1. Introduction.

VALVE

Data Rate Throttling Filter

by John Walker

This program copies input to output, waiting as required to limit the transfer to a specified mean data transfer rate. This can be used to prevent bulk data transfers from monopolising disc or network bandwidth to the detriment of other tasks.

```
#define REVDATE "16th␣December␣2004"
#define TRUE 1
#define FALSE 0
```

2. Overall Program Structure.

⟨ Preprocessor definitions ⟩
⟨ Application include files 12 ⟩
⟨ System include files 13 ⟩
⟨ Windows-specific include files 15 ⟩
⟨ Global variables 16 ⟩
⟨ Utility functions 23 ⟩
⟨ Main program 3 ⟩

3. Main program.
The exit status returned by the main program is 0 for normal completion, 1 if an error occurred, and 2 for invalid options or file name arguments.

```
⟨ Main program 3 ⟩ ≡
  int main(int argc, char *argv[])
  {
    ⟨ Local variables 10 ⟩;
    ⟨ Process command-line options 20 ⟩;
    ⟨ Check options for consistency 21 ⟩;
    ⟨ Process command-line file name arguments 22 ⟩;
    ⟨ Force binary I/O where required 8 ⟩;
    ⟨ Initialise for unbuffered I/O and allocate I/O buffer 9 ⟩;
    ⟨ Compute initial estimate of sleep interval 4 ⟩;
    ⟨ Transcribe the file, enforcing the requested transfer rate 5 ⟩;
    ⟨ Report actual transfer rate 7 ⟩;
    return 0;
  }
```
This code is used in section 2.

4. Before we start copying the file, "*ve know noss-eenk*" about the actual data transfer rate we'll experience during the copy. So, in the interest of conservatism and doing the patient no harm (not to be Hippocratical!), we initially assume the physical I/O transfer rate is infinite—that it consumes zero time. Based on this assumption, we make an initial estimate of the inter-block wait time as the requested transfer rate divided by the block size. After the transfer is underway, the every second predictor/corrector (⟨ Update predictor/corrector for sleep interval 6 ⟩) will trim this to converge upon the optimal value.

```
⟨ Compute initial estimate of sleep interval 4 ⟩ ≡
  secblocks = transferRate / blocksize;
  estsleep = 1 · 10^9 / (transferRate / blocksize);
  nts.tv_sec = (time_t)(estsleep / 1 · 10^9);
  nts.tv_nsec = (long)(estsleep − nts.tv_sec);
  nblocks = 0;
  nnomsec = 0;
  if (verbose) {
    fprintf(stderr, "I/O block size: %s bytes.\n", commas((double) blocksize));
    if (¬measure) {
      fprintf(stderr, "Requested transfer rate: %s bytes/second.\n", commas(transferRate));
      fprintf(stderr, "Blocks per second: %.0f  Initial sleep time estimate: %.0f nsec\n",
          secblocks, estsleep);
    }
  }
```
This code is used in section 3.

5. This where the real work gets done. Having in hand file descriptors for the input and output files and the desired transfer rate, we loop reading a buffer-full of data and writing to the output file. After each buffer is transferred, we sleep the computed number of nanoseconds to result in the requested transfer rate. When we've copied a nominal second's worth of buffers, we run the predictor/corrector again and update the sleep interval to trim the transfer rate.

```
⟨ Transcribe the file, enforcing the requested transfer rate 5 ⟩ ≡
  gettimeofday(&tstart, &tz);
  while ((lr = read(fdi, iobuf, blocksize)) > 0) {
    lw = write(fdo, iobuf, lr);
    if (lw ≠ lr) {
      fprintf(stderr, "Error␣writing␣%u␣bytes␣to␣output␣file.␣␣%u␣bytes␣reported␣written.\n",
          lr, lw);
      return 1;
    }
    nbytes += lr;
    if (¬measure) {
      if (always_wait ∨ (nts.tv_sec ≠ 0) ∨ (nts.tv_nsec > min_sleep_nsec)) {
        NANOSLEEP(&nts, &ntsrem);
      }
    }
    nblocks++;
    ⟨ Update predictor/corrector for sleep interval 6 ⟩;
  }
```

This code is used in section 3.

6. Every time we complete copying a nominal second's worth of blocks, we calculate the actual transfer rate, compare it with the requested transfer rate, and adjust the per-block sleep interval to attain the desired transfer rate.

```
⟨ Update predictor/corrector for sleep interval 6 ⟩ ≡
  if (¬measure ∧ (nblocks ≥ secblocks)) {
    struct timeval tcblock;
    double actime;
    gettimeofday(&tcblock, &tz);
    nnomsec++;
    actime = (tcblock.tv_sec + (tcblock.tv_usec/1 · 10^6)) − (tstart.tv_sec + (tstart.tv_usec/1 · 10^6));
    estsleep = estsleep/(transferRate/(nbytes/actime));
    if (verbose) {
      fprintf(stderr,
          "Nominal␣time:␣%ld␣seconds.␣␣Actual␣time:␣%.2f␣seconds.""␣␣Adjusted␣sleep␣in\
          terval␣%.0f\n", nnomsec, actime, estsleep);
      fprintf(stderr, "%s␣bytes␣transferred␣at␣%s␣bytes␣per␣second.\n", commas((double)
          nbytes), commas(nbytes/actime));
    }
    nts.tv_sec = (time_t)(estsleep/1 · 10^9);
    nts.tv_nsec = (long)(estsleep − nts.tv_sec);
    nblocks = 0;
  }
```

This code is cited in section 4.

This code is used in section 5.

7. If requested, at the end of the transfer compute and display the number of bytes copied and the actual transfer rate. If the `--measure` option is specified, this serves as the report of transfer rate.

```
⟨ Report actual transfer rate 7 ⟩ ≡
  if (showtrans) {
    gettimeofday(&tend, &tz);
    duration = (tend.tv_sec + (tend.tv_usec/1 · 10^6)) − (tstart.tv_sec + (tstart.tv_usec/1 · 10^6));
    fprintf(stderr, "%s bytes transferred at %s bytes per second.\n", commas((double)
        nbytes), commas(nbytes/duration));
  }
```

This code is used in section 3.

8. On Win32, if a binary stream is the default of *stdin* or *stdout*, we must place this stream, opened in text mode (translation of CR to CR/LF) by default, into binary mode (no EOL translation). If you port this code to other platforms which distinguish between text and binary file I/O (for example, the Macintosh), you'll need to add equivalent code here.

The following code sets the already-open standard stream to binary mode on Microsoft Visual C 5.0 (Monkey C) if `_WIN32` is defined, and the Cygwin equivalent if `HAVE_CYGWIN` is defined. If you're using a different version or compiler, you may need some other incantation to cancel the text translation spell. Note that on Cygwin, the mode setting is required only if Cygwin has been configured to use DOS/Windows end of line convention, which is generally a terrible idea.

```
⟨ Force binary I/O where required 8 ⟩ ≡
#ifdef FORCE_BINARY_IO
  if (in_std) {
#ifdef _WIN32
    _setmode(_fileno(fi), O_BINARY);
#endif
#ifdef HAVE_CYGWIN
    setmode(_fileno(fi), O_BINARY);
#endif
  }
  if (out_std) {
#ifdef _WIN32
    _setmode(_fileno(fo), O_BINARY);
#endif
#ifdef HAVE_CYGWIN
    setmode(_fileno(fo), O_BINARY);
#endif
  }
#endif
```

This code is used in section 3.

9. Since our goal is to *restrict* transfer rate, we wish to work as closely as possible to low-level I/O, so we use the system *read*( ) and *write*( ) functions rather than buffered stream I/O. However, since we will often serve as a component in a pipeline, and such programs are handed already-opened buffered streams *stdin* and *stdout*, we obtain the file handles from these streams and to simplify logic, open command line file arguments as streams in ⟨ Process command-line file name arguments 22 ⟩.

```
⟨ Initialise for unbuffered I/O and allocate I/O buffer 9 ⟩ ≡
  fdi = fileno(fi);
  fdo = fileno(fo);
  iobuf = malloc(blocksize);
  if (iobuf ≡ Λ) {
    fprintf(stderr, "Unable to allocate %u byte I/O buffer.\n", blocksize);
    return 1;
  }
```

This code is used in section 3.

10. The following variables are local to the *main*( ) program.

```
⟨ Local variables 10 ⟩ ≡
  int f, opt;
#ifdef FORCE_BINARY_IO
  int in_std = TRUE, out_std = TRUE;
#endif
  double nbytes = 0;      /* Number of bytes transferred */
  unsigned int blocksize;      /* I/O block size */
  double transferRate = 1 · 10^6;      /* Desired transfer rate, bytes per second */
  double blockSize = −1;      /* Block size in bytes */
  struct timeval tstart, tend;      /* Start and end times of the transfer */
  struct timezone tz;      /* Throwaway argument for compatibility */
  byte *iobuf;      /* Dynamically allocated I/O buffer */
  unsigned int lr, lw;      /* Lengths (in bytes) read and written */
  double duration, secblocks, estsleep;
    /* double values of time in microseconds, seconds per block, and estimated sleep time. */
  struct TIMESPEC nts, ntsrem;      /* nanosleep (real or emulated) duration specifications */
  long nblocks, nnomsec;      /* Blocks transferred and nominal seconds */
```

This code is used in section 3.

11. Definitions and declarations.

12. The following application include files specify the host system configuration and define our private version of *getopt_long*.

⟨ Application include files 12 ⟩ ≡
```
#include "config.h"      /* System-dependent configuration */
#include "getopt.h"      /* Use our own getopt, which supports getopt_long */
```
This code is used in section 2.

13. We include the following POSIX-standard C library files. Conditionals based on a probe of the system by the `configure` program allow us to cope with the peculiarities of specific systems.

⟨ System include files 13 ⟩ ≡
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
#include <math.h>
#include <sys/time.h>
#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#ifdef HAVE_STRING_H
#include <string.h>
#else
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif
#endif
```
See also section 14.

This code is used in section 2.

14. The following baroque construction is what it takes to handle systems which may or may not define the various flavours of precision sleep functions. We handle absence of *nanosleep* and/or *usleep* by emulating them with the first available lower-level function, with *setitimer* at the bottom of the totem pole. Since absence of *nanosleep* probably also means absence of the *timespec* structure it uses, we define our own *em_timespec* used when we're emulating *nanosleep*.

```
⟨ System include files 13 ⟩ +≡
#ifdef HAVE_NANOSLEEP
#define NANOSLEEP nanosleep
#define TIMESPEC timespec
#define TIMER "nanosleep"
#else
#define NANOSLEEP em_nanosleep
#define TIMESPEC em_timespec
  struct em_timespec {
    time_t tv_sec;
    long tv_nsec;
  };
#define EMULATE_NANOSLEEP
#ifdef HAVE_USLEEP
#define USLEEP usleep
#define TIMER "usleep"
#else
#define EMULATE_USLEEP
#define TIMER "setitimer"
#include <sys/signal.h>
#define USLEEP em_usleep
#endif
#endif
```

15. The following include files are needed in Win32 and Cygwin builds to permit setting already-open I/O streams to binary mode. Yes, the specifications for these systems duplicate one another, but doubtless they will eventually diverge.

```
⟨ Windows-specific include files 15 ⟩ ≡
#ifdef _WIN32
#define FORCE_BINARY_IO
#include <io.h>
#include <fcntl.h>
#endif
#ifdef HAVE_CYGWIN
#define FORCE_BINARY_IO
#include <io.h>      /* For setmode() */
#include <fcntl.h>
#endif
```

This code is used in section 2.

16. These variables are global to all procedures; many are used as "hidden arguments" to functions in order to simplify calling sequences. We'll declare additional global variables as we need them in successive sections.

```
⟨ Global variables 16 ⟩ ≡
  typedef unsigned char byte;      /* Byte type */
  static FILE *fi;      /* Input file */
  static FILE *fo;      /* Output file */
  static int fdi, fdo;      /* Input and output file descriptors */
  static int verbose = FALSE;      /* Verbose output */
  static int measure = FALSE;      /* Measure transfer rate only */
  static int showtrans = FALSE;      /* Print actual transfer rate at end */
#ifndef MIN_SLEEP_NSEC
#define MIN_SLEEP_NSEC  10000000      /* Default minimum sleep is 10 msec */
#endif
  static unsigned long min_sleep_nsec = MIN_SLEEP_NSEC;
    /* Minimum sleep time in nanoseconds */
  static int always_wait = FALSE;      /* Always wait, even if interval shorter than min_sleep_nsec ? */
```

See also section 19.

This code is used in section 2.

17. Command line parsing.

18. Procedure *usage* prints how-to-call information.

```
⟨ Print how-to-call information 18 ⟩ ≡
  static void usage(void)
  {
    printf("%s␣␣--␣␣Copy␣data,␣enforcing␣bandwidth␣limit.␣␣Call:\n", PRODUCT);
    printf("␣␣␣␣␣␣␣␣␣␣␣␣%s␣[options]␣[infile]␣[outfile]\n", PRODUCT);
    printf("\n");
    printf("Options:\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣-a,␣--alwayswait␣␣␣␣Always␣wait␣between␣blocks,␣\
        regardless""␣of␣transfer␣rate\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣-b,␣--blocksize␣n␣␣␣Copy␣blocks␣of␣n␣bytes\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣--copyright␣␣␣␣␣␣␣␣␣Print␣copyright␣information\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣-m,␣--measure␣␣␣␣␣␣␣Measure␣transfer␣rate␣only␣(no␣limitation)\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣-r,␣--rate␣n␣␣␣␣␣␣␣␣Limit␣transfer␣rate␣to␣n␣bytes/second\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣-s,␣--summary␣␣␣␣␣␣␣Print␣actual␣transfer␣rate\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣--timingtest␣n␣␣␣␣␣␣Run␣minimum␣wait␣timing␣test␣for␣n␣seconds\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣-u,␣--help␣␣␣␣␣␣␣␣␣Print␣this␣message\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣-v,␣--verbose␣␣␣␣␣␣␣Verbose:␣print␣transfer␣rate␣statistics\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣--version␣␣␣␣␣␣␣␣␣␣Print␣version␣number\n");
    printf("␣␣␣␣␣␣␣␣␣␣␣␣--waitmin␣n␣␣␣␣␣␣␣␣Set␣minimum␣wait␣time␣to␣n␣nanoseconds\n");
    printf("\n");
    printf("by␣John␣Walker\n");
    printf("http://www.fourmilab.ch/\n");
  }
```

This code is used in section 23.

19. Here we define the command-line options. Note that any long option which has a single letter counterpart must also be added to the *single_letter_options* string.

```
⟨ Global variables 16 ⟩ +≡
  static const struct option long_options[] = {
  {"alwayswait", 0, Λ, 'a'},
  {"blocksize", 1, Λ, 'b'},
  {"copyright", 0, Λ, 'c'},
  {"help", 0, Λ, 'u'},
  {"measure", 0, Λ, 'm'},
  {"rate", 1, Λ, 'r'},
  {"summary", 0, Λ, 's'},
  {"timingtest", 1, Λ, 202},
  {"verbose", 0, Λ, 'v'},
  {"version", 0, Λ, 201},
  {"waitmin", 1, Λ, 'w'},
  {0, 0, 0, 0}
  };
  static const char single_letter_options[] = "ab:chr:suvw:";
  static int option_index = 0;
```

20. We use *getopt_long* to process command line options. This permits aggregation of single letter options without arguments and both `-d`*arg* and `-d` *arg* syntax. Long options, preceded by `--`, are provided as alternatives for all single letter options and are used exclusively for less frequently used facilities.

```
⟨ Process command-line options 20 ⟩ ≡
  while ((opt = my_getopt_long(argc, argv, single_letter_options, long_options, &option_index)) ≠ −1) {
    switch (opt) {
    case 'a':    /* -a --alwayswait Always wait after each block */
      always_wait = TRUE;
      break;
    case 'b':    /* -b --blocksize n Block size in bytes */
      blockSize = sizearg(my_optarg);
      break;
    case 'c':    /* --copyright */
      printf("This␣program␣is␣in␣the␣public␣domain.\n");
      return 0;
    case 'm':    /* -m --measure Measure transfer rate only */
      showtrans = measure = TRUE;
      break;
    case 'r':    /* -r --rate size Transfer rate in bytes / second */
      transferRate = sizearg(my_optarg);
      break;
    case 's':    /* -s --summary Show actual transfer rate */
      showtrans = TRUE;
      break;
    case 202:    /* --timingtest n Run timing test for n seconds */
      timingtest(atoi(my_optarg));
      return 0;
    case 'u':    /* -u --help Print how-to-call information */
      case '?':
      usage( );
      return 0;
    case 'v':    /* --verbose */
      showtrans = verbose = TRUE;
      break;
    case 201:    /* --version */
      ⟨ Show program version information 26 ⟩;
      return 0;
    case 'w':    /* -w --waitmin nsec Minimum wait in nanoseconds */
      {
        double d = sizearg(my_optarg);
        if ((d < 0) ∨ (d ≥ pow(2, 32))) {
          fprintf(stderr, "Invalid␣--waitmin␣value␣\"%s\".␣␣Must␣be␣>=␣0␣and␣<␣2^32.\n",
              my_optarg);
          return 2;
        }
        min_sleep_nsec = (unsigned long) d;
      }
      break;
    default:
      fprintf(stderr, "***Internal␣error:␣unhandled␣case␣%d␣in␣option␣processing.\n", opt);
      return 1;
    }
```

```
}
```

This code is used in section 3.

21. After processing the command-line options, we need to check them for consistency.

```
⟨ Check options for consistency 21 ⟩ ≡
  if (transferRate ≤ 0) {
    fprintf(stderr, "Transfer␣rate␣specification␣(%.0f␣bytes/sec)␣invalid.␣␣Must␣be␣>=␣1.\n",
        transferRate);
    return 2;
  }
  if (blockSize ≡ −1) {
    blockSize = transferRate/4;
    if (blockSize > (1 ≪ 20)) {
      blockSize = 1 ≪ 20;
    }
  }
  if (blockSize < 1) {
    fprintf(stderr, "Block␣size␣specification␣(%.0f␣bytes)␣invalid.␣␣Must␣be␣>=␣1.\n",
        blockSize);
    return 2;
  }
  blocksize = (unsigned int)(blockSize + 0.5);
```

This code is used in section 3.

22. This code is executed after *getopt* has completed parsing command line options. At this point the external variable *my_optind* in *getopt* contains the index of the first argument in the *argv*[ ] array. The first two arguments specify the input and output file. If either argument is omitted or "-", standard input or output is used.

On systems which distinguish text and binary I/O (for end of line translation), we always open the input and output files in binary mode.

⟨ Process command-line file name arguments 22 ⟩ ≡

```
  /* Some C compilers don't allow initialisation of static variables such as fi and fo with their library's
     definitions of stdin and stdout, so we initialise them at runtime. */
  fi = stdin;
  fo = stdout;
  f = 0;
  for ( ; my_optind < argc; my_optind++) {
    char *cp = argv[my_optind];
    switch (f) {
    case 0:
      if (strcmp(cp, "-") ≠ 0) {
        if ((fi = fopen(cp,
#ifdef FORCE_BINARY_IO
        "rb"
#else
        "r"
#endif
        )) ≡ Λ) {
          fprintf(stderr, "Cannot␣open␣input␣file␣%s\n", cp);
          return 2;
        }
#ifdef FORCE_BINARY_IO
        in_std = FALSE;
#endif
      }
      f++;
      break;
    case 1:
      if (strcmp(cp, "-") ≠ 0) {
        if ((fo = fopen(cp,
#ifdef FORCE_BINARY_IO
        "wb"
#else
        "w"
#endif
        )) ≡ Λ) {
          fprintf(stderr, "Cannot␣open␣output␣file␣%s\n", cp);
          return 2;
        }
#ifdef FORCE_BINARY_IO
        out_std = FALSE;
#endif
      }
      f++;
      break;
    default: fprintf(stderr, "Too␣many␣file␣names␣specified.\n");
```

```
        usage( );
        return 2;
      }
    }
```

This code is cited in section 9.

This code is used in section 3.

23. Utility functions.
These functions are local to the program, but defined outside and before the *main* function.

```
⟨ Utility functions 23 ⟩ ≡
  ⟨ Print how-to-call information 18 ⟩;
  ⟨ Parse byte length argument 27 ⟩;
  ⟨ Edit double value with commas between thousands 24 ⟩;
  ⟨ Emulate usleep on systems which don't support it 30 ⟩;
  ⟨ Emulate nanosleep on systems which don't support it 29 ⟩;
  ⟨ Run timing test 25 ⟩;
```

This code is used in section 2.

24. Procedure *commas* edits a double precision floating point number with no decimal places and groups of three digits separated by commas. To permit this function to be called multiple times to edit arguments in *printf*, etc. function calls, values are edited into a rotating set of eight string buffers. If you need more than eight values edited into a single message, you're probably doing something wrong.

```
⟨ Edit double value with commas between thousands 24 ⟩ ≡
  static char *commas(const double f)
  {
#define StringBufLength 96
    static char res[8][StringBufLength];
    static int nr = 0;
    static char s[StringBufLength];
    int i;
    char *p = res[nr];
    char *e = p + (StringBufLength - 1);
    int l;
    nr = (nr + 1) % 8;
    *e = 0;
    i = snprintf(s, StringBufLength, "%.0f", f);
    if ((i < 0) ∨ (i ≥ StringBufLength)) {
      return "*Truncated*";
    }
    l = strlen(s) - 1;
    for (i = l; i ≥ 0; i--) {
      if ((i < l) ∧ (((l - i) % 3) ≡ 0)) {
        *--e = ',';
      }
      *--e = s[i];
    }
    return e;
  }
```

This code is used in section 23.

25. This function is invoked when the `--timingtest` option is specified. It runs a timing test for the specified number of seconds and reports the measured minimum delay time of the configured sleep function, which it identifies.

```
⟨ Run timing test 25 ⟩ ≡
  static void timingtest(const int nsec)
  {
    int s = nsec;
    time_t t = time(Λ);
    int i;
    struct TIMESPEC nsi, nso;
    unsigned long n = 0;
    printf("Using␣minimum␣wait␣time␣(--waitmin)␣=␣%s␣nanoseconds\n", commas((double)
        min_sleep_nsec));
    if (s ≤ 0) {
      printf("Invalid␣timing␣test␣duration.␣␣Using␣60␣seconds.\n");
      s = 60;
    }
    printf("Running␣"TIMER"␣timing␣test␣for␣%d␣seconds.\n", nsec);
    nsi.tv_sec = 0;
    nsi.tv_nsec = 1;
    while ((time(Λ) − t) < s) {
      for (i = 0; i < 500; i++) {
        NANOSLEEP(&nsi, &nso);
        n++;
      }
    }
    printf("Minimum␣nanosleep()␣time:␣%s␣nsec.\n", commas(((time(Λ) − t) ∗ 1 · 10^9)/n));
  }
```

This code is used in section 23.

26. Show program version information in response to the `--version` option.

```
⟨ Show program version information 26 ⟩ ≡
  printf("%s␣%s\n", PRODUCT, VERSION);
  printf("Last␣revised:␣%s\n", REVDATE);
  printf("The␣latest␣version␣is␣always␣available\n");
  printf("at␣http://www.fourmilab.ch/webtools/valve/\n");
```

This code is used in section 20.

27. Procedure *sizearg* parses a byte length argument. These arguments are decimal numbers, which may be written with commas (which are ignored) and optionally followed by a suffix specifying a multiplicative factor as follows, where a trailing "`B`" after an ISO prefix denotes a power of 10, as opposed to the closest power of two.

`c`	character	2^0	1
`w`	word	2^1	2
`b`	block	2^9	512
`KB, kB`	kilo	10^3	1,000
`K, k`	kibi	2^{10}	1,024
`MB`	mega	10^6	1,000,000
`M`	mebi	2^{20}	1,048,576
`GB`	giga	10^9	1,000,000,000
`G`	gibi	2^{30}	1,073,741,824
`TB`	tera	10^{12}	1,000,000,000,000
`T`	tebi	2^{40}	1,099,511,627,776
`PB`	peta	10^{15}	1,000,000,000,000,000
`P`	pebi	2^{50}	1,125,899,906,842,624
`EB`	exa	10^{18}	1,000,000,000,000,000,000
`E`	exbi	2^{60}	1,152,921,504,606,846,976
`ZB`	zetta	10^{21}	1,000,000,000,000,000,000,000
`Z`	zebi	2^{70}	1,180,591,620,717,411,303,424
`YB`	yotta	10^{24}	1,000,000,000,000,000,000,000,000
`Y`	yobi	2^{80}	1,208,925,819,614,629,174,706,176

```
⟨ Parse byte length argument 27 ⟩ ≡
  static double sizearg(const char *arghhh)
  {
    char *arg = malloc(strlen(arghhh) + 1);
    int decimalPower = FALSE;
    static char SIpowers[] = "KMGTPEZY";
    static char legacyPowers[] = "cwb";
    static int legacyFactors[] = {1, 2, 512};
    char *a;
    const char *s;
    double count, factor = 1.0;
    if (arg ≡ Λ) {
      fprintf(stderr, "Unable␣to␣allocate␣argument␣parsing␣buffer.\n");
      exit(1);
    }
    a = arg;
    s = arghhh;      /* Elide commas from numeric specification. */
    while (1) {
      char c = *s++;
      if (c ≠ ',') {
        *a++ = c;
      }
      if (c ≡ 0) {
        break;
      }
    }
    ;      /* Parse size suffix, if any. */
    a -= 2;
```

```
  if ((a > arg) ∧ ((*a ≡ 'B') ∨ (*a ≡ 'D'))) {
    decimalPower = TRUE;
    a−−;
  }
  if ((a > arg) ∧ (¬isdigit(*a))) {
    if (*a ≡ 'k') {
      *a = 'K';      /* Kludge for "k" */
    }
    if ((s = strchr(SIpowers, *a)) ≠ Λ) {
      int p = (s − SIpowers) + 1;
      *a = 0;
      factor = decimalPower ? pow(10.0, 3.0 * p) : pow(2.0, 10.0 * p);
    }
    else if ((s = strchr(legacyPowers, *a)) ≠ Λ) {
      *a = 0;
      factor = legacyFactors[s − legacyPowers];
    }
    else {
      fprintf(stderr, "Unknown␣factor␣suffix␣'%c'␣in␣size␣argument␣\"%s\".\n", *a, arghhh);
      exit(2);
    }
  }     /* Parse number and scale by factor. */
  count = strtod(arg, &a);
  if (*a ≠ 0) {      /* Make sure we ate the whole thing */
    fprintf(stderr, "Numeric␣syntax␣error␣in␣size␣argument␣\"%s\".\n", arghhh);
    exit(2);
  }
  return count * factor;
}
```

This code is used in section 23.

28. Emulation for legacy systems.

In order to effectively adjust the transfer rate, we need a precision wait function, *sleep*( ) is hopelessly too coarse-grained for our purposes. Our first choice is the POSIX *nanosleep*( ) function, which potentially has the best resolution and is better behaved vis-à-vis signals. Failing that, we use *usleep*( ) if it's available, and if it's not there either, we fall back to *setitimer*( ). That's the last resort; if none of these functions are present, the job just can't be done.

Note that all of these timing functions require various kinds of fancy footwork if they're interrupted by signals while execution is suspended. We don't include any handling of that here for the very simple reason that we don't use signals for anything but timing—all other signals are either ignore or fatal. If you're planning to use this code in another program which *does* use signals, you're going to have to learn all about `EINTR` and restarting sleep calls.

29. If the host system does not implement the *nanosleep* function, we emulate it using the *usleep* function which, in turn, may be emulated with *setitimer* if necessary.

```
⟨ Emulate nanosleep on systems which don't support it 29 ⟩ ≡
#ifdef EMULATE_NANOSLEEP
  void em_nanosleep(const struct em_timespec *req, struct em_timespec *rem)
  {
    double usecs = (req→tv_sec * 1 · 10^6) + (req→tv_nsec/1000);
    time_t slsec;
    unsigned long slusec;
    if (usecs < 1) {
      usecs = 1;
    }
    slsec = (time_t)(usecs/1 · 10^6);
    slusec = (unsigned long)(usecs − (slsec * 1000000));
    if (slsec > 0) {
      sleep(slsec);
    }
    USLEEP(slusec);
  }
#endif
```

This code is used in section 23.

30. If the host system does not implement the *usleep* function, we emulate it using the *setitimer* function. If *that's* not supported, we're at the end of our string.

```
⟨ Emulate usleep on systems which don't support it 30 ⟩ ≡
#ifdef EMULATE_USLEEP
#ifndef HAVE_SETITIMER
  error error error System has neither nanosleep, usleep, nor setitimer
#endif
      static int volatile waiting;
      static RETSIGTYPE(*oldsig)( ); static RETSIGTYPE getalrm(int i)
      {
        waiting = 0;
        signal(SIGALRM, getalrm);
      }
      void em_usleep(unsigned long t)
      {
        static struct itimerval it, ot;
        it.it_value.tv_sec = t/1000000;
        it.it_value.tv_usec = t % 1000000;
        oldsig = signal(SIGALRM, getalrm);
        waiting = 1;
        if (setitimer(ITIMER_REAL, &it, &ot)) {
          return;      /* Error */
        }
        while (waiting) {
          pause( );
        }
        signal(SIGALRM, oldsig);
      }
#endif
```

This code is used in section 23.

31. Development Log.

2004 December 16

Initial release, version 1.0.

32. Index. The following is a cross-reference table for `valve`. Single-character identifiers are not indexed, nor are reserved words. Underlined entries indicate where an identifier was declared.

VALVE