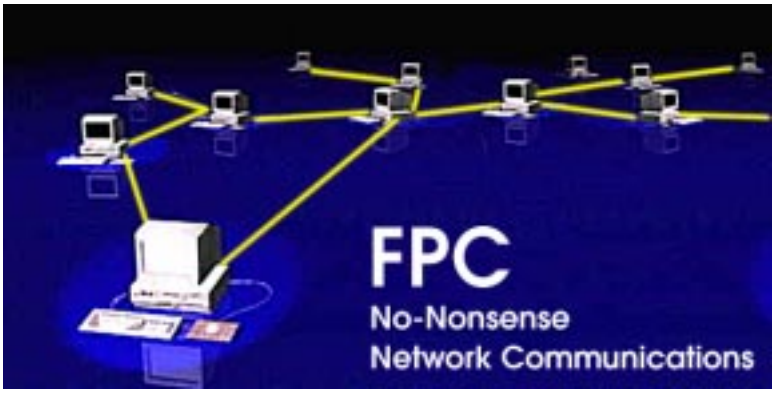
FPC

Fourth Planet Communicator
Version 2



FOURTH
PLANET

Printed in the United States of America.
July 2000.

Document FPC-2-Manual 7/30/2000

Contents

1

Introduction

Welcome to Fourth Planet Communicator (FPC) from Fourth Planet, Inc. FPC is a network communication system that gives you everything you need to create a wide range of distributed computing applications. Whether you are developing multi-user games, data collection systems, or parallel computing programs, FPC provides you with the easy way to quickly and flexibly share data and information.

1.1 Overview

When you are building a distributed computing application, sharing data among multiple processes is the hardest part. Guaranteeing that data is distributed efficiently and flexibly is a difficult problem which frequently prolongs development time. The problem is challenging enough when your application is on one machine, but it can be daunting when you have processes on different computers with different operating systems.

Fourth Planet Communicator (FPC) is a system for distributing data across computer networks. FPC uses a *publish and subscribe* framework which provides efficient, dynamically reconfigurable, and scalable data distribution. With FPC, it is easy to distribute information and to communicate between multiple processes, whether they are running on the same computer or not.

Furthermore, since FPC has been designed to be easy to use and deploy, you can develop distributed computing applications with minimal time and effort. Because FPC does all the work, you have no need to worry about the intricate details of network communications. FPC lets you concentrate on developing your application, not on distributing the data.

1.2 Features

FPC provides distributed, dynamic network communications using a *publish and subscribe* framework. In this model, a process which sends data is called a *producer* and a process which receives data is called a *consumer*. Producers *publish* data: they announce that they are generating some type of information. Consumers *subscribe* to data: they request that they receive certain types of information when it becomes available. Thus, when a producer produces data, it is delivered only to those consumers who want to consume it.

To avoid some of the problems associated with publish and subscribe (network congestion, complex consumer coding), FPC uses a centralized data cache, the FPC server, for managing data delivery to consumers. The FPC server operates like an efficient assistant: receiving and storing data whenever it becomes available, discarding data when it becomes outdated, and delivering data only when it is requested.

Here is a summary of the features provided by FPC:

- supports multiple producers and multiple consumers
- supports multiple “virtual networks” for isolating sets of producers and consumers from other producers and consumers
- simple, easy-to-use C programming interface
- reliable and efficient messaging via centralized cache
- supports multiple operating systems: Linux, Microsoft Windows, Silicon Graphics IRIX, Sun Solaris
- license management (*contact for availability)*
- other languages (*contact for availability*): Java, TCL, Perl

1.3 Using this manual

This brief introduction has given you a broad overview of FPC. The rest of this manual provides the details you will need to install and use FPC. Chapter 2 describes how to install and test the FPC distribution. Chapter 3 provides an in-depth look at the architecture underlying FPC. Chapter 4 gives step-by-step instructions for programming with FPC, including several detailed examples. Chapter 5 documents FPC functions and error codes. Chapter 6 contains a glossary of frequently used terms.

2

Installation

2.1 Installation Overview

This chapter will help you install FPC on to your computer. To install FPC, you will need to perform the following steps:

- Pre-install planning
- Obtain the distribution file
- Unpack and install the distribution
- Install the FPC license
- Configure your user environment
- Test the installation

Please be sure to read the release notes (`relnotes.txt`) accompanying the FPC distribution for changes or late-breaking information that could not be included in this manual.

2.2 Pre-install planning

System requirements

FPC is currently supported on the following operating systems:

- Red Hat Linux 5.0 and 5.1
- Microsoft Windows 95 and NT (Service Pack 3 or later)
- Silicon Graphics IRIX 6.2, 6.3, and 6.4
- Sun Solaris 2.5.x

FPC requires *thread* support for each of the above operating systems. The Linux, IRIX, and Solaris versions of FPC use POSIX threads *(pthreads*). The Windows 95 and NT versions use Microsoft WIN32 threads. Table 1

describes what action (if any) you need to take to obtain thread support for each of these operating systems.

Table 1 **- Operating System Thread Support**

Operating System	Required Action
red Hat Linux 5.0 and 5.1	none (POSIX thread support is built-in)
Microsoft Windows 95 and Windows NT (SP3 or later)	none (WIN32 thread support is built-in)
Silicon Graphics IRIX (6.2, 6.3, 6.4)	IRIX 6.2 requires the “6.2 POSIX Patch Set” (contact Silicon Graphics). IRIX 6.3 and 6.4 both have POSIX thread support built-in, but we highly recommend that you install the latest “POSIX Patch Set” for 6.3 and 6.4
Sun Solaris (2.5.x)	none (POSIX thread support is built-in)

Choosing locations

Before you install FPC, you will need to do the following:

- Select an *installation directory*. This is the location into which you will unpack all FPC files. It can be located anywhere, but you must have permissions to create directories and write files.
- Select the *FPC server host*. This is the computer on which you will run the FPC server. Since the FPC server must be running in order to use FPC, you should choose a computer which is reliable and on which you can start programs. This computer must be able to access the *installation directory*.

2.3 Obtain the distribution

CD-ROM

If your distribution of FPC is on CD-ROM, place your FPC CD-ROM into the CD-ROM drive.

UNIX Distribution

Place your FPC CD-ROM into the CD-ROM drive. `cd` to the proper CD-ROM mount point for your system. For IRIX, this will typically be `/CDROM`. Proceed to **2.4 Install the distribution**.

Windows NT Distribution

The installation program SETUP.EXE should run automatically. Proceed to **2.4 Install the distribution**.

World Wide Web

If you have downloaded the distribution from a web site, you will first have to unpack the distribution before installing it.

UNIX Distribution

The UNIX distribution, fpc_2_0.tar.gz, is a gzipped tar archive. cd into a temporary directory (like /tmp), and extract the distribution using the command:

```
cat fpc_2_0.tar.gz | gunzip -d | tar -xf -
```

Windows NT Distribution

The Windows NT distribution, fpc_2_0.exe, is a self-extracting archive. Place the distribution file in a temporary location (like C:\temp), and double-click on the archive to extract the files in the distribution.

2.4 Install the distribution

UNIX

FPC can be installed by any user (i.e., you do not need to be the superuser). However, depending on where you decide to install FPC (e.g., on a shared filesystem), you may need to get administrator permission to create directories and to write files.

From the directory containing the distribution:

```
% ./install.sh
```

The script will install the distribution files into the *installation directory* of your choice. Please consult the MANIFEST file accompanying the FPC distribution for a description of the installed files.

Windows NT

Under Windows, launch SETUP.EXE to start the standard InstallShield installation process. InstallShield will guide you through the steps of the installation.

If you select the ***Standard*** *installation option, only the libraries for Windows NT will be installed. This is the normal way to install FPC. If you would like libraries for all available platforms to be installed, select the* ***Custom*** *installation option and make your choice.*

2.4.1 Getting the host identifier

You may need to generate a license key using the Fourth Planet licensemgr ***utilities before you can use FPC, depending on how FPC was compiled. The following discussion assumes that you do need to generate a license.***

You may need to obtain a license key using the Fourth Planet **licensemgr** utilities and place it into the FPKEYS file in order to use FPC. The license key is an alphanumeric string which provides authorization for using FPC. To obtain your license key, you will first need to provide a *host identifier* to **licensemgr**. The *host identifier* is an alphanumeric string which uniquely identifies the *server host(* the machine on which you installed and will run the FPC Server).

***The* host identifier *used for the Fourth Planet license key may not be the same as the hostid reported by other utilities which you may have at your disposal, and an incorrect host identifier will prevent FPC from running. Be sure to use the Fourth Planet utility given below to determine the proper* host identifier *for the* server host.**

UNIX

To find the *host identifier* of the *server host* under UNIX, run the FPC License Install script as follows:

```
% ./install-license
```

The *host identifier* will be displayed on the console.

Windows NT

To find the *host identifier* of the *server host*, run the FPC License Installer as follows:

`Start→Programs→Fourth Planet FPC→Install License`

The *host identifier* is displayed in the window labeled **`Hostid`**.

2.4.2 Obtaining a license key

Once you've determined your *host identifier*, you can generate a FPC license key (see the licensemgr **genkey** utility).

2.4.3 Installing the license key

After you've received your license key, you will need to place it into a file called FPKEYS.

UNIX

Run the license installer utility:

```
% ./install-license
```

This will give the location of the FPKEYS file. Open the FPKEYS file and paste your license key into it.

The* FPKEYS *file is owned by the user that installed FPC. In order to write to the* FPKEYS *file, you may need to become that user.

Windows NT

Run the license installer utility:

`Start→Programs→Fourth Planet FPC→Install License`

and paste your license key into the window provided.

2.5 Configure your user environment

Before you can start using FPC, you need to add the FPKEYS variable to your user environment. FPC uses the FPKEYS variable to locate the FPKEYS file.

UNIX

To permanently configure your user environment, edit your `.login` or `.cshrc` file. For example, if you have created the FPKEYS file for the FPC license key in the default location, add the following line to your `.login` or `.cshrc` file:

```
setenv FPKEYS /usr/local/fpc/FPKEYS
```

This command can also be directly executed to temporarily configure your user environment.

Windows NT

The FPC installation script normally takes care of setting your environment variables to their default values. To verify that they are set, or to change them, go to:

Start→Settings→Control Panel

Then, open **System**, and select the **Environment** tab at the top of the window. This window is divided into two parts: the top part lists the **System Variables**, valid for all users. The bottom part lists the **User Variables**, defined for the current user.

By default, FPC installs its environment variables **User Variables**. If you want these variables defined globally for all users on this machine, move them to or redefine them in **System Variables**.

To modify a variable, select it in the list and enter the new value. Then click **Set**. Once you are done modifying variables, click **Apply** or **Ok**.

To remove a variable, select it and click **Delete**. Once you are done, click **Apply** or **Ok**

2.6 Test the installation

At this point, FPC should be completely installed and configured on your computer. To test the installation, you should try to start the FPC server.

UNIX

Under UNIX, launch `fpcServer` as follows (this example assumes that you're using a Silicon Graphics workstation and have installed FPC in the default location):

```
% /usr/local/fpc/bin/mips_irix/fpcServer
```

If FPC is correctly installed, the server will start running and print a status message.

Windows NT

Under Windows, launch `fpcServer` from the **Start** menu as follows:

Start→Programs→Fourth Planet FPC→fpcServer

If FPC is correctly installed, the server will start running and print a status message.

3

Concepts

3.1 The FPC Communications Model

Publish and subscribe

FPC provides distributed, dynamic network communications using a *publish and subscribe* framework. In this model, a process which produces data (sends messages) is called a *producer* and a process which consumes data (receives messages) is called a *consumer*. Producers *publish* data: they announce that they are generating (sending) some type of information. Consumers *subscribe* to data: they announce that they want to consume (receive) certain types of information when it becomes available. Then, when a producer produces data, it is delivered only to those consumers who want to consume it.

The *publish and subscribe* framework is extremely powerful. It enables flexible applications: there is no need for explicitly or rigidly specifying communication routing (e.g., process A on machine B is sending data of type C to process D on machine E). It allows dynamically expandable and scalable applications: any number of consumers and producers can exist at any time. Most importantly, it provides an intuitive, easy-to-understand model for designing interprocess communications.

At the same time, however, *publish and subscribe* can cause problems. It can very easily result in unnecessary network congestion, especially if data is sent[1] to every subscribed consumer whenever a producer produces new data. *Publish and subscribe* may also require complex programming, particularly for data consumers. For example, many consumers may subscribe to the same publication but may want to receive the data differently (e.g., some may only want the most recent data, others all the data).

1. “distributed”, “delivered”, “pushed”, etc.

The FPC server

To address these problems, FPC uses a centralized data cache (the FPC server) for managing data delivery. Whenever a producer produces new data, the FPC server stores the data in a FIFO[1] queue for each consumer which has subscribed to this data. Consumers can the query the FPC server for the contents of their queue at any time (i.e., whenever they wish to receive new instances of subscribed data). Additionally, consumers can control the following by sending a request to the FPC server:

- the queue depth (the *cache limit*)
- the amount of data sent by the FPC server in response to a query (the *read limit*)

The *cache limit* parameter controls the amount of published data that the FPC server stores for a consumer. By setting a small cache limit, consumers can guarantee that they always receive recent data without having to process a potentially large backlog of "old" data. This is useful for applications in which consumers have to work with producers that produce data at high rates. Conversely, by using a large (or even infinite[2]) cache limit, consumers can guarantee that they do not miss too much (or any data) that is produced.

The *read limit* parameter provides consumers control over how they receive published data. By setting a small *read limit*, consumers can guarantee that they will only have to process a limited amount of data whenever they query the FPC server. This is important for time-critical applications (e.g., real-time graphical interfaces) which cannot afford to be overloaded with too much data at once. Conversely, by using a large (or unconstrained[3]) *read limit*, consumers can guarantee that they always process a significant portion (or all) of the published data stored by the FPC server.

An efficient postal system

FPC allows producers and consumers to communicate as if they were using an efficient postal system. Producers send mail (publish data), consumers receive it (subscribe and read). Producers can send mail to the post office (the FPC server) at any time. Consumers can get their mail at any time by "going to the post office" (querying the FPC server). Any number of producers can send the same mail (publish the same type of data) to all consumers who have subscribed to it.

1. First-In First-Out
2. Truth in advertising: infinite means "until all available memory has been allocated"
3. unconstrained means "all messages currently queued by the server"

The post office (FPC server) stores mail from producers into post office boxes (FIFO queues) of subscribed consumers. The post office boxes can be different sizes (*cache limit*) to suit the needs of each consumer. If a box becomes "full" (i.e., the *cache limit* is reached), the FPC server makes room for new mail by throwing out old mail. When consumers "check" their post office boxes, they can take as little or as much out of it as they desire (*read limit*).

Thus, with FPC producers can send mail (produce data) at any time to any number of consumers, but each consumer controls how and when they receive it. If only it could be so easy in the real world!

3.2 FPC Clients

An application built with FPC can be a producer, a consumer, or both. Regardless of type, these applications are all *FPC clients* because they all communicate via the FPC server. In fact, since the FPC server delivers data from producers to consumers, an FPC server must be in operation before any client-to-client communication can occur.

Channels

Unlike other communication systems, FPC does not multiplex different types of data on a single client-server communications link. Instead, FPC clients use one or more c*hannels* to communicate with the FPC server[1]. Producers use a different channel for each type of data they are producing: each channel carries one type of data from producer to server. Consumers generally also use different channels for each type of data they are consuming: each channel carries one type of data from server to consumer[2].

Channels provide tremendous flexibility for application design. For example, if you are building an application which distributes both time-critical (though low-rate) "control" information as well as non-timely (though high-rate) "data", you can create a *control channel* and a separate *data channel* in all your producers and consumers. Then when the application is running, consumers will be able to process the two channels in parallel. In other words, consumers will be able to receive urgent messages

1. FPC clients are not restricted to a single FPC server, but may use multiple channels to communicate with multiple FPC servers.
2. Consumers can use a single channel to communicate with the FPC server, but doing so will mix different types of data together (i.e., the data will be multiplexed). Since FPC does not provide a mechanism for distinguishing data, the consumer itself must be able to identify different data arriving on the same channel.

from the *control channel* without having to first wade through a stream of routine data on the *data channel*.

Producers

An FPC producer does the following for each type of data it produces:

- connects (opens a channel) to the FPC server
- specifies the channel's publication (i.e., what type of data will be sent on the channel)
- sends (produces) data on the channel

Connections to the FPC server are performed using the FPC library function `fpcConnect`. The producer and FPC server may both be run on the same computer or on different computers. The only requirement is that they share a TCP/IP network connection.

A publication is specified in FPC as a simple text string using the FPC library function `fpcPublish`. So, "data", "Temperature : value", and "Captain Janeway" are all valid FPC publications. If you do not specify a publication for a channel, the default NULL[1] publication is used.

More than one producer can publish the same publication. However, since all FPC clients are considered equal peers, there is no arbitration among multiple producers of the same data. Consequently, if a consumer subscribes to a publication produced by multiple producers, it will receive data from all producers in no specific order.

FPC producers use the `fpcSend` function to send (produce) data to the FPC server. Once connected, producers operate completely independently and asynchronously of the server. Thus, producers can produce data at any time and at any rate.

Consumers

A FPC consumer does the following for each type of data it consumes:

- connects (opens a channel) to the FPC server
- specifies the channel's subscription (i.e., what type of data will be received on the channel)
- specifies the channel's *handler function* (to process new data)
- optionally set the channel's *cache limit* and *read limit* parameters
- reads (consumes) data on the channel

1. The NULL publication is actually the NULL C-string = "" .

As with producers, FPC consumers connect to the FPC server using the FPC library function `fpcConnect`. The consumer and FPC server may both be run on the same computer or on different computers. The only requirement is that they share a TCP/IP network connection.

A subscription is specified in FPC as a simple text *pattern* with the function `fpcSubscribe`. Patterns are matched according to the "wildcard" rules used by UNIX command shells (i.e., globbing)[1] to match filenames. Thus, "data", "d*a", and "[d]a?a" are all valid subscription patterns which will match a publication called "data". You can use the "*" pattern to receive all publications (i.e., "*" matches everything, including the NULL publication). If you do not specify a subscription for a channel, the default NULL subscription is used.[2]

A consumer may have multiple subscriptions on the same channel. However, since FPC does not provide a mechanism for distinguishing data, the consumer, specifically the handler function, must itself be able to identify different data (subscriptions) arriving on the same channel.

If a consumer subscribes to a publication produced by multiple producers, it will receive data from all producers in no specific order[3]. Thus, if message order or message priority is important to your application, you will have to design the message data appropriately (e.g., encode a priority level in the data and have the handler function arbitrate among messages based on this level).

FPC consumers use the `fpcRead` function to poll the FPC server whenever they want to receive data. The FPC server then sends any cached messages (subject to each consumer's *read limit* parameter), which are processed by the *handler function*. The handler will be called one time for each data message that is sent from the FPC server to the consumer. The `fpcRead` function reads (consumes) data synchronously with the server. (For information on reading asynchronously, see `fpcReadBackground` on page 37.)

3.3 FPC Message Data

All communication in FPC uses a single data type: a NULL terminated string (also known as a *C string*). Thus, producers must produce data as strings and consumers must consume data as strings. Because the strings

1. The pattern matching function used by FPC is BSD's `fnmatch`.
2. The NULL subscription will not match any publication.
3. Actually, the data will be arranged in the order it arrives at the FPC server. However, due to network variability (latency, congestion, etc.) it is not possible to predict or guarantee the order data will arrive at any time.

are NULL terminated, it is not possible to send or receive a NULL (ASCII value 0) character. However, all other ASCII characters (values 1-255) can be used in FPC. Additionally, FPC restricts the string length to a maximum of `FPC_DATALEN_MAX`[1] bytes (including the NULL terminator).

1. Currently 1024 bytes (see the `fpc.h` file)

4

Programming with FPC

This chapter provides an in-depth look at programming applications with FPC. We begin with a brief overview of the FPC communication model, the FPC server, and compiling programs with FPC. Then, we describe how to create a simple producer and a simple consumer. Finally, we provide some programming recommendations and discuss the limitations of FPC.

4.1 Overview

FPC Communications Model

Communication with FPC works as follows (see **Chapter 3** for details):

- The FPC server delivers messages between FPC clients (producers and consumers) using channels.
- Each FPC client may have multiple channels connected to multiple FPC servers.
- Producers send messages to the FPC server for deferred (cached) delivery to all subscribing consumers.
- The FPC server caches all messages in a FIFO for each consumer (subject to each consumer's *cache limit* parameter).
- Consumers poll the FPC server for cached messages whenever they want to receive data. The FPC server then sends any cached messages (subject to each consumer's *read limit* parameter), which are processed by a *handler function*. There is one *handler function* per channel.

Starting the FPC server

The FPC server is a stand-alone application which must be operational in order for FPC clients to communicate. To start the FPC server with default parameters, simply run it without additional command line arguments.

For example, if you have are using FPC on a Silicon Graphics workstation and have installed FPC in `/usr/local/fpc`, do the following:

```
% /usr/local/fpc/bin/mips_irix/fpcServer
```

This will start the FPC server running with the default (compiled) port number = 4242. The FPC server also accepts several command line arguments (see page 27 for details).

Compiling programs

To create programs using FPC, you must do the following:

- include the FPC header in your sources: `#include "fpc.h"`
- call the function `fpcInit` before any other FPC library function
- link the appropriate FPC library for your computer

For example, if you are using FPC on a Silicon Graphics workstation and have installed FPC in `/usr/local/fpc`, you could compile the program `blah.c` as follows:

```
% cc -I/usr/local/fpc/include blah.c \
-L/usr/local/fpc/lib/mips_irix -lfpc
```

The FPC distribution includes a number of example programs and a Makefile for compiling these programs.

Windows 95 and NT Libraries

Under Windows NT, three different versions of the FPC library are available:

- libfpc.lib: single-threaded
- libfpcMT.lib: multi-threaded (command line equivalent: /MT)
- libfpcMD.lib: multi-threaded DLL (command line equivalent: /MD)

Make sure to use the version you need, based on your project's requirements. If you are building a client program using multi-thread support, you will need to link against either one of the multi-threaded versions of FPC.

4.2 A simple FPC producer

As described in **Chapter 3**, a FPC producer does the following for each type of data it produces:

1. connect (open a channel) to the FPC server
2. specify the channel's publication (i.e., what type of data will be sent on the channel)
3. send (produce) data on the channel

So, to create a simple FPC producer, we just need to do these steps for a single channel. Let's look at each step in detail.

1. First, we need to connect the producer to the FPC server. We do this using the FPC library function `fpcConnect` which creates a communication channel. For example, we can connect to a FPC server running on host machine `foobar` and port 5000 as follows:

```
int channel;

/* connect to the comm server, failing after 5 attempts */
if (0 > fpcConnect (foobar, 5000, NULL, 5, &channel)) {
  fprintf (stderr, "error: could not connect to foobar:5000\n");
  fpcPerror ("producer");
  return (-1); /* failure */
}
```

Note that we allow `fpcConnect` *to try the connection up to 5 times before giving up. This is useful if the network is unreliable (i.e., the connection between the producer and the server is flaky) or if the server is not yet running. When the connection succeeds,* `fpcConnect` *will set* `channel` *to the opened channel number.*

2. Next, we need to specify what type of data will be published on the channel. We can use the `fpcPublish` function to specify that data of type "blah" will be produced:

```
/* publish messages */
if (0 > fpcPublish (channel, "blah")) {
  fprintf (stderr, "error: could not publish <blah>\n");
  return (-1); /* failure */
}
```

3. Now we are ready to start producing "blah" data. Whenever we want to send data to the server, we use the `fpcSend` function. For example, to send the data "the cow jumped over the moon", we would do this:

```
if (0 > fpcSend (channel, "the cow jumped over the moon")) {
  fpcPerror ("producer");
  return (-1); /* failure */
}
```

You should notice that, if `fpcSend` fails we call `fpcPerror` to print an error message describing the FPC error.

And that's all we need to do to create a working FPC producer!

For a complete producer example, you should look at `producer.c` (contained in the `progs` directory of your FPC distribution).

4.3 A simple FPC consumer

As described in **Chapter 3**, a FPC consumer does the following for each type of data it consumes:

1. connect (open a channel) to the FPC server
2. specify the channel's subscription (i.e., what type of data will be received on the channel)
3. specify the channel's *handler function* (to process new data)
4. optionally set the channel's *cache limit* and *read limit* parameters
5. read (consume) data on the channel

So, to create a simple FPC consumer, we just need to do these steps for a single channel. As we did for the simple FPC producer, let's look at each of these steps in detail.

1. First, we need to connect the consumer to the FPC server. As for the simple FPC producer, we can use `fpcConnect` to connect to a FPC server running on host machine `foobar` and port 5000 as follows:

```
int channel;

/* connect to the comm server, failing after 5 attempts */
if (0 > fpcConnect (foobar, 5000, NULL, 5, &channel)) {
  fprintf (stderr, "error: could not connect to foobar:5000\n");
  fpcPerror ("producer");
  return (-1); /* failure */
}
```

2. Next, we need to specify what type of data will be consumed on the channel. We can use the `fpcSubscribe` function to specify that data of type "blah" will be consumed:

```
/* subscribe to messages of type "blah" */
if (0 > fpcSubscribe (channel, "blah")) {
  fprintf (stderr, "error: could not subscribe to <blah>\n");
  return (-1); /* failure */
}
```

3. Now we need to install a handler function for processing the data messages the consumer receives on the channel. Here's the handler function:

```
int messageHandler (char *pData, void* pPrivateHandlerData)
{
```

```
  if (!pData) {
    fprintf (stderr, "messageHander: error: received NULL data\n");
    return (-1); /* failure */
  }

  printf ("messageHandler: received: %s\n", pData);
  return (0); /* success */
}
```

It's extremely important to note that the message data passed to a handler (via the `pData` ***pointer shown above) should be considered volatile and read-only. FPC does not guarantee that data referenced by this pointer will persist once the handler returns. Thus, if the consumer needs to use this data at a later time, the handler should save a copy.***

Here's how we install the handler using `fpcHandlerRegister`:

```
/* register message handler (no private data) */
if (0 > fpcHandlerRegister (numChannel, messageHandler, (void *)
NULL)) {
  fprintf (stderr, "error: could not register message handler\n");
  return (-1); /* failure */
}
```

Note that if we wanted to pass a private parameter to the handler (e.g., pointer to a storage area, a function, etc.) we would pass it as the last parameter to `fpcHandlerRegister` (instead of NULL).

4. *If needed, we could set the channel's cache limit and read limit parameters using the functions* `fpcLimit` *and* `fpcReadLimit` *respectively. But, if we do not call these functions, the FPC server will use default parameters*[1] *which should work well for most applications.*

5. Now we are ready to start consuming data. Whenever we want to consume any "blah" data that has been sent to the server, we poll the FPC server with the `fpcRead` function:

```
if (0 > fpcRead (channel)) {
   fpcPerror ("consumer");
   return (-1); /* failure */
}
```

If the FPC server has cached any "blah" data messages for us to consume, it will send them. Each message that is received will be processed by our handler function. Note that if for some reason `fpcRead` fails, we call `fpcPerror` to print an error message describing the FPC error.

1. The default cache limit is `FPC_MAXQUEUE_DEFAULT` (100 messages) and the default read limit is `FPC_MAXQUEUE_UNLIMITED` (send all messages).

And that's all we need to do to create a working FPC consumer!

For a complete consumer example, you should look at `consumer.c` (contained in the `progs` directory of your FPC distribution).

4.4 Example Programs

The FPC distribution contains several complete example programs. Here is a brief description of each one:

`benchConsume.c`	benchmark program which tests FPC's message rate (messages sent or received per second) and bandwidth (bytes sent or received per second). Should be started before running `benchProduce.c`
`benchProduce.c`	benchmark program. Start after running `benchConsume.c`
`consumer.c`	consumer which subscribes to a specified type of data. Also demonstrates use of the *cache limit* and *read limit* parameters. Use with `producer.c`
`consumerLic.c`	consumer which demonstrates use of FPC client licensing (contact Fourth Planet for information).
`munch.c`	complex consumer which demonstrates the use of multiple channels, multiple subscriptions, and the *cache limit* and *read limit* parameters. Use with `spew.c`
`producer.c`	producer which publishes a specified type of data. Use with `consumer.c`
`serve-and-consume.c`	consumer and FPC server in a single application. The consumer portion uses multiple subscriptions on a single channel. The server portion runs "in the background" using a process thread[1]. Use with `producer.c`
`serve-and-produce.c`	producer and FPC server in a single application. The producer portion uses multiple channels to produce multiple publications.

1. At this time, FPC provides the capability to create client applications with a built-in FPC server as an unsupported feature. If you are building applications which require or can utilize this feature, please contact Fourth Planet, Inc.

	The server portion runs “in the background” using a process thread. Use with `consumer.c`
`spew.c`	complex producer which demonstrates the use of multiple channels and high-rate, multiple productions. Use with `munch.c`

4.5 Tips and Recommendations

Here are a few tips and recommendations for using FPC:

- *Use multiple consumer channels and the read limit parameter.* This is particularly important when you have handle multiple types of data which have varying message rates. For example, if a consumer has to receive both high-rate “data” and low-rate (but urgent) “control” messages, use two channels and set a small *read limit* on both. This way, the consumer will be always be able to receive “control” messages in a timely manner, even if a flood of “data” messages are in the system.

- ***Remember that producers use the NULL publication by default***. If you do not change this (via `fpcPublish`), the only way for a consumer to receive these publications is to subscribe (with `fpcSubscribe`) to “*”.
- *Producers are serviced by the FPC server in parallel.* Thus, you do not have to worry about “blocking” (i.e., preventing it from handling other clients) the server if you produce messages at high-rates. Consumers, however, can potentially block the server (see the Limitations section below)
- *Take advantage of subscription patterns*. For example, if different producers publish “xfer_data_1”, “data-Z”, and “producer1: data: source A”, a consumer can receive all these publications simply by subscribing to “*data*”. Of course, the same consumer handler must be able to process the three different types of data.
- *Don’t forget that data passed to a handler function is volatile.* If you ever need to use or reference the message data at a time after the handler returns, make sure to save a copy.
- *Set an appropriate cache limit and read limit.* Both of these parameters can greatly impact how an application performs. The example programs `spew.c` and `munch.c` demonstrate how effective this can be for coping with high-rate data (or when you only care about receiving the latest publication of specific data).

- ***Do not specify an unlimited cache limit unless absolutely necessary***. If a consumer specifies a cache limit of unlimited, the FPC server will attemptto satisfy this request by caching messages until it runs out of memory. Thus, if you have a slow consumer receiving publications from a fast producer, make sure the consumer uses a reasonnably sized cache limit such as `FPC_CACHELIMIT_DEFAULT`[1].
- *Use the `fpcVerbosity` function*. If you are having problems debugging your program, change FPC's verbosity with the `fpcVerbosity` function. Additionally, you might find it useful to increase the FPC server verbosity.
- *Check function return values*. It is good programming practice to always check the return values given by FPC library functions. All of the FPC library functions which return a value indicate status in this manner. A non-zero return value indicates an error and that the external variable `fpcErrno` has been set to indicate the cause of the error.
- *Use the `fpcPerror` and `fpcStrerror` functions*. These functions translate the `fpcErrno` value to a descriptive error message. To be of most use, the argument string to `fpcPerror` should include the name of the program that incurred the error.

4.6 Limitations

The following is a list of FPC's known limitations:

- FPC will only operate as well as the underlying TCP/IP network. Since FPC uses TCP[2] sockets to guarantee reliable message transport, FPC performance will suffer if the TCP/IP network is congested, unreliable, or slow.
- The maximum message length (`FPC_DATALEN_MAX`) is 1024. FPC will automatically truncate data strings which exceed this size.
- The FPC server is single-threaded and when a consumer polls calls `fpcRead` (or `fpcReadBackground`) on a channel, the server will only service this channel (i.e., it will ignore all other connected clients) until either (1) all messages are sent or (2) the channel's *read limit* has been reached. Thus, consumers can block the server (from servicing other connected clients) if they attempt to read too many messages at a time.

1. Currently 100 messages (see the `fpc.h` file)
2. Transmission Control Protocol

- The FPC server only supports a maximum of 256 simultaneous channels.
- FPC clients can only open 256 simultaneous channels, even if connected to multiple FPC Servers.
- Producers can only produce one publication per channel (i.e., one type of data per channel). If `fpcPublish` is called more than once for a channel, only the last call will be used (i.e., subsquent publications replace previous publications).
- Consumers can add multiple subscriptions to a channel but cannot remove subscriptions. Thus, if `fpcSubscribe` is called more than once for a channel, all the subscriptions will be added to the channel and the consumer will receive any publication which matches any of the subscriptions.
- FPC messages cannot contain the NULL character (‘\0’), since this character is used to terminate C strings.

4.7 FPC performance

Benchmarking the performance of network communication applications is an extremely difficult task. To some extent, performance is affected by application characteristics (application structure, compiler, code tuning, etc.). These factors, however, are largely overshadowed by network characteristics (interface efficiency, link bandwidth, congestion, physical structure, etc.). For example, transport layer (UDP, TCP) throughput benchmarks can be strongly biased by the underlying network hardware (e.g., the presence of bridging hubs).

Thus, it is often not possible to produce consistent and accurate benchmarks. More troubling is that benchmarks tend to represent ideal performance and do not accurately reflect actual application performance. The bottom line is that, in reality, the only criteria which *really* matters is whether an application performs sufficiently well to accomplish its tasks.

However, to give you some insight into FPC, here are results from the `benchProduce.c` and `benchConsume.c` programs. These figures are not intended as a guarantee of performance, but rather to provide you with guidelines for designing FPC applications. All tests were performed using an increasing number of messages until both `benchProduce` and `benchConsume` generated consistent statistics.

Local performance

In this test, `benchProduce`, `benchConsume` and the FPC server were all run on the same computer.

Table 2 - **FPC throughput (messages per second)**

Computer System	message size (bytes)			
	4	64	256	1024
IBM PC compatible RedHat Linux 4.2 (kernel 2.0.3) Pentium-166, 32 MB RAM	400	400	400	900
Silicon Graphics O2 IRIX 6.3 R5000-180, 256 MB RAM	1800	1500	1200	1100
Silicon Graphics Octane IRIX 6.4 2xR10000-195, 256 MB RAM	4700	4700	3900	3000
Sun UltraSparc 1 Solaris 2.5.1 Ultra1-140, 128 MB RAM	250	500	1000	600

Remote performance

In this test, `benchProduce`, `benchConsume` and the FPC server were run on three identical Silicon Graphics O2 workstations, connected on the same 10 Mbit ethernet segment. However, the location of each program was varied among the three machines.

Table 3 - **FPC throughput (messages per second)**

Configuration (program machine)			message size (bytes)			
FPC Server	bench Produce	bench Consume	4	64	256	1024
A	A	B	1900	1700	1300	600
A	B	A	3100	2600	1900	700
A	B	C	3100	2000	800	400

5

Reference

Error Codes

NAME

fpcServer – FPC server

USAGE

UNIX

fpcServer [-h] [-k] [-p port] [-v level] [config_file]

Windows NT

fpcServer [-h] [-p port] [-v level] [config_file]

PARAMETERS

-h	help
-k	disable keyboard controls (*not available under Windows*)
-p port	set server port to port (default is 4242)
-v level	set verbosity to level (default is 0)
config_file	configuration file specifying above parameters

DESCRIPTION

This command starts the FPC server on the local host computer. For FPC communications to occur, the FPC server must be running and all clients (producers and consumers) must connect to the server port. The server port number can be any legal system port number, which for UNIX systems is any port between 1000 and 65535.[1] The default (compiled) server port is 4242. More than one FPC server can be concurrently run on the same computer. However, each server requires a different port.

The ***-k*** option may be used to disable keyboard controls. This is useful for running the FPC server “in the background” or from command scripts.

The ***-v*** option may be used to control the level of debugging messages generated by the FPC server. Valid levels are: 0 (fatal errors), 1 (status messages), 2 (debug messages), 3 (extremely verbose).

An optional ***config_file*** may be used to specify server parameters. This is an ASCII file containing one or more of the following entries:

```
port number
verbosity level
```

1. Ports lower than 1000 can be used if you have superuser privileges. However, we recommend that you avoid this for system security reasons.

Blank lines and lines beginning with # are ignored. For example, the following may be used to configure a server for port 5000 and verbosity level 2 (debug messages):

```
# set server port to 5000
port 5000

# set verbosity level to 2
verbosity 2
```

EXAMPLES

Start FPC with default parameters (port = 4242, verbosity = 0):

```
% fpcServer
```

Start FPC with port = 5000, verbosity = 2:

```
% fpcServer -p 5000 -v 2
```

Start FPC with port = 5000, verbosity = 2 in the background:

```
% fpcServer -k -p 5000 -v 2 &
```

Start FPC with using a configuration file called `/etc/fpc.conf`

```
% fpcServer /etc/fpc.conf
```

NAME

fpcCacheLimit – limit the number of messages cached by the server

C SPECIFICATION

int **fpcCacheLimit** (int numChannel, int maxQueue);

PARAMETERS

numChannel channel number

maxQueuemax # of messages to queue

DESCRIPTION

fpcCacheLimit limits the maximum number of messages cached by the server into each channel's FIFO queue. The default limit is set (in the server) to `FPC_CACHELIMIT_DEFAULT` (see `fpc.h`). For infinite caching (i.e., until the server runs out of memory), *maxQueue* should be set to `FPC_CACHELIMIT_UNLIMITED`.

On success **fpcCacheLimit** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_INTERNAL`

`FPC_ERROR_NOTINITIALIZED`

SEE ALSO

fpcReadLimit

NAME

fpcClose – close an open channel

C SPECIFICATION

int **fpcClose**(int numChannel);

PARAMETERS

*numChannel*a FPC channel previously opened with **fpcConnect**

DESCRIPTION

fpcClose closes an open FPC channel.

On success **fpcClose** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_CLOSECHANNEL`

SEE ALSO

fpcConnect

NAME

fpcConnect – open a channel from a client to the FPC server

C SPECIFICATION

int **fpcConnect** (char *pHostname, int port, char *pClientName, int maxRetries, int *pNumChannel);

PARAMETERS

pHostname server hostname (if NULL or "", use default host)

port server port (if 0, use default port)

pClientName name of this client (if NULL or "" connect unlicensed)

maxRetries # of times (1..FPC_FOREVER) to retry connection

pNumChannel opened channel

DESCRIPTION

fpcConnect opesn a channel from a client to a FPC server. The connection will be attempted once per second for ***maxRetries*** attempts. If pClientName is given, the server will check for a valid Fourth Planet, Inc. floating license. If a license exists and is available, the connection will succeed. Otherwise, the connection will fail and `fpcErrno` is set to FPC_ERROR_NOLICENSE. Please contact Fourth Planet, Inc. if you wish to use this feature.

If pClientName is NULL or "", no license validation will be performed. and the connection will be attempted using the unlicensed FPC_ANON client name.

On success **fpcConnect** returns 0 and fills ***pNumChannel*** with the opened channel number. Otherwise, it returns -1 and sets `fpcErrno` to one of:

```
FPC_ERROR_CONNECTFAILED

FPC_ERROR_MAXCHANNELS

FPC_ERROR_NEGRETRIES

FPC_ERROR_NOLICENSE

FPC_ERROR_NOTINITIALIZED

FPC_ERROR_NOUNIQUEID
```

SEE ALSO

fpcClose

NAME

fpcHandlerRegister – register a message handler for a channel

C SPECIFICATION

int **fpcHandlerRegister** (int numChannel,
FPC_HANDLER theHandler, void *pPrivateHandlerData);

PARAMETERS

numChannel channel number

theHandler a FPC handler function (see fpc.h)

pPrivateHandlerData pointer to data which will be passed to the handler wwhenever it is called

DESCRIPTION

fpcHandlerRegister registers a message handler for a channel.

On success, **fpcHandlerRegister** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_NOTINITIALIZED`

SEE ALSO

fpcRead, fpcReadAll

NAME

fpcInit – initialize FPC

C SPECIFICATION

int **fpcInit** (void);

PARAMETERS

none

DESCRIPTION

fpcInit initializes FPC. This function must be called before using any other FPC library function. This function may be safely called multiple times, however, only the first call actually initializes FPC.

On success **fpcInit** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to:

```
FPC_ERROR_SIGHND
```

SEE ALSO

fpcShutdown

NAME

fpcPerror – print a message describing the last FPC error

C SPECIFICATION

void **fpcPerror** (const char *pString);

PARAMETERS

pString string to print before error message (optional)

DESCRIPTION

fpcPerror prints FPC error message on the standard error output describing the last error encountered during a call to a FPC library function. The argument string *pString* is printed first, then a colon and a blank, then the message and a newline. However, if *pString* is a null pointer or points to a null string, the colon is not printed. To be of most use, the argument string referenced by *pString* should include the name of the program that incurred the error. The error number is taken from the external variable `fpcErrno`, which is set when errors occur but not cleared when non-erroneous FPC function calls are made.

SEE ALSO

fpcStrerror

NAME

fpcPublish – specify a publication for an open channel

C SPECIFICATION

int **fpcPublish** (int numChannel, char *pPublication);

PARAMETERS

numChannelchannel number

pPublicationpointer to a publication string

DESCRIPTION

fpcPublish informs the server that the client is a producer on *numChannel* and will produce data of type *pPublication* whenever **fpcSend** is called.

On success **fpcPublish** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_INTERNAL`

`FPC_ERROR_NOTINITIALIZED`

SEE ALSO

fpcSend

NAME

fpcRead – read from an open channel

C SPECIFICATION

int **fpcRead** (int numChannel);

PARAMETERS

numChannel an open FPC channel

DESCRIPTION

fpcRead reads from an open channel by polling the server for cached data. The handler for the channel (registered with **fpcHandlerRegister**) is called for each received message.

On success **fpcRead** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_INTERNAL`

`FPC_ERROR_LOSTSERVER`

`FPC_ERROR_READCHANNEL`

`FPC_ERROR_SENDCHANNEL`

SEE ALSO

fpcHandlerRegister, fpcReadAll

NAME

fpcReadBackground – read from an open channel in background

C SPECIFICATION

int **fpcReadBackground** (int numChannel);

PARAMETERS

numChannel an open FPC channel

DESCRIPTION

fpcReadBackground reads from an open channel by polling the server for cached data in the background. A reading thread is started that will poll for data. The handler for the channel (registered with **fpcHandlerRegister**) is called for each received message during the next call to **fpcReadBackground** or **fpcRead**.

You must still call fpcRead or fpcReadBackground in order to process data. The background thread is taking care of "collecting" the data from the network. Unless fpcRead or fpcReadBackground is invoked again, the data will not be passed to the function handler.

On success **fpcReadBackground** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_INTERNAL`

`FPC_ERROR_LOSTSERVER`

`FPC_ERROR_READCHANNEL`

`FPC_ERROR_SENDCHANNEL`

SEE ALSO

fpcHandlerRegister, fpcReadAllBackground

NAME

fpcReadAll – read from all open channels

C SPECIFICATION

int **fpcReadAll** (void);

PARAMETERS

none

DESCRIPTION

fpcReadAll reads from all open channels. The handler for each channel is called for each received message.

On success **fpcReadAll** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_INTERNAL`

`FPC_ERROR_LOSTSERVER`

`FPC_ERROR_READCHANNEL`

`FPC_ERROR_SENDCHANNEL`

SEE ALSO

fpcRead

NAME

fpcReadAllBackground – read from all open channels in the background

C SPECIFICATION

int **fpcReadAllBackground** (void);

PARAMETERS

none

DESCRIPTION

fpcReadAllBackground reads from all open channels in the background. The handler for each channel is called for each received message during the next call to **fpcRead**, **fpcReadAll**, **fpcReadBackground** or **fpcReadAllBackground**. See **fpcReadBackground** for a detailed description of background reading.

On success **fpcReadAllBackground** returns 0. Otherwise, it returns -1 and sets
`fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_INTERNAL`

`FPC_ERROR_LOSTSERVER`

`FPC_ERROR_READCHANNEL`

`FPC_ERROR_SENDCHANNEL`

SEE ALSO

fpcReadBackground

NAME

fpcReadLimit – limit the number of messages for each fpcRead

C SPECIFICATION

int **fpcReadLimit** (int numChannel, int maxRead);

PARAMETERS

numChannel channel number

maxRead max # of messages to retrieve during each fpcRead

DESCRIPTION

fpcReadLimit limits the maximum number of messages retrieved from the server during each **fpcRead**. The default limit is set (in the server) to `FPC_READLIMIT_DEFAULT` (see `fpc.h`). To always retrieve all messages cached by the server during each **fpcRead**, *maxRead* should be set to `FPC_READLIMIT_UNLIMITED`.

On success **fpcReadLimit** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_INTERNAL`

`FPC_ERROR_NOTINITIALIZED`

SEE ALSO

fpcCacheLimit

NAME

fpcSend – send to an open channel

C SPECIFICATION

int **fpcSend** (int numChannel, char *pData);

PARAMETERS

numChannel an open FPC channel

pData pointer to message data (a C string)

DESCRIPTION

fpcSend sends the data message (a C string) pointed to by *pData* to *numChannel*. A maximum of `FPC_DATALEN_MAX` bytes (see `fpc.h`) will be transferred from *pData*.

On success **fpcSend** returns 0. Otherwise, it returns -1 and sets fpcErrno to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_LOSTSERVER`

`FPC_ERROR_SENDCHANNEL`

SEE ALSO

fpcSendAll

NAME

fpcSendAll – send to all open channels

C SPECIFICATION

int **fpcSendAll** (char *pData);

PARAMETERS

pData pointer to message data (a C string)

DESCRIPTION

fpcSendAll sends the data message (a C string) pointed to by *pData* to all open channels. A maximum of `FPC_DATALEN_MAX` bytes (see `fpc.h`) will be transferred from *pData.*

On success **fpcSendAll** returns 0. Otherwise, it returns -1 and sets fpcErrno to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_LOSTSERVER`

`FPC_ERROR_SENDCHANNEL`

SEE ALSO

fpcSend

NAME

fpcShutdown – shutdown FPC

C SPECIFICATION

int **fpcShutdown** (void);

PARAMETERS

none

DESCRIPTION

fpcShutdown closes all open channels.

On success **fpcShutdown** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

```
FPC_ERROR_CLOSECHANNEL
```

SEE ALSO

fpcInit

NAME

fpcSigInstall – install a signal handler

C SPECIFICATION

#include <signal.h>

int **fpcSigInstall** (int signalNum,
FPC_SIGNAL_HANDLER handler);

PARAMETERS

signalNum number of the signal to trap

handler signal handling function

DESCRIPTION

fpcSigInstall installs *handler* to handle *signalNum* signals. *handler* must be a function of type `FPC_SIGNAL_HANDLER` (see `fpc.h`) which accepts an int and other architecture dependent args. For example:

IRIX void signalHandler (int signalNum, int code,
struct sigcontext *sc)

Linux void signalHandler (int signalNum, int code, void *sc)

Solaris void signalHandler (int signalNum, int code,
struct sigaction *sc)

WinNT void signalHandler (int signalNum, int code)

On success **fpcSigInstall** returns 0. Otherwise, it returns -1.

SEE ALSO

fpcSigRemove

NAME

fpcSigRemove – remove a signal handler

C SPECIFICATION

int **fpcSigRemove** (int signalNum);

PARAMETERS

signalNumnumber of the signal to free

DESCRIPTION

fpcSigRemove removes any installed signal handler for *signalNum* and restores the default signal handling.

On success **fpcSigRemove** returns 0. Otherwise, it returns -1.

SEE ALSO

fpcSigInstall

NAME

fpcSleep – suspend execution for a period

C SPECIFICATION

int **fpcSleep** (float seconds);

PARAMETERS

seconds time to sleep

DESCRIPTION

fpcSleep suspends program execution for a period of *seconds*. Because of other system activity, or because of the time spent in processing the call, the actual suspension time may be longer than the amount of time specified.

You should not depend on **fpcSleep** for sleeps less than a millisecond.

On success **fpcSleep** returns 0. Otherwise, it returns -1.

NAME

fpcStrerror – retrieve a message string describing the last FPC error

C SPECIFICATION

char * **fpcStrerror** (void);

PARAMETERS

DESCRIPTION

fpcStrerror retrieves an error message string describing the last error encountered during a call to a FPC library function. This function uses the same set of error messages as **fpcPerror**. The error number is taken from the external variable `fpcErrno`, which is set when errors occur but not cleared when non-erroneous FPC function calls are made.

fpcStrerror always returns a pointer to an error message string. This string should not be overwritten.

SEE ALSO

fpcPerror

NAME

fpcSubscribe – add a subscription for an open channel

C SPECIFICATION

int **fpcSubscribe** (int numChannel, char *pSubscription);

PARAMETERS

numChannelchannel number

pSubscriptionpointer to a subscription pattern (a string containing a UNIX shell-style globbing style pattern

DESCRIPTION

fpcSubscribe informs the server that the client is a consumer on *numChannel* and will consume data of type *pSubscription* whenever **fpcRead** is called.

Consumers can add multiple subscriptions to a channel but cannot remove subscriptions. Thus, if **fpcSubscribe** is called more than once for a channel, all the subscriptions will be added to the channel and the consumer will receive any publication which matches any of the subscriptions.

On success **fpcSubscribe** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_INTERNAL`

`FPC_ERROR_NOTINITIALIZED`

SEE ALSO

fpcRead

NAME

fpcVerbosity – set debugging verbosity

C SPECIFICATION

int **fpcVerbosity** (int verbosity);

PARAMETERS

verbosityFPC verbosity level

DESCRIPTION

fpcVerbosity sets the debugging verbosity to *verbosity* for all FPC functions. Valid levels of *verbosity* are 0 (fatal errors), 1 (status messages), 2 (debug messages), 3 (extremely verbose).

On success **fpcVerbosity** returns 0. Otherwise, it returns -1 and sets `fpcErrno` to one of:

`FPC_ERROR_BADPARAM`

`FPC_ERROR_NOTINITIALIZED`

Error Codes

Table 4 shows all of the error codes and error messages generated by FPC. FPC library functions will return the appropriate error code when an error occurs. The corresponding error message can be retrieved by calling `fpcErrorNo`. Error messages are self-explanatory.

Table 4 - Error Codes and Messages

Error Code	Error Message
FPC_ERROR_NONE	"no FPC error"
FPC_ERROR_NOTINITIALIZED	"FPC not initialized call fpcInit first"
FPC_ERROR_MAXCHANNELS	"exceeded maximum number of FPC channels"
FPC_ERROR_NEGRETRIES	"FPC connect retries cannot be negative"
FPC_ERROR_CONNECTFAILED	"unable to connect to FPC server"
FPC_ERROR_NOUNIQUEID	"unable to obtain unique FPC client ID"
FPC_ERROR_NOLICENSE	"no license available for FPC client"
FPC_ERROR_SIGHND	"unable to install FPC signal handler"
FPC_ERROR_CLOSECHANNEL	"error closing FPC channel"
FPC_ERROR_BADPARAM	"bad parameter"
FPC_ERROR_INTERNAL	"internal FPC error"
FPC_ERROR_SENDCHANNEL	"problem sending to FPC channel"
FPC_ERROR_READCHANNEL	"problem reading from FPC channel"
FPC_ERROR_LOSTSERVER	"server disconnected"

6

Glossary

C string — A sequence of non-NULL ASCII characters (values 1-255) terminated by a NULL character (value 0).

cache limit — Parameter which specifies the maximum number of published data messages the **FPC server** can cache for later delivery to a consumer. Each **channel** has its own cache limit.

channel — The communication link connecting a FPC **client** (either a **producer** or **consumer**) to a **FPC server**.

client — See **FPC client**.

consumer — Process which subscribes to and consumes data.

FPC — Fourth Planet Communicator - a killer product that gives you no-nonsense network communications.

FPC client — A program which communicates with the **FPC server**.

FPC server — Stand-alone program which manages network communication between clients (consumers and producers) using a centralized, **publish and subscribe** message cache.

globbing — The type of pattern matching performed by UNIX shells for matching files. Globbing uses wildcards such as *, ?, and [a-z] to specify patterns.

handler function — **Consumer** function invoked to process each new data message sent from the **FPC server** to the consumer.

producer Process which publishes and produces data.

publication A simple text string specifying (naming) a type of data. For example, “data”, “Temperature: value”, and “Captain Bringdown”.

publish and subscribe A communication model in which some processes publish (announce that they are producing) some named type of data and in which other processes subscribe (announce that they are consuming) some named type of data.

read limit Parameter which specifies the number of data messages sent by the **FPC server** to a consumer in response to a query. Specified per channel.

server See **FPC server**.

server host The machine running the **FPC server**.

string See **C string**.

subscription A simple text pattern following the rules used by UNIX command shells. For example, “blah”, “*data*”, “[a-c]cow??42”.

Index

G

H

I

L

M

O

P

Q

R

S

T