CFLOW User's Guide

User Documentation
Interactive Powerflow Program

Prepared by:

Planning Methods Section
System Planning
Bonneville Power Administration
P.O. Box 3621 — TEOS
Portland, OR 97208

PLANNING METHODS SECTION
SYSTEM PLANNING
BONNEVILLE POWER ADMINISTRATION
P.O. BOX 3621, M/S - TEOS
PORTLAND, OR 97208

NOTICE TO NON-BPA USERS

The Bonneville Power Administration (BPA) releases BPA-developed computer programs under the following conditions:

1. BPA does not charge for program development costs; however, a fee to cover costs incurred in answering inquiries is assessed against the organization receiving the material. This fee typically includes costs for personnel and computer resources, reproduction, shipping, and postage.
2. BPA cannot provide assistance with conversion to other computers or consulting services to the program users.
3. In consideration of receipt and acceptance of these programs — or portions thereof — if sold, assigned, or transferred to another organization, you and your organization agree to advise any third-party recipient in writing that the program(s) and/or documentation are in the public domain and available from BPA. The intent of this agreement is to ensure that BPA-developed or supplied programs, and/or documentation, whether in whole or in part, that are in the public domain, are identified as such to recipients.

"LEGAL NOTICE"

Neither BPA nor any person acting on behalf of BPA:

1. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe upon privately owned rights; or
2. Assumes any liability with respect to the use of, or for damages resulting from the use of any information, apparatus, method or process disclosed in this report.

TABLE OF CONTENTS

INTRODUCTION

WRITING CFLOW PROGRAMS

CFLOW LIBRARY FUNCTIONS

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

With the addition of CFLOW to the Interactive Power Flow Program (IPF), users can access the bus and branch data within the powerflow data structures in a highly flexible way. Once data is retrieved from the IPF "database engine", it can be manipulated within the user-written CFLOW C program and either output to a file, screen, or plotter, or sent back to the IPF "database engine" itself.

CFLOW performs analogous functions to the WSCC COPE language: Computationally Oriented Programming Environment. COPE is a stand-alone language integrated with the WSCC equivalent of BPA's IPF, the Interactive Powerflow System (IPS). CFLOW is a library of C language functions, that have "sister" functions within the IPF. The function pairs form a "remote procedure call" library. All the power of the C language is immediately available to CFLOW users, whereas COPE users are limited to the COPE language and environment. Another example of similar functionality is PTI's IPLAN which is interpreted similar to the way COPE is.

CFLOW is more powerful and flexible than COPE. A CFLOW program is a totally separate process running concurrently with the "powerflow solution and database server" process and communicates with it via an IPC (Inter Process Communication) protocol, so CFLOW requires an operating system capable of multi-processing and that supports "sockets" for IPC.

In order to use CFLOW, you must have some knowledge of the C language, and a C compiler on your computer platform. Once a CFLOW program has been written, compiled, linked, and debugged, it is stored as an executable which can be run from the command line or from the GUI using the "PROCESS -- CFLOW" menu option.

The CFLOW product is completely compatible with ANSI C language and compilers and portable to all environments that have an ANSI C compiler. To use CFLOW, one writes a program in standard C language.

1.2 CHAPTER SUMMARIES

Chapter 1 Provides a description of CFLOW and its benefits and gives C language book recommendations.

Chapter 2 Provides a description of the CFLOW program development cycle and the development environment. CFLOW has been developed for the VMS and UNIX platforms, and can be ported to any other platform (hardware/software

environment) that the IPF application is ported to.

Chapter 3 Shows you how to use CFLOW functions to write IPF reports.

Chapter 4 Describes CFLOW functions.

1.3 AUDIENCE

This manual assumes that you are a beginning to mid-level C language programmer. This means you should have successfully written programs in languages such as FORTRAN, BASIC, Pascal, or C. These need not be complex programs, simply programs such as you would be required to complete in an undergraduate college programming course. If you already know C, you are ready to write CFLOW programs. If not, you should probably take a class, invest in a computer tutorial course, and/or spend some time with a good C language book.

1.4 RECOMMENDED BOOKS

The following C programming books are recommended for those users needing introductory or refresher information:

- Brakakati, Nabajyoti. *The Waite Group's Microsoft C Bible*. Howard W. Sams & Company, 1988. This MS-DOS environment reference book clearly describes ANSI C compatibility for each function.
- Harbison, Samuel P. and Guy L. Steele. *C: A Reference Manual*. 3rd ed. Prentice-Hall, 1991. This book shows ANSI C facilities contrasted with traditional or alternate facilities. If you are well acquainted with C programming, but want to make sure your program complies with ANSI C, look here.
- Johnsonbaugh, Richard and Martin Kalin. *Applications Programming in ANSI C*. MacMillan, 1990. This is a textbook used in beginning undergraduate college courses.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Prentice-Hall, 1988. This is the standard book for learning the language, updated to a second edition. Experienced programmers will do well with this book. Beginners should use Kochan's book.
- Kernighan, Brian W. and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984. This book describes how to develop programs in the UNIX operating system.
- Kochan, Stephen G. *Programming in ANSI C*. Howard W. Sams & Company, 1988. This book is a comprehensive tutorial for the beginning programmer.
- Plauger, P. J. *The Standard C Library*. Prentice-Hall, 1992. This book shows you how to use the standard ANSI and ISO C library functions. It provides code examples for implementing many of the library functions.

1.5 A NOTE ON TYPOGRAPHICAL CONVENTIONS

C language identifiers, variables, function names, and all information that you type or that might appear on your screen are all printed in the Courier plain font, for example, `while`, `basekv`, and `pf_cflow_init`.

CHAPTER 2

CFLOW PROGRAM DEVELOPMENT

This chapter provides a brief introduction to CFLOW program development. Refer to the C programming books cited in Chapter 1 for more information.

2.1 CREATING A PROGRAM

CFLOW is a library of functions that a C program can link to (with an object file linker) to access IPF data and control IPF execution. The program you write is a C program. CFLOW source programs are created using a text editor such as EDT (VMS) or vi (UNIX).

The program lines can be entered in “free format,” since there are no column restrictions. Indentation is recommended when designing nested logical constructs to reduce logic errors and enhance readability.

For example:

```
if(condition) {/* beginning of first “if” block */
   if(condition) {/* beginning of second “if” block */
      statement;
      statement;
   }/* end of second “if” block */
   statement;
   statement;
}/* end of first “if” block */
statement;
```

2.2 HEADER FILES

Include the CFLOW header file, called `cflowlib.h`, in each of your CFLOW files.

2.3 COMPILING A CFLOW PROGRAM

Use a standard C compiler and standard linker to compile your CFLOW C program. You need to know how to do this for the computer system you are working on. Under the Unix operating system, this is usually done with a "`make`" file. The file which makes the sample programs is included with IPF. You will want to change this for your own platform, environment, directory structure, and CFLOW programs. Refer to your system documentation for more guidance.

2.4 RUNNING A CFLOW PROGRAM

2.4.1 FROM THE GUI

There are two ways to execute a CFLOW program from the GUI: Either in its own terminal window, or in the same terminal window that the GUI is running in (background).

CFLOW programs that are run in the same terminal window as the GUI that read/write to `stdin` or `stdout` (i.e. use `readln` or `printf`) have their I/O intermixed in the same terminal window as the GUI and IPF server. The background mode should usually only be used for programs that generate a report or other output to a file.

To run a CFLOW program from the GUI, you start up the GUI, and set up whatever conditions are required for your program to work, such as loading and solving a case, if the program is designed to report on a currently loaded case. Then select Process - Run CFLOW. You will get a file selection window. Double-click on a directory name, or change the filter field and click the Filter button, to change the file list. When you see the file you want to run (your executable CFLOW program), select it, and it will appear in the Selection field. Select either Window (default) or Background, and, if your program has command line arguments, type these into the CFLOW Program Arguments field. Then click the Launch CFLOW button. The Wait field (default 30 seconds) is provided just in case your program takes longer than 30 seconds to start up. The wait time is the length of time that the GUI/IPFSRV will wait for the CFLOW program to start up and establish a socket (inter-process communication) connection. The value can be set from 15 to 300 seconds. If the CFLOW program has not connected within the wait time, then the CFLOW run is aborted and control is returned to the GUI.

The program is run synchronously. This means that you cannot use the GUI until the CFLOW program is finished running. If the CFLOW program fails, control is returned to the GUI. However, if the CFLOW program hangs (as in an infinite loop), you need to kill the CFLOW process through operating system resources. (For example, on VMS this can be done with the stop process command, on UNIX, this can be done with the `kill` command.) See your computer system documentation or your system administrator for help.

When the program completes, the CFLOW window goes away and control returns to the GUI. You can then load a different file, or make other changes, and select Process - Run CFLOW again to rerun. The scripts `run_cflow_win` (window) and `run_cflow_bg` (background) are used to run the CFLOW program. These scripts can be customized for your system.

2.4.2 FROM IPFBAT

For batch, also called background or terminal window interactive processing, the `ipfbat` program is provided. This program reads a control file rather than connecting to and receiving commands from the GUI process.

A CFLOW program can be run by including the following command in the control file:

```
/CFLOW, PROGRAM =
[ directory path ] < CFLOW executable file or script file >
[ , WINDOW ]
[ , WAIT = < max wait time for socket connect > ]
[ , ARGS = < command line arguments for the specified program > ]
```

The brackets ([ ]) denote optional items. The command is free format with the restriction that any "word" (like the [path] < file >) must be all on the same line (not continued on the next line) with no imbedded blanks or any of the following: ",=\n" . On a UNIX system, for example, you can use the following:

```
/CFLOW, PROGRAM = my_cflow_program
```

This "launches" the program if it is in your directory search path.

`ARGS =` is required only if the CFLOW program requires command line arguments. Without the WINDOW option, any I/O from the CFLOW program goes to standard input or standard output, and will be to and from the same terminal window that the `ipfbat` program is run from (intermixed with any I/O from the `ipfbat` program). The scripts `run_cflow_win` (window) and `run_cflow_bg` (background) are use to run the CFLOW program. These scripts can be customized for your system.

2.4.3 FROM COMMAND LINE

To run your CFLOW program from the command line, simply run the program as you would any other program. If your system needs some setup to be done to allow your program to accept command line arguments (VMS), you can use the script `run_cflow_bg` , which can be customized for your system. For example, the following could be used:

```
my_cflow_program [ arg1 ] [ arg2 ] ...

run_cflow_bg my_cflow_program  [ arg1 ] [ arg2 ] ...
```

The CFLOW program will launch the IPFSRV program and establish a connection. The script `run_ipfsrv_cf` is used to run the IPFSRV program. This script can be customized for your system. There are two optional command line arguments that are helpful in debugging. These options must precede any of the arguments for the CFLOW program.

`-n`	**n**oserver, do not launch IPFSRV
`-w < wait time >`	max **w**ait time for socket connect

2.5 ENVIRONMENT

CFLOW is available for every environment which supports the GUI version of IPF.

2.6 DEBUGGING

Use the standard system debugger for your computer system to debug CFLOW C programs.

Although it is possible to do debugging using the GUI/IPFSRV or the IPFBAT programs, it is recommended that you debug by running your CFLOW program from the command line.

If your program expects some setup (i.e. a case already loaded and solved), because your program is used like a subroutine that generates a report or other output based on whatever is currently there, then for debugging purposes, you can create a function that does the setup (e.g. load base, apply changes, solve) and call that function at the beginning of your program. When the program is debugged, you can "comment out" the call to the setup function.

The `pf_cflow_init()` function uses the C library function `system("command line")` to launch the script `run_ipfsrv_cf` which in turn runs the IPFSRV program with the output redirected to a file. There is a default wait (time out) of 30 seconds for the IPFSRV program to extablish a socket connection, but this can be increased, if needed, to up to 300 seconds with the -w option. See running from the command line above. In general, when debugging, you want to "step over" (versus "step into") the `pf_cflow_init()` function, however, if you decide to step through the function, be aware that part of the code has the time out in effect and will cause a "failed connect" to occur if you proceed too leisurely.

If you experience problems with debugging that you suspect are related to the "system" call that launches the IPFSRV program, you can use the -n (noserver) option as follows to debug from two windows:

In one window run your CFLOW program.

```
my_cflow  -n  -w 300  [ arg1 ] [ ... ]
```

Wait for the message "using socket nnnn" (this will happen when the `pf_cflow_init()` function is executed), then in another window run the IPFSRV program.

```
ipfsrv -socket nnnn
```

where "nnnn" is the same as what the CFLOW program stated. The socket connection should happen in less than 10 seconds (usually a couple of seconds) depending upon the speed of your system.

You can then debug your program in one window while the IPFSRV program runs in the other.

CHAPTER 3

WRITING CFLOW PROGRAMS

This chapter presents a general discussion of CFLOW functions and includes some sample CFLOW applications with comments.

3.1 CFLOW AND C

When you write a CFLOW program, you will really be writing a program in the C language, which will be calling various CFLOW functions. You must have a C compiler and a linker/loader on your system in order to write programs. Also, you must conform to the syntax standards of your compiler (ANSI standard recommended).

If you have never taken a class in C, or have not recently written any C programs, you will need a reference manual.

3.2 CFLOW FUNCTIONS OVERVIEW

The CFLOW library is a set of functions, written in the C language, that allows access to IPF data. Behind the scenes, the routines communicate with the IPF program via an interprocess communication channel known as a *socket* (similar to a *pipe* or *stream*). The routines are organized, as much as possible, as database access routines, since the powerflow program is playing the role of database and compute "server."

There are four major classes of functions:

- Simple IPF "command" functions.
- Record-oriented IPF functions.
- Buffer-oriented IPF "command" functions.
- Utility functions and local data translation.

The library currently accesses only the Powerflow program. In the future, a similar approach could be used to provide a CFLOW interface for other programs. The various functions are documented in Chapter 4. All of those that access Powerflow start with `pf_`. All functions return a non-zero integer for an error condition and a zero for successful completion.

3.3 SIMPLE POWERFLOW COMMAND FUNCTIONS

Examples of simple functions are `pf_area_of_zone`, `pf_del_zone`, `pf_rename_area`, `pf_rename_zone`, and `pf_rename_bus`. These functions perform an operation that requires little

or no input data, other than a command, and usually return only a status, or a single piece of data such as `pf_area_of_zone` does.

3.4 RECORD-ORIENTED FUNCTIONS

Examples of record-oriented functions are `pf_rec_bus` and `pf_rec_cbus`. These functions use C language structures to manipulate a Powerflow *record*. Both generic and record-type specific structure definitions are provided, so that field names specific to the record type can be used for a little better "self-documenting" code. For example, the variable containing TAP2 for a transformer contains B2 for an E type line, and the minimum phase shift for a type RM regulating phase shifter. This same variable can be accessed by using the names `r.i.branch.tap2, r.i.pf_E.b2`, and `r.i.pf_RM.min_phase_shift_deg`.

The functions all use an *action code* to specify what is to be done with the record, such as D for delete, F2 to retrieve the first branch record associated with two named buses, and O to retrieve solution (output) data for a bus or branch.

3.5 BUFFER-ORIENTED OPERATIONS

The buffer-oriented operations are all accessed through one function: `pf_cflow_ipc`. This function sends a buffer to Powerflow containing a command, command options, and usually input data. A buffer is returned that contains the results of the request. Any command that is in the IPF Advanced User Manual can be put in the buffer, with records separated by the '\n' (linefeed) character. Much of the data sent and received in the buffers is WSCC-formatted data.

3.6 UTILITY FUNCTIONS

Most of the utility functions are provided to form an "abstraction layer" between your program logic and WSCC formatted ascii records. There are functions to translate between the C structures and WSCC ascii records, as well as functions to initialize the C structures.

3.7 A SIMPLE REPORT FROM POWERFLOW

The ANSI C program listed below was derived from a COPE program. This program illustrates how the CFLOW library and ANSI standard C may accomplish many of the same tasks as COPE programs. Detailed discussion follows the program code.

The following program outputs a Shunt Reactive Summary report to the screen. A loaded base case in the Powerflow "database engine" provides the data for the report.

```
/* shreac.c
 *  The following is an example of a COPE program, re-written
 *  in the "C" programming language using the CFLOW library.
 *  It does a Shunt Reactive Summary report on the currently
 *  loaded case, for a user-entered zone.  Output to gui T/W.
 */
```

```
#include <stdio.h>
#include <string.h>
/*#include <cflowlib.h> /* use this form if cflowlib.h is in the
                          "standard" include area for  C                    */
#include "cflowlib.h"  /* use this form if "cflowlib.h" is in a "user
                        library" include area.  Your compile procedure
                        should use the "-I or /I" option that specifies
                        the path to "cflowlib.h"                        */


int readln( char *s, int lim )  /* Function to read input from the T/W. */
{
   int   i;
   char  c;

   for ( i=0; i < lim - 1  &&  ( c = getchar() ) != EOF  &&  c != '\n'; ++i )
      s[i] = c;
   s[i] = '\0';
   return i;
}

main( int argc, char *argv[] )  /* Main Program */
{
   pf_rec  r;      /* CFLOW structure */
   char zn[3];
   int    error, status ;
   float   q_avail_react_tot, q_avail_cap_tot,
           q_used_react_tot, q_used_cap_tot,
           q_unused_react_tot, q_unused_cap_tot,
           q_unsched_react_tot, q_unsched_cap_tot;

   pf_cflow_init( argc, argv );  /* IPC connection function, required. */

/* Ask user for zone to report */
   printf("Enter Zone to report Shunt Reactive Summary > ");
   readln( zn, sizeof(zn) );
   zn[sizeof(zn)] = '\0';

   printf("\n\n Shunt Reactive Summary for Zone %s \n\n",zn);
   printf(" Avail_caps Avail_reac Used_caps Used_reac Unus_caps Unus_rx Unsch_caps
Unsch_rx\n\n");

      q_avail_react_tot   = q_avail_cap_tot   =
      q_used_react_tot    = q_used_cap_tot    =
      q_unused_react_tot  = q_unused_cap_tot  =
      q_unsched_react_tot = q_unsched_cap_tot = 0.0;

      /* Compute zone quantities */

      error = pf_rec_bus( &r, "F" ); /* get first bus in case */
      status = pf_rec_bus( &r, "O" ); /* get solution data for first bus */
      while ( !error && !status ) {         /* Loop through all buses in case */
         if ( strcmp(r.i.ACbus.zone, zn )==0)
         {                                     /* If bus is in the zone */
            q_avail_react_tot   += r.s.ACbus.Bshunt_sch_rx;
```

```
            q_used_react_tot    += r.s.ACbus.Bshunt_used_rx;
            q_avail_cap_tot     += r.s.ACbus.Bshunt_sch_cap;
            q_used_cap_tot      += r.s.ACbus.Bshunt_used_cap;
            if( r.s.ACbus.Qunsch < 0 ) {
               q_unsched_react_tot -= r.s.ACbus.Qunsch;
            }
            else
            {
               q_unsched_cap_tot   += r.s.ACbus.Qunsch;
            }
         }
         error = pf_rec_bus( &r, "N" ); /* get next bus in case */
         status = pf_rec_bus( &r, "O" ); /* get solution data for next bus */
      }

      q_unused_react_tot = q_avail_react_tot - q_used_react_tot;
      q_unused_cap_tot   = q_avail_cap_tot   - q_used_cap_tot;

      /* Print zone summary */

   printf("   %6.1f    %6.1f   %6.1f   %6.1f   %6.1f   %6.1f   %6.1f   %6.1f \n\n",
            q_avail_react_tot,   q_avail_cap_tot,
            q_used_react_tot,    q_used_cap_tot,
            q_unused_react_tot,  q_unused_cap_tot,
            q_unsched_react_tot, q_unsched_cap_tot );

    pf_cflow_exit();      /* Drop IPC connection  */
}
```

C programs usually have header files "included" somewhere near the beginning after a comment header. Header files may be standard or programmer-created. One standard header file required is `stdio.h` if the program does any I/O operations. Various macros and definitions make up these files. `string.h` is also a standard header file. `cflowlib.h` is a special header file for the CFLOW library. All CFLOW programs must include this file.

All C programs must include a call to `main`, which is where program execution starts.

Immediately after main follow a series of declarations of variables local to `main`. The variable `r` is a special CFLOW structure of type `pf_rec`. This is the basic powerflow record structure used to retrieve both input and output data for all types of records.

A three character array representing a two character bus zone code follows. (Strings should always be declared one larger than needed to account for the terminating `NULL` (`\0`).) Two integer variables, `error` and `status`, are declared next. These will be used to store the return value from calls to pf_rec_bus. The variables to collect the zone total quantities are declared as `float` (single precision real numbers).

`pf_cflow_init` is called to establish the IPC socket connection, which the program will use to communicate with either the IPFSRV or IPFBAT programs.

We print a question to the terminal window, and use `readln` to retrieve the user selection of a zone

to report on, and make sure the string is null-terminated by storing a null in the last element of the array (remember that C indexes array elements from 0 to n-1). The input zone id is echoed back in the heading of the report, and the floating point variables are initialized.

Now we begin the actual processing. The first bus record is retrieved by calling `pf_rec_bus` with an action code of "F" (for First). This stores the input record data for the first bus in the currently loaded system in the local structure `r`. But we want the output (solution) quantities. So we call `pf_rec_bus` again, with action code "O" (for Output). The necessary id fields have been stored in `r` by the first call, and these are passed back to `ipfsrv` so it knows what bus you want output values for.

A while loop now executes. The purpose of the while loop is to sequentially access Powerflow bus records and gather floating point data related to the shunt reactance. `pf_rec_bus` initially used action code "`F`" to go to the first bus record in the base case. After this, `pf_rec_bus` uses action code "`N`" to retrieve the next bus record. When the end of all records has been reached, `pf_rec_bus` returns a -1, which causes the while loop to terminate.

Since only records from the user-specified zone are desired in this program, a test is first performed on each bus record to see if it has the correct zone. The program could be made more efficient by not bothering to retrieve output values except for the right buses; however it has been left this way for simplicity in providing an example. If the bus is in the right zone, then the appropriate floating point values are totaled in assignment statements. This program assumes that the zone has no dc buses; if there are any, then the dc solution variables stored in the same fields as the ac shunt would give you weird results, to say the least! Of course, it would be possible to also test for bus type, along with testing for the zone, in order to avoid this problem.

When the loop teminates, the reactance totals are printed to the screen with `printf` statements formatted for decimal output. Then we exit, and release the socket connection, by calling `pf_cflow_exit`.

3.8 REAL WORLD CFLOW PROGRAMS

3.8.1 Standard Line Flow Summary

The `slfs.c` program was translated from a COPE procedure used by WSCC Technical Staff. It reads an input data file of headings and branches to be reported, looks these up in the currently loaded solved system, and sends the report to a file. The report includes the input headings, and group totals where called for, in an attractive format for printing.

This is what the input data file looks like. Heading lines are those with neither 'LIN' or 'TOT' on them. They are printed as encountered. 'LIN' cards identify a branch to be reported; 'TOT' cards call for a total to be printed. The "2" in column 5 indicates that the reverse flow is to be reported.

```
CANADA AND NORTHWEST
--------------------
```

```
  1. Alberta - British Columbia
LIN   LANGDON  500 CBK500   500
LIN   LNGDN500 500 CRANBROK 500
LIN   LNGDN500 500 CBK500   500
LIN 2 POCA TAP 138 EMC138   138
TOTAL
  2. Canada - Northwest
LIN   ING500   500 CUSTER W 500
LIN   INGLEDOW 500 CUSTER W 500
LIN 2 NLY230   230 BOUNDARY 230
LIN 2 NELWAY   230 BOUNDARY 230
LIN 2 NLYPHS   230 BOUNDARY 230
LIN   SELPHS-1 230 MARSHALL 230
LIN   SELPHS-2 230 MARSHALL 230
LIN   SELPHS-1 230 BEACON N 230
LIN   SELPHS-2 230 BEACON S 230
TOTAL
  3. Northwest - California
LIN   MALIN    500 ROUND MT500.
LIN   DELTA    115 CASCADE 115.
LIN   CAPTJACK 500 OLINDA  500.
TOTAL
  4. Celilo - Sylmar
LIN 2 SYLMAR2I106. SYLMARLA230.
LIN 2 SYLMAR1I106. SYLMARLA230.
LIN 2 SYLMAR2R106. SYLMARLA230.
LIN 2 SYLMAR1R106. SYLMARLA230.
TOTAL
NORTHEAST
---------
  1. MPC High Line
LIN 2 CONRAD   115 CUT BANK115.
LIN   GT FALLS 161 HAVRE   161.
            .
            .
            .
```

The program slfs.c prompts the user for the output file name to put the line flow listing in. The input file name is hard-coded in the program.

```
/*  slfs.c

      This CFLOW procedure looks up flows and creates a report
      of flows between buses as listed in an input data file.

      Before the CFLOW procedure is called, a solved power flow case
      must be resident in the powerflow server.

      This CFLOW procedure prompts the user for two file names:
           the output file name to put the line flow listing in,
           the input file name of a file to get line data from.

      The input file has a LIN card for each branch to be monitored.
      If there are multiple lines between the same buses, slfs.c
      picks up all lines.  A TOT card flags printing of total flow
```

```
      since the last TOT card (or since the beginning).  Input lines
      without either LIN or TOT are printed directly to the output file.

      Each data card has LIN in columns 1-3 and the Branch identifiers
      in columns 7 to 31. First Bus name and KV in columns 7-18,
      Second bus name and KV in columns 20-31.  Column 5 is a flag to
      tell slfs.c whether to use the "Pin" or "Pout" data quantity for
      the total flow. If the flag is "2", then "Pout" is used; otherwise
      "Pin" is used.  The flag corresponds to the metering point.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cflowlib.h"

/* cf_debug = 1;   /* Put this where you want the buffers displayed,for debugging   */

int readln( char *s, int lim )   /* Read a line from the terminal input */
{
   int   i;
   char  c;

   for ( i=0; i < lim - 1  &&  ( c = getchar() ) != EOF  &&  c != '\n'; ++i )
      s[i] = c;
   s[i] = '\0';
   return i;
}

int freadln( FILE *fp, char *s, int lim )    /* Read a line from a file */
{
   int   i;
   char  c;

   for ( i=0; i < lim - 1  &&  ( c = getc( fp ) ) != EOF  &&  c != '\n'; ++i )
      s[i] = c;
   s[i] = '\0';
   return i;
}



main ( int argc, char *argv[] )
{
   pf_comments  comments;          /* CFLOW structure */
   pf_rec     r;                   /* CFLOW structure */
   FILE   *out, *dat;
   char   card[82], cout[31];
   char   skv1[5], skv2[5];
   char   direc;
   int    error, lin, tot, head;
   float  brflow, totalflow = 0.0;
   float kv1, kv2;

   pf_cflow_init( argc, argv );
```

```
/*  Open the input file.  This could be prompted for, but Don says
    he always uses the same input file, which will NOT be in the
    execution directory,so full pathname is supplied.
    If the file cannot be opened, the program terminates
*/

dat=fopen( "/home/ipf/cflow_progs/slfs.dat", "r" );
if ( dat == NULL ) {
   printf("Can't open data file\n");
   exit(0);
}

/*  Prompt the user for the output file name, and open the file.
    Normally, file will be created in the execution directory,
    so cout is only 30 characters.
    If the file cannot be opened, the program terminates.
*/

printf("Enter output file name > ");
readln( cout, sizeof(cout) );
printf("\n");
out=fopen( cout, "w" );
if ( out == NULL ) {
   printf("Can't open output file\n");
   exit(0);
}

/* Retrieve caseid and description, and print heading    */

error = pf_rec_comments( &comments, "G" );
fprintf(out,"\n%s\n\n", &comments.h[0][33] );
totalflow = 0.0;

/*  Loop for every line in the data file
   Read a line from the data file, if not "LIN" or "TOT" then
        print the text as is.
   If "LIN" retrieve and print line data.
   If "TOT" print totalflow and reinitialize it.
*/

/* "C" array indices start at zero; below we use explicit starting index
   of 1 for "card" character array (string), so that column numbers
   will match array indices.
*/

while ( freadln( dat, &card[1], sizeof( card ) - 1 ) )
{

   lin = (!( strncmp( &card[1], "LIN", 3 )) );
   tot = (!( strncmp( &card[1], "TOT", 3 )) );
   head = lin + tot;     /* = 0 if not either */

   if (head == 0)       /* Heading card - print the text */
   {
        fprintf(out,"%s\n",&card[1]); /* Print from col. 1 - 1st char. is NULL! */
   }                            /* end if Heading  */
```

```
      if (lin)        /* LIN data card - process line */
      {
   /* Using CFLOW pf_init_branch function; bus kv's must be real numbers  */
         strncpy(skv1, &card[15], 4 ); skv1[4] = '\0';
         strncpy(skv2, &card[28], 4 ); skv2[4] = '\0';
         kv1 = atof(skv1);
         kv2 = atof(skv2);
/* pf_init_branch stores the passed ID fields in r, and zeroes all the other fields */
      pf_init_branch ( &r, "L", &card[7],kv1,&card[20],kv2,"*",0);

    /* Call CFLOW pf_rec_branch to retrieve output solution data   */
       error = pf_rec_branch( &r, "O" );

    /* If branch not found, do not print anything.  This happens quite often, since
       the input file is canned, and used on all cases.                          */
       if ( error ) { continue; }          /* This sends it back to the 'while' */

   /* Otherwise, retrieve flows. r is defined in this pgm as a structure of type pf_rec
     i indicates input data, s indicates output data.  See cflowlib.h for definitions.
    */
         direc = card[5];    /* Meter flag for Pin or Pout */
         if ( direc == '2' )
         {                    /* meter at second bus */
            brflow = r.s.branch.Pout;
            totalflow += brflow;
            fprintf( out,"%s    - %s(M) %s kV    num ckts %d      %7.1f\n",
                         r.i.branch.bus1_name, r.i.branch.bus2_name, skv2,
                         r.s.branch.num_ckt, brflow );
         }
         else               /* meter at first bus */
         {
            brflow = r.s.branch.Pin;
            totalflow += brflow;
            fprintf( out,"%s(M) - %s    %s kV    num ckts %d      %7.1f \n",
                         r.i.branch.bus1_name, r.i.branch.bus2_name, skv2,
                         r.s.branch.num_ckt, brflow );
         }
      }                             /* end if LIN */

      if (tot)              /* TOT data card */
      {
            fprintf(out,
           "\n                                    Total flow is %7.1f \n\n",totalflow);
            totalflow = 0.0 ;
      }                            /* end else if TOT */

}                                 /* end while */
   pf_cflow_exit();
}                                 /* end main */
```

Note that both input values (`r.i.branch.`) and output values (`r.s.branch.`) are reported. The "i" stands for "input" and the "s" stands for "solution".

The variable `cf_debug` is provided for convenience in debugging a CFLOW program. When it is "true" (set non-zero), all the input and output buffers will be dumped to the terminal window, so you can see exactly what your program is sending and getting back. Since this can be very voluminous, you would only want to turn it on in the area where you are having a problem.

3.8.2 INCREM Program

The INCREM program was translated from a COPE procedure used by WSCC Technical Staff. It reads an input data file of branches to be reported, and another file of buses to change generation on. The starting case is hard-coded, and so is the bus that you want to study power transfer from, in this case GADSBY 3 13.8. For each bus in the second file, the generation at GADSBY is increased, that of the other bus decreased, area intertie schedules are adjusted as necessary, the case is solved, and the flow is retrieved for all the branches in the first input file and stored in an array. The report goes to a file; it consists mostly of a matrix showing the effect of the generation changes on the monitored line flows.

The COPE procedure used the IPS feature INCREM to accomplish this task. IPF has no built-in incrementals function, so the CFLOW program just does what is described in the paragraph above. It is *not* a general-use incremental program mimicking the IPS function.

For simplicity, a lot of names are hard-coded in this program. The user would have to decide whether it would be more efficient to change them in the code for each study, or fix the code to be general and then have to type them in over again for each run.

```
/* increm.c
   This CFLOW procedure creates an incremental line flow listing of
   selected lines, sorted by areas. The incremental flows are computed
   as the change in flows from the base case to the incremental case.

   Several incremental cases can be submitted. The limit here is 10,
   compared with 64 for the INCREM COPE procedure for IPS. However, this
   limit can be extended with attendent changes in the incremental
   storage arrays and in the output reports.

   Two aspects makes this CFLOW procedure more complicated than the COPE
   equivalent.

   1. IPF uses area intertie "I" records to define the net area export.
      If these records are present, an interarea transfer is effected
      only by changing the scheduled interarea export. If "I" records exist
      but the particular Area1-Area2 "I" record does not exist, then a
      new "I" record must be added for Area1-Area2 with an export value
      of the desired transfer. If no "I" records exist, then the interarea
      transfer is effected by the ordinary means, namely, by increasing the
      area1 export and decreasing the area2 export.

   2. Process INCREM does not exist in IPF.  Consequently, the sorting
      and listing of branches was implemented entirely within this CFLOW
      procedure.
```

```
    The code is intentionally batch. It could be made interactive by
    prompting for file names and bus names.  All file names and bus names
    are hard coded.  Changing these requires re-editing, recompiling, and
    relinking the program.  Fortunately, these steps can be performed in
    a short time.

    The program's execution plan is as follows.

    1. Load in base case history file.
    2. Open branch data file, bus data file, and output report file.
    3. Process the branch data file. For each branch, obtain the base case
       line flow.
    4. Process the bus data file to identify each transfer pair of buses:
       "busname1" and "busname2".  The "busname1" is a hard-coded global
       variable.
       a. For each bus pair, perturb busname1's generation +100MW (and its
          associated areaname1's export +100MW) and busname2's generation
          -100MW (and its associated areaname2 export -100MW).
       b. Solve the case.
       c. Loop through the monitored branches, obtaining the line flows
          for the perturbed case.
    5. Print the output report.
       a. Use a branch index array "keysrt" in conjunction with a
          user-written compare routine (to be used with qsort) to obtain a
          double-entry list of monitored branches sorted by the following
          fields: area1, bus1, area2, bus2, id, and section.
       b. Print out the monitored lines flows using the sort index.
 */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#include "../cflow/cflowlib.h"
#include "../cflow/ft.h"

#define NULLC    '\0'
#define MAXFLOWS 1000
#define MAXCASES 10
#define TRUE     1
#define FALSE    0
#define LINES_PER_PAGE 64

/* Declare global variables */

struct FLOWS {
  char   bus1[13];         /* bus1, base1 field                */
  char   bus2[13];         /* bus2, base2 field                */
  char   id;               /* id field                         */
  int    section;          /* section field                    */
  char   area1[11];        /* area name corresponding to bus1  */
  char   area2[11];        /* area name corresponding to bus2  */
  float  Pin[MAXCASES+1];  /* Tie line flows in MWs            */
  float  Pout[MAXCASES+1]; /* Tie line flows in MWs            */
  } br_flow[MAXFLOWS];
```

```
/* Declare hard-coded files and bus names */

char *progname = "increm";
char *basefile = "../cflow_progs/98hs3inc.bse";
char *branchfile = "../cflow_progs/line.dat";
char *busname1 = "GADSBY 313.8";
char *busfile = "../cflow_progs/name2.dat";
char *outfile = "98hs3inc.out";

/* declare function prototypes */

FILE *efopen (char *, char *);
int compare( const void *, const void * );
/* static int compare( const void *key1, const void *key2 ); */
int increment (char *, char *, float, int);
int get_bus_area (char *, char *);
int mod_bus_gen (char *, float);
int mod_area_export (char *, float);
int mod_itie_export (char *, char *, float);
void title (int *, int *, FILE *, pf_comments *, int, char *);

FILE *efopen (char *file, char *mode) /* fopen file, return message */
      /* if can't                   */
{
  FILE *fp;

  if ((fp = fopen(file, mode)) != NULL)
    return fp;
  fprintf (stderr, " %s: can't open file %s mode %s\n",
    progname, file, mode);
  return NULL;
}

main(int argc, char *argv[])
{
  pf_comments   c;
  pf_rec        br, itie;

  int i, j, k, status, numcases, keysrt[2*MAXFLOWS], numbr, lineno = 0,
      pageno = 0, intertie_flag = FALSE, section;
  FILE *fp_busfile, *fp_branchfile, *fp_outfile;
  char id, base[5], busname2[13], oldarea[11], string[133], areaname1[11];

  pf_cflow_init( argc, argv);

  /* Load the history data file */

  status = pf_load_oldbase (basefile);
  if (status) {
    fprintf (stderr, " Unable to open history file %s status %d",
      basefile, status);
    pf_cflow_exit();
    return 1;
  }
```

```
 /* Determine if any Area Intertie "I" records exist in the base case */

 pf_rec_init (&itie, PF_REC);   /* Initialize structure prior
                                   to calling pf_rec_itie.
                                   PF_REC is in header file "ft.h"  */
 status = pf_rec_itie (&itie, "F");
 if (status == 0) intertie_flag = TRUE;  /* If success, "I" records exist */

 /* Obtain case comments */

 pf_rec_init (&c, PF_REC);   /* Initialize structure prior
                                to calling pf_rec_comments.
                                PF_REC is in header file "ft.h"  */
 status = pf_rec_comments (&c, "G");

/* Open the monitored branch data file for read only */

 fp_branchfile = efopen(branchfile, "r");
 if (fp_branchfile == NULL) {
   fprintf (stderr, " Unable to open monitored branch file %s \n",
     branchfile);
   pf_cflow_exit();
   return 1;
 }

 /* Open the bus data file for read only */

 fp_busfile = efopen(busfile, "r");
 if (fp_busfile == NULL) {
   fprintf (stderr, " Unable to open buses list file %s \n",
     busfile);
   pf_cflow_exit();
   return 1;
 }

/* Open the output report file for write only */

 fp_outfile = efopen(outfile, "w");
 if (fp_outfile == NULL) {
   fprintf (stderr, " Unable to open INCREM output report file %s \n",
     outfile);
   pf_cflow_exit();
   return 1;
 }

 fprintf (fp_outfile, " Base case from history file %s \n", basefile);
 for (i=0; i < 3; i++) {
   if (strlen (c.h[i]) > 0)
     fprintf (fp_outfile, " %s \n", &c.h[i][1]);
 }
/*
 Process each record in the monitored branch file. The procedure invokes
 the following steps.

 1. Parse the ID fields of each branch entity into the branch data
    structure.
```

```
2. A call to routine "pf_rec_branch" with appropriate arguments will
   obtain the branch output quantitys Pin and Pout.

3. Parse the ID fields again of each branch entity into arrays used
   for sorting and printing the output report. The "area1" and "area2"
   arrays are obtained indirectly through the zone. First the zone is
   obtained from the bus data structure of each terminal bus. From each
   zone, the corresponding the area name is obtained via the call
   "pf_area_of_zone".

4. An slightly unusual convention is employed here for subscrip "numbr".
   Normally, a C-arrays begins with subscript "0".  Here, it begins with
   subscript "1".  The reason is that it is necessary to distinquish
   normal branches from transposed branches in the "keysrt" array, which
   will be built after all lines have been read in and all incremental
   cases have been processed. The distinction is done by positive
   and negative subscripts: +n denotes the normal orientation, -n its
   transpose.  (C cannot distinquish between +0 and -0.)
*/

fprintf (stderr, " Reading in monitored line data \n");

numbr = 0;
while ( fgets (string, 132, fp_branchfile) != NULL) {
  /*
     Parse the branch data id fields from "string" into structure
     "br.i.branch" and call "pf_rec_branch" with the "O" option to
     retrieve the base case flow.

     Copy the parsed branch data id fields into a second structure
     "br_flow[]" for storing the flow results.

     Note that the strncpy command requires that the strings be
     explicitly null-terminated.
  */

  pf_rec_init (&br, PF_REC);   /* Initialize structure prior
                                  to calling pf_rec_branch.
                                  PF_REC is in header file "ft.h"  */
  strcpy (br.i.branch.type, "L ");
  strncpy (br.i.branch.bus1_name, &string[6], 8);
  br.i.branch.bus1_name[8] = NULLC;
  strncpy (base, &string[14], 4);
  base[4] = NULLC;
  br.i.branch.bus1_kv = atof (base);
  strncpy (br.i.branch.bus2_name, &string[19], 8);
  br.i.branch.bus2_name[8] = NULLC;
  strncpy (base, &string[27], 4);
  base[4] = NULLC;
  br.i.branch.bus2_kv = atof (base);
  id = string[31];
  if (id  == '\n' || id == '\t' || id == '\0') id = ' ';
  br.i.branch.ckt_id = id;
  if (strlen (string) > 32) {
    base[0] = string[32];
```

```
        base[1] = NULLC;
        section = atoi (base);
      } else {
        section = 0;
      }
      br.i.branch.section = section;
 /*
      The branch data is now loaded in structure "branch". A call to
      "pf_rec_branch" will retrieve the desired information.
 */
      status = pf_rec_branch (&br, "O");
      if (!status) {
        numbr++;
        strcpy (br_flow[numbr].bus1, &string[6]);
        br_flow[numbr].bus1[12] = NULLC;
        strcpy (br_flow[numbr].bus2, &string[19]);
        br_flow[numbr].bus2[12] = NULLC;
        br_flow[numbr].id = br.i.branch.ckt_id;
        br_flow[numbr].section = br.i.branch.section;
        br_flow[numbr].Pin[0] = br.s.branch.Pin;
        br_flow[numbr].Pout[0] = br.s.branch.Pout;

        /* Get the area name associated with bus1 indirectly through the
 zone of bus1 */

        status = get_bus_area (br_flow[numbr].bus1, br_flow[numbr].area1);
        if (status) {
          fprintf (stderr, " Unable to retrieve area associated with bus1 of \
monitored branch %s %s \n", br_flow[numbr].bus1, br_flow[numbr].bus2);
pf_cflow_exit();
return 1;
        }

        /* Get the area name associated with bus2 indirectly through the
 zone of bus2 */

        status = get_bus_area (br_flow[numbr].bus2, br_flow[numbr].area2);
        if (status) {
          fprintf (stderr, " Unable to retrieve area associated with bus2 of \
monitored branch %s %s \n", br_flow[numbr].bus1, br_flow[numbr].bus2);
pf_cflow_exit();
return 1;
        }
      }
    }

    /* Begin the incremental bus loop. This is the list of buses in
       testdc3.bdat, processed one-by-one. */

    while ( fgets (string, 80, fp_busfile) != NULL) {

      strncpy (busname2, string, 12);
      busname2[12] = NULLC;
      status = increment (busname1, busname2, 100.0, intertie_flag);
      if (!status) {
```

```
      numcases++;

      /* Solve the modified case */

      status = pf_solution ();

      /* Write change case comments */

      if (status) {

fprintf (fp_outfile, "\n Incremental case number %d failed \n",
  numcases);
fprintf (fp_outfile, "    100 MW scheduled from \"%s\" to \"%s\"\n",
  busname1, busname2);
fprintf (stderr, " Incremental case number %d failed \n",
  numcases);
for (i = 1; i <= numbr; i++) {
  br_flow[i].Pin[numcases] = 0.0;
  br_flow[i].Pout[numcases] = 0.0;
        }
      } else {

fprintf (fp_outfile, "\n Incremental case number %d \n", numcases);
fprintf (fp_outfile, "    100 MW scheduled from \"%s\" to \"%s\"\n",
  busname1, busname2);
fprintf (stderr, "\n Incremental case number %d \n", numcases);
fprintf (stderr, "    100 MW scheduled from \"%s\" to \"%s\"\n",
  busname1, busname2);

for (i = 1; i <= numbr; i++) {
          pf_rec_init (&br, PF_REC);   /* Initialize structure prior
                                          to calling pf_rec_branch.
                                          PF_REC is in header file "ft.h"  */
          strcpy (br.i.branch.type, "L ");
  strncpy (br.i.branch.bus1_name, br_flow[i].bus1, 8);
  br.i.branch.bus1_name[8] = NULLC;
  br.i.branch.bus1_kv = atof (&br_flow[i].bus1[8]);
  strncpy (br.i.branch.bus2_name, br_flow[i].bus2, 8);
  br.i.branch.bus2_name[8] = NULLC;
  br.i.branch.bus2_kv = atof (&br_flow[i].bus2[8]);
  br.i.branch.ckt_id = br_flow[i].id;
          status = pf_rec_branch (&br, "O");
  br_flow[i].Pin[numcases] = br.s.branch.Pin;
  br_flow[i].Pout[numcases] = br.s.branch.Pout;
}
      }
    }
  }

  /* Obtain sorted double entry index "keysrt" for branch flows
     keysrt[i] > 0 means "br_flow" is processed as is
               < 0 means "br_flow" is processed transposed
   */

  for (i = 0; i < numbr; i++) {
    keysrt[2*i] = i + 1;
```

```
      keysrt[2*i+1] = -keysrt[2*i];
    }
    qsort ( keysrt, 2*numbr, sizeof (keysrt[0]), compare );

    title (&lineno, &pageno, fp_outfile, &c, numcases, "");

    oldarea[0] = NULLC;
    for (i = 0; i < 2*numbr; i++) {
      j = keysrt[i];
      if (j > 0) {
        strcpy (areaname1, br_flow[j].area1);
      } else {
        strcpy (areaname1, br_flow[-j].area2);
      }
      if (strcmp (areaname1, oldarea) != 0) {
        if (lineno+4 > LINES_PER_PAGE) {
          lineno = 0;
          title (&lineno, &pageno, fp_outfile, &c, numcases, areaname1);
        } else {
          fprintf (fp_outfile, "\n From area %s \n\n", areaname1);
          lineno += 3;
        }
        strcpy (oldarea, areaname1);
      }
      if (lineno+1 > LINES_PER_PAGE) {
        lineno = 0;
        title (&lineno, &pageno, fp_outfile, &c, numcases, areaname1);
      }
      if (j > 0) {
        fprintf (fp_outfile, " %s %s %c %s %8.2f  ", br_flow[j].bus1,
br_flow[j].bus2, br_flow[j].id, br_flow[j].area2,
br_flow[j].Pin[0]);
        for (k = 1; k <= numcases; k++) {
fprintf (fp_outfile, " %7.2f", br_flow[j].Pin[k] - br_flow[j].Pin[0]);
        }
      } else {
        fprintf (fp_outfile, " %s %s %c %s %8.2f  ", br_flow[-j].bus2,
br_flow[-j].bus1, br_flow[-j].id, br_flow[-j].area1,
-br_flow[-j].Pout[0]);
        for (k = 1; k <= numcases; k++) {
fprintf (fp_outfile, " %7.2f",
            -br_flow[-j].Pout[k] + br_flow[-j].Pout[0]);
        }
      }
      fprintf (fp_outfile, "\n");
      lineno++;
    }

    pf_cflow_exit();
    fclose (fp_busfile);
    fclose (fp_branchfile);
    fclose (fp_outfile);
    if (!status) {
      exit (0);
    } else {
      fprintf (stderr, " %s aborted with errors \n", progname);
```

```
      exit (1);
    }
  }

  int get_bus_area (char *name, char *area)

  /* This routine returns the area name given the bus name. It is obtained
     indirectly through the bus bus data structure. First, essential information
     is inserted into the bus data structure and the remaining data in the
     structure is obtained after calling "pf_rec_bus". From the zone name in
     the bus structure, the area name is obtained by calling "pf_area_of_zone".
   */

  {
    pf_rec bus;
    char base[5], zone[3];
    int len, status;

    pf_rec_init (&bus, PF_REC);   /* Initialize structure prior
                                     to calling pf_rec_bus.
                                     PF_REC is in header file "ft.h"  */
    strcpy (bus.i.ACbus.type, "B ");
    strncpy (bus.i.ACbus.name, name, 8);
    bus.i.ACbus.name[8] = NULLC;
    strcpy (base, &name[8]);
    bus.i.ACbus.kv = atof (base);
    status = pf_rec_bus (&bus, "G");
    if (status) {
      fprintf (stderr, " Bus %s is not in history file %s \n", name, basefile);
      return 1;
    }
    strncpy (zone, bus.i.ACbus.zone, 2);
    zone[2] = NULLC;
    status = pf_area_of_zone (area, zone);
    if (status) {
      fprintf (stderr, " No area data in history file %s \n", basefile);
      return 1;
    }
    return 0;
  }

  int increment (char *busname1, char *busname2, float delta_gen, int flag)

  /* This routine applies +/- delta_gen to pairs of buses, areas, and/or
     intertie records to effect the desired transfer. */

  {
    pf_rec      b, a, itie;
    char areaname1[11], areaname2[11];
    int status;

    /* The following are declared "static" such that they can be initialized
       to NULL, are local, and are not volatile, i.e., they retain their value
       after the routine is executed. In FORTRANese, they are similar to local
       variables, initialized with a DATA statement, and retained with a
       SAVE statement.
```

```
*/
  static char oldbus1[13] = "", oldarea1[11] = "", oldbus2[13] = "",
              oldarea2[13] = "";

 /* Get areaname1 associated with busname1 */

 status = get_bus_area (busname1, areaname1);
 if (status) {
   fprintf (stderr, " Unable to retrieve area associated with bus %s \n",
    busname1);
   pf_cflow_exit();
   return 1;
 }

 /* Get areaname2 associated with busname2 */

 status = get_bus_area (busname2, areaname2);
 if (status) {
   fprintf (stderr, " Unable to retrieve area associated with bus %s \n",
    busname2);
   pf_cflow_exit();
   return 1;
 }

 /* Restore original Pgen on "oldbus1" and modify Pgen on "busname1".
    In this case, restoring the original Pgen on "oldbus1" is not
    necessary because it never changes its name once it is assigned.
    The same is true for area "oldarea1" and for intertie "oldarea1-oldarea2".
    This extra logic (two lines in each instance) is added for generality.
 */

 if (strcmp (busname1, oldbus1) != 0) {
   if (strlen (oldbus1) > 0)
     status = mod_bus_gen (oldbus1, -delta_gen);
   status = mod_bus_gen (busname1, delta_gen);
 }

 /* Restore original Pgen on oldbus2 and modify Pgen on busname2 */

 if (strcmp (busname2, oldbus2) != 0) {
   if (strlen (oldbus2) > 0)
     status = mod_bus_gen (oldbus2, delta_gen);
   status = mod_bus_gen (busname2, -delta_gen);
 }
 if (flag) {

   /* Area Intertie "I" records exist and these records define the net
      area interchange export. Restore the original export on intertie
      "oldarea1-oldarea2" and modify the export on intertie
      "areaname1-areaname2" */

   if (strlen (oldarea1) > 0 && strlen (oldarea2) > 0)
     status = mod_itie_export (oldarea1, oldarea2, -delta_gen);
   status = mod_itie_export (areaname1, areaname2, delta_gen);
 } else {
```

```
      /* Area intertie "I" records do not exist. Work directly on the
         area records. Modify export on each area "areaname1" and
         "areaname2" */

      if (strlen (oldarea1) > 0)
        status = mod_area_export (oldarea1, -delta_gen);
      status = mod_area_export (areaname1, delta_gen);
      if (strlen (oldarea2) > 0)
        status = mod_area_export (oldarea2, delta_gen);
      status = mod_area_export (areaname2, -delta_gen);
    }
    strcpy (oldbus1, busname1);
    strcpy (oldbus2, busname2);
    strcpy (oldarea1, areaname1);
    strcpy (oldarea2, areaname2);
    return status;
  }

  int mod_bus_gen (char *busname, float delta_gen)

  /* This routine changes Pgen on bus "busname" an amount "delta_pgen" */

  {
    pf_rec      b;
    int status;

    pf_rec_init (&b, PF_REC);   /* Initialize structure prior
                                   to calling pf_rec_bus.
                                   PF_REC is in header file "ft.h"  */
    strcpy (b.i.ACbus.type, "B ");
    strncpy (b.i.ACbus.name, busname, 8);
    b.i.ACbus.name[8] = NULLC;
    b.i.ACbus.kv = atof (&busname[8]);

    status = pf_rec_bus (&b, "G");
    if (status) {
      fprintf (stderr, " Bus %s is not in history file %s \n",
        busname, basefile);
      pf_cflow_exit();
      return 1;
    }

    /* Add "delta_gen" to bus generation */

    b.i.ACbus.Pgen = b.i.ACbus.Pgen + delta_gen;
    status = pf_rec_bus (&b, "M");
    return status;
  }

  int mod_area_export (char *areaname1, float delta_export)

  /* This routine changes Export on Area "areaname1" an amount "delta_export" */

  {
    pf_rec      a;
    int status;
```

```
  pf_rec_init (&a, PF_REC);   /* Initialize structure prior
                                 to calling pf_rec_bus.
                                 PF_REC is in header file "ft.h"  */
  strcpy (a.i.area.type, "A ");
  strcpy (a.i.area.name, areaname1);
  status = pf_rec_area (&a, "G");
  if (status) {
    fprintf (stderr, " Area %s is not in history file %s \n",
      a.i.area.name, basefile);
    pf_cflow_exit();
    return 1;
  }

  /* Add "delta_export" to area export */

  a.i.area.sched_export = a.i.area.sched_export + delta_export;
  status = pf_rec_area (&a, "M");
  return status;
}

int mod_itie_export (char *areaname1, char *areaname2, float delta_export)

/* This routine changes Export on area intertie "areaname1-areaname2" and
   amount "delta_export" */

{
  pf_rec      itie;
  int status;

  pf_rec_init (&itie, PF_REC);   /* Initialize structure prior
                                    to calling pf_rec_itie.
                                    PF_REC is in header file "ft.h"  */
  if (strcmp (areaname1, areaname2) == 0) {
    return 0;
  } else {
    strcpy (itie.i.itie.type, "I ");
    strcpy (itie.i.itie.area1_name, areaname1);
    strcpy (itie.i.itie.area2_name, areaname2);
    status = pf_rec_itie (&itie, "G");
    if (status) {
      itie.i.itie.sched_export = delta_export;
      status = pf_rec_itie (&itie, "A");  /* Add a new "I" record       */
    } else {
      itie.i.itie.sched_export = itie.i.itie.sched_export + delta_export;
      status = pf_rec_itie (&itie, "M");  /* Modify existing "I" record */
    }
    return status;
  }
}

int compare( const void *key1, const void *key2 )
/* static int compare( const void *key1, const void *key2 ) */

/* This comparing function obtains the relative sort order of *key1 and
   *key2 upon the following fields:
```

```
    1. area1.
    2. bus1.
    3. area2.
    4. bus2.
    5. id.

 */

{
  int i = *((int *) key1), j = *((int *) key2), comp;
  char *area11, *area12, *area21, *area22, *bus11, *bus12, *bus21, *bus22,
       id1, id2;
  int sect1, sect2;

  if (i == j) {
    return 0;
  } else {
    if (i > 0) {
      area11 = br_flow[i].area1;
      area12 = br_flow[i].area2;
      bus11 = br_flow[i].bus1;
      bus12 = br_flow[i].bus2;
      id1 = br_flow[i].id;
      sect1 = br_flow[i].section;
    } else {
      area11 = br_flow[-i].area2;
      area12 = br_flow[-i].area1;
      bus11 = br_flow[-i].bus2;
      bus12 = br_flow[-i].bus1;
      id1 = br_flow[-i].id;
      sect1 = br_flow[-i].section;
    }
    if (j > 0) {
      area21 = br_flow[j].area1;
      area22 = br_flow[j].area2;
      bus21 = br_flow[j].bus1;
      bus22 = br_flow[j].bus2;
      id2 = br_flow[j].id;
      sect2 = br_flow[j].section;
    } else {
      area21 = br_flow[-j].area2;
      area22 = br_flow[-j].area1;
      bus21 = br_flow[-j].bus2;
      bus22 = br_flow[-j].bus1;
      id2 = br_flow[-j].id;
      sect2 = br_flow[-j].section;
    }
    comp = strcmp (area11, area21);
    if (comp == 0) comp = strcmp (bus11, bus21);
    if (comp == 0) comp = strcmp (area12, area22);
    if (comp == 0) comp = strcmp (bus12, bus22);
    if (comp == 0) comp = (unsigned int)id1 - (unsigned int)id2;
    if (comp == 0) comp = sect1 - sect2;

    return comp;
```

```
  }
}

void title ( int *lineno, int *pageno, FILE *fp_outfile, pf_comments *c,
             int numcases, char *areaname)
{
  /* Write base case comments */

  int i;

  fprintf (fp_outfile, "\f\n\n");
  (*pageno)++;
  for (i=0; i < 3; i++) {
    if (strlen (c->h[i]) > 0)
      fprintf (fp_outfile, " %s \n", &c->h[i][1]);
      (*lineno)++;
  }
  fprintf (fp_outfile, "\n INCREMENTAL report (DELTA) line flows in MW \
page No. %d \n\n", *pageno);
  fprintf (fp_outfile, " From bus     To bus     cir To area   Base Flow   ");
  for (i = 1; i <= numcases; i++) {
    fprintf (fp_outfile, "Incr %2d ", i);
  }
  (*lineno) += 6;
  if (strlen (areaname) > 0) {
    fprintf (fp_outfile, "\n From area %s \n\n", areaname);
    (*lineno) += 3;
  }
}
```

CHAPTER 4

CFLOW LIBRARY FUNCTIONS

This chapter presents the CFLOW function library in reference format. Global variables and utility function are described first, followed by a detailed description of each function, including syntax, description, returned value, and an example. The detailed descriptions are presented in alphabetical order by function name.

4.1 CFLOW LIBRARY FUNCTIONS

The organization of the following CFLOW library functions is alphabetical. Each function entry starts at the top of a page. The function entries are structured like typical UNIX operating system and C language “man pages” (manual pages).

The manual page headings are defined as follows:

SYNTAX	The full ANSI C prototype format is shown. The argument declarations are also shown along with definitions.
DESCRIPTION	This shows how to use the function. It may also provide additional information giving you greater understanding of the function.
RETURNS	This indicates the value returned by the function whether there is an error or the call is successful.
EXAMPLE	The example shows you one way the function can be used in actual code. Look at all the examples to see several different ways of using the function.
SEE ALSO	Additional related functions are listed here.

4.2 NOTES

Below are a few notes, reminders, and definitions relating to the C language and the CFLOW library functions, for your convenience.

Strings

Many of the function parameters, such as bus names and action codes, are described as "strings." In C, the technical definition of a string is as follows: An array of characters, with a null (ASCII character '\0') in the element following the last valid character.

The library functions, like most C library functions, expect the strings you supply to conform to this definition. The easiest way is to enclose the value you want in *double quotes* (e.g. `"AMBROSIA"`). You can also store a value like this when you initially declare the character array. (When you declare a 'char' array, remember to always size it one larger than you need to store your actual string, so there is room for the null character!) However, you cannot put this value in an array with an assignment statement (`name = "AMBROSIA"` is illegal). To store a string in a character array, use the standard C `strcpy` or `strncpy` function.

A single character can be placed in one element of a character array by enclosing it in single quotes (`name[0] = 'A';` is a legal C assignment statement). If you do this, be sure to store a null in the last character (`name[8] = '\0';`).

Arrays

In C, an array dimension is declared to be the actual number of elements in the array:

```
char name[9];   /* Sized to hold an 8-character bus name, plus a null.*/
```

But when you reference the array elements, the indices run from zero to one less than the declared dimension. In the example above, `name[0]` has the first character of the bus name; `name[7]` has the last character, and `name[8],` which is the last element of the array, contains a null.

Function Types

If a function is declared as "void", then no return value is expected, and it may be invoked without a place being provided to receive the returned value. Example:

```
pf_init_bus( &b, "B", "AMBROSIA", 230.0 );
```

Most of the CFLOW functions are "int", meaning that they return an integer value, which is usually zero for success and non-zero for any kind of error. These must be called in some way which is compatible with the name of the function being a variable with a value. Example:

```
int error;
error = pf_rec_bus( &b, "G" );
```

Main Program

Your main program must have the standard arguments argc and argv:

```
main ( int argc, char *argv[] )  {    ...    }
```

You cannot merely call it "main", as you may have done for programs in a C class, or which you will see as examples in the books. Look at the sample CFLOW programs for guidance.

Include Libraries

In every program you write, you must include the standard C libraries `stdio.h` and `string.h`, and also the CFLOW library, `cflowlib.h`. This is done with a preprocessor statement:

`#include <stdio.h>` For libraries in the standard C include directory.
`#include "cflowlib.h"` For libraries in some other directory.

Linking Your Program

Your CFLOW program can be compiled by itself, but in order to execute, it must be linked with the cflow library, `libcflow.a .` The simplest way to do this (on a Unix system) is to use a `makefile`. One is provided in the delivery, along with the sample programs.

Buffers

CFLOW communicates with `ipfsrv` by using buffers which are passed back and forth (see the next section, Global Buffers, for detailed information). If you want to see what is in the buffers, set the variable `cf_debug` to one. As long as it has this value, the contents of every buffer passed will be displayed in the terminal window. Since this output will go by so fast you can't read it, and it can be very voluminous, you will want to limit the number of buffers actually displayed to the ones you are interested in. Set `cf_debug` back to zero to turn off the display.

Languages

CFLOW routines do not have to be written in C, except for the main program. If you are adept at the intricacies of calling C routines from a FORTRAN program and vice versa, you can write your main processing and reporting routines in FORTRAN, or use code you already have. In general, it will be easier to write a C program to perform the function, rather than trying to retrofit.

4.3 GLOBAL BUFFERS AND VARIABLES

The CFLOW code defines four global buffers as described in Table 4-3.

Table 4-1. Global Buffers

Buffer	Description
char *pf_cflow_inbuf	A buffer of the data most recently received from Powerflow.
char *pf_cflow_outbut	A buffer of the data most recently sent to Powerflow.
char *err_buf	A buffer containing a null terminated list of error messages from the most recent Powerflow communication.
char *reply_buf	A buffer containing a null terminated list of data from the most recent Powerflow communication.

The CFLOW code defines two global variables as described in Table 4-3.

Table 4-2. Global Variables

Buffer	Description
int cf_debug	= 1 to turn on debug output. = 0 to turn off debug output.
int cv_error	Contains the most recent conversion error code.

4.4 UTILITY FUNCTIONS

The functions `get_fld_a` and `put_fld_a` are used to read and write field values in bus records. You specify the card, field, and value to be read or written, and the functions take care of formatting the value regardless of whether it is an integer, floating point, or character value. These two functions use values defined in the header file `ft.h`. Be sure to `#include ft.h` when you use `get_fld_a` or `put_fld_a`.

The functions `readln` and `freadln` are provided to read input from the terminal window, and from a file, respectively. The standard `stdio.h` library has the corresponding functions `printf` and `fprintf`.

4.4.1 FREADLN

SYNTAX

```
int freadln ( FILE *fp, char *s,int lim );
```

`FILE *fp`	A pointer to the file.
`char *s`	A pointer to the string read from the file.
`int lim`	An integer specifying the length of the expected string.

DESCRIPTION

`freadln` reads one line from the file designated by `fp`.

RETURNS

`freadln` returns the number of characters read.

4.4.2 GET_FLD_A

SYNTAX

```
int get_fld_a ( char *str, char *card, int fld );
```

`char *str`	A pointer to a string containing bus record data.
`char *card`	A pointer to a string containing a bus record identifier.
`int fld`	An integer specifying the field from which the `str` data is to be retrieved. See the header file `ft.h` for the integer-to-field decoding.

DESCRIPTION

`get_fld_a` reads data from one bus record field at a time. The function takes care of correctly formatting the field data returned. The data can be integer, floating point, or character (text) data.

RETURNS

`get_fld_a` returns 0 if it is successful; otherwise, it returns an error number.

4.4.3 PUT_FLD_A

SYNTAX

```
int put_fld_a ( char *card, char *str, int fld );
```

`char *card`	A pointer to a string containing a bus record identifier.
`char *str`	A pointer to a string containing bus record data.
`int fld`	An integer specifying the field where the `str` data is to be stored. See the header file `ft.h` for the integer-to-field decoding.

DESCRIPTION

`put_fld_a` writes data into one bus record field at a time. The function takes care of correctly formatting the field data. The data can be integer, floating point, or character (text) data.

RETURNS

`put_fld_a` returns 0 if it is successful; otherwise, it returns an error number.

4.4.4 READLN

SYNTAX

```
int readln ( char *s,int lim );
```

`char *s`	A pointer to the string retrieved from the terminal.
`int lim`	An integer specifying the length of the expected string.

DESCRIPTION

`readln` retrieves user input from the terminal window where CFLOW is running.

RETURNS

`readln` returns the number of characters read.

4.5 POWERFLOW FUNCTIONS

The powerflow functions retrieve and/or modify powerflow data, as well as issue powerflow commands.

4.5.1 PF_AREA_OF_ZONE Find the area that a zone is in

SYNTAX

```
int pf_area_of_zone ( char *area_name, char *zone_name );
```

`*area_name`	A pointer to an array of 11 characters in which the area name is returned.
`*zone_name`	A string holding a zone name.

DESCRIPTION

`pf_area_of_zone` finds the name of the area that a zone is in.

RETURNS

`pf_area_of_zone` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
int cnt;
char zones[32][3];   /* array for zone list */
pf_get_list((char *)zones,10,ZONE_LIST,"");
for (cnt=0; cnt < 10; ++cnt) {
   char area_name[11];
   int error;
   error = pf_area_of_zone(area_name, zones[cnt]);
   printf("zone %-5s is in area %-10s\n",zones[cnt],area_name);
}
```

4.5.2 PF_BUS_EXISTS See if a bus exists

SYNTAX

```
int pf_bus_exists ( char *name, float kv );
```

`char *name` A pointer to a string containing the name.

`float kv` A real value representing the bus base kV.

DESCRIPTION

`pf_bus_exists` inquires whether or not a given bus is in the currently loaded case.

RETURNS

`pf_bus_exists` returns 0 if the bus exists; otherwise, it returns 1.

EXAMPLE

```
int found;
found = pf_bus_exists(new_name, kv);
if (found == 0)
   printf(" - This node already exists! %s\n", b.i.ACbus.name);
```

4.5.3 PF_CASE_INFO Retrieve case info

(Not implemented yet.)

SYNTAX

```
int pf_case_info ( pf_case_stats *info );
```

`pf_case_stats *info` A pointer to a structure of type `pf_case_stats`.

DESCRIPTION

`pf_case_info` retrieves data from a Powerflow base case and puts it in the `info` structure.

The structure members are accessed as follows:

`info.PF_version[11]`	Ten character array containing Powerflow version information.
`info.base_mva`	Base MVA of the base case (normally 100.0).
`info.num_DC_systems`	An integer count of the number of dc systems in the case.
`info.num_areas`	An integer count of the number of areas in the case.
`info.num_ities`	An integer count of the number of interties in the case.
`info.num_zones`	An integer count of the number of zones in the case.
`info.num_owners`	An integer count of the number of owners in the case.
`info.num_buses`	An integer count of the number of buses in the case (both ac and dc).
`info.num_area_slack_buses`	An integer count of the number of area slack buses in the case.
`info.num_DC_buses`	An integer count of the number of dc buses in the case.
`info.num_AGC_buses`	An integer count of the number of buses with AGC control in the case.
`info.num_BX_buses`	An integer count of the number of BX buses in the case.
`info.buses_pct_var_ctrl`	An integer count of the number of buses with percent VAR control in the case.

`info.num_branches`	An integer count of the number of branches in the case.
`info.num_lines`	An integer count of the number of lines in the case. All parallels count as one line.
`info.num_LTC_xfmrs`	An integer count of the number of LTC transformers in the case.
`info.num_phase_shifters`	An integer count of the number of phase shifters in the case.
`info.case_soln_status`	An integer containing the solution status. Corresponds to enumerated variables as follows: 1 = `NO_CASE` (no case data loaded) 2 = `UNSOLVED` (netdata loaded) 5 = `SOLVED` (successful solution, or solved case loaded) 6 = `SAVED` (solved case has been saved) 7 = `DIVERGED` (unsuccessful solution - diverged)

RETURNS

`pf_case_info` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
int error;
pf_case_stats *ci;
error = pf_case_info (&ci);
if (!error) {
fprintf (out,"Number of areas = %d\n",ci.num_areas,
"Number of zones = %d\n",ci.num_zones,
"Number of buses = %d\n",ci.num_buses,
"Number of connections = %d\n",num_lines, /* Not including parallels*/
"Number of lines = %d\n",ci.num_branches);
}
if (ci.case_soln_status == SOLVED) {
fprintf (out,"This was a solved case.\n\n");
}
```

4.5.4 PF_CFLOW_EXIT Close the data link to powerflow

SYNTAX

```
void pf_cflow_exit();
```

DESCRIPTION

`pf_cflow_exit` is called in the user's program to "disconnect" properly from `ipfsrv`.

RETURNS

`pf_cflow_exit` has no return; it calls the `exit` function.

EXAMPLE

```
#include "cflowlib.h"

main(int argc, char *argv[]) {
  pf_cflow_init(argc, argv);
  printf("pf_cflow_init\n");
  printf("pf_load_netdata=%d\n",pf_load_netdata("fgrove.net"));
  printf("pf_solution=%d\n",pf_solution());
  pf_cflow_exit();
}
```

4.5.5 PF_CFLOW_INIT Initialize the data link to powerflow

SYNTAX

```
void pf_cflow_init ( int argc, char *argv[] );
```

`int argc`	An integer providing the count of command line arguments picked up by the `main` function.
`char *argv[]`	A pointer to a character array that stores all the command line arguments picked up by the `main` function.

DESCRIPTION

`pf_cflow_init` establishes a socket connection with the Powerflow process. Other command line arguments that have been collected by the `argv` mechanism may be used by the CFLOW program. The command line arguments are "shifted left" such that `*argv[1]` contains the first command line argument intended for the CFLOW program and `argc` is updated to reflect the count of those arguments only.

RETURNS

`pf_cflow_init` returns 0 if it is successful; otherwise, it calls the `exit` function.

EXAMPLE

```
#include "cflowlib.h"

main(int argc, char *argv[]) {
  pf_cflow_init(argc, argv);
  printf("pf_cflow_init\n");
  printf("pf_load_netdata=%d\n",pf_load_netdata("fgrove.net"));
  printf("pf_solution=%d\n",pf_solution());
  pf_cflow_exit();
}
```

4.5.6 PF_CFLOW_IPC Buffer interface to powerflow

SYNTAX

```
pf_cflow_ipc ( );
```

DESCRIPTION

`pf_cflow_ipc` is a low-level interface to the interprocess communication that uses two global buffers `pf_cflow_inbuf` and `pf_cflow_outbuf`. This routine is used by most of the other pf functions; however, you can also use it directly. You put valid PCL commands and associated WSCC-formatted data records into `pf_cflow_outbuf`, call `pf_cflow_ipc`, then look for the results in `pf_cflow_inbuf`.

A list and description of the valid PCL commands is in the *IPF Advanced User's Guide*.

RETURNS

`pf_cflow_ipc` returns 0 if it is successful; otherwise, it calls the exit function.

4.5.7 PF_DEL Functions Delete by entity (area, zone)

The `pf_del_xxx` functions delete an AREA or ZONE along will all buses in the AREA or ZONE.

4.5.8 PF_DEL_AREA

SYNTAX

```
int pf_del_area ( char *area_name );
```

`char *area_name` A string representing an area name.

DESCRIPTION

`pf_del_area` deletes all buses in an area. All associated branches are also deleted.

RETURNS

`pf_del_area` returns 0 (zero) if it is successful; otherwise, it returns 1.

EXAMPLE

```
char *area_name = "ARIZONA   ";
error = pf_del_area ( area_name );
```

SEE ALSO

`pf_del_zone`

4.5.9 PF_DEL_ZONE

SYNTAX

```
int pf_del_zone ( char *zone_name );
```

`char *zone_name` A string representing a zone name.

DESCRIPTION

`pf_del_zone` deletes all buses in a zone. All associated branches are also deleted.

RETURNS

`pf_del_zone` returns 0 (zero) if it is successful; otherwise, it returns 1.

EXAMPLE

```
char *zone_name = "NA";
error = pf_del_zone ( zone_name );
```

SEE ALSO

`pf_del_area`

4.5.10 PF_GET_LIST Retrieve various lists: owners, areas, etc.

SYNTAX

```
int pf_get_list ( char *list, int size, int type );
```

`char *list`	A pointer to a two dimensional character array of appropriate size.
`int size`	An integer specifying the maximum number of list elements.
`int type`	An integer specifying the type of list: `AREA_LIST`, `KV_LIST`, `REC_TYPE_LIST`, `OWNER_LIST`, `ZONE_LIST`. See Table 4-3.

DESCRIPTION

`char *list` should be a two dimensional character array. For example, for `AREA_LIST`, the array might be declared as `char area_names[20][11]` if you know that there are not more than 20 area names in the case. `pf_case_info` gives information on the number of areas, zones, owners, etc. The size argument prevents the routine from exceeding the bounds of your array.

Table 4-3. Type Values

Type Code	List Name	Max Size[a]	String Length	Meaning
1	AREA_LIST	50	11	A list of the different area names in the case.
2	KV_LIST	150	4	A list of the different bus kV's in the case.
3	REC_TYPE_LIST	50	3	A list of the different record types in the case.
4	OWNER_LIST	450	4	A list of the different owners in the case.
5	ZONE_LIST	150	3	A list of the different zones in the case.

a. Max Size lists the IPF server's current internal limit on each quantity.

RETURNS

`pf_get_list` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
int cnt;
char owners[64][4]];
pf_get_list((char *)owners,64,OWNER_LIST );
printf("owners=\n");
for (cnt=0; cnt < 64; ++cnt) {
   printf("%-5s",owners[cnt]);
}
```

SEE ALSO

pf_case_info

4.5.11 PF_INIT FUNCTIONS Initialize data structures

4.5.12 PF_INIT_AREA

SYNTAX

```
void pf_init_area (pf_rec *r, char *type, char *name);
```

`pf_rec *r`	A pointer to a structure of type `pf_rec`, supplied by the calling routine.
`char *type`	A string that specifies the record type (must be A).
`char *name`	A string that contains the area name.

DESCRIPTION

`pf_init_area` initializes all area data fields to 0 except ID fields which are initialized to the values passed as parameters. This function is used to store ID fields for a specific area before calling `pf_rec_area` to retrieve the values for that area.

EXAMPLE

```
printf (”Initialize area record == \n”);
pf_init_area (&r,”A”,”NORTHWEST”);
printf (”Type = %s  Area Name = %s  Scheduled export = %7.1f\n\n”,
          r.i.area.type,r.i.area.name,r.i.area.sched_export);
```

4.5.13 PF_INIT_BRANCH

SYNTAX

```
void pf_init_branch(pf_rec *r, char *type, char *name1, float kv1,
          char *name2, float kv2, char *cid, int sid);
```

`pf_rec *r`	A pointer to a structure of type `pf_rec`, supplied by the calling routine.
`char *type`	A string which specifies the record type (L, T, E, or specific R-type).
`char *name1`	A string that contains the bus 1 name.
`float kv1`	A floating point value representing the base kv1.
`char *name2`	A string that contains the bus 2 name.
`float kv2`	A floating point value representing the base kv2.
`char *cid`	A string that contains the circuit ID. For solution data, "*" will retrieve the sum of all parallel circuits.
`int sid`	A integer value representing the section id. For solution data, a value of 0 will retrieve the total equivalent line.

DESCRIPTION

`pf_init_branch` initializes all branch data fields to 0 except ID fields which are initialized to the values passed as parameters. This function is used to store ID fields for a specific branch before calling `pf_rec_branch` to retrieve the values for that branch.

EXAMPLE

```
printf("Initialize branch record == \n");
pf_init_branch(&r,"L","WESTMESA",345.0,"FOURCORN",345.0,"1",0);
printf("Bus1 Name = %s%5.1f Bus2 Name = %s%5.1f  R = %7.1f X = %7.1f\n\n",
        r.i.branch.bus1_name, r.i.branch.bus1_kv, r.i.branch.bus2_name,
        r.i.branch.bus2_kv, r.i.branch.r, r.i.branch.x);
```

4.5.14 PF_INIT_BUS

SYNTAX

```
void pf_init_bus(pf_rec *r, char *type, char *name, float kv);
```

`pf_rec *r`	A pointer to a structure of type `pf_rec`, supplied by the calling routine.
`char *type`	A string that specifies the record type (either B or any bus type is legal).
`char *name`	A string that contains the bus name.
`float kv`	A floating point value representing the base kV.

DESCRIPTION

`pf_init_bus` initializes all bus data fields to 0 except ID fields which are initialized to the values passed as parameters. This function is used to store ID fields for a specific bus before calling `pf_rec_bus` to retrieve the values for that bus.

EXAMPLE

```
int error;
pf_rec *b;
/* This function is normally used to find data for a specific bus.
 * pf_init_bus stores the ID fields in the structure. */
pf_init_bus (&b, "B", "SJUAN G1",22.0);

/* Then pf_rec_bus retrieves the data. */
error = pf_rec_bus (&b, "G");   /* Gets rest of bus data (input). */
if (!error) {
  error = pf_rec_bus (&b, "O"); /* Gets output bus data. */
}
```

4.5.15 PF_INIT_CBUS

SYNTAX

```
void pf_init_cbus(pf_rec *r, char *type, char *owner,
                  char *name, float kv, char *year);
```

`pf_rec *r`	A pointer to a structure of type `pf_rec`, supplied by the calling routine.
`char *type`	A string that specifies the record type (must be +).
`char *owner`	A string that contains the owner name.
`char *name`	A string that contains the bus name.
`float kv`	A floating point value representing the base kV.
`char *year`	A string that contains the code year.

DESCRIPTION

`pf_init_cbus` initializes all continuation bus data fields to 0 except ID fields which are initialized to the values passed as parameters. This function is used to store ID fields for a specific bus before calling `pf_rec_cbus` to retrieve the continuation record values for that bus.

EXAMPLE

```
fprintf (out,"Initialize cbus record == \n");
pf_init_cbus (&r,"+","PNM","SAN JUAN",345.0," ");
fprintf (out,"Bus Name = %s%5.1f Owner = %s  Load = %7.1f\n\n",
          r.i.cbus.name,r.i.cbus.kv,r.i.cbus.owner,r.i.cbus.Pload);
```

4.5.16 PF_INIT_ITIE

SYNTAX

```
void pf_init_itie(pf_rec *r, char *type, char *area1, char *area2);
```

`pf_rec *r`	A pointer to a structure of type `pf_rec`, supplied by the calling routine.
`char *type`	A string that specifies the record type (must be I).
`char *area1`	A string which contains the area 1 name.
`char *area2`	A string which contains the area 2 name.

DESCRIPTION

`pf_init_itie` initializes all intertie data fields to 0 except ID fields that are initialized to the values passed as parameters. This function is used to store ID fields for a specific tie line before calling `pf_rec_itie` to retrieve the values for that line.

EXAMPLE

```
fprintf (out,"Initialize itie record == \n");
pf_init_itie (&r,"I","NORTHWEST","BC=HYDRO");
fprintf (out,"Area 1 = %s  Area 2 = %s  Scheduled flow = %7.1f\n\n",
     r.i.itie.area1_name,r.i.itie.area2_name,r.i.itie.sched_export);
```

4.5.17 PF_INIT_QCURVE

SYNTAX

```
void pf_init_qcurve(pf_rec *r, char *type, char *name, float kv);
```

`pf_rec *r`	A pointer to a structure of type `pf_rec` supplied by the calling routine.
`char *type`	A string that specifies the record type (must be QP).
`char *name`	A string that contains the bus name.
`float kv`	A floating point value representing the base kV.

DESCRIPTION

`pf_init_qcurve` initializes all P-Q curve data fields to 0 except ID fields that are initialized to the values passed as parameters. This function is used to store ID fields for a specific bus before calling `pf_rec_qcurve` to retrieve the generator capability curve values for that bus.

EXAMPLE

```
fprintf (out,"Initialize Q=curve record == \n");
pf_init_bus (&r,"QP","SJUAN G1",22.0);
fprintf (out,"Bus Name = %s%5.1f Status code = %s\n\n",
          r.i.qcurve.bus_name,r.i.qcurve.bus_kv,r.i.qcurve.active);
```

4.5.18 PF_INIT_REC

SYNTAX

```
void pf_init_rec ( pf_rec *r, enum TYPE );
```

`pf_rec *r` A pointer to a structure of type `pf_rec`, supplied by the calling routine.

`enum TYPE` An enumerated variable defined in `ft.h`

DESCRIPTION

`pf_init_rec` initializes a data buffer of type `pf_rec` to blanks and zeros, in order to clear out old data before calling one of the `pf_rec` routines to store new data in it. Its use is not necessary, but is recommended.

EXAMPLE

```
pf_init_rec(r,AREA);
pf_init_rec(r,L_LINE);
pf_init_rec(r,AC_BUS);
pf_init_rec(r,CBUS);
pf_init_rec(r,ITIE);
pf_init_rec(r,QCURVE);
```

4.5.19 PF_LOAD Functions Tell powerflow to load a file

The `pf_load_xxx` access the powerflow load file capability.

4.5.20 PF_LOAD_CHANGES

SYNTAX

```
int pf_load_changes ( char *filename );
```

`char *filename`	A string representing a file name, or "`*\n`", followed by valid change records.

DESCRIPTION

`pf_load_changes` passes an ASCII change file name to `ipfsrv` so that it can read and interpret the file.

Or

`char *filename` contains an `*\n` followed by change records which are to be processed by `ipfsrv`. The records must be separated by `\n`.

RETURNS

`pf_load_changes` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_load_oldbase ("43bus.bse");
fprintf (out,"Loaded old base 43bus.bse, status = %d \n\n",error);

if (!error) {
   error = pf_load_changes("43bus.chg");
   printf("Loaded change file 43bus.chg, status = %d\n\n",error);
}
```

4.5.21 PF_LOAD_CONTROL

SYNTAX

```
int pf_load_control ( char *filename );
```

`char *filename`	A string representing a file name of a file containing PCL commands [and data].

DESCRIPTION

`pf_load_control` passes an ASCII control file name to `ipfsrv` so that it can read and interpret the file.

RETURNS

`pf_load_control` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
error = pf_load_control("34bus.pcl");
printf("Executed control file 34bus.pcl, status = %d\n\n",error);
```

4.5.22 PF_LOAD_NETDATA

SYNTAX

```
int pf_load_netdata ( char *filename );
```

`char *filename` — A string representing a file name or an `*\n` followed by bus and branch records.

DESCRIPTION

`pf_load_netdata` passes a network data file name to the `ipfsrv` process so that it can read and interpret the network data file.

Or

`char *filename` contains an `*\n` followed by bus and branch records which are to be processed by `ipfsrv`. The records must be separated by `\n`.

Note: If a case is currently loaded, it is overwritten and the data is lost. The case loaded is *not* usable by GUI after CFLOW has completed.

RETURNS

`pf_load_netdata` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
#include "cflowlib.h"

main(int argc, char *argv[]) {
  pf_cflow_init(argc, argv);
  printf("pf_cflow_init\n");
  printf("pf_load_netdata=%d\n",pf_load_netdata("fgrove.net"));
  printf("pf_solution=%d\n",pf_solution());
  pf_cflow_exit();
}
```

4.5.23 PF_LOAD_OLDBASE

SYNTAX

```
int pf_load_oldbase ( char *filename );
```

`char *filename` A string representing a file name followed by an optional `,rebuild = ON|OFF`.

DESCRIPTION

`pf_load_oldbase` passes a base case filename to the `ipfsrv` process so that it can read and interpret the file as an "oldbase".

Note: If a case is currently loaded, it is overwritten and the data is lost.

RETURNS

`pf_load_oldbase` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_load_oldbase (“j98cy94.bse, rebuild = ON”);
```

4.5.24 PF_LOAD_REFBASE

SYNTAX

```
int pf_load_refbase ( char *filename );
```

`char *filename` A string representing a file name.

DESCRIPTION

`pf_load_refbase` passes a base case filename to the `ipfsrv` process so that it can read and interpret the file as a "reference base" (also referred to as an "alternate base"). This is done prior to requesting difference plots or comparison (difference) reports.

Note: If a reference case is currently loaded, it is overwritten and the data is lost.

RETURNS

`pf_load_refbase` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_load_refbase ("j98cy94.bse");
```

4.5.25 PF_PLOT Create a Hardcopy Plot

SYNTAX

```
int pf_plot ( char *cor_file, char *ps_file, char *options );
```

`char *cor_file`	A string representing the name of a coordinate file.
`char *ps_file`	A string representing the name of the postscript file to be created.
`char *options`	An optional string (may be NULL), representing a list of comments and options, separated by newline (`\n`). Each option must begin with an "@" character.

DESCRIPTION

`pf_plot` causes powerflow (IPF) to generate a plot. Difference plots may be made by first loading an reference (alternate) base case with `pf_load_refbase` and providing a difference plot coordinate file. `pf_plot` sends a command constructed as follows:

```
/plot
<cor_filename>
<ps_filename>
<options>
```

RETURNS

`pf_plot` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
pf_load_oldbase("A98CY94.BSE");
pf_load_refbase("J98CY94.BSE");
pf_plot("500BUS_DIF.COR", "A98CY94.PS", "");
system("print/queue=EOHQMS_PS A98CY94.PS");
```

4.5.26 PF_PUT_INREC Send WSCC change record to powerflow

SYNTAX

```
int pf_put_inrec ( char *record );
```

`char *record` A string containing WSCC formatted data for a Powerflow input data record.

DESCRIPTION

`pf_put_inrec` changes, adds, or deletes an input data record for Powerflow.

Note: See the *IPF Batch User's Guide* for a description of input record types and rules for adding, changing, and deleting.

This function provides a means of inputting records for which there is not a specific function such as factor change (P), although it can be used for inputting any WSCC input record.

RETURNS

`pf_put_inrec` returns 0 (zero) if it is successful; otherwise, it returns 1.

EXAMPLE

The following program uses `pf_put_inrec` to change an input data record in Powerflow and then outputs a success or failure message to the screen.

```
int error;
char record [130];
/* record needs to contain valid change, add, or delete input data */
error = pf_put_inrec ( record );
if (!error)
   printf ( "Successfully changed, added, or deleted input record.\n"
);
else
   printf ( "Invalid record.\n" );
}
```

4.5.27 PF_REC Functions Manipulate powerflow records

The `pf_rec_xxx` functions allow powerflow input (network data) and output (solution) data to be retrieved, as well as allowing input data (network data) to be added, modified, or deleted.

4.5.28 PF_REC_A2B

SYNTAX

```
int pf_rec_a2b ( char *net_data, pf_rec *r, char *action );
```

`char *net_data` A pointer to a source string of network or output data.

`pf_rec *r` A pointer to a structure of type `pf_rec`.

`char *action` A pointer to a string designating the action to be performed. See Table 4-15 for the codes and their meanings.

DESCRIPTION

`pf_rec_a2b` converts an ascii record in WSCC format to a `pf_rec`.

Table 4-4 Action Codes

Code	Meaning
I	Converts network data string to binary input data.
O	Converts output data string to binary solution data.

RETURNS

`pf_rec_a2b` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
pf_rec b;
char net_data[80];
printf("Enter branch identifying data: ");
gets(net_data);
pf_rec_a2b(net_data, &b, "I");
pf_rec_branch(&b, "O");
```

4.5.29 PF_REC_AREA

SYNTAX

```
int pf_rec_area ( pf_rec *a, char *action );
```

`pf_rec *a` — A pointer to a structure of type `pf_rec` supplied by the calling routine.

`char *action` — A string designating the action to be performed on an area record. See Table 4-15 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_area` retrieves, modifies, adds, or deletes area data.

Table 4-5. Area Action Codes

Code	Meaning
F	Retrieves the first area record.
N	Retrieves the next area record. (Area name must be valid.)
G	Retrieves the rest of the area record. (Area name must be valid.)
D	Deletes an area record. (Area name must be valid.)
A	Adds an area record. (All required data fields must be valid. See the *IPF Batch User's Guide*.)
M	Modifies an area record. (All required data fields must be valid. See the *IPF Batch User's Guide*.)
E	Eliminates all area records. (area_rec is ignored and can be set to zero. This code does not delete any zones, buses, etc. It places all zones in area "blank.")
O	Retrieves the solution output data. (The case must be solved and the area name must be valid.)

The input values available are:

`a.i.area.type[3]` — Record type; here A for area record.

`a.i.area.name[11]` — Ten character area name.

`a.i.area.sbus_name[9]`	Eight character area slack bus name.
`a.i.area.sbus_kv`	Base kV of the area slack bus.
`a.i.area.sched_export`	Scheduled export power from an area.
`a.i.area.zones[50][3]`	Zones defined to be in an area.
`a.i.area.num_zones`	The number of zones in the area.
`a.i.area.max_Vpu`	Maximum per unit voltage.
`a.i.area.min_Vpu`	Minimum per unit voltage.

The solution values available are:

`a.s.area.Pgen`	Total area generation.
`a.s.area.Pload`	Total area load.
`a.s.area.Ploss`	Total area losses.
`a.s.area.Pexport`	Total actual export.
`a.s.area.num_inties_input`	Number of input intertie records associated with the area.
`a.s.area.num_ities_internal`	Number of internal intertie records for the area.

RETURNS

`pf_rec_area` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
    pf_rec  a;                    /* CFLOW structure */
    int    error, status;
    FILE *out;

    pf_cflow_init( argc, argv );
    out = fopen (“misc.rpt”, “w”);

/* Area data */

       fprintf (out,”\n******** AREA DATA ********\n\n”);
       cf_debug = 1;
       error = pf_rec_area( &a, “F” ); /* get first area */
       cf_debug = 0;
```

```
status = pf_rec_area( &a, "O" ); /* get first area output*/

while ( !error && !status )
{
  fprintf (out,"Type Area Name  Slack Bus   NZn  Export   Pgen
     Pload   Ploss Pexport  Vmax  Vmin\n");
  fprintf (out," %s  %s %s%5.1f %d %7.2f %7.1f %7.1f %7.1f %7.1f
     %6.4f %6.4f\n", a.i.area.type, a.i.area.name,
     a.i.area.sbus_name,
     a.i.area.sbus_kv,a.i.area.num_zones,a.i.area.sched_export
     , a.s.area.Pgen, a.s.area.Pload, a.s.area.Ploss,
     a.s.area.Pexport, a.i.area.max_Vpu,a.i.area.min_Vpu);
  fprintf (out,"Zones in Area:  %s  %s  %s  \n\n",
     a.i.area.zones[0],a.i.area.zones[1],a.i.area.zones[2]);

  error = pf_rec_area( &a, "N" ); /* get next area */
  status = pf_rec_area( &a, "O" ); /* get next area output*/
}
```

4.5.30 PF_REC_B2A

SYNTAX

```
int pf_rec_b2a ( char *net_data, pf_rec *r, char *action );
```

`char *net_data`	A pointer to a destination string for network data.
`pf_rec *r`	A pointer to a structure of type pf_rec.
`char *action`	A pointer to a string designating the action to be performed on a bus record. See Table 4-15 for the codes and their meanings.

DESCRIPTION

`pf_rec_b2a` converts a `pf_rec` to an ascii record in WSCC format. All data fields in `pf_rec` must be valid.

Table 4-6 Action Codes

Code	Meaning
I	Writes network data record.
D	Writes change record to delete.
A	Writes change record to add.
M	Writes change record to modify.
O	Writes solution record. (The case must be solved.)

RETURNS

`pf_rec_b2a` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
pf_rec b;
char net_data[80];
pf_rec_bus(&b, "F");
pf_rec_b2a(net_data, &b, "I");
printf("%s\n", net_data);
```

4.5.31 PF_REC_BRANCH

SYNTAX

```
int pf_rec_branch ( pf_rec  *br, char *action );
```

`pf_rec *br` — A pointer to a structure of type `pf_rec`, supplied by the calling routine.

`char *action` — A string designating the action to be performed on a branch record. See Table 4-7 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_branch` retrieves, modifies, adds, or deletes branch records.

Table 4-7. Branch Action Codes

Code	Meaning
F3	Retrieves the first branch record associated with bus1, bus2, and circuit ID. bus1_name, bus1_kV, bus2_name, bus2_kV, and ckt_id must be valid.
N3	Retrieves the next branch record associated with bus1, bus2, and circuit ID. See notes below.
F2	Retrieves the first branch record associated with bus1 and bus2. bus1_name, bus1_kV, bus2_name, bus2_kV must be valid.
N2	Retrieves the next branch record associated with bus1 and bus2. See notes below.
F1	Retrieves the first branch record associated with bus1. bus1_name and bus1_kV must be valid.
N1	Retrieves the next branch record associated with bus1. See notes below.
F	Retrieves the first branch record disregarding bus association. All id fields may be null or zero.
N	Retrieves the next branch record disregarding bus association. See notes below.
G	Retrieves the rest of the branch record. All id fields must be valid to get a specific record.
D	Deletes a branch record. All id fields must be valid.
A	Adds a branch record. All fields appropriate for the branch type must be valid.

Table 4-7. Branch Action Codes

Code	Meaning
M	Modifies a branch record. All fields appropriate for the branch type must be valid.
O	Retrieves the solution output data.

Notes:

1) with a "wildcard" circuit ID of " " or "*" and
a section code of zero (or blank) --
"G" and "F3" are the same as "F2".

2) with a valid (non-"wildcard") circuit ID and
a section code of zero (or blank) --
"G" is the same as "F3".

3) codes "F", "N", "N1", "N2", and "N3" do not need any data specified in the `pf_rec` , however the first use of "N", "N1", "N2", and "N3" relies on initialization with a "F", "F1", "F2", "F3", or "G" code on a previous call.

The columns in Table 4-8 represent the input values that are defined for branch records. In the table, "N/A" means that the data item does not apply to that record type; "-" means the variable name is the same as the generic variable name (left most column). See the *IPF Batch User's Guide* for descriptions of the various branch records.

To access individual fields of the records use the union member name, for example, using a declaration of `pf_rec br`; then `br.i.branch.tap1` would reference the input `tap1` field for a transformer. If you were looking at an E-type line instead, the variable `br.i.branch.tap1` would contain the `g2` value, but to make it more obvious what you were actually doing, you would probably want to use the union member name `br.i.E.g2`, instead. All of the structures and unions are declared in `cflowlib.h` .

Table 4-8. Branch Types and .branch. Structure - Input

Generic branch variable name and type	E	L	T	TP	R,RN, RQ,RV	RM, RP	RZ	LD	LM
char type[3]	E	L	T	TP	R,RV,RQ,RN	RM, RP	RZ	LD	LM
char owner[4]	-	-	-	-	-	-	-	-	-
char bus1_name[9]	-	-	-	-	-	-	-	-	-
float bus1_kv	-	-	-	-	-	-	-	-	-
int meter	-	-	-	-	var_tap_side	var_tap_side	var_tap_side	-	-
char bus2_name[9]	-	-	-	-	-	-	-	-	-
float bus2_kv	-	-	-	-	-	-	-	-	-
char ckt_id	-	-	-	-	N/A	N/A	-	l_or_R_control	N/A
int section	-	-	-	-	N/A	N/A	-	N/A	N/A
float total_rating	-	-	-	-	N/A	N/A	I_rate	-	-
int num_ckts	-	-	-	-	num_taps	num_taps	rani_type	N/A	N/A
float r	-	-	-	-	N/A	N/A	Pc_max	R	R
float x	-	-	-	-	N/A	N/A	Pc_min	L_mh	L_mh
float g	g1	-	-	-	N/A	N/A	Xij_max	C_uf	C_uf
float b	b1	-	-	-	N/A	N/A	Xij_min	P_sched	N/A
float tap1	g2	miles	tap1	phase_shift_deg	max_tap	max_phase_shift_deg	Bis_max	V_sched	N/A
float tap2	b2	N/A	tap2	tap2	min_tap	min_phase_shift_deg	Bis_min	miles	miles
float alpha_N_deg	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-	N/A
float gamma_0_deg	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-	N/A

Table 4-8. Branch Types and .branch. Structure - Input

Generic branch variable name and type	E	L	T	TP	R,RN, RQ,RV	RM, RP	RZ	LD	LM
char descrip[9]	N/A	descrip [9]	N/A	N/A	rmt_ bus_ name [9]	rmt_ bus_ name [9]	N/A	N/A	N/A
char date_in[4]	-	-	-	-	-	-	N/A	N/A	-
char date_out[4]	-	-	-	-	-	-	N/A	N/A	-
float thermal_rating	-	-	-	-	rmt _bus _kv	rmt _bus _kv	N/A	-	-
float bottleneck_rating	-	-	-	-	Qmax	Pmax	N/A	-	-
float emergency _rating	N/A	N/A	-	-	Qmin	Pmin	N/A	N/A	N/A

The structure members for solution variables are accessed as follows:

`br.s.branch.Pin` Real power flow at bus1, in MW. Positive indicates flow from bus1 toward bus2.

`br.s.branch.Qin` Reactive power flow at bus1, in Mvar. Positive indicates flow from bus1 toward bus2.

`br.s.branch.Pout` Real power flow at bus2, in MW. Negative indicates flow from bus1 to bus2 (i.e., without losses, `Pout = -Pin`).

`br.s.branch.Qout` Reactive power flow at bus2, in Mvar. Negative indicates flow from bus1 to bus2.

`br.s.branch.Ploss` Real losses, in MW (`Pin+Pout`).

`br.s.branch.Qloss` Reactive losses, in Mvar (`Qin+Qout`).

`br.s.branch.max_flow_amps` Largest current in any section of a line.

`br.s.branch.max_flow_mva` Largest flow in a transformer.

`br.s.branch.rating_code` Type of rating used – N, B, T, or E.

`br.s.branch.rating`	Actual value of rating used (amps or MVA).
`br.s.branch.pct_load`	Percent loading on the branch, using the indicated rating.
`br.s.branch.pct_comp`	Percent line compensation (total negative reactance divided by total positive reactance).
`br.s.branch.tap1`	Final tap at bus1 of transformer, in kV for normal tap, in degrees for phase shifter.
`br.s.branch.tap2`	Final tap at bus2 of transformer, in kV.

RETURNS

`pf_rec_branch` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
pf_rec br;
int err1;
for
(err1=pf_rec_branch(&br,"f1");err1==0;err1=pf_rec_branch(&br,"n1"))
{
    printf("      %s, kv= %6.1f,name2= %s, kv2= %6.1f\n",
      br.i.branch.bus1_name,br.i.branch.bus1_kv,
      br.i.branch.bus2_name,br.i.branch.bus2_kv);    }
```

4.5.32 PF_REC_BUS

SYNTAX

```
int pf_rec_bus ( pf_rec *b, char *action );
```

`pf_rec *b`	A pointer to a structure of type `pf_rec` supplied by the calling routine.
`char *action`	A string designating the action to be performed on an intertie record. See Table 4-9 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_bus` retrieves, modifies, adds, or deletes bus records.

Table 4-9. Bus Action Codes

Code	Meaning
F	Retrieves the first bus record.
N	Retrieves the next bus record. (Name and kV must be valid.)
G	Retrieves the rest of the bus record. (Name and kV must be valid.)
D	Deletes a bus record. (Name and kV must be valid.)
A	Adds a bus record. (All data fields must be valid. See the *IPF Batch User's Guide*.)
M	Modifies a bus record. (All data fields must be valid. See the *IPF Batch User's Guide*.)
O	Retrieves the solution output data. (The case must be solved and name and kV must be valid.)

`pf_rec` is a union of both input data (i) and solution data (s).

To access individual fields of the records use the union member name, for example, using a declaration of `pf_rec b`; then `b.i.ACbus.Pload` would contain the MW load for an ac bus, and the smoothing reactance for a dc bus, but to make it more obvious what you were actually doing, you would probably want to use the union member name `b.i.DCbus.smooth_rx_mh` instead, when dealing with a dc bus. If you are retrieving all buses, you can use the `ACbus` designation for the `type, owner, name, kv,` and `zone` fields; the contents of these fields is the same regardless of the bus type.

The structure members for an ac bus are accessed as follows:

`b.i.ACbus.type[3]`	Two character bus record type, for example, B, BS, BQ, etc.
`b.i.ACbus.owner[4]`	Three character bus owner.
`b.i.ACbus.name[9]`	Eight character bus name.
`b.i.ACbus.kv`	Base kV of the bus.
`b.i.ACbus.zone[3]`	Two character zone name.
`b.i.ACbus.Pload`	Real load in MW.
`b.i.ACbus.Qload`	Reactive load in Mvar.
`b.i.ACbus.Pshunt`	Real shunt in MW.
`b.i.ACbus.Qshunt`	Reactive shunt in Mvar.
`b.i.ACbus.Pmax`	Maximum real load in MW.
`b.i.ACbus.Pgen`	Scheduled real power in MW.
`b.i.ACbus.Qsch_Qmax`	Scheduled reactive load in Mvar (Qsch) or a real number designating maximum reactive power in Mvar (Qmax).
`b.i.ACbus.Qmin`	Minimum reactive power in Mvar.
`b.i.ACbus.Vhold_Vmax`	Voltage to hold in per unit (Vhold) or a real number designating a maximum voltage limit in per unit (Vmax), depending on the bus type.
`b.i.ACbus.Vmin_Vdeg`	Minimum voltage limit in per unit, or voltage angle for the BS bus.
`b.i.ACbus.rmt_name[9]`	Eight character remote bus name.
`b.i.ACbus.rmt_kV`	Base kV of a remote bus.
`b.i.ACbus.pct_vars`	Percent vars supplied for control of remote bus.

The solution values for an ac bus are accessed as follows:

`b.s.ACbus.Pgen`	Solved real power in MW.
`b.s.ACbus.Qgen`	Solved reactive power in MW.

`b.s.ACbus.Vmag` Solved reactive voltage in per unit.

`b.s.ACbus.Vdeg` Solved voltage angle in degrees.

`b.s.ACbus.Pload` Solved real load in MW.

`b.s.ACbus.Qload` Solved reactive load in Mvar.

`b.s.ACbus.Bshunt_used` Total shunt used, net of capacitors (+) and reactors (-).

`b.s.ACbus.Bshunt_sch` Total shunt available, net of capacitors and reactors.

`b.s.ACbus.Bshunt_used_cap` Capacitive shunt used, Mvar.

`b.s.ACbus.Bshunt_sch_cap` Capacitive shunt available, Mvar.

`b.s.ACbus.Bshunt_used_rx` Reactive shunt used, Mvar.

`b.s.ACbus.Bshunt_sch_rx` Reactive shunt available, Mvar.

`b.s.ACbus.Qunsch` Mvars produced, on a type BS or BE bus.

The fields `type, owner, name, kv,` and `zone` are the same for both ac and dc buses. Other structure members for a dc bus are accessed as follows:

`b.i.DCbus.bridges_per_ckt` Number of dc bridges per circuit.

`b.i.DCbus.smooth_rx_mh` Smoothing reactance in millihenries.

`b.i.DCbus.alpha_min_deg` $Alpha_{min}$ in degrees.

`b.i.DCbus.alpha_stop_deg` $Alpha_{stop}$ in degrees.

`b.i.DCbus.valve_drop_per_bridge_volts` Voltage drop per valve.

`b.i.DCbus.bridge_current_rating_amps` DC current rating.

`b.i.DCbus.alpha_gamma_N_deg` $Alpha_N$ or $gamma_N$ in degrees.

`b.i.DCbus.gamma_0_deg` $Gamma_0$ in degrees.

`b.i.DCbus.P_sched` Scheduled power MW.

`b.i.DCbus.V_sched` Scheduled voltage kV.

`b.i.DCbus.commutating_bus_name[9]` Eight character commutating bus name.

`b.i.DCbus.commutating_bus_kv`	Commutating bus kV.
`b.i.DCbus.converter_code`	One character converter code.

The solution values for a dc bus are accessed as follows:

`b.s.DCbus.P_DC`	AC real power into the DC bus, positive at the rectifier and negative at the inverter.
`b.s.DCbus.Q_DC`	AC reactive power into the DC bus, returned as a positive number.
`b.s.DCbus.V_DC`	DC terminal voltage (final voltage at commutating bus, in kV).
`b.s.DCbus.converter_deg`	Converter angle, α for rectifier, γ for inverter, in degrees.
`b.s.DCbus.P_valve_losses`	Difference between AC power and DC power.
`b.s.DCbus.Q_valve_losses`	Difference between AC power and DC power (same as Q_DC).

RETURNS

`pf_rec_bus` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
pf_rec r;
int err;
for (err=pf_rec_bus(&r,"f"); err == 0; err=pf_rec_bus(&r,"n")){
   pf_rec_bus(&r,"o");
   printf("name= %s, kv= %6.1f, vmag=%6.1f, vdeg=%6.1f\n",
     r.i.ACbus.name, r.i.ACbus.kv, r.s.ACbus.Vmag,
     r.s.ACbus.Vdeg);
}
```

4.5.33 PF_REC_CBUS

SYNTAX

```
int pf_rec_cbus ( pf_rec *cb, char *action );
```

`pf_rec *cb`	A pointer to a structure of type `pf_rec`, supplied by the calling routine.
`char *action`	A string designating the action to be performed on a continuation bus record. See Table 4-10 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_cbus` retrieves, modifies, adds, or deletes continuation bus (+) data. Note that cbus data is always associated with particular buses.

Table 4-10. Continuation Bus Action Codes

Code	Meaning
F1	Retrieves the first continuation bus record associated with a given bus (name, kV).
N1	Retrieves the next cbus record associated with a given bus. (All id fields must be valid. See the *IPF Batch User's Guide*.)
G	Retrieves the rest of the cbus record. (All id fields must be valid. See the *IPF Batch User's Guide*.)
D	Deletes a cbus record. (All id fields must be valid. See the *IPF Batch User's Guide*.)
A	Adds a cbus record. (All data fields must be valid. See the *IPF Batch User's Guide*.)
M	Modifies a cbus record. (All data fields must be valid. See the *IPF Batch User's Guide*.)
O	Retrieves the output data. (The case must be solved; all id fields must be valid. See the *IPF Batch User's Guide*.)

The structure members are accessed as follows:

`cb.i.cbus.type[3]`	Two character bus record type, for example, +, +A, etc.
`cb.i.cbus.owner[4]`	Three character bus owner.

`cb.i.cbus.name[9]`	Eight character bus name.
`cb.i.cbus.kv`	Base kV of the bus.
`cb.i.cbus.code_year[3]`	Two character extension of type.
`cb.i.cbus.Pload`	Real load in MW belonging to this owner.
`cb.i.cbus.Qload`	Reactive load in Mvar belonging to this owner.
`cb.i.cbus.Gshunt`	Fixed real shunt in MW.
`cb.i.cbus.Bshunt`	Fixed reactive shunt in Mvar.
`cb.i.cbus.Pgen`	Scheduled real power in MW for this owner.
`cb.i.cbus.Qgen_Qmax`	Scheduled reactive power in Mvar (Qgen) or maximum reactive power in Mvar (Qmax).
`cb.i.cbus.Qmin`	Minimum reactive power in Mvar.

The solution values for a continuation bus are accessed as follows:

`cb.s.cbus.Pload`	Solved real load (same as input).
`cb.s.cbus.Qload`	Solved reactive load (same as input).
`cb.s.cbus.Gshunt`	Solved real shunt.
`cb.s.cbus.Bshunt`	Solved reactive shunt.

RETURNS

`pf_rec_cbus` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
/* cbus.c
 *  This program is designed to test the CFLOW retrieval of
 *  continuation bus data from the currently loaded case.
 */
#include <stdio.h>
#include <string.h>
#include "cflowlib.h"

main( int argc, char *argv[] )
{
```

```
      pf_rec  r;                    /* CFLOW structure */
      int    error, status;
      FILE *out;

      pf_cflow_init( argc, argv );
      out = fopen ("cbus.rpt", "w");

   /* CBUS DATA */

        fprintf (out,"\n******** CBUS DATA ********\n\n");
        error = pf_rec_bus( &r, "F" );         /* get first bus in case */
        status = pf_rec_cbus( &r, "F1" );  /* is there a cbus record?  */

       while (!error)
       {
         do {
         error = pf_rec_bus( &r, "N" );         /* get next bus in case */
         status = pf_rec_cbus( &r, "F1" );    /* is there a cbus record?
   */
           } while (status);

          while ( !status )       /* loop on bus with cbus record(s) */
             {
            status = pf_rec_cbus( &r, "O" );
           fprintf (out,"Type Own  Bus Name       Pload  Qload  Gshunt
   Bshunt    Pgen  Qgn-mx    Qmin  \n");
           fprintf (out," %s %s  %s%5.1f %7.1f %7.1f %7.1f %7.1f %7.1f
   %7.1f %7.1f\n\n",
   r.i.cbus.type,r.i.cbus.owner,r.i.cbus.name,r.i.cbus.kv,
   r.s.cbus.Pload,r.s.cbus.Qload,r.s.cbus.Gshunt,r.s.cbus.Bshunt,
   r.i.cbus.Pgen ,r.i.cbus.Qgen_Qmax,r.i.cbus.Qmin);
            status = pf_rec_cbus( &r, "N1" );
            }
       }

      fclose (out);

      pf_cflow_exit();
   }
```

4.5.34 PF_REC_COMMENTS

SYNTAX

```
int pf_rec_comments ( pf_comments *comm, char *action );
```

`pf_comments *comm` — A pointer to a structure of type `pf_comments`.

`char *action` — A string designating the action to be performed . See Table 4-11 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_comments` retrieves or modifies the case name, project title, and case comments.

Table 4-11. Comments Action Codes

Code	Meaning
G	Retrieves the case comments.
M	Modifies the case comments. All data is updated with the contents of the record.

The structure members are accessed as follows:

`comm.case_name[11]` — Ten character string containing caseid.

`comm.case_descrip[21]` — Twenty character string containing case description.

`comm.h[3][133]` — A character array containing case headers. The first one, `h[0]`, is generated by IPF, and contains the program version, the caseid and description, and the date of the run. The other two are user-specified.

`comm.c[20][121]` — A character array containing case comments.

RETURNS

`pf_rec_comments` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
pf_comments   c;              /* CFLOW structure */
int    error;
```

```
 FILE *out;

 pf_cflow_init( argc, argv );
 out = fopen ("init.rpt", "w");

/* Obtain case comments */
error = pf_rec_comments (&c, "G");

fprintf (out,"Current case is: %s Description: %s\n\n",
    c.case_name,c.case_descrip);
fprintf (out,"%s\n",c.h[0]);
fprintf (out,"%s\n",c.h[1]);
fprintf (out,"%s\n\n",c.h[2]);
fprintf (out,"%s\n",c.c[0]);
fprintf (out,"%s\n",c.c[1]);
fprintf (out,"%s\n\n",c.c[2]);

fclose (out);
```

4.5.35 PF_REC_ITIE

SYNTAX

```
int pf_rec_itie ( pf_rec *it, char *action );
```

`pf_rec *it` — A pointer to a structure of type `pf_rec`, supplied by the calling routine.

`char *action` — A string designating the action to be performed on an intertie record. See Table 4-12 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_itie` retrieves, adds, modifies, and deletes intertie data.

Table 4-12. Intertie Action Codes

Code	Meaning
F	Retrieves the first intertie record.
N	Retrieves the next intertie record. (Name1 and Name2 must be valid.)
G	Retrieves the rest of the intertie record. (Name1 and Name2 must be valid.)
D	Deletes an intertie record. (Name1 and Name2 must be valid.)
A	Adds an intertie record. (All data fields must be valid. See the *IPF Batch User's Guide*.)
M	Modifies an intertie record. (All data fields must be valid. See the *IPF Batch User's Guide*.)
O	Retrieves the solution output. (The case must be solved and name1 and name2 must be valid.)

The structure members are accessed as follows:

`it.i.itie.type[3]` — Two character record type, here "I" for intertie record.

`it.i.itie.area1_name[11]` — Ten character area1 name.

`it.i.itie.area2_name[11]`	Ten character area2 name.
`it.i.itie.sched_export`	Scheduled export power.
`it.s.itie.Pexport`	Solution export power.
`it.s.itie.Pcirc`	Solution circulating current.
`it.s.itie.input_exists`	An integer indicating whether intertie values are internally or externally generated; `0` = no input record (internally generated intertie values); `1` = input data is from input record.

RETURNS

`pf_rec_itie` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
pf_rec itie;
int status;
char areaname1[11], areaname2[11];

pf_init_itie("I ", areaname1, areaname2);
status = pf_rec_itie (&itie, "G");
```

4.5.36 PF_REC_QCURVE

SYNTAX

```
int pf_rec_qcurve ( pf_rec *pq, char *action );
```

`pf_rec *pq` — A pointer to a calling routine-supplied structure of type `pf_rec`.

`char *action` — A string designating the action to be performed on `qcurve` record. See Table 4-13 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_qcurve` retrieves, modifies, adds, or deletes PQ curve data.

Table 4-13. Q Curve Action Codes

Code	Meaning
G	Retrieves the rest of the Q curve records associated with a given bus. (Name and kV must be valid. See the *IPF Batch User's Guide*.)
D	Deletes a Q curve record. (Name and kV must be valid. See the *IPF Batch User's Guide*.)
M	Modifies a Q curve record. (Valid only for activation or inactivation. See the *IPF Batch User's Guide*.)

The structure members are accessed as follows:

`pq.i.qcurve.type[3]` — Two character record type – here QP.

`pq.i.qcurve.PU_code[3]` — Two character code – PU for per unit or blank for kV values.

`pq.i.qcurve.active` — One character code – A for active or * for inactive.

`pq.i.qcurve.bus_name[9]` — Eight character bus name.

`pq.i.qcurve.bus_kV` — Base kV of the bus.

`pq.i.qcurve.Pgen[10]` — Real power levels in MW, for the reactive capability curve.

`pq.i.qcurve.Qmax[10]` Corresponding maximum reactive power values in Mvar.

`pq.i.qcurve.Qmin[10]` Corresponding minimum reactive power values in Mvar.

RETURNS

`pf_rec_qcurve` returns 0 if it is successful; otherwise, it returns 1.

4.5.37 PF_REC_XDATA

SYNTAX

```
int pf_rec_xdata ( pf_rec *x, char *action );
```

`pf_rec *x` — A pointer to a structure of type `pf_rec`, supplied by the calling routine.

`char *action` — A string designating the action to be performed on a switched reactance record. See Table 4-14 for the codes and their meanings. Either upper or lower case is acceptable.

DESCRIPTION

`pf_rec_xdata` retrieves, modifies, and adds switched reactance (X) data.

Note: The delete function is handled by changing the BX bus to another bus type or deleting the BX bus.

Table 4-14. Switched Reactance Action Codes

Code	Meaning
F	Retrieves the first xdata record in a case.
N	Retrieves the next xdata record in a case. (Name and kV must be valid. See the *IPF Batch User's Guide*.)
G	Retrieves the xdata record associated with bus_name and bus_kV.
A	Adds an xdata record. (All required data must be valid. See the *IPF Batch User's Guide*.)
M	Modifies an xdata record. (All required data must be valid. See the *IPF Batch User's Guide*.)
O	Retrieves the output data. (The case must be solved; all id fields must be valid. See the *IPF Batch User's Guide*.)

The structure members are accessed as follows:

`x.i.xdata.type[3]` — Two character array designating the record type — here "X" for switched reactance record.

`x.i.xdata.owner[4]`	Three character array designating an owner.
`x.i.xdata.bus_name[9]`	Eght character bus name.
`x.i.xdata.bus_kv`	Base kV of the BX bus.
`x.i.xdata.rmt_name[9]`	Eight character remote bus name.
`x.i.xdata.rmt_kv`	Remote bus base kV.
`x.i.xdata.step[8]`	Number of each reactance value available (integer).
`x.i.xdata.delta[8]`	Magnitude of reactance values available.
`x.s.xdata.step[8]`	Actual number of steps used.
`x.s.xdata.delta[8]`	Actual reactance amounts used.

RETURNS

`pf_rec_xdata` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

4.5.38 PF_RENAME Functions Rename various entities

The `pf_rename_xxx` functions allow renaming of various entities in the powerflow. These functions utilize the powerflow change records of type `Z`.

4.5.39 PF_RENAME_AREA

SYNTAX

```
int pf_rename_area ( char *oldname, char *newname );
```

`char *oldname`	A string representing an area name to be changed.
`char *newname`	A string representing an area name that will become the new name.

DESCRIPTION

`pf_rename_area` renames an area.

RETURNS

`pf_rename_area` returns 0 (zero) if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_rename_area ( "NORTHWEST", "NW AREA");
```

SEE ALSO

`pf_rename_zone`

4.5.40 PF_RENAME_BUS

SYNTAX

```
int pf_rename_bus ( char *oldname, float kv, char *newname,
 float newkv );
```

`char *oldname`	A string representing a bus name to be changed.
`float kv`	A floating point number representing the base kV of the bus.
`char *newname`	A string representing the new name.
`float newkv`	A floating point number representing the new base kV.

DESCRIPTION

`pf_rename_bus` renames a bus and re-maps all associated data to the new name.

RETURNS

`pf_rename_bus` returns 0 (zero) if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_rename_bus ( "BELL", 69., "BELL1", 60.6 );
```

4.5.41 PF_RENAME_ZONE

SYNTAX

```
int pf_rename_zone ( char *oldname, char *newname );
```

`char *oldname` A string representing a zone name to be replaced.

`char *newname` A string representing the new zone name.

DESCRIPTION

`pf_rename_zone` renames a zone. All zone fields for all records in a zone are updated. If the new zone name already exists, a combined zone results if adjacency permits; otherwise, it is an error.

RETURNS

`pf_rename_zone` returns 0 (zero) if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_rename_zone (" 7", "N7");
```

SEE ALSO

`pf_rename_area`

4.5.42 PF_SAVE Functions Tell powerflow to save data to a file

The `pf_save_xxx` functions allow use of the various powerflow save file functions. For more information on save file capability, see the *IPF ADVANCED* manual.

4.5.43 PF_SAVE_CHANGES

SYNTAX

```
int pf_save_changes ( char *filename );
```

`char *filename` A string representing a file name.

DESCRIPTION

`pf_save_changes` saves to a change file the input data changes you have made to the currently resident base case data.

RETURNS

`pf_save_changes` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_save_changes ("mychanges.chg");
```

4.5.44 PF_SAVE_NETDATA

SYNTAX

```
int pf_save_netdata (char *filename, char *dialect, char *ratings,
            int size);
```

`char *filename`	A string representing a file name.
`char *dialect`	A string having the following possible values: `BPA`, `WSCC`, `WSCC1`, or `PTI`. These refer to different forms that the output file can take. See Appendix F of the *IPF Batch User's Guide* for the differences in dialects.
`char *ratings`	A string having the following possible values: `EXTENDED`, `NOMINAL`, or `MIN_EXTENDED`. See Appendix F of the *IPF Batch User's Guide* for a description of these options for ratings.
`int size`	An integer representing the size of output records — either 80 or 120. The choice of 120 is valid only with the `BPA` dialect.

DESCRIPTION

`pf_save_netdata` saves data from a Powerflow base case in ASCII format.

RETURNS

`pf_save_netdata` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_save_netdata (“mychanges.net”,”WSCC”,”NOMINAL”,80);
```

4.5.45 PF_SAVE_NEWBASE

SYNTAX

```
int pf_save_newbase ( char *filename );
```

`char *filename` A string representing a file name.

DESCRIPTION

`pf_save_newbase` saves the currently resident base case in its current state to the specified filename.

RETURNS

`pf_save_newbase` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_save_newbase ("mychanges.bse");
```

4.5.46 PF_SAVE_WSCC_STAB_DATA

SYNTAX

```
int pf_save_wscc_stab_data ( char *filename, char *type);
```

`char *filename`	A string representing a file name.
`char *type[7]`	A string representing the type of file format of the saved file. The type values are either ASCII or BINARY.

DESCRIPTION

`pf_save_wscc_stab_data` saves the power flow data required for input to the WSCC Stability program in either ASCII or binary form depending on the type argument value.

RETURNS

`pf_save_wscc_stab_data` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
error = pf_save_wscc_stab_data ("mychanges.asif","ASCII" );
```

4.5.47 PF_SELECT_BASE Select OLDBASE or REFBASE

SYNTAX

```
int pf_select_base( char *base );
```

`char *base;` A character indicating which set of data other commands act upon:
'O' OLDBASE data
'R' REFBASE data.

DESCRIPTION

`pf_select_base` allows the `pf_rec` functions to access the input and solution data in either the primary base case (OLDBASE) loaded with `pf_load_oldbase` or the reference (alternate) base case loaded with `pf_load_refbase`. The accessed base case initially defaults to the OLDBASE data.

RETURNS

`pf_select_base` returns 0 if it is successful; otherwise, it returns 1.

EXAMPLE

```
pf_rec br;

pf_load_refbase("J98CY94.BSE");
pf_select_base('R');
pf_rec_branch(&br, "F");
```

4.5.48 PF_SOLUTION Solve the current case

SYNTAX

```
int pf_solution( char *options );
char *options;      solution options separated by newline (\n)
```

DESCRIPTION

`pf_solution` causes the powerflow process to initiate a solution on the currently resident base case data. If you wish to use the default options, then use a null string (`""`) for the options string. See the *IPF BATCH* manual for more information on solution options.

RETURNS

`pf_solution` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
#include "cflowlib.h"

main(int argc, char *argv[]) {
  pf_cflow_init(argc, argv);
  printf("pf_cflow_init\n");
  printf("pf_load_netdata=%d\n",pf_load_netdata("fgrove.net"));
  printf("pf_solution=%d\n",pf_solution(""));
  pf_cflow_exit();
```

4.5.49 PF_USER Functions Access to powerflow User Analysis

The `pf_user_xxx` functions provide a means of using the User Analysis features of the powerflow.

See the *IPF BATCH* and *IPF ADVANCED* manuals for further details on User Analysis capabilities of the powerflow.

4.5.50 PF_USER_BRANCH

SYNTAX

```
int pf_user_branch ( char *symbol, pf_rec *r, char *type );
```

`char *symbol` A pointer to a string containing a symbol name.

`pf_rec *r` A pointer to a structure of type `pf_rec`.

`char *type` A pointer to a string indicating the quantity type.

DESCRIPTION

`pf_user_branch` builds a symbol definition and corresponding comment card based on the data in a `pf_rec` structure and the supplied symbol and type and loads them into the user-analysis arrays in powerflow (IPF). It sends a command constructed as follows:

```
/LOADDEF
> DEFINE_TYPE <symbol_type>
LET <symbol_name> = <bus1_name> <bus1_kv>[*] <bus2_name> <bus2_kv>[*]
C <symbol_name> = $<symbol_name>/F15.7
```

Table 4-15 Branch Flow Type Codes

type	<symbol_type>
P	BPANCH_P
Q	BRANCH_Q

Bus names and voltages are derived from the pf_rec branch structure. Blanks in the bus names are replaced by pound signs (#). An asterisk (*) determines at which terminal the line flow is computed. If the metering point in the branch data is 0 or 1, the first bus is selected, if 2 then the second. Symbol names are limited to six characters and are case insensitive, but retain case for the comment card.

RETURNS

pf_user_branch returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
float p;
pf_rec r;

pf_rec_branch(&r, "F");
pf_user_init_def();
pf_user_branch("L1", &r, "P");
pf_user_sub_def("base");
pf_user_quantity("L1", "", &p);
printf("BRANCH_P = %6.2f\n", p);
```

4.5.51 PF_USER_BUS

SYNTAX

```
int pf_user_bus ( char *symbol, pf_rec *r, char *suffix );
```

`char *symbol` A pointer to a string containing a symbol name.

`pf_rec *r` A pointer to a structure of type `pf_rec`.

`char *suffix` A pointer to a string containing the BUS_INDEX suffix.

DESCRIPTION

`pf_user_bus` builds a symbol definition and corresponding comment card based on the data in a `pf_rec` structure and the supplied symbol and suffix and loads them into the user-analysis arrays in powerflow (IPF). It sends a command constructed as follows:

```
/LOADDEF
> DEFINE_TYPE  BUS_INDEX
LET <symbol_name> = <bus_name> <bus_kv>
C <symbol_name><index_suffix> = $<symbol_name><index_suffix>/F15.7
```

Bus name and voltage is derived from the `pf_rec` bus structure. Blanks in the bus names are replaced by pound signs (#). Symbol names are limited to six characters and are case insensitive, but retain case for the comment card. The suffix must contain the period (i. e. ".VK").

RETURNS

`pf_user_bus` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
float v;
pf_rec r;

pf_rec_bus(&r, "F");
pf_user_init_def();
pf_user_bus("V1", &r, ".VK");
pf_user_sub_def("base");
pf_user_quantity("V1", ".VK", &v);
printf("V in kV  = %4.1f\n", v);
```

4.5.52 PF_USER_COMMENT

SYNTAX

```
int pf_user_comment ( char *symbol, char *suffix, char *format );
```

`char *symbol` A pointer to a string containing a symbol name.

`char *suffix` A pointer to a string containing an index suffix.

`char *format` A pointer to a string containing format code.

DESCRIPTION

`pf_user_comment` builds a comment record and loads it into the user-analysis arrays in powerflow (IPF). It sends a command constructed as follows:

```
/LOADDEF
C <symbol_name><symbol_suffix> = $<symbol_name><symbol_suffix><format>
```

All data is case insensitive, however, the case is preserved. Symbol names are limited to six characters. The suffix is optional and is used for BUS_INDEX and ZONE_INDEX data types. Use a null string ("") if the suffix is not applicable. The format obeys the FORTRAN convention and can be either floating point (i. e. /F8.3) or text (i. e. /A7). The default is /F6.0. The format string must include the slash (/).

Comment cards constructed by `pf_user_comment` are designed to be processed by either `pf_user_quantity` to retrieve floating point values or `pf_user_string` to retrieve textual information.

RETURNS

`pf_user_comment` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
float p_in, kv;
char  user_name[8];

pf_user_init_def();
pf_user_define("L1", "ASTOR#TP 115 SEASIDE 115", "BRANCH_P");
pf_user_define("B1", "CHIEF#JO 500", "BUS_INDEX");
pf_user_define("U1", "USER", "OLDBASE");
pf_user_comment("L1", "", "/F8.3");
pf_user_comment("B1", ".VK", "/F8.3");
pf_user_comment("U1", "", "/A7");
pf_user_sub_def("base");
pf_user_quantity("L1", "", &p_in);
pf_user_quantity("B1", ".VK", &kv);
pf_user_string("U1", 7, user_name);
printf("P_IN = %6.2f, kV = %4.1f, User is %s\n", p_in, kv, user_name);
```

4.5.53 PF_USER_DEFINE

SYNTAX

```
int pf_user_define ( char *symbol, char *id, char *type );
```

`char *symbol` A pointer to a string containing a symbol name.

`char *id` A pointer to a string containing quantity identity.

`char *type` A pointer to a string containing symbol type.

DESCRIPTION

`pf_user_define` builds a symbol definition and loads it into the user-analysis arrays in powerflow (IPF). It sends a command constructed as follows:

```
/LOADDEF
> DEFINE_TYPE <symbol_type>
LET <symbol_name> = <id_of_computed_quantity>
```

If blanks are part of the quantity id, substitute them with pound signs (#). All data is case insensitive. Symbol names are limited to six characters. Use blanks or commas to separate identity items.

RETURNS

`pf_user_define` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
float p_in, kv;
char  user_name[8];

pf_user_init_def();
pf_user_define("L1", "ASTOR#TP 115 SEASIDE 115", "BRANCH_P");
pf_user_define("B1", "CHIEF#JO 500", "BUS_INDEX");
pf_user_define("U1", "USER", "OLDBASE");
pf_user_comment("L1", "", "/F8.3");
pf_user_comment("B1", ".VK", "/F8.3");
pf_user_comment("U1", "", "/A7");
pf_user_sub_def("base");
pf_user_quantity("L1", "", &p_in);
pf_user_quantity("B1", ".VK", &kv);
pf_user_string("U1", 7, user_name);
printf("P_IN = %6.2f, kV = %4.1f, User is %s\n", p_in, kv, user_name);
```

4.5.54 PF_USER_INIT_DEF

SYNTAX

```
int pf_user_init_def ();
```

DESCRIPTION

`pf_user_init_def` initializes the user analysis arrays in powerflow (IPF). It should be called prior to other user analysis functions. It sends the command `/INITDEF` to powerflow.

RETURNS

`pf_user_init_def` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
char user_def[1000];

strcpy(user_def,"> DEFINE_TYPE BRANCH_P\n");
strcat(user_def," LET A1 = ASTOR#TP 115 SEASIDE 115\n");
strcat(user_def,"C ASTOR#TP 115 SEASIDE 115 P_in = $A1");
pf_user_init_def();
pf_user_load_def(user_def);
pf_user_sub_def("base");
printf("%s\n", reply_pf);
```

4.5.55 PF_USER_ITIE

SYNTAX

```
int pf_user_itie ( char *symbol, pf_rec *r, char *type );
```

`char *symbol` A pointer to a string containing a symbol name.

`pf_rec *r` A pointer to a structure of type pf_rec.

`char *type` A pointer to a string indicating the quantity type.

DESCRIPTION

`pf_user_itie` builds a symbol definition and corresponding comment card based on the data in a `pf_rec` structure and the supplied symbol and type and loads them into the user-analysis arrays in powerflow (IPF). It sends a command constructed as follows:

```
/LOADDEF
> DEFINE_TYPE <symbol_type>
LET <symbol_name> = <area1_name> <area2_name>
C <symbol_name> = $<symbol_name>/F15.7
```

Table 4-16 Intertie Flow Type Codes

type	<symbol_type>
P	INTERTIE_P
Q	INTERTIE_Q
S	INTERTIE_P_SCHEDULED

Area names are derived from the `pf_rec` intertie structure. Blanks in the area names are replaced by pound signs (#). Symbol names are limited to six characters and are case insensitive, but retain case for the comment card.

RETURNS

`pf_user_itie` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
float p;
pf_rec r;

pf_rec_itie(&r, "F");
pf_user_init_def();
pf_user_itie("I1", &r, "P");
pf_user_sub_def("base");
pf_user_quantity("I1", "", &p);
printf("INTERTIE_P = %6.2f\n", p);
```

4.5.56 PF_USER_LOAD_DEF

SYNTAX

```
int pf_user_load_def ( char *definitions );
```

`char *definitions` A pointer to a string containing User Analysis Define statements.

DESCRIPTION

`pf_user_load_def` loads the user analysis arrays in powerflow (IPF) with the specified symbol definitions. It sends the command `/LOADDEF` , followed by a newline (`\n`) separated list of definitions, to powerflow.

RETURNS

`pf_user_load_def` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
char user_def[1000];

strcpy(user_def,"> DEFINE_TYPE BRANCH_P\n");
strcat(user_def," LET A1 = ASTOR#TP 115 SEASIDE 115\n");
strcat(user_def,"C ASTOR#TP 115 SEASIDE 115 P_in = $A1");
pf_user_init_def();
pf_user_load_def(user_def);
pf_user_sub_def("base");
printf("%s\n", reply_pf);
```

4.5.57 PF_USER_QUANTITY

SYNTAX

```
int pf_user_quantity ( char *symbol, char *suffix, float *quantity );
```

`char *symbol` A pointer to a string containing a symbol name.

`char *suffix` A pointer to a string containing an index suffix.

`float *quantity` A pointer to a float to hold the retrieved quantity.

DESCRIPTION

`pf_user_quantity` searches the `reply_pf` buffer after a `pf_user_sub_def` function is called for "<symbol_name><index_suffix> = ", where the suffix is optional, and scans in the floating point value that immediately follows. Comment cards can be built with `pf_user_comment` or any other applicable method. The case of the symbol and suffix must match. The suffix is optional and is used for BUS_INDEX and ZONE_INDEX data types. Use a null string ("") for the suffux, if the data type does not use a suffix. The suffix must include the period (i.e. ".VK").

RETURNS

`pf_user_quantity` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
float p_in, kv;
char  user_name[8];

pf_user_init_def();
pf_user_define("L1", "ASTOR#TP 115 SEASIDE 115", "BRANCH_P");
pf_user_define("B1", "CHIEF#JO 500", "BUS_INDEX");
pf_user_define("U1", "USER", "OLDBASE");
pf_user_comment("L1", "", "/F8.3");
pf_user_comment("B1", ".VK", "/F8.3");
pf_user_comment("U1", "", "/A7");
pf_user_sub_def("base");
pf_user_quantity("L1", "", &p_in);
pf_user_quantity("B1", ".VK", &kv);
pf_user_string("U1", 7, user_name);
printf("P_IN = %6.2f, kV = %4.1f, User is %s\n", p_in, kv, user_name);
```

4.5.58 PF_USER_REPORT

SYNTAX

```
int pf_user_report ( char *filename, char *output );
```

`char *filename` A character string representing a user analysis file name.

`char *output` A character string representing an output report file name.

DESCRIPTION

`pf_user_report` loads a user analysis file for generating customized analysis listings. The requested report is appended to the output file, which is created if it doesn't already exist.

RETURNS

`pf_user_report` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
pf_cflow_init(argc, argv);
pf_load_oldbase("fgrove.bse");
pf_solution();
pf_user_report("sum_define.user", "sum_define.rep");
```

4.5.59 PF_USER_STRING

SYNTAX

```
int pf_user_string ( char *symbol, int length, char *info );
```

`char *symbol`	A pointer to a string containing a symbol name.
`int length`	An integer specifying the number of characters to scan.
`char *info`	A pointer to a destination string for the scanned data.

DESCRIPTION

`pf_user_string` searches the `reply_pf` buffer after a `pf_user_sub_def` function is called for "<symbol_name> = " and scans into `*info` the number of immediately following characters specified by length. Comment cards can be built with `pf_user_comment` or any other applicable method. The case of the symbol name must match. Where applicable include the suffix in the symbol name string.

RETURNS

`pf_user_string` returns 0 if it is successful; otherwise, it returns -1.

EXAMPLE

```
float p_in, kv;
char  user_name[8];

pf_user_init_def();
pf_user_define("L1", "ASTOR#TP 115 SEASIDE 115", "BRANCH_P");
pf_user_define("B1", "CHIEF#JO 500", "BUS_INDEX");
pf_user_define("U1", "USER", "OLDBASE");
pf_user_comment("L1", "", "/F8.3");
pf_user_comment("B1", ".VK", "/F8.3");
pf_user_comment("U1", "", "/A7");
pf_user_sub_def("base");
pf_user_quantity("L1", "", &p_in);
pf_user_quantity("B1", ".VK", &kv);
pf_user_string("U1", 7, user_name);
printf("P_IN = %6.2f, kV = %4.1f, User is %s\n", p_in, kv, user_name);
```

4.5.60 PF_USER_SUB_DEF

SYNTAX

```
int pf_user_sub_def ( char *source );
```

`char *source` A pointer to a string designating the source base case as "BASE" or "ALTERNATE_BASE" (may be abreviated to "ALT").

DESCRIPTION

`pf_user_sub_def` performs character string substitutions using computed base case quantities upon the tokens defined with the >DEFINE statements within comment records sent to powerflow either through the `pf_user_load_def` function or a User Analysis file. The return message is available in the global buffer `reply_pf`. It sends the command `/SUBDEF, SOURCE=<name>`, where name is either "BASE" or "ALTERNATE_BASE" ("ALT").

RETURNS

`pf_user_sub_def` returns 0 if it is successfull; otherwise, it returns -1.

EXAMPLE

```
char user_def[1000];

strcpy(user_def,"> DEFINE_TYPE BRANCH_P\n");
strcat(user_def," LET A1 = ASTOR#TP 115 SEASIDE 115\n");
strcat(user_def,"C ASTOR#TP 115 SEASIDE 115 P_in = $A1");
pf_user_init_def();
pf_user_load_def(user_def);
pf_user_sub_def("base");
printf("%s\n", reply_pf);
```

INDEX