IPPC: A LIBRARY FOR INTER-PROGRAM PIPE COMMUNICATION

J.W. Bandler, Q.J. Zhang, G. Simpson and S.H. Chen

SOS-90-10-L

September 1990

IPPC: A LIBRARY FOR INTER-PROGRAM PIPE COMMUNICATION

J.W. Bandler, Q.J. Zhang, G. Simpson and S.H. Chen

Abstract IPPC is a library for inter-program communication in the UNIX environment. Such communication is done with pipes as a vehicle for data transfer. The basic form of communication is between two programs, a parent program and a child program. IPPC also permits communication of one parent with several children and grandchildren. The library has been tested on the Apollo, HP and SUN workstations. This document lists the source code of the IPPC library.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant STR0040923 and in part by Optimization Systems Associates Inc.

The authors are with the Simulation Optimization Systems Research Laboratory and the Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada L8S 4L7. J.W. Bandler, Q.J. Zhang and S.H. Chen are also with Optimization Systems Associates Inc., P.O. Box 8083, Dundas, Ontario, Canada L9H 5E7.

I. INTRODUCTION

IPPC is a library for inter-program pipe communication in the UNIX environment. The basic form of communication is between two programs, a parent program and a child program. IPPC also permits communication of one parent with several children and grandchildren. The library is written in C language. It has been tested on the Apollo DN3500 Domain/IX SR 9.7, HP 9000/340 HP-UX 6.5 and SPARCstation 1 SunOS 4.0.3c.

This document lists the source code of the IPPC library. The users manual together with illustrative examples is found in [1]. The library contains a header file "ippc.h" and a source file "ippc.c". There are 9 functions as listed in Table I.

TABLE I

LIST OF FUNCTIONS IN THE IPPC LIBRARY

	Function	Listing from Page
1	pipe_initialize()	3
2	pipe_open()	3
3	pipe_close()	4
4	pipe_write()	4
5	pipe_write2()	5
6	pipe_read()	5
7	pipe_read2()	6
8	pipe_initialize2()	6
9	pipe_get_cid()	6

II. REFERENCE

[1] J.W. Bandler, Q.J. Zhang, G. Simpson and S.H. Chen, "IPPC: a library for inter-program pipe communication", Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada, Report SOS-90-10-U, 1990.

III. LISTING OF THE IPPC LIBRARY SOURCE CODE

```
/***********************************************************************
* File: ippc.h
* Date: Sep. 10, 1990
*
* Description:  header file for inter-program pipe communications
***********************************************************************/

int pipe_loop_stop;

int pipe_open(),
    pipe_close(),
    pipe_read(),
    pipe_write(),
    pipe_read2(),
    pipe_write2(),
    pipe_initialize2();

#define pipe_initialize(cid) (pipe_write(&pipe_loop_stop, sizeof(int), 1, cid))


/***********************************************************************
 * File: ippc.c
 * Date: Sep. 10, 1990
 *
 * Description: IPPC: inter-program pipe communications
 *    This version has been tested on the Apollo, HP and SUN
 *
 **********************************************************************/

#include <stdio.h>
#include <signal.h>
#include "ippc.h"

#define MAX_N_PIPE_BYTES   4000        /* this may depend on the platform */

#define VACANT          0
#define OCCUPIED        1
#define READ            0
#define WRITE           1
#define ERROR          -1
#define NO_ERROR        0
#define MAX_N_CID     128

int PipeProcessNumber = 0;  /* largest No. of children simultaneously lived */
int PipeProcessID[MAX_N_CID], PipeProcessStatus[MAX_N_CID];
int PipeFDRead[MAX_N_CID], PipeFDWrite[MAX_N_CID];

/* local function */

int pipe_get_cid();

/***********************************************************************
 * int pipe_open ( char *cmd )
 *
 * Description: opens pipes between a parent and a child
 *    The name of the child (executable) is given by *cmd
 *    Returns -1 if error.
 *    Returns a pipe ID number if successful
 *
 **********************************************************************/
int pipe_open(cmd)
char *cmd;
{
   int p[2], q[2], cid;

   cid = pipe_get_cid();
   if (cid == -1) return(ERROR);
   PipeProcessStatus[cid] = OCCUPIED;

   /* turn on the flag for routine pipe_loop_start() */

   pipe_loop_stop = 0;

   if (pipe(p) < 0)     /* open pipe p: data from parent to child */
```

```
      return (ERROR);
   if (pipe(q) < 0)     /* open pipe q: data from child to parent */
      return (ERROR);

   if (!(PipeProcessID[cid] = fork( ))) {
      /* this is the child process */

      close (p[WRITE]);    close (q[READ]);   /* close parent side of pipes */
      close (0);           close (1);         /* close standard I/O  */
      dup(p[READ]);        dup(q[WRITE]);     /* duplicate p[READ] and p[WRITE]
                                                 to standard I/O     */
      close(p[READ]);      close(q[WRITE]);   /* close child side of pipes */

      execl("/bin/sh", "sh", "-c", cmd, 0);   /* execute child process */
      _exit(1);  /* disaster, should never get here */
   }
   else {   /* this is the parent process */
      if (PipeProcessID[cid] == -1) return (ERROR);
      close(p[READ]);     close(q[WRITE]);   /* close child side of pipes */
      PipeFDWrite[cid] = p[WRITE];
      PipeFDRead [cid] = q[READ];
      return (cid);
   }
}


/************************************************************************
 * int pipe_close ( int cid )
 *
 * Description: closes the pipes to the child identified by "cid"
 *    Returns the status value given by wait()
 *
 ************************************************************************/
int pipe_close(cid)
int cid;
{
   void (*hstat) ( ), (*istat) ( ), (*qstat) ( );
   int r, status, p_loop_stop;

   p_loop_stop = 1;   /* turn on flag to stop the infinite loop in child */
   pipe_write(&p_loop_stop, sizeof(int), 1, cid); /* send flag to stop child */

   /* close parent side of pipes in the parent process, i.e., p[WRITE] and
      q[READ] in procedure pipe_open() */

   close(PipeFDRead[cid]);
   close(PipeFDWrite[cid]);
   PipeProcessStatus[cid] = VACANT;

   istat = signal(SIGINT,  SIG_IGN);  /* ignore interrupt signal */
   qstat = signal(SIGQUIT, SIG_IGN);  /* ignore quit signal */
   hstat = signal(SIGHUP,  SIG_IGN);  /* ignore hang up signal */

   while ( (r = wait(&status)) != PipeProcessID[cid] && r != -1) ;

   if (r == -1) status = -1;

   signal(SIGINT,  istat);  /* restore flag for interrupt signal */
   signal(SIGQUIT, qstat);  /* restore flag */
   signal(SIGHUP,  hstat);  /* restore flag */

   return (status);
}


/************************************************************************
 * int pipe_write ( char *a, int s, int n, int cid )
 *
 * Description: sends data via pipe from the parent to a child
 *    The child is identified by cid
 *    Sends n data items of size s
 *    Returns 0 if successful, -1 if error.
 *
 ************************************************************************/
int pipe_write(a, s, n, cid)
char *a;
int cid, s, n;
```

```
{
   register int n_bytes, n_bytes_buffered;

   n_bytes = s * n;

   while (n_bytes > 0) {
      if (n_bytes > MAX_N_PIPE_BYTES) n_bytes_buffered = MAX_N_PIPE_BYTES;
      else n_bytes_buffered = n_bytes;

      n_bytes_buffered = write(PipeFDWrite[cid], a, n_bytes_buffered);
      if (n_bytes_buffered == -1) return(ERROR);

      n_bytes -= n_bytes_buffered;
      a += n_bytes_buffered;
   }
   return(NO_ERROR);
}


/************************************************************************
 * int pipe_write2 ( char *a, int s, int n )
 *
 * Description: sends data via pipe from a child to its parent
 *    Sends n data items of size s
 *    Returns 0 if successful, -1 if error.
 *
 ************************************************************************/
int pipe_write2(a, s, n)
char *a;
int s, n;
{
   register int max_n_items, n_items;

   max_n_items = MAX_N_PIPE_BYTES / s;

   while (n > 0) {
      if (n > max_n_items) n_items = max_n_items;
      else n_items = n;

      n_items = fwrite(a, s, n_items, stdout);
      if (n_items == -1) return(ERROR);
      fflush(stdout);

      n -= n_items;
      a += (n_items * s);
   }
   return(NO_ERROR);
}


/************************************************************************
 * int pipe_read ( char *a, int s, int n, int cid )
 *
 * Description: reads data via pipe from a child to the parent
 *    The child is identified by cid
 *    Reads n data items of size s
 *    Returns 0 if successful, -1 if error.
 *
 ************************************************************************/
int pipe_read(a, s, n, cid)
char *a;
int cid, s, n;
{
   register int n_bytes, n_bytes_buffered;

   n_bytes = s * n;

   while (n_bytes > 0) {
      if (n_bytes > MAX_N_PIPE_BYTES) n_bytes_buffered = MAX_N_PIPE_BYTES;
      else n_bytes_buffered = n_bytes;

      n_bytes_buffered = read(PipeFDRead[cid], a, n_bytes_buffered);
      if (n_bytes_buffered == -1) return(ERROR);

      n_bytes -= n_bytes_buffered;
      a += n_bytes_buffered;
   }
   return(NO_ERROR);
}


/***************************************************************************
 * int pipe_read2 ( char *a, int s, int n )
 *
 * Description: reads data via pipe from the parent to the child
 *    Reads n data items of size s
 *    Returns 0 if successful, -1 if error.
 *
 ***************************************************************************/
int pipe_read2(a, s, n)
char *a;
int s, n;
{
   register int max_n_items, n_items;

   max_n_items = MAX_N_PIPE_BYTES / s;

   while (n > 0) {
      if (n > max_n_items) n_items = max_n_items;
      else n_items = n;

      n_items = fread(a, s, n_items, stdin);
      if (n_items == -1) return(ERROR);

      n -= n_items;
      a += (n_items * s);
   }
   return(NO_ERROR);
}


/***************************************************************************
 * int pipe_initialize2 ()
 *
 * Description: reads the loop-stopping signal from the parent
 *    The child commits suicide when the signal is set to 1
 *
 ***************************************************************************/
int pipe_initialize2()
{
   if (pipe_read2(&pipe_loop_stop, sizeof(int), 1) == ERROR)
      return(ERROR);
   if (pipe_loop_stop == 1) exit();
   return(NO_ERROR);
}


/***************************************************************************
 * int pipe_get_cid ()
 *
 * Description: local function not to be used by users.
 *              Get the lowest available child id.
 *              Return value is -1 if error occurs, otherwise
 *              return value is the child id.
 *
 ***************************************************************************/
int pipe_get_cid()
{
   int cid;

   if (PipeProcessNumber > 0) {
         for (cid = 1; cid < PipeProcessNumber; cid++) {
              if (PipeProcessStatus[cid] == VACANT) return(cid);
         }
   }
   PipeProcessNumber++;
   if (PipeProcessNumber >= MAX_N_CID) return(ERROR);
   cid = PipeProcessNumber;
   return(cid);
}
```