```
;/* CompareIO.c - Execute me to compile me with Lattice 5.10b
LC -b0 -cfistq -v -y -j73 CompareIO.c
Blink FROM LIB:c.o,CompareIO.o TO CompareIO LIBRARY
LIB:LC.lib,LIB:Amiga.lib,lib:debug.lib
quit ;*/

/* CompareIO.c uses packet level I/O to copy the standard input channel to the */
/* standard output channel (as set up by the standard startup code, c.o).      */
/* CompareIO uses both synchronous and asynchronous I/O to perform the copy    */
/* and reports the time it takes to do each.                                   */


#include <exec/types.h>
#include <dos/dosextens.h>
#include <devices/timer.h>

#include <clib/dos_protos.h>
#include <clib/timer_protos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }                    /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }
#endif


#define BUFSIZE 8192

UBYTE *vers = "\0$VER: CompareIO 37.14 Nov-12-92";

ULONG AsyncLoop(void);
ULONG SyncLoop(void);

extern struct Library *DOSBase;
struct Library *TimerBase;

struct MsgPort *myport;

struct FileHandle *in, *out;
BPTR results, in_start, out_start;

struct DosPacket *sp_read, *sp_write;

UBYTE buffer[BUFSIZE*2];

struct timeval time_start, time_finish;
struct timerequest timer_io;

ULONG vfprintfargs[2]; /* An array of pointers */

void main(void)
{
  if (DOSBase->lib_Version >= 37)
  {
    if (results = Open("*", MODE_NEWFILE)) /* This is for printing the results.      */
    {                                      /* Since the example is already using the */
                                           /* standard I/O channels for its own      */
                                           /* purposes, there needs to be a separate */
                                           /* channel to output the results.         */
      if (!OpenDevice(TIMERNAME, UNIT_MICROHZ, &timer_io, 0L))
      {
        TimerBase = (struct Library *)timer_io.tr_node.io_Device;

        if (myport = CreateMsgPort())
        {
          in_start  = Input();  /* Need to hold on to input and output so no one can */
          out_start = Output(); /* change them while this example is using them.     */
          if (in = (struct FileHandle *)BADDR(in_start))
          {
            if (out = (struct FileHandle *)BADDR(out_start))
            {
              if (sp_read = AllocDosObject(DOS_STDPKT, NULL))
              {
                if (sp_write = AllocDosObject(DOS_STDPKT, NULL))
                {
                      /* When AllocDosObject() allocates a StandardPacket, it takes */
                      /* care of linking together the Message and DosPacket.        */
                      /* AllocDosObject() points the DosPacket's dp_Link field at   */
                      /* the StandardPacket's Message structure.  It also points    */
                      /* the Message's mn_Node.ln_Name field at the DosPacket:      */
                      /*                  sp_read->dp_Link = sp_Msg;                */
                      /*                  sp_Msg->mn_Node.ln_Name = (STRPTR)sp_read; */

                  sp_read->dp_Type = ACTION_READ;   /* Fill out ACTION_READ packet.  */
                  sp_read->dp_Arg1 = in->fh_Arg1;

                  sp_write->dp_Type = ACTION_WRITE; /* Fill out ACTION_WRITE packet. */
                  sp_write->dp_Arg1 = out->fh_Arg1;

                  VFPrintf(results, "\n      Method     Seconds   Micros\n", NULL);
                  VFPrintf(results,   "   ------------  -------   ------\n", NULL);

                  GetSysTime(&time_start);
                  if (AsyncLoop())
                  {
                    GetSysTime(&time_finish);
                    SubTime(&time_finish, &time_start);
                    vfprintfargs[0] = time_finish.tv_secs;
                    vfprintfargs[1] = time_finish.tv_micro;
                    VFPrintf(results,
                           "   Asynchronous:  %3ld     %7ld\n", &vfprintfargs[0]);

                    GetSysTime(&time_start);
                    if (SyncLoop())
                    {
                      GetSysTime(&time_finish);
                      SubTime(&time_finish, &time_start);
                      vfprintfargs[0] = time_finish.tv_secs;
                      vfprintfargs[1] = time_finish.tv_micro;
                      VFPrintf(results,
                             "    Synchronous:  %3ld     %7ld\n", &vfprintfargs[0]);
                    }
                    else
                      VFPrintf(results, "         *******  Stop  ******\n", NULL);
                  }
                  else
                    VFPrintf(results, "         *******  Stop  ******\n", NULL);


                  FreeDosObject(DOS_STDPKT, sp_write);
                }
                FreeDosObject(DOS_STDPKT, sp_read);
              }
            }
          }
          DeleteMsgPort(myport);
        }
        CloseLibrary(TimerBase);
      }
      Close(results);
    }
  }
}

ULONG AsyncLoop()
{
struct StandardPacket *mysp;
UBYTE *buf;

LONG amount_read;

BOOL sp_read_busy  = TRUE,                    /*    Is the ACTION_READ packet busy?  */
     sp_write_busy = FALSE,                   /*    Is the ACTION_WRITE packet busy? */
     done          = FALSE;                   /*    Is the program finished?         */
ULONG ok           = TRUE;


  if (!((out->fh_Arg1) && (in->fh_Arg1)))    /* Don't bother if in or out uses NIL:  */
    return(FALSE);
  sp_read->dp_Arg2 = (LONG)buffer;   /* The buffer to fill in.           */
  sp_read->dp_Arg3 = BUFSIZE;        /* The size of the Arg2 buffer.     */
```

```
SendPkt(sp_read, in->fh_Type, myport);   /* Send initial read request. */

sp_write->dp_Type = ACTION_WRITE; /* Fill out the ACTION_WRITE packet. */
sp_write->dp_Arg1 = out->fh_Arg1;
sp_write->dp_Arg2 = (LONG)&buffer[BUFSIZE];  /* Arg2 points to the buffer to write */
sp_write->dp_Arg3 = 0L;                      /* out.  At first glance, it might    */
sp_write->dp_Res1 = 0L;                      /* seem odd to bother setting Arg2    */
                                 /* when the program hasn't read anything yet.  */
                                 /* This is to set up for the main loop.  The   */
                                 /* main loop swaps the ACTION_READ buffer with */
                                 /* the ACTION_WRITE buffer when it receives    */
                                 /* a completed read.  Likewise, dp_Arg3 and    */
                                 /* dp_Res1 are set to make the ACTION_READ     */
                                 /* look like it has a valid return value so    */
                                 /* main loop won't fail the first time through */
                                 /* the loop.                                   */

              /* main() has already taken care of sending the initial read to the  */
              /* handler.  Because we need the data from that read before we can   */
while (!done) /* do anything, the first thing to do is wait for its return.        */
{
  do                                    /*      Wait for the ACTION_READ to return. */
  {
    WaitPort(myport);
    while (mysp = (struct StandardPacket *)GetMsg(myport))   /* ...empty the port. */
    {
                        /* If this message is the ACTION_READ packet, mark it as   */
                        /* no longer busy so we can use it to start another read.  */
      if (mysp->sp_Pkt.dp_Type == ACTION_READ)    sp_read_busy  = FALSE;

                        /* If this message is instead the ACTION_WRITE packet,     */
                        /* mark it as not busy.  We need to check for this because */
                        /* the WRITE_PACKET from the previous interation through   */
                        /* the loop might have come back before the ACTION_WRITE   */
                        /* from the previous interation.                           */
      else
        if (mysp->sp_Pkt.dp_Type == ACTION_WRITE) sp_write_busy = FALSE;
    }
  } while (sp_read_busy);                    /* End of "wait for ACTION_READ" loop. */

                                        /* Get ready to send the next ACTION_READ. */
  buf = (UBYTE *)(sp_read->dp_Arg2);    /* Hold on to the important stuff from the */
  amount_read = sp_read->dp_Res1;       /* ACTION_READ we just got back so we can  */
                                        /* reuse the packet to start a new read    */
                                        /* while processing the last read's data.  */

  while (sp_write_busy)           /* Because this example only uses two buffers and */
  {                               /* the ACTION_WRITE might be using one of them,   */
                                  /* this example has to wait for an outstanding    */
                                  /* ACTION_WRITE to return before reusing the      */
                                  /* ACTION_WRITE packet's buffer.                  */
    WaitPort(myport);
    while (mysp = (struct StandardPacket *)GetMsg(myport))
      if (mysp->sp_Pkt.dp_Type == ACTION_WRITE) sp_write_busy = FALSE;
  }

  if (SetSignal(0L, SIGBREAKF_CTRL_C) & SIGBREAKF_CTRL_C)
  {
    done = TRUE;
    ok   = FALSE;
  }
  else
  {
             /* This tests the return values from the ACTION_READ and ACTION_WRITE */
             /* packets.  The ACTION_READ packet returns the number of bytes it    */
             /* read in dp_Res1, which was copied earlier into amount_read. If it  */
             /* is 0, the read packet found the EOF.  If it is negative, there was */
             /* an error.  In the case of ACTION_WRITE, an error occurs if the     */
             /* number of bytes that ACTION_WRITE was supposed to write (Arg3)     */
             /* does not match the actual number it wrote, which ACTION_WRITE re-  */
             /* turns in Res1.  This test is the reason dp_Res1 and dp_Arg3 were   */
             /* set to zero when the ACTION_WRITE packet was set up in main().     */
    if ((amount_read > 0) && (sp_write->dp_Res1 == sp_write->dp_Arg3))
    {
      sp_read->dp_Arg2 = sp_write->dp_Arg2;  /* ACTION_WRITE is finished with its  */
                                             /* buffer, use it in the next read.   */
```

```
      SendPkt(sp_read, in->fh_Type, myport); /* Send the next ACTION_READ and mark */
      sp_read_busy = TRUE;                   /* the ACTION_READ as busy.           */

      /* Process Buffer.  This example doesn't do anything with the data from the  */
      /* last ACTION_READ, it just passes it on to the STDOUT handler.             */


      sp_write->dp_Arg2 = (LONG)buf;         /*    Set up the ACTION_WRITE packet. */
      sp_write->dp_Arg3 = amount_read;
      SendPkt(sp_write, out->fh_Type, myport);   /* Send the next ACTION_WRITE and */
      sp_write_busy = TRUE;                      /* mark the ACTION_WRITE as busy. */
    }
    else                             /* A packet returned with a failure, so quit. */
    {
      done = TRUE;
      if ((amount_read < 0) || (sp_write->dp_Res1 != sp_write->dp_Arg3)) ok = FALSE;
    }
  }
}
return(ok);
}

ULONG SyncLoop()
{
BOOL  done = FALSE;
ULONG ok   = TRUE;
BPTR lock;

  if (!((out->fh_Arg1) && (in->fh_Arg1)))    /* Don't bother if in or out uses NIL:  */
    return(FALSE);

  sp_read->dp_Arg2 = (LONG)buffer;
  sp_read->dp_Arg3 = BUFSIZE*2;
  sp_write->dp_Arg2 = (LONG)buffer;

  if (lock = DupLockFromFH(in_start))
  {
    UnLock(lock);                             /* Make sure this is a filesystem and not */
    Seek(in_start, 0, OFFSET_BEGINNING);      /* a console.  If this is a filesystem,   */
  }                                           /* go to the beginning of the file.       */

  while (!done)
  {
    if (SetSignal(0L, SIGBREAKF_CTRL_C) & SIGBREAKF_CTRL_C)
    {
      done = TRUE;
      ok = FALSE;
    }
    else
    {
      SendPkt(sp_read, in->fh_Type, myport);
      WaitPort(myport);
      while (GetMsg(myport));

      if (sp_read->dp_Res1 > 0)
      {
        sp_write->dp_Arg3 = sp_read->dp_Res1;
        SendPkt(sp_write, out->fh_Type, myport);
        WaitPort(myport);
        while (GetMsg(myport));
        if (sp_write->dp_Res1 != sp_write->dp_Arg3)
        {
          done = TRUE;
          ok   = FALSE;
        }
      }
      else
      {
        done = TRUE;
        if (sp_read->dp_Res1 < 0) ok = FALSE;
      }
    }
  }
  return(ok);
}
```

```
;/* InOutCTRL-C.c - Execute me to compile me with Lattice 5.10b
LC -b0 -cfistq -v -y -j73 InOutCTRL-C.c
Blink FROM LIB:c.o,InOutCTRL-C.o TO InOutCTRL-C LIBRARY
LIB:LC.lib,LIB:Amiga.lib,lib:debug.lib
quit ;*/

/* InOutCTRL-C.c uses packets to copy the standard input channel to the        */
/* standard output channel using asynchronous I/O.  This example does a better */
/* job checking for a user break than the accompanying example, CompareIO.c.   */

#include <exec/types.h>
#include <dos/dosextens.h>

#include <clib/dos_protos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }                    /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }
#endif


#define BUFSIZE 8192

UBYTE *vers = "\0$VER: InOutCTRL-C 37.9 Nov-12-92";

void MainLoop(void);

extern struct Library *DOSBase;

struct MsgPort  *myport;
ULONG portsignal, signals, sigmask;

struct FileHandle *in, *out;
struct DosPacket *sp_read, *sp_write;

UBYTE buf1[BUFSIZE], buf2[BUFSIZE];

void main(void)
{
  if (DOSBase->lib_Version >= 37)                                      /* 2.0 only */
  {
    if (myport = CreateMsgPort())
    {
      if (in = (struct FileHandle *)BADDR(Input()))        /* Need file handle to    */
      {                                                    /* get to Handler process */
        if (out = (struct FileHandle *)BADDR(Output()))
        {
          if (sp_read = AllocDosObject(DOS_STDPKT, NULL))    /* Allocate two         */
          {                                                  /* StandardPackets: one */
            if (sp_write = AllocDosObject(DOS_STDPKT, NULL)) /* for reading, and one */
            {                                                /* for writing. */

              sp_read->dp_Type = ACTION_READ;    /* Fill out the ACTION_READ packet. */
              sp_read->dp_Arg1 = in->fh_Arg1;
              sp_read->dp_Arg2 = (LONG)buf1;     /* The buffer to fill in.           */
              sp_read->dp_Arg3 = BUFSIZE;        /* The size of the Arg2 buffer.     */

                       /* When AllocDosObject() allocates a StandardPacket, it takes */
                       /* care of linking together the Message and DosPacket.        */
                       /* AllocDosObject() points the DosPacket's dp_Link field at   */
                       /* the StandardPacket's Message structure.  It also points    */
                       /* the Message's mn_Node.ln_Name field at the DosPacket:      */
                       /*                  sp_read->dp_Link = sp_Msg;                */
                       /*                  sp_Msg->mn_Node.ln_Name = (STRPTR)sp_read; */


              if (!((out->fh_Arg1) && (in->fh_Arg1)))     /* Don't bother if in or   */
                return;                                   /*          out uses NIL:  */

              SendPkt(sp_read, in->fh_Type, myport);   /* Send initial read request. */

              portsignal = 1L<<myport->mp_SigBit;          /* Record the signal bits */
              sigmask    = SIGBREAKF_CTRL_C | portsignal;  /*          for later use. */

              sp_write->dp_Type = ACTION_WRITE; /* Fill out the ACTION_WRITE packet. */
              sp_write->dp_Arg1 = out->fh_Arg1;
              sp_write->dp_Arg2 = (LONG)buf2;  /* Arg2 points to the buffer to write */
              sp_write->dp_Arg3 = 0L;          /* out.  At first glance, it might    */
              sp_write->dp_Res1 = 0L;          /* seem odd to bother setting Arg2    */
                                        /* when the program hasn't read anything yet.  */
                                        /* This is to set up for the main loop.  The   */
                                        /* main loop swaps the ACTION_READ buffer with */
                                        /* the ACTION_WRITE buffer when it receives    */
                                        /* a completed read.  Likewise, dp_Arg3 and    */
                                        /* dp_Res1 are set to make the ACTION_READ     */
                                        /* look like it has a valid return value so    */
                                        /* main loop won't fail the first time through */
                                        /* the loop.                                   */
              MainLoop();
              FreeDosObject(DOS_STDPKT, sp_write);
            }
            FreeDosObject(DOS_STDPKT, sp_read);
          }
        }
      }
      DeleteMsgPort(myport);
    }
  }
}


void MainLoop()
{
struct StandardPacket *mysp;
UBYTE *buf;

LONG amount_read;

BOOL sp_read_busy  = TRUE,                    /*    Is the ACTION_READ packet busy?  */
     sp_write_busy = FALSE,                   /*    Is the ACTION_WRITE packet busy? */
     done          = FALSE;                   /*    Is the program finished?         */

                /* main() has already taken care of sending the initial read to the  */
                /* handler.  Because we need the data from that read before we can   */
  while (!done) /* do anything, the first thing to do is wait for its return.        */
  {
    do                                 /*      Wait for the ACTION_READ to return. */
    {
      signals = Wait(sigmask);         /*          Wait for port signal or CTRL-C. */

      if (signals & portsignal)        /* If a message arrived at the port, ...    */
      {
        while (mysp = (struct StandardPacket *)GetMsg(myport)) /* ...empty the port. */
        {
                          /* If this message is the ACTION_READ packet, mark it as   */
                          /* no longer busy so we can use it to start another read.  */
          if (mysp->sp_Pkt.dp_Type == ACTION_READ)    sp_read_busy  = FALSE;

                          /* If this message is instead the ACTION_WRITE packet,     */
                          /* mark it as not busy.  We need to check for this because */
                          /* the WRITE_PACKET from the previous interation through   */
                          /* the loop might have come back before the ACTION_WRITE   */
                          /* from the previous interation.                           */
          else
            if (mysp->sp_Pkt.dp_Type == ACTION_WRITE) sp_write_busy = FALSE;
        }
      }

      if (signals & SIGBREAKF_CTRL_C) /* If someone hit CTRL-C, start to quit. */
      {
        done = TRUE;                     /* If the ACTION_READ is still out, try to */
        if (sp_read_busy)                /* abort it.  As of V39, AbortPkt() does   */
          AbortPkt(in->fh_Type, sp_read); /* not do anything, so this function has   */
      }                                  /* no effect. Maybe a later release of the */
                                         /* OS will support packet aborting.        */


    } while (sp_read_busy);                  /* End of "wait for ACTION_READ" loop. */

                                         /* Get ready to send the next ACTION_READ. */
    buf = (UBYTE *)(sp_read->dp_Arg2);   /* Hold on to the important stuff from the */
```

```
    amount_read = sp_read->dp_Res1;       /* ACTION_READ we just got back so we can  */
                                          /* reuse the packet to start a new read    */
                                          /* while processing the last read's data.  */

    while (sp_write_busy)          /* Because this example only uses two buffers and */
    {                              /* the ACTION_WRITE might be using one of them,   */
                                   /* this example has to wait for an outstanding    */
                                   /* ACTION_WRITE to return before reusing the      */
                                   /* ACTION_WRITE packet's buffer.                  */
      signals = Wait(sigmask);

      if (signals & portsignal)           /*  If a message arrived at the port, ...  */
      {                                              /*           ... empty the port. */
        while (mysp = (struct StandardPacket *)GetMsg(myport))
          if (mysp->sp_Pkt.dp_Type == ACTION_WRITE) sp_write_busy = FALSE;
      }

      if (signals & SIGBREAKF_CTRL_C)     /*   If someone hit CTRL-C, start to quit. */
      {
        done = TRUE;                      /* If the ACTION_READ is still out, try to */
        if (sp_write_busy) AbortPkt(out->fh_Type, sp_write);            /* abort it. */
      }
    }

               /* Make sure the user didn't hit CTRL-C.  If the user hit CTRL-C dur- */
    if (!done) /* ing one of the "wait for packet" loops, done == TRUE.  Notice that */
    {          /* this example does not actually break for the CTRL-C until after it */
               /* gets back both packets.                                            */

               /* This tests the return values from the ACTION_READ and ACTION_WRITE */
               /* packets.  The ACTION_READ packet returns the number of bytes it    */
               /* read in dp_Res1, which was copied earlier into amount_read. If it  */
               /* is 0, the read packet found the EOF.  If it is negative, there was */
               /* an error.  In the case of ACTION_WRITE, an error occurs if the     */
               /* number of bytes that ACTION_WRITE was supposed to write (Arg3)     */
               /* does not match the actual number it wrote, which ACTION_WRITE re-  */
               /* turns in Res1.  This test is the reason dp_Res1 and dp_Arg3 were   */
               /* set to zero when the ACTION_WRITE packet was set up in main().     */
      if ((amount_read > 0) && (sp_write->dp_Res1 == sp_write->dp_Arg3))
      {
        sp_read->dp_Arg2 = sp_write->dp_Arg2;  /* ACTION_WRITE is finished with its  */
                                               /* buffer, use it in the next read.   */

        SendPkt(sp_read, in->fh_Type, myport); /* Send the next ACTION_READ and mark */
        sp_read_busy = TRUE;                   /* the ACTION_READ as busy.           */

        /* Process Buffer.  This example doesn't do anything with the data from the  */
        /* last ACTION_READ, it just passes it on to the STDOUT handler.             */


        sp_write->dp_Arg2 = (LONG)buf;         /*    Set up the ACTION_WRITE packet. */
        sp_write->dp_Arg3 = amount_read;
        SendPkt(sp_write, out->fh_Type, myport);   /* Send the next ACTION_WRITE and */
        sp_write_busy = TRUE;                      /* mark the ACTION_WRITE as busy. */
      }
      else                              /* A packet returned with a failure, so quit. */
        done = TRUE;
    }
  }
}
```