UCRL-MA-108964 Rev. 1

PPC User’s Manual

Stewart A. Brown
Kelly A. Barrett
Dennis Braddy

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Table of Contents

Acknowledgements

A special acknowledgement is due to Lee Minner who helped with the socket and remote execution facility in PPC.

1.0 Introduction

The Portable Process Control library (PPC) is a set of routines to execute and communicate with other processes in environments which permit such operations. The goal is to establish an easy to use interface which can be implemented on a wide variety of platforms. The standard C library I/O interface was taken as a model. In this view, processes are treated as much like files as is possible. In particular, a process is opened and closed as is a file and that means that it is forked by the parent to start with and killed at the end. While a process is open it may be written to and read from with PPC functions which are analogous to the standard C I/O functions **fprintf** and **fgets**. In addition, there are functions to monitor the status of the child processes. These do not have precise analogs to file routines, but since the basic interface has been established by the open/close and read/write routines, these have obvious usages.

PPC is one part of PACT, Portable Application Code Toolkit. See the section **Other PACT Documentation** at the end for more information about PACT.

2.0 PPC Model

This section describes the interprocess communication (IPC) model used in PPC and gives a high level overview of some of the key features of the library. Much of this discussion is involved with UNIX since that environment is one of the richest (and hence most confusing) environments for IPC. We believe that the concepts are fundamental and hence permit PPC to be ported to environments with different semantics. Also the prevalence of the TCP/IP standard makes it a reasonable one to study in the context of distributed IPC.

2.1 Pipes, Sockets, and Pseudo Terminals

In UNIX environments there are three ways for processes to communicate that seem suitable for PPC. They are pipes, sockets, and pseudo terminals (PTY's). Although one might argue that for local processes there is no real advantage to using pipes over sockets or vice versa, PPC supports them both to guard against platforms where one or the other has some non-standard feature. By contrast, PTY's have a crucial significance in that some programs with which a code may wish to communicate behave differently when talking to a terminal than they do when talking to a socket or pipe. FTP is a classic example of this phenomenon. Only sockets are available for remote processes.

When an application opens a child process it must specify the IPC medium. In this way the application developer has control over how the parent and child communicate.

2.2 Interrupt Driven and Multiplexed I/O

When communicating with one or more child processes, an application is often in the position of having or wanting to get input from the terminal or one or more child processes. There are three ways that are commonly used to do this:

1. unblocking the input channels and polling in the application
2. using a multiplex I/O system call (e.g. select or poll)
3. using interrupts on input channels

PPC supports all three of these options.

2.3 Remote File Access

Sometimes applications need to access files on remote systems which cannot be mounted via some standard mechanism such as NFS or AFS. Using the IPC machinery inherent in its design, PPC also supplies a facility to do I/O on remote files. The model here is in two parts.

2.3.1 File I/O Interface

SCORE defines the following set of function pointers:

io_open_hook	default value fopen
io_tell_hook	default value ftell
io_read_hook	default value fread
io_write_hook	default value fwrite
io_setvbuf_hook	default value setvbuf
io_close_hook	default value fclose
io_seek_hook	default value fseek
io_printf_hook	default value fprintf
io_puts_hook	default value fputs
io_getc_hook	default value fgetc
io_ungetc_hook	default value ungetc
io_flush_hook	default value fflush
io_gets_hook	default value fgets

and macros

```
#define io_open    (*io_open_hook)
#define io_setvbuf (*io_setvbuf_hook)
#define io_tell    (*io_tell_hook)
#define io_read    (*io_read_hook)
#define io_write   (*io_write_hook)
#define io_close   (*io_close_hook)
#define io_seek    (*io_seek_hook)
#define io_printf  (*io_printf_hook)
#define io_puts    (*io_puts_hook)
#define io_getc    (*io_getc_hook)
#define io_ungetc  (*io_ungetc_hook)
#define io_flush   (*io_flush_hook)
#define io_gets    (*io_gets_hook)
```

These give a call compatible interface to the major portion of the standard C file I/O library. It also provides a simple way for an application to supply its own functions to make variations on the functionality. In particular, PPC supplies a set of functions to access files on remote hosts. The function **PC_io_connect** toggles between the default set of functions and the remote access versions.

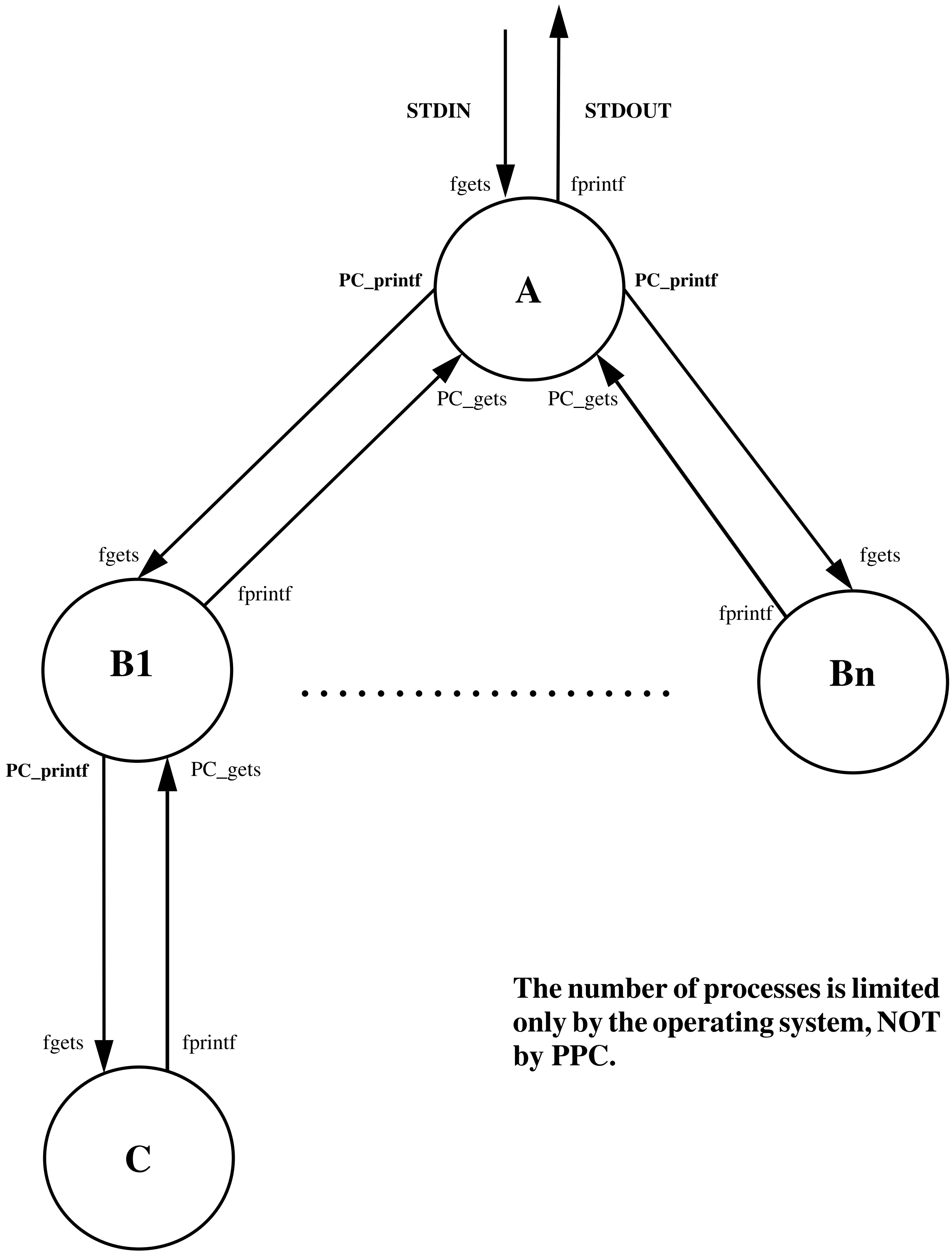
2.3.2 File Access Server

The remote file access functions mentioned in the last section depend on a server running on the remote host. The utililty **pcexec** is that server.

When a request to open a file with a name using the syntax discussed in the section **PCEXEC** indicates a file on a remote host, **pcexec** is started on that host in server mode. It then handles all file access requests made by the other calls in the previous section. Any number of files can be managed per host per user. When the last file that the remote **pcexec** session knows about is closed, **pcexec** exits. NOTE: if your application crashes, there will most likely be an orphaned **pcexec** running on the remote host and you will have to kill it yourself.

2.4 Communication Paths and PPC Routines

Use PPC routines **PC_printf** and **PC_gets** to communicate with child processes. Use C routines **fgets** and **fprintf** to communicate with parent processes.



3.0 The PPC Library

This section describes the PPC library and related information necessary to use PPC in your applications.

3.1 The PPC API

These routines form the interface between applications programs and the low level operating system dependent process control and communication calls. As such they may lack some of the flexibility which the low level routines afford, but they are much easier to use and for most applications they do everything that is necessary.

Most of these routines put an error message into a global variable called **PC_err**. The message contains the name of the function in which the error occurred thus eliminating the need for a cross reference document on error codes. In this way applications programs can check for error conditions themselves and decide in what manner to use the PPC error messages instead of having error messages printed by the system routines. Error messages are not stacked and must be processed by the application before any other PPC calls are made in order to avoid potential overwrites.

int PC_block(PROCESS *pp)

Set the PROCESS *pp* to be blocked (wait for messages).

Input: *pp*, a pointer to a PROCESS.

Output: TRUE, if successful, FALSE otherwise.

int PC_block_fd(int fd)

Set the file descriptor to be blocked (wait for messages).

Input: *fd*, an integer file descriptor.

Output: TRUE, if successful, FALSE otherwise.

int PC_block_file(FILE *fp)

Set the FILE to be blocked (wait for messages).

Input: *fp*, a pointer to a FILE.

Output: TRUE, if successful, FALSE otherwise.

int PC_close(PROCESS *pp)

Kill the process specified by *pp*. This is used to terminate and remove a process when it is not needed (even if the executable process has terminated).

Input: *pp*, a pointer to a PROCESS.

Output: TRUE, if successful, FALSE otherwise.

int PC_echo_off_fd(int fd)

Set the file descriptor input to be unechoed.

Input: *fd*, an integer file descriptor.

Output: TRUE, if successful, FALSE otherwise.

int PC_echo_off_file(FILE *fp)

Set the FILE input to be unechoed.

Input: *fp*, a pointer to a FILE.

Output: TRUE, if successful, FALSE otherwise.

int PC_echo_on_fd(int fd)

Set the file descriptor input to be echoed.

Input: *fd*, an integer file descriptor.

Output: TRUE, if successful, FALSE otherwise.

int PC_echo_on_file(FILE *fp)

Set the FILE input to be echoed.

Input: *fp*, a pointer to a FILE.

Output: TRUE, if successful, FALSE otherwise.

int PC_flush(PROCESS *pp)

Flush the input and output streams for the given process.

Input: *pp*, a pointer to a PROCESS.

Output: TRUE, if successful, FALSE otherwise.

int PC_gets(char *bf, int len, PROCESS *pp)

Read a string from a process into the buffer provided. This behaves exactly like **fgets** except that if there is no input available, NULL is returned without waiting.

Input: *bf*, a pointer to an ASCII string.

len, an integer length of the buffer *bf*.

pp, a pointer to the PROCESS from which to read the input.

Output: a pointer to *bf* if successful or NULL if nothing is available to be read.

int PC_io_callback_fd(int fd, PFVoid fnc)

Change the state of the specified file descriptor so that the specified function will be called when there is input available (interrupt driven or multiplexed).

Input: *fd*, an integer file descriptor.

fnc, a pointer to a function returning nothing which will handle the input.

Output: TRUE, if successful, FALSE otherwise.

int PC_io_callback_file(FILE *fp, PFVoid fnc)

Change the state of the specified FILE so that the specified function will be called when there is input available (interrupt driven or multiplexed).

Input: *fp*, a pointer to a FILE.

fnc, a pointer to a function returning nothing which will handle the input.

Output: TRUE, if successful, FALSE otherwise.

int PC_io_connect(int flag)

If *flag* is PC_REMOTE set the I/O hooks (see **File I/O Interface** section) to the functions for remote file access and if *flag* is PC_LOCAL set them to the standard C library calls for local access.

Input: *flag*, an integer flag.

Output: TRUE, if successful, FALSE otherwise.

PROCESS *PC_open(char **argv, char **envp, char *mode)

Execute a process with command line arguments from *argv* in some (as yet unspecified) mode. The arguments are handled exactly like those of C programs where *argv[0]* is the name of the process/program and the remaining entries in *argv* are null terminated strings each corresponding to a command line argument.

The legal modes are expressed similarly to those for the standard C fopen call:

(’r’ | ’w’ | ’a’)[’p’ | ’s’ | ’t’][’b’]

where

- r Read only (child stdout only connected to parent)
- w Write only (child stdin only connected to parent)
- a Append or read/write (child stdin and stdout connnected to parent)
- p Communicate via pipe
- s Communicate via socket
- t Communicate via PTY
- b Binary data exchanged

For example, for bidirectional communication with a local child via a pseudo TTY in plain ASCII mode use "at". At this point read only and write only are not fully implemented.

The default is "as" for remote processes and "ap" for local processes.

Input: *argv*, an array of pointers to the arguments.

envp, an array of pointers to the environment strings.

mode, an ASCII string indicating the IPC mode.

Output: a pointer to a PROCESS.

int PC_printf(PROCESS *pp, char *fmt, ...)

Write the arguments to the PROCESS *pp* according to the format *fmt*.

Input: *pp*, a pointer to a PROCESS.

fmt, an ASCII string which specifies the output format.

..., the arguments specified in the format.

Output: TRUE, if successful and FALSE otherwise.

int PC_read(void *bf, char *type, size_t ni, PROCESS *pp)

Do a binary read of *ni* items of type *type* from the PROCESS *pp* into the buffer *bf*.

Input: *bf*, a pointer to memory into which the data is to be read.

type, an ASCII string which specifies the data type of items to be read.

ni, an integer (size_t) number of items to be read.

pp, a pointer to a PROCESS.

Output: The number of items successfully read.

int PC_set_attr(PROCESS *pp, int i, int state)

Set the status flags for the specified PROCESS. The flags which can be set are:

PC_LINE	line at a time input
PC_NDELAY	non-blocking I/O
PC_APPEND	append (writes guaranteed at the end)
PC_SYNC	synchronous write option
PC_ASYNC	interrupt-driven I/O for sockets

Input: *pp*, a pointer to a PROCESS.

i, an integer value containing a bit pattern indicating attribute settings.

state, an integer value indicating to set or reset.

Output: ***i*** if successful and -1 otherwise.

int PC_set_fd_attr(int fd, int i, int state)

Set the file status flags for a specified file descriptor. The flag which can be set are:

PC_LINE	line at a time input
PC_NDELAY	non-blocking I/O
PC_APPEND	append (writes guaranteed at the end)
PC_SYNC	synchronous write option
PC_ASYNC	interrupt-driven I/O for sockets

Input: *fd*, an integer file descriptor.

i, an integer value containing a bit pattern indicating attribute settings.

state, an integer value indicating to set or reset.

Output: ***i*** if successful and -1 otherwise.

void PC_signal_handler(int signo)

On receipt of a signal that a child process status has changed, loop forever asking about children with changed status until the system says there are no more.

Input: *signo*, an integer signal number.

Output: None.

int PC_status(PROCESS *pp)

Return the execution status of the PROCESS *pp*.

Input: *pp*, a pointer to a PROCESS.

Output: an integer value, one of RUNNING, STOPPED, EXITED, COREDUMPED, or SIGNALED.

int PC_unblock(PROCESS *pp)

Set the PROCESS to be unblocked (do not wait for messages).

Input: *pp*, a pointer to a PROCESS.

Output: TRUE, if successful, FALSE otherwise.

int PC_unblock_fd(int fd)

Set the file descriptor to be unblocked (do not wait for messages).

Input: *fd*, an integer file descriptor.

Output: TRUE, if successful, FALSE otherwise.

int PC_unblock_file(FILE *fp)

Set the FILE to be unblocked (do not wait for messages).

Input: *fp*, a pointer to a FILE.

Output: TRUE, if successful, FALSE otherwise.

int PC_write(void *bf, char *type, size_t ni, PROCESS *pp)

Do a binary write of *ni* items of type *type* to the PROCESS *pp* from the buffer *bf*.

Input: *bf*, a pointer to memory containing the data to be written.
type, an ASCII string which specifies the data type of items to be written.
ni, an integer (size_t) number of items to be written.
pp, a pointer to a PROCESS.

Output: The number of items successfully written.

3.2 PPC Constants

The following #define'd constants should be used in the contexts indicated:

RUNNING	0	return value of PC_status indicating process running
STOPPED	1	return value of PC_status indicating process stopped
CHANGED	2	return value of PC_status indicating process status changed
EXITED	4	return value of PC_status indicating process exited
COREDUMPED	8	return value of PC_status indicating process crashed
SIGNALED	16	return value of PC_status indicating process signalled
PC_LOCAL	102	value indicating process or file on current CPU
PC_REMOTE	103	value indicating process or file on remote host
USE_PTYS	50	value indicating IPC medium is a pseudo terminal
USE_SOCKETS	51	value indicating IPC medium is a socket
USE_PIPES	52	value indicating IPC medium is a pipe
PC_NDELAY		used with **PC_set_attr** to set non-blocking reads
PC_APPEND		used with **PC_set_attr** to specify writes at end
PC_SYNC		used with **PC_set_attr** to specify synchronous writes
PC_LINE		used with **PC_set_attr** to specify line-at-a-time input

3.3 PPC Variables

PPC provides the following global variables:

char PC_err[]	Buffer for error messages from PPC routines
int PC_io_interrupts_on	Flag which iff TRUE enables I/O interrupts

3.4 Compiling and Loading

To compile your C programs you must use the following

```
#include <ppc.h>
```

in the source files which deal with the library routines.

To link your application you must use the following libraries in the order specified.

-lppc -lpdb -lpml -lscore [-lm ...]

Although this is expressed as if for a UNIX linker, the order would be the same for any system with a single pass linker. The items in [] are optional or system dependent.

Each system has different naming conventions for its libraries and the reader is assumed to understand the appropriate naming conventions as well as knowing how to tell the linker to find the installed PACT libraries on each system that they use.

3.5 Data Structures for PPC

PROCESS

The data structure which underlies PPC is the PROCESS. It is analogous in purpose to the FILE structure used for file I/O. PROCESS structures contain the information necessary for PPC routines to monitor and communicate with child processes. They are passed to PPC routines the way the FILE structure is passed in the standard C file I/O routines.

3.6 Example

The following example is a basic test of PPC in which a small polling loop gets messages from the controlling terminal and passes them to a child process while polling the child process for messages and sending them to the controlling terminal. This program should be entirely transparent to the application.

```
#include "ppc.h"

main(argc, argv, envp)
   int argc;
   char **argv, **envp;
   {PROCESS *pp;
    char s[BIGLINE];

/* open the process */
    if ((pp = PC_open(argv+1, envp, "w")) == NULL)
       {printf("\nFailed to open: %s\n\n", argv[1]);
        exit(1);};

    printf("\nRunning process: %s\n\n", argv[1]);

/* unblock stdin and turn stdout buffering off */
    PC_unblock_file(stdin);
    setbuf(stdout, NULL);

    while (TRUE)
       {PC_err[0] = '\0';
        while (PC_gets(s, BIGLINE, pp) != NULL)
            printf("%s", s);

/* check the status of the process */
        if (PC_status(pp) != RUNNING)
           {printf("\nProcess %s terminated (%d %d)\n\n",
                   argv[1], pp->status, pp->reason);
            break;};

/* get any messages from tty, if available */
        if (fgets(s, BIGLINE, stdin) != NULL)
           PC_printf(pp, "%s", s);
```

```
        if (PC_err[0] != '\0')
           {printf("\nERROR: %s\n\n", PC_err);
            break;};};

/* close the process */
    PC_close(pp);

    printf("\nProcess test %s ended\n\n", argv[1]);

/* turn on blocking for stdin (very important) */
    PC_block_file(stdin);

    exit(0);}
```

4.0 PCEXEC

PPC includes an application program called **pcexec**. It began as a test code for PPC but it has expanded to the point where it has utility in its own right. In fact, **pcexec** does two jobs. First, it can be used to run other programs in any of the modes discussed in the section **The PPC Model**. Second, it acts as a file access server for the remote file access capability in PACT.

4.1 Usage

Run *prog* as a child process:

pcexec [-p | -s | -t] [-i] [-q] *prog* [arg_1 ...]

Act as a file access server on *host* (triggered by -f):

pcexec -f [-l] *host*

The forms for *prog* are:

name	run *name* on local *host*
host:*name*	run *name* on remote *host*
CPU@name	run *name* on processor *CPU*
host:*CPU@name*	*r*un *name* on processor *CPU* on remote *host*

The last two are not yet completed.

The full syntax for *host* is one of the following:

hostname

hostname,*username*

hostname,*username*,*passwd*

Note: whitespace is NOT allowed

Options:

i	Poll explicitly instead of using system call
l	When acting as file server, log transactions to PC_fs.log in home directory
q	Print only messages from the child
p	Use pipes for communications (default for local processes)
s	Use sockets for communications (only mode for remote processes)
t	Use pseudo terminals for communications

All three modes (pipes, sockets, pseudo terminals) are available for local processes.

5.0 Other PACT Documentation

PPC uses functions from the PACT SCORE and PDB libraries. The interested reader is referred to the other PACT documentation:

The list of PACT Documents is:

PACT User's Guide, UCRL-MA-112087

SCORE User's Manual, UCRL-MA-108976 Rev.1

PPC User's Manual UCRL-MA-108964 Rev.1 (this document)

PML User's Manual, UCRL-MA-108965 Rev.1

PDBLib User's Manual, M-270 Rev.2

PGS User's Manual, UCRL-MA-108966 Rev.1

PANACEA User's Manual, M-276 Rev.2

ULTRA II User's Manual, UCRL-MA-108967 Rev.1

PDBDiff User's Manual, UCRL-MA-108975 Rev.1

PDBView User's Manual, UCRL-MA-108968 Rev.1

SX User's Manual, UCRL-MA-112315