U.S. DEPARTME
National Bureau of Standards

NBS Technical Note 1208

PIPE/1000: An Implementation of Piping on an HP-1000 Minicomputer

N. L. Seidenman

National Bureau of Standards

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, the Institute for Computer Sciences and Technology, and the Center for Materials Science.

The National Measurement Laboratory

Provides the national system of physical and chemical measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; provides advisory and research services to other Government agencies; conducts physical and chemical research; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

- Basic Standards[2]
- Radiation Research
- Chemical Physics
- Analytical Chemistry

The National Engineering Laboratory

Provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

- Applied Mathematics
- Electronics and Electrical Engineering[2]
- Manufacturing Engineering
- Building Technology
- Fire Research
- Chemical Engineering[2]

The Institute for Computer Sciences and Technology

Conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

- Programming Science and Technology
- Computer Systems Engineering

The Center for Materials Science

Conducts research and provides measurements, data, standards, reference materials, quantitative understanding and other technical information fundamental to the processing, structure, properties and performance of materials; addresses the scientific basis for new advanced materials technologies; plans research around cross-country scientific themes such as nondestructive evaluation and phase diagram development; oversees Bureau-wide technical programs in nuclear reactor radiation research and nondestructive evaluation; and broadly disseminates generic technical information resulting from its programs. The Center consists of the following Divisions:

- Inorganic Materials
- Fracture and Deformation[3]
- Polymers
- Metallurgy
- Reactor Radiation

[1] Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Gaithersburg, MD 20899.

[2] Some divisions within the center are located at Boulder, CO 80303.

[3] Located at Boulder, CO, with some elements at Gaithersburg, MD.

NBS Technical Note 1208

PIPE/1000: An Implementation of Piping on an HP-1000 Minicomputer

N. L. Seidenman
Office of Standard Reference Data
National Measurement Laboratory
National Bureau of Standards
Gaithersburg, MD 20899

March 1985

U.S. Department of Commerce
Malcolm Baldrige, Secretary
National Bureau of Standards
Ernest Ambler, Director

National Bureau of Standards
Technical Note 1208
Natl. Bur. Stand. (U.S.),
Tech. Note 1208,
50 pages (Mar. 1985)
CODEN: NBTNAE

U.S. Government Printing Office
Washington: 1985

For sale by the Superintendent
of Documents,
U.S. Government Printing Office
Washington, DC 20402

PIPE/1000

Table of Contents

ABSTRACT

Piping is a system by which programs can communicate so as to coordinate their respective functions in a synchronized effort aimed at the completion of a given task. Piping is one of the strong points of the increasingly popular operating system UNIX, developed at Bell Laboratories and licensed by AT&T. This paper describes an implementation of piping in a non-UNIX environment; in particular, on an HP-1000 minicomputer.

KEYWORDS

C; computer languages; piping; programming; UNIX.

DISCLAIMER

This report was prepared under the sponsorship of the United States Government. Neither the United States, nor the United States National Bureau of Standards, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information or data disclosed, or represent that its use would not infringe upon privately owned rights.

Certain trade names and company products are mentioned in the text in order to specify or identify the products adequately where generic designations would be insufficient. In no case does such identification imply recommendation or endorsement, or conversely, disparagement or criticism by the National Bureau of Standards, nor does it imply that any specific product is necessarily the best or worst available for the purpose.

PIPE/1000: An Implementation of Piping on an HP-1000 Minicomputer

N. L. Seidenman
Office of Standard Reference Data
National Measurement Laboratory
National Bureau of Standards
Gaithersburg, MD 20899

INTRODUCTION

In 1975, Western Electric, a subsidiary of AT&T, began licensing the UNIX(tm)[1] operating system. This system, developed at AT&T's Bell Laboratories, provides an environment in which programs can be made to work together toward the completion of a given task. Central to the UNIX system user interface is the SHELL which acts as a command interpreter, file manager, and scheduler. One of the more powerful features of the SHELL is its ability to reroute or redirect the default input (stdin) and default output (stdout) of a program or command. This is done using a structure called a PIPE.

Pipes in UNIX are links between the stdout of one program and the stdin of another. We are accustomed to directing output from a program to a file or to a device such as printer or tape drive. The idea of directing the output to another program, however is somewhat new. Actually, it shouldn't seem all that strange since most file processing is done by several programs anyway. The first program does whatever processing it needs to do on the file and finishes. The next program is then executed using the output of the first program as its input. Several programs may be involved in this process but the pattern is always the same; read a file, process as you go, and write it out to another file. Piping eliminates the "middle-man" files by redirecting the output of one program directly into the input of the next.

Implementations of piping are largely dependent on the particular computer. In a true UNIX environment pipes are connections between programs which run concurrently, each processing the data

---

UNIX is a registered trade mark of Bell Laboratories in a synchronous fashion. The SHELL takes care of rerouting the input and output (I/O) so that programs written for a UNIX environment need only make calls to functions which, by default, use stdin and stdout in order to take advantage of piping.

In the implementation discussed here files are used to simulate pipes. These files are created as they are needed and destroyed when the programs using them have finished. The user never sees this happen, however. The file names are guaranteed to be unique so that more than one session can use the PIPE system without concern for mixups in ownership of a given pipe file. The system described in this paper has come to be known as PIPE/1000 or PIPE for short.

ENVIRONMENT

It is important to give some information here about the computer and operating system on which PIPE was developed since they had a direct bearing on the way it was finally implemented.

The computer that was used is a 16-bit Hewlett/Packard (HP) 1000 F-series model 65 with hardware floating point processor, firmware vector instruction set and one megabyte of main memory. Disc storage totals 469 megabytes. The resident operating system is HP's RTE-6/VM virtual memory system using the C.83 revision. C.83 is the first version of RTE-6 to support a hierarchal file system commonly found on the HP-1000 A-series or on VAX computers. The system was "gen'ed" with roughly 4K words of System Available Memory (SAM) which is used by the operating system (opsys) as scratch memory. The significance of this will be seen in the next section.

The routines which comprise the PIPE system are written almost entirely in C[2] (actually HP/C[3], a very close dialect of C). FORTRAN was used in those places where using C would have required inordinate gymnastics on the part of the program (and the programmer!) to make the file system (File Management Package or FMP)[4] calls work properly. This paper does assume a minimal knowledge of C. The reader should also note that as of this writing only programs written in C may take advantage of PIPE.

DESIGN

The overall goal in designing PIPE was the creation of a system which resembled, as closely as possible, true UNIX piping. The term close was understood to mean that a programmer accustomed to UNIX would see little difference between PIPE and UNIX piping. The only difference, if any, would be the addition of a few PIPE system calls. No special I/O functions would be used; only those found in the standard C I/O library (stdio). To regular UNIX users there would be no difference. This includes redirection to other programs (|), files or devices(>), and appending to existing files (>>).

Another concern was portability. PIPE is not likely to be used on systems other then the 1000, but it can be shown that portable programs are also cleaner programs. There is very little system dependent code in PIPE and what little there is can be changed without harm to the rest of the system.

As has been stated already, there are basically two methods of implementation. The first is to use actual program to program communication via mailbox or class i/o. There are two disadvantages to this method. First, class i/o uses SAM to create class buffers. If SAM becomes filled or near filled too quickly, the system crashes. This is a worst-case which could occur easily if one program is generating class buffers at a rate which exceeds the next programs rate of "consuming" or retrieving them. A method of "handshaking" was considered whereby a program would be suspended by a slower running program further down the pipe until a sufficient number of class buffers had been consumed. This adds a level of complexity which in turn adds another degree of vulnerability to failure to the system.

A second disadvantage lies in the way class i/o works. Briefly, a program executes a class "write/read" to create the buffer in SAM. This buffer is of fixed length. The receiving program must then do a class "get" to consume the buffer thus releasing the allocated SAM. The problem lies in the fact that the receiving program must know in advance how big the buffer will be. This becomes particularly difficult when formatted output is used. For instance, doing a formatted print in PIPE is simply a printf() call which is no different than a normal formatted print. Using class i/o the same operation would have required first doing an internal formatted print (sprintf) then taking the length of the resulting string (strlen) , creating the buffer or buffers for the class write/read calls, transmitting the length of the incoming buffers to the next program, and finally sending the buffers themselves down the pipe. Again, this adds another level of complexity and also requires more contorsions on the part of the programmer. In the final product there are only three required function calls, one to initiate redirection, one to continue, and one to complete the operation.

As is evident in the last paragraph, using class i/o requires introducing obtuse and somtimes obstructive code not found in UNIX programs thus violating the basic design premise.

One last problem with a true pipe implementation on the HP-1000 F-series, also pertaining to limited memory resources, is the fact that all of the programs in a true pipe must be running concurrently. Memory on the F-series is divided into a fixed number of partitions. Thus, if there are twenty partitions in the system, and a pipe containing 10 programs is executed, one pipe is now using half of the system memory resources. Two such pipes would effectively monopolize the system.

True UNIX piping could also have been accomplished by writing highly system-dependent code which would connect the Equipment Table (EQT) entry of one program's stdout with another's EQT entry for stdin. This would have required actually writing a shell either as a substitute for or front end to the resident command interpreter, CI[5]. This is not an unlikely possibility in terms of future versions of the system but given the limited personnel resources, the current version offered the greatest return for the least investment.

Before moving on it should be mentioned that the HP-1000 A-series computer architecture along with the RTE-A operating system could possibly support true piping.

As it turns out the method of using pipe files (which heretofore will be referred to as pipefiles) allows the closest approximation to UNIX piping available on the F-series. This will hopefully change as HP migrates to UNIX as its standard operating system in the years to come.

Pipefiles have a major advantage over true pipes on the HP-1000 in that the problem of synchronization is removed. Without bottlenecks between programs the chance of a system crash due to insufficient SAM is eliminated. The dynamic allocation of disc storage allows the pipefile to be as small or as large as necessary without the headache of fixed length records. All communications are now relegated to the opsys, thereby freeing the programmer (and the program) to worry about other things.

The actual code is written in C. There were several reasons for choosing this language as opposed to FORTRAN-77. To begin with, C offers a more or less standard solution to the problem of retrieving runstring information - a key task for the PIPE system; FORTRAN does not. Second, C code is more concise, more compact, and more powerful than FORTRAN. Third, the HP/C product comes with a symbolic debugger which made the task of testing and error catching considerably easier. There is a symbolic debugger available from HP which supports FORTRAN, PASCAL, and the MACRO assembler which is actually quite powerful but was unavailable on the development system at the time.

Program design proceeded in a "top-down" fashion. Lower level functions were kept right where they belong - in the lower levels. Higher level functions know nothing about the routines on which they call. The code is highly modular which simplifies debugging and modification. The end result is a system which works as it was originally intended to, required relatively little time to implement, and is easily modified for porting to other systems.

IMPLEMENTATION

The final product consists of a library of support routines and an "include" file used in "pipeable" programs. With the exception of calls to three routines in this library, a C programmer need do nothing different insomuch as I/O processing is concerned. This is in keeping with the design philosophy of making the system as close to true piping as possible. Below are two programs. The first does not use PIPE; the second does. Notice that there is really very little difference between the two.

```
#include <stdio.h>            /* Without piping */

main(argc, rgv, rmpar)
int argc, rmpar[];
char *argv[];
{

   while (--argc > 0)
      printf((argc > 1) ? "%s " : "%s\n", *++argv);
}


           /* * * * * * * * * * * * * * * */


#include <stdio.h>            /* With piping */
#include <pipe.h>

main(argc, rgv, rmpar)
int argc, rmpar[];
char *argv[];
{
   if (popen(argc, argv, rmpar) == NULL)
      fprintf(stderr, "clog in pipe!\n");
   else
      while (--argc > 0)
         printf((argc > 1) ? "%s " : "%s\n", *++argv);

   pclose(argc, argv, rmpar);
}
```

As is easily seen, there is no difference in the way I/O is performed. The PIPE functions popen and pclose simply initiate and complete redirected I/O, respectively. The same would apply if scanf, getc, or putc calls were used. Before describing a typical application using PIPE there are several routines which need mentioning.

The first two have already been introduced. These are popen and pclose. Popen performs two functions; it calls the runstring processor runstr, and it redirects stdout if necessary. Pclose does the opposite. It constructs a new runstring for the next program in the pipe (if there is a pipe), purges the input pipefile (if any), closes stdout, and schedules the next program in the pipe, suspending itself during this program's execution.

Runstr calls five other subprograms which determine what kind of input redirection is requested (intype), determine the kind of output redirection requested (outtype), build the input file list (bldinlst), build the option list (bldoplst), and return the index in the argv array of the output file (outfdes) if specified. These routines are not considered user callable and thus have not been included in the programmer reference.

There are four kinds or "types" of I/O recognized by the PIPE system. These types are returned as integer tokens by the intype and outtype routines. The first is the default STANDARD type. This is returned by intype or outtype when no redirection is specified. Next is FILEIO. This is returned when I/O is redirected to or from a user specified file. In the case of output being type FILEIO and the specified file does not exist it is created. APPEND is used when the >> operator appears in the runstring. This indicates that output from stdout is to be appended to the specified file. Last and most important is the type PIPE. This token indicates that I/O is to be redirected to and/or from another program. This token should not be confused with the name of the system itself. With the exception of APPEND all types can refer either to input or to output redirection.

Bldinlst is the routine which scans the argv array and builds a list of pointers to parameters which it believes to be input file names. Scanning begins with the second parameter in argv (the first is always the program name) and continues until either an output redirection operator ("|" or ">" or ">>") is encountered or the last parameter has been reached. When an input file is found, bldinlst copies the argv pointer to the inlist array and increments a counter. Once finished, the routine returns this counter to the caller.

Bldoplst does essentially the same thing as bldinlst only its task is easier since all it need look for is a "+" or "-" at the beginning of a parameter. Bldoplst also returns the number of options found. There are two user callable routines associated with this one; options and optval. Options is passed a string argument for which it then scans the oplist. If a match is found then the index in the oplist array is returned. Otherwise an EOF is returned. Optval is used to extract integral data from an option. For example, suppose an option "-c130" appears in the runstring. Options("c") would be used to determine if and where the "C" option appears in the oplist and optval would be called to extract the integer 130 from it.

Outfdes returns the index in the argv array of the output file if FILEIO redirection is used in the outgoing direction. It searches the argv array for a ">" or ">>" operator and stops when either one of these is found or if a piping operator ("|") is found. If a FILEIO or APPEND output redirection operator is found, outfdes then skips over any intervening options and returns the index of the next filename it finds. In the event that a user should enter a faulty runstring such as

```
    proga infile.dat > | progb
```

which would seem to redirect the output to a file named "|", the system ignores the value returned by outfdes. Piped redirection, then, takes precedence over any other type.

There is a third subroutine which is used to handle the redirection of stdin. This is the pnext routine. Pnext will sequentially reopen files in the input list (inlist) built by bldinlst until the end of the list is reached. Once this happens an EOF is then returned. If an error occurs while trying to open a file, pnext returns a NULL. Pnext will always return a positive integer if the input type is STANDARD. Note that unlike output files, input files must exist before they can be opened. An attempt to open a non-existent file will result in an error and pnext returns a NULL.

There are several other routines which are used to initiate and close the spooling of output to system devices; in particular the line printers. These are the spoolon and spooloff family of routines. They are by nature highly system dependent since they make use of the Spool Management Package (SMP) calls in RTE-6. There are only two programs which use them in the set of utilities written at NBS for the HP-1000; lpr which spools output to the high speed line printer, and dpr which spools output to the letter-quality printer.

One last user-callable routine is the datex function. This routine is used to read the system clock and return a six element, short integer array to the caller. The array contains the month, day, year, hours, minutes, and seconds as read from the clock. This is used by the PIPE system itself when creating the output pipefile the name of which is of the form

```
/pipe/_hhmmss.end
```

Using this routine enables PIPE to create pipefiles in the /pipe global directory whose names are, for all intents and purposes, unique.

The testing of the code was facilitated by the inclusion of "debugging" code generated by the HP/C compiler using the -db option. This along with the symbolic debugger SEBUG (Systematic Error eradication and deBUG) included with the HP/C product enabled close examination of PIPE system objects in a run-time environment. The general approach in the initial test phase was to use so-called "loop-back" testing. That is, have a program pipe its output to itself. Lpr was the first program to be "fitted" with the pipe routines. Once this program was working it became the benchmark by which all the others were (and still are) tested since it uses virtually all of the PIPE system calls in one way or another.

HOW IT WORKS

Probably the best way to explain the internal workings of PIPE is to use an example. File redirection is fairly straightforward and so we will look instead at how program to program piping works.

Suppose there are two programs which have been augmented to support PIPE and for clarity we will call them proga and progb. Given the runstring

```
proga input.dat -o | progb final.txt
```

where input.dat and final.txt are sequential files and -o is some arbitrary option.

First proga calls popen which then calls runstr. Runstr determines that the input type is FILEIO and the output type is

PIPE. Next, runstr calls bldinlst and bldoplst to build the input list and option list, respectively, as well as return the number of input files specified and the number of options. Having returned to popen, execution continues with popen recognizing that the output type is PIPE. Popen calls pipename (an internal routine) to create a pipefile name which is unique. It then creates and redirects stdout to this file. In addition, this pipename is saved in the global string buffer pname.

Proga, having checked the value returned by popen to see that it is not NULL goes into the main process loop. This loop is generally of the form

```
while ((input = pnext()) != EOF)
   if (input != NULL)
      dostuff();
```

Pnext starts by opening stdin to input.dat and incrementing a counter. On all but the first call to pnext, stdin is closed and reopened to the next file in the input list. If pnext sees that the counter is equal to the number of input files in the list, it returns an EOF. This counter is internal to pnext and cannot be altered by any other routine (unless it is very clever).

Once the EOF condition is detected the loop is exited and pclose is called to do the cleanup. First pclose closes stdout. Since the output type is PIPE pclose must now build a runstring for and then execute progb. The routine bldrustr is called to build the runstring. This is done by concatenating the parameters to the right of but not including the "|" operator and inserting the pipename (stored in pname) between the first and second parameters. At this point the runstring for progb would be

```
progb /pipe/_hhmmss.end >> final.txt
```

Progb is now scheduled and is passed the above runstring. Assuming no error occurred on the schedule attempt, proga suspends itself and waits for progb to finish.

Progb now goes through the same setup as did proga but this time the input type is found to be PIPE. This becomes significant when pclose is called since the pipefile will otherwise be regarded as an input file. Stdout is reopened to final.txt and execution proceeds until an EOF is returned from pnext. Now when pclose is called it purges the pipefile in addition to closing stdout, and (in this case) passes control back to proga and terminates. Once back in proga the pipe is completed and proga terminates.

Up to thirty objects (filenames, operators, program names, and options) may appear in a pipe not to exceed 100 characters. These limits are fairly arbitrary and can be increased if the need arises. For the great majority of applications, however, they have been found to be more than adequate.

SHORTCOMINGS

PIPE currently offers a simple but powerful implementation of piping, but there are still a few places in which it falls short. The primary weakness which will (-hopefully) be eliminated in future versions is program dependence.

If a program is not fitted to use PIPE, it cannot be used in pipes with other programs that are. This is symptomatic of the file handling system in RTE-6. In operating systems such as UNIX information about a file is maintained by the system. A cell in the "per process data region" is then used to connect a given program's or process's logical I/O references to the actual file or device. In RTE, this information is maintained by the process itself with the system keeping track of how many processes a file is open to (maximum of seven) and to whom the file is open. Whereas the shell can redirect stdin and stdout by file manipulation at the system level, RTE must do so at the process level and, hence, PIPE is program or process dependent.

A good example of this is the word formatter used to initially prepare this article, WOLF. WOLF (Word Oriented Line Formatter) is a public domain word formatter which is distributed by the international HP-1000 users' group, INTEREX. It was written in FORTRAN-IV and, since FORTRAN is not yet supported in PIPE, it is rather difficult to interface with the PIPE system. Currently, the text is sent into a file and the file is then printed using lpr or dpr (which are PIPE'd). It would be preferable to simply include WOLF (or any process for that matter) in a pipe without having to worry about whether or not it has been adapted for PIPE. An interim solution will be to provide a library of routines callable from FORTRAN, PASCL (HP's PASCAL), and MACRO as well as C. The long term solution will be to write a shell which will enable HP-1000 users to enjoy the benefits of UNIX along with the power of a real-time system.

The HP-1000 A-series was mentioned earlier. This machine represents a total rethinking of the HP-1000's architecture and accompanying opsys, particularly in terms of the way it handles I/O and the user interfaces available. The greater part of I/O on the A-series is now handled by the interface cards (multiplexers, controllers, etc.) themselves, leaving the computer free to compute. The CPU is used only to initiate the I/O and then turns the continuation/completion phase over to the card. On the F-series the CPU is involved in all phases of I/O from initiation to completion.

The way the user interfaces interact with the opsys is also different in that although it does not support as sophisticated a multi-user accounting system as RTE-6, it does allow for the inclusion of user interfaces other than CI and FMGR. The Session Monitor accounting system on the F-series currently allows only CI and FMGR.

These improvements pave the way for an interface which will incorporate the benefits of the UNIX shell in a real-time environment.

SUMMARY

PIPE/1000 has by no means attained fixed status. The future holds revisions and improvements which will make the already useful system even better. PIPE does help meet the need for an environment in which program units can be easily combined sufficiently well to make it a system which can be used and is being used today.

REFERENCES

[1] K. Christian, The UNIX Operating System, (John Wiley and Sons, 1983)

[2] B. Kernighan and D. Ritchie, The C Programming Language (Bel l Laboratories, Inc, 1978)

[3] HP/C Reference Manual For the CCS C Compiler (Corporate Computer Systems, Inc., 1983)

[4] RTE-6/VM Programmers Reference Manual, (Hewlett/Packard Company, 1983)

[5] RTE-6/VM CI User's Manual, (Hewlett/Packard Company, 1983)

APPENDIX A: Source Listings

This section contains source listings for all of the PIPE/1000 routines.

It includes both user-callable and internal (not intended for direct use by programmer) routines. User-callable routines are flagged with a "UC" in the page heading.

The source code is available through INTEREX, the International Association of Hewlett/Packard Computer Users along with the binary relocatable files for those who have HP-1000's.

ARGBUFS

This module is used by the HP/C runtime system to hold the argument strings pointed to by argv.

```
MACRO,Q
      NAM _ABUF,7 V1.0  Argv buffer for PIPE system
      ENT _ABUF,_ALEN,.ABUF,_ARGV
      SPC 1
***********************************************************
* THIS MODULE DEFINES THE BUFFER TO BE USED FOR ACQUIRING *
* THE RUN STRING FROM A C PROGRAM. THE SIZE OF _ABUF MUST *
* BE ONE WORD LONGER THAN THE NUMBER OF WORDS (_ALEN/2)   *
* THAT WILL BE RETRIEVED.                                 *
*                                                         *
* History;                                                *
*                                                         *
* Augmented 841029 by N. Seidenman to allow for 30 argu-  *
* ments in a runstring 100 characters long.               *
*                                                         *
***********************************************************
      SPC 1
_ABUF BSS 51          RUN STRING BUFFER
_ALEN DEC -100        NEGATIVE CHARACTER LENGTH OF _ABUF
.ABUF DBL _ABUF       BYTE ADDRESS OF _ABUF
*
_ARGV BSS 31          ROOM FOR 30 ARGUMENTS
      END
```

BLDINLST

This routine is used to build the globally accessible list of input file names, inlist. C and a are the argument count and argument list respectively.

```
int bldinlst(argc, argv)    /* Build the pointer list   */
int argc;                   /* for the input files.     */
char *argv[];  {

   int n, m;

   for (n=0, m=1; m < argc; m++)
      if (isinput(argv[m]))
      inlist[n++] = argv[m];
      else if (isoption(argv[m]))
         continue;
      else
         break;

   return n;
}
```

BLDOPLST

Bldoplst builds the options list in the global array oplist. Options are distinguished by a "+" or "-" in the first characters position of a parameter.

```
extern int isoption();
extern char *optlist[];

int bldoplst(argc, argv)            /* Build options list */
int argc;
char *argv[];  {

   int m, n;

   for (m=1, n=0; m < argc; m++)
           /* If it's an option, keep it. */
      if (isoption(argv[m]))
         optlist[n++] = argv[m];
      else if (*argv[m] == '|')  /* Ooops! Hit a joint. */
         break;

   return n;
}
```

BLDRUSTR

This routine is used to construct the new runstring for the next program in a pipe.

```
extern int ot;
extern char *pname;


char *bldrustr(argc, argv)
int argc;
char *argv[];
{

   int i = 0;
   char a[80], b[80];

   while (*argv[i] != '|' && i < argc)
      ++i;


   if (i < argc)  {
      if (ot == PIPE)
         sprintf(b, "RU,%s,%s", argv[++i], pname);
      else
         sprintf(b, "RU,%s",argv[++i]);

      while (++i < argc)   {
         sprintf(a, "%s,%s", b, argv[i]);
         sprintf(b, "%s", a);
      }
      return b;
   }
   else
      return EOF;
}
```

BUFFERS

This module contains all of the buffers globally accessible by PIPE and by the user's code.

```
#define SIZE 2000

char *inlist[MXPTRS],   /* Put input file lst here */
     *oplist[MAXPTRS],             /* Options list */
     pname[MXFDBS];          /* Pipe·end file name */

int it,                              /* Input type */
    ot,                             /* Output type */
    ni,                   /* Number of input files */
    no,                       /* Number of options */
    ox;            /* Index in argv of output file */

        /* Size-up dynamic memory block */
        /* and runtime stack          */
int _mem[SIZE], _mlen = SIZE;
int _sspc[SIZE], _skln = SIZE;
```

DATEX (UC)

Datex is not limited to PIPE/1000 but can be used for any application in which a gregorian date and/or 24-hour time are required. Datex returns the month, day, year (last two digits), hours, minutes, and seconds as read from the system clock. Note that only the method of accessing the system clock need be changed in order to port this routine to other systems.

```
extern exec();

      /* Days-in-the-month table  */

static int daytab[2][13] = {
   {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
   {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

datex(d)
int d[];
{

   int i, leap;
   int itime[5];
   int year, yearday;

   exec(11, *itime, year);
   yearday = itime[4];
   leap = year % 4 == 0 && year % 100 != 0 ||
                      year % 400 == 0;

   for (i = 1; yearday > daytab[leap][i]; i++)
      yearday -= daytab[leap][i];

   d[0] = i;            /* Month   */
   d[1] = yearday;      /* Day     */
   d[2] = year % 100    /* Year    */
   d[3] = itime[3];     /* Hours   */
   d[4] = itime[2];     /* Minutes */
   d[5] = itime[1];     /* Seconds */
}
```

DSPOOLOF (UC)

Spooling is initiated and completed by a family of four routines called dspoolon, dspoolof, spoolon, and spooloff. Their names are sufficiently descriptive such that further delineation is unnecessary. It should be said, however, that only lpr and dpr use these routines since they are usually at the end of a pipe anyway. Dspoolof and spooloff turn off spooling to the letter-quality printer and draft printer, respectively.

```
extern int runit() alias "FmpRunProgram";
extern long cstrdsc();

static char runstr[80] = "XQ,DSPOFF";

int dspoolof()
{
   char rpname[8], *rpnamptr, ofstr[30];
   int dummy[5], error;
   long fstr = cstrdsc(rpname);


   return (runit(cstrdsc(runstr), dummy, fstr));
}
```

DSPOOLON

This routine turns on spooling to the letter-quality printer.

```
extern int runit() alias "FmpRunProgram";
extern long cstrdsc();

static char runstr[80] = "RU,DSPON";

int dspoolon()
{
   int dummy[5], error;
   char rpname[10];

   return (runit(cstrdsc(runstr), dummy, cstrdsc(rpname)));
}
```

INTYPE

Intype returns an integer token which will indicate to the PIPE system what type of input will occur; STANDARD, FILEIO, or PIPE (see UDEFS).

```
extern int isinput(), isoption();

int intype(argc, argv, rmpar)
int argc, rmpar[];
char *argv[];  {

   int i = 0, j = 1;

   if (rmpar[0] == 0 && rmpar[2] != 0)
                 /* Input type is PIPE */
      if (isinput(argv[1]))
         return PIPE;
      else
         return NULL;  /* Error condition:     */
                       /* No end file found in runstring */

   while (j < argc && i == 0)  /* Look for the first */
                               /* input file.   */
                               /* This will say that  */
                               /* input type is FILEIO */
      if (*argv[j] == '|' || *argv[j] == '>')
         break;
      else if (isoption(argv[j]))
         ++j;
      else
         ++i;

   if (i != 0)
      return FILEIO;
   else
      return STANDARD;  /* None of the above.  */
}
```

ISINPUT

Routine called by bldinlst and intype to determine if a parameter is an input file descriptor.

```
extern int isoption();

int isinput(s)
char s[];  {

   if (*s != '>' && *s != '|' && !isoption(s))
      return 1;
   else
      return 0;
}
```

ISOPTION

Routine used by bldoplst, bldinlst, and outfdes to determine whether or not a parameter is an option.

```
int isoption(s)
char *s;  {

   if (*s == '+' || *s == '-')
      return 1;
   else
      return 0;
}
```

OPTIONS (UC)

This is one of several user-callable routines in PIPE/1000 which enable the user to interact with the system. The caller passes a string for which the option list (oplist) is then found. If a match occurs, the index of the option within the list is returned; otherwise the return value is an EOF.

```
extern int no;
extern char *optlist[], *sloc();

int options(op)
char *op;
{

   int n = 0;

   while (n < no)
      if (sloc(optlist[n], op) != NULL)
         return n;
      else
         n++;

   return EOF;
}
```

OPTVAL (UC)

Integer function which returns a number appended to an option. For example, suppose the option string is "-f14". Optval would be passed the string and a skip value of 2 since the -f is not part of the value.

The return would be 14.

```
int optval(op, skip)
char *op;
int skip;
{

   int k = 0;

   while (skip--)
      *op++;

   while (*op)
      k = (k * 10) + (*op++ - 48);

   return k;
}
```

OUTFDES

This function returns the index in argv of the output file descriptor.

The return is meaningless if the output type is not FILEIO or APPEND.

```
extern int isoption();

int outfdes(argc, argv)
int argc;
char *argv[];  {

   int i=1;

   while (i < argc)
      if (*argv[i] == '>')
         break;
      else if (*argv[i] == '|')
         return NULL;
      else
         ++i;

   while (isoption(argv[++i]));

   return ((argv[i] == NULL) ? NULL : i);
}
```

OUTTYPE

Companion routine to intype. This routine will return an integer token indicating the type of output to perform; STANDARD, FILEIO, APPEND or

PIPE'd.

```
extern char *sloc();

int outtype(argc, argv)
int argc;
char *argv[];  {

   int i = 0;

   while (++i <= argc)
      switch (*argv[i])  {
         case '>':   if (sloc(argv[i], ">>") != NULL)
                        return APPEND;
                     else
                        return FILEIO;
                     break;
         case '|':   return PIPE;
                     break;
      }

   return STANDARD;   /* Default case.     */
}
```

PCLOSE (UC)

Routine to close a pipe and perform cleanup operations.

```
extern ru_it() alias "FmpRunProgram";
extern char *bldrustr();
extern int  ot;
extern char *pname;

int pclose(argc, argv, rmpar)
int argc, rmpar[];
char *argv[];
{
   int error;
   char rs[80];
   char runam[10];

   if (ot == PIPE)  {
      rmpar[0] = 0;
      rmpar[2] = 100;
      sprintf(rs, "%s", bldrustr(argc, argv));

      freopen(1, "w", stdout);

      error = ru_it(cstrdsc(rs), *rmpar, cstrdsc(runam));
      if (error != NULL)
         fprintf(stderr,
                 "\npipe clogged!\ncan't %s\n", rs);
      fpurge(pname);
   }
   exit();
}
```

PIPENAME

This function returns a unique file descriptor to be used as a pipefile descriptor.

```
extern int sprintf();
extern datex();

static char fmt[30] = {
   "_%02d%02d%02d::PIPE"
};

char *pipename()  /* Return a pipe end name */
{
   char temp[64];
   int d[6];

   datex(d);
   sprintf(temp, fmt, d[3], d[4], d[5]);
   return temp;
}
```

PNEXT (UC)

Pnext is used to maintain PIPE system input operations such as opening the next input file in inlist and redirecting input from a pipe.

```
extern char *inlist[];
extern int ni;

static int nextfile = 0;

int pnext()
{
   if (nextfile == ni)
      return EOF;
   else
      return (freopen(inlist[nextfile++], "r", stdin));
}
```

POPEN (UC)

Popen is called to initiated PIPE'd communications. It is passed the argc, argv, and rmpar parameters passed to the main.

```
extern runstr();
extern int  ot,
            ox;
extern char *pname, *pipename();

int popen(argc, argv, rmpar)
int argc, rmpar[];
char *argv[];
{

   extern int _error;
   static char pname2[64];


   runstr(argc, argv, rmpar);

   /* Reopen stdout to the appropriate file.  */

   if (ot == PIPE)   {
      sprintf(pname, "%s", pipename());
      sprintf(pname2, "%s:4:100", pname);
      if (freopen(pname2, "gw", stdout) == NULL)  {
         fprintf(stderr, "_error = %d\n", _error);
         return NULL;
      }
      else
         return 1;
   }
   else if (ot == FILEIO)
      return (freopen(argv[ox], "gw", stdout));
   else if (ot == APPEND)
      return (freopen(argv[ox], "a", stdout));
   else
      return 2;
}
```

RUNSTR

Procedure for processing the runstring parameters. Runstr is passed the argc, argv, and rmpar parameters and calls the routines necessary to determine input and output types, build the input and option lists, and find the output file descriptor (if any).

```
extern int it, ot, ox, no, ni;
extern   intype(),
         outtype(),
         outfdes(),
         bldinlst(),
         bldoplst();

runstr(argc, argv, rmpar)
int argc, rmpar[];
char *argv[];
{

   it = intype(argc, argv, rmpar);
   ot = outtype(argc, argv);
   ox = outfdes(argc, argv);
   ni = bldinlst(argc, argv);
   no = bldoplst(argc, argv);
}
```

SPOOLOFF (UC)

Same as dspoolof only output is sent to draft-quality printer.

```
extern int runit() alias "FmpRunProgram";
extern long cstrdsc();

static char runstr[80] = "XQ,SPOFF";

int spooloff()
{
   char rpname[8], *rpnamptr, ofstr[30];
   int dummy[5], error;
   long fstr = cstrdsc(rpname);

   return (runit(cstrdsc(runstr), dummy, fstr));
}
```

SPOOLON (UC)

Same as dspoolon only output is sent to draft-quality printer.

```
extern int runit() alias "FmpRunProgram";
extern long cstrdsc();

static char runstr[80] = "RU,SPON";

int spoolon()
{
   int dummy[5], error;
   char rpname[10];

   return (runit(cstrdsc(runstr), dummy, cstrdsc(rpname)));
}
```

APPENDIX B: Sample Program Using PIPE/1000

This is the lpr print spooler routine. It calls virtually all of the routines and functions in the PIPE library. It can be used as a model for other PIPE applications.

```
hpc,1,mc,"lpr File lister with headings and paging";
/** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                            *
*                            > > >  LPR  < < <                               *
*                                                                            *
*      Description;                                                          *
*         LPR is the file listing utility found on most UNIX(tm) systems.   *
*         This implementation is  designed to employ most of the features   *
*         found  in the UNIX version.  It  can  be  used  for  generating   *
*         "nice" copy on the scheduling terminal (default)  or  a  desig-   *
*         nated line printer.                                               *
*                                                                            *
*      Usage;                                                                *
*         [ru] lpr InputList [> output] -options                             *
*                                                                            *
*         Like most UNIX utilities, lpr parameters are passed via options   *
*         in the runstring.  Parameters beginning with a + or - character   *
*         are regarded as options.   If  no  '>' appears in the runstring   *
*         then the  output is directed to the scheduling terminal and all   *
*         non-option parameters are treated as input files.                 *
*                                                                            *
*      Explanation of Arguments;                                             *
*         InputList                                                          *
*             One  to  twenty file descriptors which make up the list  of   *
*             files to be used as input.  If a file cannot be opened,  an   *
*             error message is printed on the output device and execution   *
*             continues with the next file.                                 *
*                                                                            *
*         > output                                                           *
*             if a device other than the user's terminal  is  to  be used   *
*             for output then the file or device must be specified  after   *
*             a '>' parameter.                                              *
*                                                                            *
*         -options                                                           *
*             -n          line numbering                                     *
*             -h          headings                                           *
*             -t          trailers                                           *
*             -cn         page width is n columns (132)                      *
*             -lm         page length is m lines (59)                        *
*                                                                            *
*      Example(s);                                                       *
*         lpr heading.txt nice1.txt kp.dat > 6 -n -l40 -c78              *
*             list  files  heading.txt  nice1.txt  kp.dat  to  LU 6 (line *
*             printer) with line numbering, lines are  78 characters long *
*             and pages are 40 lines long.                                *
*                                                                        *
*         lpr -h this.dat -t                                             *
*             lists  this.dat  on the user's  terminal with  headings and *
*             trailers.  Note that options may  appear  anywhere  in  the *
*             runstring.                                                  *
*                                                                        *
*      History;                                                          *
*         841017 - Version 1.0 completed NLS                             *
*         841018 - Column width now applies to entire output. NLS        *
*         841023 - PIPE subroutines now enable prog to prog comm.  NLS   *
*                                                                        *
*      Language(s);                                                      *
*         HP/C Rev.  1.6                                                 *
*                                                                        *
*      Include Files;                                                    *
*         <stdio.h>                                                      *
*         <pipe.h>                                                       *
*                                                                        *
*      External Libraries;                                               *
*         Standard C library (C.LIB).                                    *
*         PIPE subroutine library (PIPE.LIB)                             *
*                                                                        *
*      Additional Comments;                                              *
*                                                                        *
*      Author: Nick Seidenman                                            *
*      Site:   OSRD Computing Facility;                                  *
*              Room B336, Physics                                        *
*              National Bureau of Standards                              *
*              Gaithersburg, MD  20899                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **/
#include <stdio.h>
#include /usr/prog/pipe.h

main(argc, argv, rmpar)
mew doyu, rmpar[];
char *argv[];  {

   int cf;
   int input;

         /* Start up */
```

```
if (popen(argc, argv, rmpar) == NULL) {
   printf("lpr: can't start due to clog in pipe\n");
   printf("%s\n", pname);
   pclose(argc, argv, rmpar);
}

      /* Set up outspooling. */

if (ot == STANDARD)
   if (spoolon())  {
      fprintf(stderr, "lpr: can't open spool file\n");
      pclose(argc, argv, rmpar);
   }
   else
      freopen(6, "w", stdout);

      /* Initialize local parameters */

init();

      /* Diagnostics */

if (options("XX") != EOF)  {
   fprintf(stderr, "it:%d ot:%d ni:%d no:%d ox:%d\n", it, ot, ni, no, ox);
   fprintf(stderr, "input list:\n");
   for (cf = 0; cf < ni; cf++)
      fprintf(stderr, "   %s\n", inlist[cf]);
   fprintf(stderr, "\noption list:\n");
   for (cf = 0; cf < no; cf++)
      fprintf(stderr, "   %s\n", optlist[cf]);
   fprintf(stderr, "\nPIPE end filename\n%s\n", pname);
}

         /* Now copy files specified in inlist to output. */
cf = 0;
while ((input = pnext()) != EOF)  {
   if (input != NULL)
      filestuff(inlist[cf++]);
}

printf(" \f");                  /* One last form feed for the road */

if (ot == STANDARD) {
   spooloff();                  /* Close spool file and queue for print. */

   exit();
}
else
```

```
      pclose(argc, argv, rmpar);   /* Close her up! */
}

            /* Global parameters */


int  pageno,                  /* Page number */
     columns,                 /* Page width  */
     lpp;                     /* Lines per page */

char datestr[12];




init()                        /* Initialize global parameters */
{
   int mdy[6], i;

   pageno = 1;                   /* Defaults */
   columns = 132;
   lpp = 60;

   datex(mdy);
   sprintf(datestr, "%d/%d/%d", mdy[0], mdy[1], mdy[2]);

   if ((i = options("C")) != EOF)  {          /* Change page width */
      columns = 0;
      *optlist[i]++;
      *optlist[i]++;
      while (*optlist[i])
         columns = (columns * 10) + (*optlist[i]++ - 48);
   }

   if ((i = options("L")) != EOF)  {          /* Change page length */
      lpp = 0;
      *optlist[i]++;
      *optlist[i]++;
      while (*optlist[i])
         lpp = (lpp * 10) + (*optlist[i]++ - 48);
   }

   if (options("H") != EOF)          /* Adjust page length for no heading */
      lpp -= 3;

   if (options("T") != EOF)          /* Adjust page length for no tariler */
      lpp -= 3;
   if (options("N") != EOF)          /* Adjust page width for line numbers */
      columns += 7;
}


extern int ifbrk();

filestuff(fname)        /* Copy files to stdout */
char *fname;
{

   int  lines = 1,
        n;
   int c, ccount, newpage = 0;

      /* Determine if line numbering is requested */

   if ((n = options("N") + 1) == NULL)
      ccount = 0;
   else
      ccount = 7;


   header(fname);
   while ((c = fgetc(stdin)) != EOF && !ifbrk()) {
      if ((lines % lpp) == 0 && newpage) { /* Bottom of page: do a trailer */
         trailer();                    /* and a header (unless suppressed). */

         ++pageno;
         header(fname);
         newpage = 0;
         if (n)
            printf("%5d: ", lines++);
      }

      if (c == '\n' || ccount > columns)  {    /* EOL wrap-around and/or */
         ccount = 0;
         if (n) {                              /* newline. */
            ccount = 7;
            printf("\n%6d: ", lines);
         }
         else
            printf("\n ");
         ++lines;
         newpage = 1;
      }
      else  {                          /* Print the character. */
```

```
         ++ccount;
         fputc(c, stdout);
      }
   }

   while (lines++ % lpp != 0)       /* If not on a page boundary when */
      printf(" \n");                /* finished then print out as many */
   trailer();                       /* filler newlines as needed.     */
   pageno++;

}


header(fn)                 /* Print heading for lpr */
char *fn;
{

   int  i;

   if (options("H") == EOF)                 /* No heading requested - */
      return;

   printf(" \fOSRD Computing Facility");    /* Heading body */
   i = columns - (31 + strlen(datestr));
   while (i-- > 0)
      printf(" ");
   printf("Printed %s\n", datestr);
   printf(" %s\n \n ", fn);
}


trailer()                          /* Print trailer for lpr */
{
   int cpos;
   char pline[30];

   if (options("T") == EOF)               /* Trailer specified. */
      return;

   printf(" \n");                            /* Trailer body  */
   sprintf(pline, "Page %d", pageno);
   cpos = (columns - strlen(pline)) / 2;     /* Center 'Page n' */
   while (cpos--)
      printf(" ");
   printf("%s\n", pline);
}
```

APPENDIX C: Potential C Incompatibilities

The following list is complete to our knowledge. It includes all features which are defined in the C language, but as of this time are not supported by HP/C.

Structures may not be initialized.

Structure tags, members, and user identifiers are taken from the same name space.

All floating point arithmetic is done in the type specified by the referenced operands. Floating numbers remain floating unless combined with double in which case arithmetic is done in double.

Single characters (even in structures) are always allocated to the uppermost byte of a 16-bit word.

Floats are not converted to double when functions are called.

The upper character of an integer contains the integers most significant bits.

Reprinted with written permission from the HP/C Reference Manual, (c) Copyright Corporate Computer Systems, October, 1983.

APPENDIX D: PIPE.H Include File

This file contains the macro definitions and external references necessary for proper execution of PIPE'd programs. It should be included right after the <stdio.h> file.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                          *
 *    pipe.h -- include file for programs which use the     *
 *       PIPE system.                                       *
 *                                                          *
 *    Version 1.0  <841119.0907>                            *
 *                                                          *
 *    OSRD Computing Facility                               *
 *                                                          *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
extern int  datex(),
            dspoolof(),
            dspoolon(),
            options(),
            optval(),
            pnext(),
            popen(),
            spooloff(),
            spoolon();

extern pclose();

#define STANDARD  1
#define FILEIO    2
#define PIPE      3
#define MXLINL  150
#define MXPTRS  100
#define MXFDBS   64

/*    +---------------------------------------+
      |  The following objects are global     |
      |  and are used either to save a value  |
      |  so that multiple function calls are  |
      |  eliminated or to save space or       |
      |  both.                                |
      +---------------------------------------+     */

extern int  it,          /* Input type */
```

```
             ot,            /* Output type */
             ox,            /* Output file des. index in argv */
             no,            /* Number of options */
             ni;            /* Number of input file descriptors */

extern char *inlist[],      /* Input file list */
            *optlist[],     /* Option list */
            *pname;         /* PIPE file name */
```

☆U.S. GOVERNMENT PRINTING OFFICE: 1985-461-105:20038

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBS/TN-1208	2. Performing Organ. Report No.	3. Publication Date March 1985

4. TITLE AND SUBTITLE

PIPE/1000: An Implementation of Piping on an HP 1000 Minicomputer

5. AUTHOR(S)

N. L. Seidenman

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Same as in item 6 above.

10. SUPPLEMENTARY NOTES

□ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

Piping is a system by which programs can communicate so as to coordinate their respective functions in a synchronized effort aimed at the completion of a given task. Piping is one of the strong points of the increasingly popular operating system UNIX, developed at Bell Laboratories and licensed by AT&T. This paper describes an implementation of piping in a non-UNIX environment; in particular, on an HP-1000 minicomputer.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

C; computer languages; piping; programming; UNIX.

13. AVAILABILITY

☒ Unlimited

□ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

□ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

50

15. Price

NBS Technical Publications

Periodical

Journal of Research—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year.

Nonperiodicals

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396). NOTE: The Journal of Physical and Chemical Reference Data (JPCRD) is published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements are available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Service, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.

U.S. Department of Commerce
National Bureau of Standards
Gaithersburg, MD 20899