriptor is for read (`PX_READ`), write (`PX_WRITE`), or both (`PXREAD | PXWRITE`).
handler	`PX_Handler` to use for activity on `fd`
on_err	Handler to call for an error condition on `fd`
on_eof	Handler to call for end-of-file on `fd`
timeout	timeout (may by a null pointer for no timeout). Timeouts are in terms of absolute time.
ctx	`pexec` context to add `fd` to

Notes

The handlers are added at the *end* of the list of existing handlers.

Synopsis

```
int PX_FD_add( fd, in_out_mask, handler, on_err, on_eof, timeout, ctx )
int             fd, in_out_mask;
PX_Handler     handler, on_err, on_eof;
struct timeval *timeout;
PX_Ctx         *ctx;
```

Location

`src/fds.c`

PX_FD_free **PX_FD_free**

PX_FD_free — Frees the handlers associated with an fd

Input Parameters

ctx	pexec context
fd	File descriptor to free; this fds handlers will be removed.

Notes

The fd is *not* closed unless the `fds` handler"s destroy function chooses to do so. Permanent handlers are left in place.

Synopsis

```
int PX_FD_free( ctx, fd )
PX_Ctx *ctx;
int    fd;
```

Location

src/fds.c

PX_Read_add **PX_Read_add**

PX_Read_add — Adds a function for reading from an FD

Input Parameters

ctx	PEXEC context
fd	File descriptor to read from
read	Function to call for actual read
extra_data	Extra data saved for function

Notes

By default, the function `PX_Read` is called to perform reads. This function allows the user to choose a different routine or to interpose others. The read function is added at the *head* of the list; it may call functions further down in the list to complete the operation. See the tracing examples for more information.

Synopsis

```
int PX_Read_add( ctx, fd, read, extra_data )
PX_Ctx *ctx;
int    fd;
int    read ANSI_ARGS((int,void *,int,PX_ReadFn));
void   *extra_data;
```

Location

src/rdwrt.c

PX_Write_add **PX_Write_add**

PX_Write_add — Adds a function for writing to an FD

Input Parameters

ctx	PEXEC context
fd	File descriptor to write to
read	Function to call for actual write
extra_data	Extra data saved for function

Notes

By default, the function `PX_Write` is called to perform writes. This function allows the user to choose a different routine or to interpose others. The write function is added at the *head* of the list; it may call functions further down in the list to complete the operation. See the tracing examples for more information.

Synopsis

```
int PX_Write_add( ctx, fd, write, extra_data )
PX_Ctx *ctx;
int    fd;
int    write ANSI_ARGS((int,void *,int,PX_WriteFn));
void   *extra_data;
```

Location

`src/rdwrt.c`

PX_FD_set_active **PX_FD_set_active**

PX_FD_set_active — Controls whether a read/write mask is set

Input Parameters

ctx	`pexec` context
fd	fd to change active set for
mask	Mask value to change. Either `PX_READ` or `PX_WRITE`.
flag	Value. `1` for set, `0` for cleared.

Synopsis

```
void PX_FD_set_active( ctx, fd, mask, flag )
PX_Ctx *ctx;
int    fd, mask, flag;
```

Location

`src/fds.c`

7.5 File Descriptor Service Functions

These routines provide useful routines for acquiring file descriptors for common operations; in particular, a number of routines for interprocess communication over sockets are provided here.

PX_FD_Accept_connection **PX_FD_Accept_connection**

PX_FD_Accept_connection — Accepts a connection and creates a new file descriptor

Input Parameters

fd File descriptor created by `PX_FD_Create_network_socket` or `PX_FD_Create_local_socket`.

Returns

New file descriptor for a new socket connection. This file descriptor is marked as non-blocking.

Synopsis

```
int PX_FD_Accept_connection( fd, sock_info )
int        fd;
void       *sock_info;
```

Location

`src/sockets.c`

PX_FD_Create_local_conn **PX_FD_Create_local_conn**

PX_FD_Create_local_conn — Creates a connection to a local process

Input Parameters

server_path Path name to socket
q_wait If true, wait for the connection to be established.

Output Parameter

is_ready Set to true if the connection is established. May be null if `q_wait` is true.

Returns

A file descriptor on which a `connect` has been *started*. If `q_wait` is false, this file descriptor is ready *only* when a `PX_FD_Establish_local_conn` has completed on it, unless `is_ready` is true.

Synopsis

```
int PX_FD_Create_local_conn( server_path, q_wait, is_ready )
char *server_path;
int  q_wait, *is_ready;
```

Location

`src/sockets.c`

PX_FD_Create_local_socket **PX_FD_Create_local_socket**

PX_FD_Create_local_socket — Creates a file descriptor for a local (Unix domain) socket

Input Parameters

name Valid name in the file system for the local socket.

max_number_of_pending_conn
Controls the number of pending connections. A reasonable value is zero.

Output Parameter

sock_info Pointer to data to give `PX_FD_Accept_connection`. Use `PX_FD_Free_sockinfo` when closing this FD.

Returns

A file descriptor or `-1`.

Notes

This creates an fd that a `listen` has been performed on. The routine, `PX_FD_Accept_connection` should be called to create an fd for communication with another process. That process should use `PX_FD_Create_local_conn` and `PX_FD_Establish_local_conn` to establish the other side of the connection.

In Master/Slave or Server/Client applications, this routine is used by the Master/Server.

Synopsis

```
int PX_FD_Create_local_socket( name, max_number_of_pending_conn, sock_info )
char *name;
int  max_number_of_pending_conn;
void **sock_info;
```

Location

`src/sockets.c`

PX_FD_Create_network_conn **PX_FD_Create_network_conn**

PX_FD_Create_network_conn — Creates a connection to a remote process

Input Parameters

server_hostname
Network name of remote processor

portnum port number.

q_wait If true, wait for the connection to be established.

Output Parameter

is_ready Set to true if the connection is established. May be null if `q_wait` is true.

Returns

A file descriptor on which a `connect` has been *started*. If `q_wait` is false, this file descriptor is ready *only* when a `PX_FD_Establish_network_conn` has completed on it, unless `is_ready` is true.

Notes

This routine uses `gethostbyname` to get information about `server_name`; as a result, this routine is not thread-safe (because `gethostbyname` is not thread safe), and should not be used in a signal handler.

Synopsis

```
int PX_FD_Create_network_conn( server_hostname, portnum, q_wait, is_ready )
char *server_hostname;
int  portnum, q_wait, *is_ready;
```

Location

`src/sockets.c`

PX_FD_Create_network_socket **PX_FD_Create_network_socket**

PX_FD_Create_network_socket — Creates a file descriptor for a network (INET) socket

Input Parameters

hostname Name of host; network address is (portnum,hostname). May be null, in which case `gethostname` is used.

portnum Port number.

max_number_of_pending_conn
Controls the number of pending connections. A reasonable value is zero.

Output Parameter

sock_info Pointer to data to give `PX_FD_Accept_connection`. Use `PX_FD_Free_sockinfo` when closing this FD.

Returns

A file descriptor or `-1`.

Notes

This creates an fd that a `listen` has been performed on. The routine, `PX_FD_Accept_connection` should be called to create an fd for communication with another process. That process should use `PX_FD_Create_network_conn` and `PX_FD_Establish_network_conn` to establish the other side of the connection.

In Master/Slave or Server/Client applications, this routine is used by the Master/Server.

Synopsis

```
int PX_FD_Create_network_socket( hostname, portnum,
     max_number_of_pending_conn, sock_info )
int  portnum, max_number_of_pending_conn;
char *hostname;
void **sock_info;
```

Location

src/sockets.c

PX_FD_Establish_local_conn **PX_FD_Establish_local_conn**

PX_FD_Establish_local_conn — Completes the establishment of a connection

Input Parameter

fd File descriptor for connection to complete.

Notes

This routine should be called only if `PX_FD_Create_local_conn` returned false for `is_ready` *and* `PX_Gselect` has determined that `fd` is available for *writing*.

Synopsis

```
int PX_FD_Establish_local_conn( fd )
int fd;
```

Location

src/sockets.c

PX_FD_Establish_network_conn **PX_FD_Establish_network_conn**

PX_FD_Establish_network_conn — Completes the establishment of a connection

Input Parameter

fd File descriptor for connection to complete.

Notes

This routine should be called only if `PX_FD_Create_network_conn` returned false for `is_ready` *and* `PX_Gselect` has determined that `fd` is available for *writing*.

Synopsis

```
int PX_FD_Establish_network_conn( fd )
int fd;
```

Location

```
src/sockets.c
```

PX_FD_Free_sockinfo **PX_FD_Free_sockinfo**

PX_FD_Free_sockinfo — Frees sockinfo information

Input Parameter

sock_info socket inforamation returned by `PX_FD_Create_network_socket` or `PX_FD_Create_local_socket`.

Synopsis

```
int PX_FD_Free_sockinfo( sock_info )
void *sock_info;
```

Location

```
src/sockets.c
```

7.6 Process Management

These routines provide a simple process management capability.

PX_Proc_fork_with_pipes **PX_Proc_fork_with_pipes**

PX_Proc_fork_with_pipes — Fork a process and setup pipes for stdin/out/err

Output Parameters

pid pid of created process.
rd_in fd for standard input to program
wrt_out fd for standard output from program
wrt_err fd for standard error from program

Synopsis

```
int PX_Proc_fork_with_pipes( pid, rd_in, wrt_out, wrt_err )
int *pid, *rd_in, *wrt_out, *wrt_err;
```

Location

`src/runpgm.c`

PX_Proc_run_on_host **PX_Proc_run_on_host**

PX_Proc_run_on_host — Runs a program on a host, using remote shell

Input Parameters

host	Name of host
pgm	Name of program
argv	Argument vector; null terminated

Output Parameters

rd_in	fd for standard input to program
wrt_out	fd for standard output from program
wrt_err	fd for standard error from program
pgmst	Process description structure

Notes

This process returns as soon as the fork to create the process to run the remote shell in completes. Robust applications should consider a `PX_Hfcn_handshake` handler (on wrt_out) and a timeout.

Synopsis

```
int PX_Proc_run_on_host( host, pgm, argv, rd_in, wrt_out, wrt_err, pgmst )
char *host, *pgm, **argv;
int  *rd_in, *wrt_out, *wrt_err;
PX_Pgm **pgmst;
```

Location

`src/rshstart.c`

PX_Proc_run_program **PX_Proc_run_program**

PX_Proc_run_program — Runs a program on this processor

Input Parameters

pgm	Name of program
argv	Argument vector; null terminated

Output Parameters

rd_in	fd for standard input to program
wrt_out	fd for standard output from program

wrt_err fd for standard error from program
pgmst Process description structure

Notes

This process returns as soon as the fork to create the process to run the program in completes.

Synopsis

```
int PX_Proc_run_program( pgm, argv, rd_in, wrt_out, wrt_err, pgmst )
char *pgm;
char **argv;
int *rd_in, *wrt_out, *wrt_err;
PX_Pgm **pgmst;
```

Location

`src/runpgm.c`

PX_Proc_set_child_handler **PX_Proc_set_child_handler**

PX_Proc_set_child_handler — Setup a signal handler for SIGCHLD events

Input Parameter

ctx Pexec context

Notes

The signal handler installed by this routine will attempt to call the appropriate handlers for closing the standard in/out/err pipes to the process, if these fds are known by the pexec context (`ctx`). The process exit handler will also be called (see the `PX_pgm` structure).

Synopsis

```
void PX_Proc_set_child_handler( ctx )
PX_Ctx *ctx;
```

Location

`src/runpgm.c`

7.7 Miscellaneous

These routines provide functions such as debugging, logging, and tracing through the use of persistent handlers.

PX_Monitor_setup **PX_Monitor_setup**

PX_Monitor_setup — Installs a simple tracing handler to all fds

Input Parameter

ctx Pexec context

Notes

The handlers installed by this call are placed at the head of each possible fd, and are persistent (they are not deleted when the fd is closed). The handlers write some information about the handlers that are actually being used to handle the data to standard output.

This is an example of composing fd handlers to provide extra debugging and monitoring information.

Synopsis

```
int PX_Monitor_setup( ctx )
PX_Ctx *ctx;
```

Location

src/monitor.c

PX_Read **PX_Read**

PX_Read — Basic read function

Input Parameters

fd File descriptor to read
buf Data area to read into
len Maximum amount of data to read
ctx Pointer to PX_ReadFn that holds this function (self)

Returns

Number of bytes read, or -1 if there is an error. If the read was aborted because of an interrupted system call (EINTR), the read is restarted.

Synopsis

```
int PX_Read( fd, buf, len, ctx )
int        fd, len;
void       *buf;
PX_ReadFn ctx;
```

Location

src/rdwrt.c

PX_Read_add **PX_Read_add**

PX_Read_add — Adds a function for reading from an FD

Input Parameters

ctx	PEXEC context
fd	File descriptor to read from
read	Function to call for actual read
extra_data	Extra data saved for function

Notes

By default, the function `PX_Read` is called to perform reads. This function allows the user to choose a different routine or to interpose others. The read function is added at the *head* of the list; it may call functions further down in the list to complete the operation. See the tracing examples for more information.

Synopsis

```
int PX_Read_add( ctx, fd, read, extra_data )
PX_Ctx *ctx;
int    fd;
int    read ANSI_ARGS((int,void *,int,PX_ReadFn));
void   *extra_data;
```

Location

src/rdwrt.c

PX_Write **PX_Write**

PX_Write — Basic write function

Input Parameters

fd	File descriptor to write
buf	Data area to write
len	Amount of data to write
ctx	Pointer to PX_WriteFn that holds this function (self)

Returns

Number of bytes written, or -1 if there is an error. If the write was aborted because of an interrupted system call (EINTR), the write is restarted.

Synopsis

```
int PX_Write( fd, buf, len, ctx )
int        fd, len;
void       *buf;
PX_WriteFn ctx;
```

Location

src/rdwrt.c

PX_Write_add **PX_Write_add**

PX_Write_add — Adds a function for writing to an FD

Input Parameters

ctx	PEXEC context
fd	File descriptor to write to
read	Function to call for actual write
extra_data	Extra data saved for function

Notes

By default, the function `PX_Write` is called to perform writes. This function allows the user to choose a different routine or to interpose others. The write function is added at the *head* of the list; it may call functions further down in the list to complete the operation. See the tracing examples for more information.

Synopsis

```
int PX_Write_add( ctx, fd, write, extra_data )
PX_Ctx *ctx;
int    fd;
int    write ANSI_ARGS((int,void *,int,PX_WriteFn));
void   *extra_data;
```

Location

src/rdwrt.c

PX_Util_Deltatime **PX_Util_Deltatime**

PX_Util_Deltatime — Returns a time in the future

Input Parameters

seconds	Number of seconds in the future
usec	Number of microseconds in the future
tval	Pointer to structure to use for timeval.

Returns

Pointer to timeval structure for the specified time in the future. This in in absolute time format, as required by PEXEC routines. The pointer is `tval` if `tval` is non-null; otherwise it is a *local static* variable and should not be used if there is any chance that this data will be overwritten.

Note that when used with `PX_FD_Add`, it is ok to use a null `tval` because `PX_FD_Add` makes its own copy of the timevalue.

Synopsis

```
struct timeval *PX_Util_Deltatime( seconds, usec, tval )
int            seconds, usec;
struct timeval *tval;
```

Location

src/util.c