```
//
//      OTVirtualServer         by Eric Okholm          Version 1.0.1
//
//      This is an OpenTransport sample server application which demonstrates
//      a fast framework for making an OpenTransport server application.
//
//      This version of the server simply opens a listener endpoint and
//      many endpoints which can accept connections.  When inbound connections
//      are received, it waits to receive a 128 byte "request", then it sends
//      a predetermined data from memory (not disk) and begins an orderly release
//      of the connection.
//
//      Future iterations of this program will retreive data from disk to return,
//      demonstrating synchronization methods, and do ADSP, demonstrating
//      protocol independence.
//
//      You are welcome to use this code in any way to create you own
//      OpenTransport applications.   For more information on this program,
//      please review the document "About OTVirtual Server".
//
//      Go Bears, beat Stanford !!!
//
//      What's new in version 1.0.1:
//
//      (1) Worked around a bug found when using AckSends and sending the same
//          buffer more than once.   See the routine SendData for details.
//
//      To do:
//
//      (1) Improve statistics window.
//      (2) General routine for processing kOTLookErrs
//      (3) Handle inbound T_ORDREL processing inside other notifications.
//      (4) Allow running on OT 1.1 by including a copy of tilisten module to install.
//

#define DoAlert(x)          { sprintf(gProgramErr, x); gProgramState = kProgramError; }
#define DoAlert1(x, y)      { sprintf(gProgramErr, x, y); gProgramState = kProgramError; }
#define DoAlert2(x, y, z)   { sprintf(gProgramErr, x, y, z); gProgramState = kProgramError;}

//
//  Program mode
//
//  Before compiling,
//  set kDebugLevel to 0 for production
//                  or 1 for debug code.
//
//  In production mode, the code attempts to recover cleanly from any problems in encounters.
//  In debug mode, the unexplained phenomenon cause an alert box highlighting the situation
//  to be delivered and then the program exits.
//

#define kDebugLevel 1

#if kDebugLevel > 0

#define DBAlert(x)          DoAlert(x)
#define DBAlert1(x, y)      DoAlert1(x, y)
#define DBAlert2(x, y, z)   DoAlert2(x, y, z)

#else

#define DBAlert(x)          { }
#define DBAlert1(x, y)      { }
#define DBAlert2(x, y, z)   { }

#endif
```



```
//
//  Include files
//
#include <Dialogs.h>
#include <Events.h>
#include <Fonts.h>
#include <GestaltEqu.h>
#include <Memory.h>
#include <Menus.h>
#include <QuickDraw.h>
#include <SegLoad.h>
#include <stdio.h>
#include <StdLib.h>
#include <String.h>
#include <strings.h>
#include <ToolUtils.h>
#include <Windows.h>

#include <OpenTptInternet.h>        // includes OpenTransport.h
#include <OpenTptClient.h>          // needed for OTReleaseBuffer()

//
//  Defines, enums, resource IDs
//
#define kInFront                    (WindowPtr) -1
#define kWindowResID                128

    // Apple Menu
#define kAppleMenuResID             128
#define kAppleMenuAbout             1

    // File Menu
#define kFileMenuResID              129
#define kFileMenuOpen               1
#define kFileMenuClose              2
#define kFileMenuQuit               4

    // Edit Menu
#define kEditMenuResID              130             // Edit menu is disabled

    // Server Menu
#define kServerMenuResID            131
#define kServerMenuTCPPrefs         1

    // Alerts, etc.
#define kAlertExitResID             128
#define kAboutBoxResID              130

    // TCP Prefs Dialog
#define kTCPPrefsDlogResID          129
#define kListenerPortDItem          2
#define kListenerQueueDepthDItem    4
#define kMaxConnectionsDItem        6
#define kReturnDataLengthDItem      8
#define kStartStopDItem             9

    // Overall program states
enum
{
    kProgramRunning     = 0,
    kProgramDone        = 1,
    kProgramError       = 2
};

    // Server states
enum
{
```

```
    kServerStopped      = 0,
    kServerRunning      = 1,
    kServerShuttingDown = 2
};

    // Bit numbers in EPInfo stateFlags fields
enum
{
    kBrokenBit                      = 0,
    kOpenInProgressBit              = 1,
    kFlushDisconnectInProgressBit   = 2,
    kIPReuseAddrBit                 = 3,
    kPassconBit                     = 4,
    kGotRequestBit                  = 5,
    kWaitingBit                     = 6
};

    // Misc stuff
enum
{
    kDontQueueIt                = 0,
    kQueueIt                    = 1,
    kTimerHitsBeforeAcceptMax   = 2,
    kTimerIntervalInSeconds     = 3,
    kTimerInterval              = (kTimerIntervalInSeconds * 1000),
    kRequestSize                = 128,
    kOTVersion113               = 0x01130000,
    kOTVersion111               = 0x01110000,
    kDataBufSize                = (16 * 1024),
    kTCPKeepAliveInSecs         = (30 * 1000)       // 30 sec in msec.
};


//
//  Globals
//
int                 gServerState                = kServerStopped;
int                 gProgramState               = kProgramRunning;
char                gProgramErr[128];
DialogPtr           gDialogPtr                  = NULL;
WindowPtr           gWindowPtr                  = NULL;
long                gSleepTicks                 = 60;
Str255              gListenerPortStr            = "\p2001";
long                gListenerPort               = 2001;
Str255              gListenerQueueDepthStr      = "\p20";
long                gListenerQueueDepth         = 20;
Str255              gMaxConnectionsStr          = "\p100";
long                gMaxConnections             = 100;
long                gMaxConnectionsAllowed      = 0;
Str255              gReturnDataLengthStr        = "\p2048";
long                gReturnDataLength           = 2048;
Boolean             gServerRunning              = false;
Str255              gStartStr                   = "\pStart";
Str255              gStopStr                    = "\pStop";
SInt32              gCntrEndpts                 = 0;
SInt32              gCntrIdleEPs                = 0;
SInt32              gCntrBrokenEPs              = 0;
SInt32              gCntrConnections            = 0;
SInt32              gCntrTotalBrokenEPs         = 0;
SInt32              gCntrTotalConnections       = 0;
SInt32              gCntrTotalBytesSent         = 0;
Boolean             gListenPending              = false;
OTLIFO              gIdleEPLIFO;
OTLIFO*             gIdleEPs                    = &gIdleEPLIFO;
OTLIFO              gBrokenEPLIFO;
OTLIFO*             gBrokenEPs                  = &gBrokenEPLIFO;
OTLIFO              gWaitingEPLIFO;
```



```
OTLIFO*             gWaitingEPs                 = &gWaitingEPLIFO;
OTConfiguration*    gCfgMaster                  = NULL;
long                gTimerTask                  = 0;
SInt32              gCntrIntervalConnects       = 0;
SInt32              gCntrIntervalBytes          = 0;
SInt32              gConnectsPerSecond          = 0;
SInt32              gConnectsPerSecondMax       = 0;
SInt32              gKBytesPerSecond            = 0;
SInt32              gKBytesPerSecondMax         = 0;
SInt32              gCntrIntervalEventLoop      = 0;
SInt32              gEventsPerSecond            = 1;
SInt32              gEventsPerSecondMax         = 1;
Boolean             gWaitForEventLoop           = false;
Boolean             gDoWindowUpdate             = true;
SInt32              gAllowNewMax                = kTimerHitsBeforeAcceptMax;
OSType              gOTVersionSelector          = 'otvr';
UInt32              gOTVersion;

struct EPInfo
{
    EndpointRef     erf;                //  actual endpoint
    OTLink          link;               //  link into an OT LIFO (atomic)
    SInt32          outstandingSends;   //  number of T_MEMORYRELEASED events expected
    unsigned char*  sendPtr;            //  ptr to next byte to send
    UInt32          sendBytes;          //  remaining bytes to send
    SInt32          rcvdBytes;          //  bytes received (we pretend 128 bytes in is a request
    UInt8           stateFlags;         //  various status fields
};
typedef struct EPInfo EPInfo;

EPInfo* gListener   = NULL;
EPInfo* gAcceptors  = NULL;

unsigned char gDataBuf[kDataBufSize];

//
//  Option structure
//
//  This is used to pass down both IP_REUSEADDR and TCP_KEEPALIVE in the
//  same option message
//

struct TKeepAliveOpt
{
    UInt32      len;
    OTXTILevel  level;
    OTXTIName   name;
    UInt32      status;
    UInt32      tcpKeepAliveOn;
    UInt32      tcpKeepAliveTimer;
};
typedef struct TKeepAliveOpt TKeepAliveOpt;

//
//  OpenTransport Networking Code Prototypes
//
static void         CheckUnbind(EPInfo*, OTResult, Boolean);
static void         DoListenAccept();
static void         DoRcvDisconnect(EPInfo*);
static void         EnterListenAccept();
static Boolean      EPClose(EPInfo*);
static Boolean      EPOpen(EPInfo*);
static void         NetInit(void);
static void         NetShutdown(void);
static pascal void  Notifier(void*, OTEventCode, OTResult, void*);
static void         ReadData(EPInfo*);
static void         Recycle(void);
```

```
static void         SendData(EPInfo*);
static void         StartServer(void);
static void         StopServer(void);
static void         TimerInit();
static void         TimerDestroy();
static pascal void  TimerRun(void*);

//
//  Macintosh Program Wrapper Prototypes
//
static void         AboutBox(void);
static void         AlertExit(Str255);
static void         C2PStr(char*, Str255);
static void         DialogClose(void);
static Boolean      EventDialog(EventRecord*);
static void         EventDrag(WindowPtr, Point);
static void         EventGoAway(WindowPtr, Point);
static void         EventKeyDown(EventRecord*);
static void         EventLoop(void);
static void         EventMouseDown(EventRecord*);
static void         MacInit(void);
static void         MacInitROM(void);
static void         MenuDispatch(long);
static void         P2CStr(Str255, char*);
static void         SetupMenus(void);
static void         TCPPrefsDialog(void);
static void         TCPPrefsReset(void);
static void         WindowClose(void);
static void         WindowOpen(void);
static void         WindowUpdate(void);


////////////////////////////////////////////////////////////////////////////////
//
//  OpenTransport Networking Code
//
//  The code in this section provides the networking portions of the
//  OpenTransport Virtual Server.
//
////////////////////////////////////////////////////////////////////////////////

//
//  CheckUnbind
//
//  This routine checks the results of an unbind.   Due to various problems
//  in OpenTransport, an OTUnbind can fail for a number of reasons.  This problem
//  is timing related so you usually won't hit it.   When an OTUnbind fails,
//  we assume the best way to recover is to throw the endpoint on the broken
//  list to be recycled.   Later, in the recycle routine, it will be closed
//  and a new endpoint will be opened to replace it.  If the OTUnbind is
//  successful, the endpoint is put back on the free list to be reused.
//
//  Since the unbind failure is timing related, a more efficient solution
//  would probably be to wait and retry the unbind in a few seconds,
//  expecting that the call would not fail on the next try.
//
static void CheckUnbind(EPInfo* epi, OTResult result, Boolean queueIt)
{
    if (result != kOTNoError)
    {
        if ( OTAtomicSetBit(&epi->stateFlags, kBrokenBit) == 0 )
        {
            //
            //  The OTAtomicSetBit guarantee's that the EPInfo won't be
            //  enqueued twice.   We only enqueue the EPInfo if the previous
            //  state of the bit was 0.
            //
```



```
            OTLIFOEnqueue(gBrokenEPs, &epi->link);
            OTAtomicAdd32(1, &gCntrBrokenEPs);
            OTAtomicAdd32(1, &gCntrTotalBrokenEPs);
        }
    }
    else
    {
        if (queueIt)
        {
            OTLIFOEnqueue(gIdleEPs, &epi->link);
            OTAtomicAdd32(1, &gCntrIdleEPs);
            if (gListenPending)
                EnterListenAccept();
        }
    }
}

//
//  EnterListenAccept
//
//  This is a front end to DoListenAccept() which is used whenever
//  it is not being called from inside the listener endpoint's notifier.
//  We do this for syncrhonization.   If we were processing an OTListen()
//  or an OTAccept() and we were interrupted at the listener endpoint's
//  notifier with a T_LISTEN, etc, it would be inconvenient and would require
//  some more sophisticated synchronization code to deal with the problem.
//  The easy way to avoid this is to do an OTEnterNotifier() on the listener's
//  endpoint.
//
//  Important note - doing OTEnterNotifier on one endpoint only prevents that
//  endpoint's notifier for interrupting us.   Since the same notifier code
//  is used for lots of endpoints here, remember that the one endpoint's
//  notifier can interrupt another.   Doing an OTEnterNotifier() on the
//  listener endpoint prevents the listener from interrupting us, but it
//  does not prevent the Notifier() routine from interrupting us via
//  another endpoint which also uses the same routine.
//
//  Important note #2 - Don't ever do an OTEnterNotifier on an acceptor endpoint
//  before doing the OTAccept().   This confuses OT and creates problems.
//
static void EnterListenAccept()
{
    Boolean doLeave;

    doLeave = OTEnterNotifier(gListener->erf);
    DoListenAccept();
    if (doLeave)
        OTLeaveNotifier(gListener->erf);
}

//
//  DoListenAccept
//
//  The handling of a T_LISTEN is greatly simplified by use
//  of the tilisten module, which serializes inbound connections.
//  This means that when doing an OTAccept we won't get a kOTLookErr
//  because another inbound connection arrived and created a T_LISTEN.
//  Without the tilisten module, we have to use the "8 step
//  listen/accept/disconnect method", which is documented elsewhere.
//  At this point, if we have a free endpoint, accept the connection.
//  If we don't, assume we are overloaded and reject the connection.
//
//  When we are called from inside the notifier due to a T_LISTEN,
//  DoListenAccept() is called directly.
//
//  When we restart delayed handling of a T_LISTEN, either because of
//  doing a throttle-back or because the program ran out of free endpoints,
```

```
//  EnterListenAccept() is called for synchronization on the listener endpoint.
//
static void DoListenAccept()
{
    TCall       call;
    InetAddress caddr;
    OTResult    lookResult;
    OTLink*     acceptor_link;
    EPInfo*     acceptor;
    OSStatus    err;

    //
    //  By deferring handling of a T_LISTEN, we can slow down inbound requests
    //  and get some time to make sure the event loop occurs.   This is important
    //  so that: (1) the user can quit the program, (2) so memory can be restructured,
    //  (3) so we can recycle broken endpoints and other administrative tasks that
    //  are not done in the notifier.
    //
    if (gWaitForEventLoop)
    {
        gListenPending = true;
        return;
    }

    //
    //  Get an EPInfo & endpoint.   If none are available, defer handling the T_LISTEN.
    //
    acceptor_link = OTLIFODequeue(gIdleEPs);
    if (acceptor_link == NULL)
    {
        gListenPending = true;
        return;
    }

    OTAtomicAdd32(-1, &gCntrIdleEPs);
    gListenPending = false;
    acceptor = OTGetLinkObject(acceptor_link, EPInfo, link);
    acceptor->stateFlags = 0;
    acceptor->rcvdBytes = 0;

    call.addr.maxlen = sizeof(InetAddress);
    call.addr.buf = (unsigned char*) &caddr;
    call.opt.maxlen = 0;
    call.opt.buf = NULL;
    call.udata.maxlen = 0;
    call.udata.buf = NULL;

    err = OTListen(gListener->erf, &call);
    if (err != kOTNoError)
    {
        //
        //  Only two errors are expected at this point.
        //  One would be a kOTNoDataErr, indicating the inbound connection
        //  was unavailable, temporarily hidden by a higher priority streams
        //  message, etc.   The more likely error is a kOTLookErr,
        //  which indicates a T_DISCONNECT on the OTLook()
        //  happens when the call we were going to process disconnected.
        //  In that case, go away and wait for the next T_LISTEN event.
        //
        OTLIFOEnqueue(gIdleEPs, &acceptor->link);
        OTAtomicAdd32(1, &gCntrIdleEPs);
        if (err == kOTNoDataErr)
            return;

        lookResult = OTLook(gListener->erf);
        if (err == kOTLookErr && lookResult == T_DISCONNECT)
            DoRcvDisconnect(gListener);
        else
            DBAlert2("Notifier: T_LISTEN - OTListen error %d lookResult %x", err, lookResult);
        return;
    }

    err = OTAccept(gListener->erf, acceptor->erf, &call);
    if (err != kOTNoError)
    {
        //
        //  Again, we have to be able to handle the connection being disconnected
        //  while we were trying to accept it.
        //
        OTLIFOEnqueue(gIdleEPs, &acceptor->link);
        OTAtomicAdd32(1, &gCntrIdleEPs);
        lookResult = OTLook(gListener->erf);
        if (err == kOTLookErr && lookResult == T_DISCONNECT)
            DoRcvDisconnect(gListener);
        else
            DBAlert2("Notifier: T_LISTEN - OTAccept error %d lookResult %x", err, lookResult);
    }
}


//
//  DoRcvDisconnect
//
//  This routine is called from the notifier in T_LISTEN handling
//  upon getting a kOTLookErr back indicating a T_DISCONNECT needs to be handled.
//
static void DoRcvDisconnect(EPInfo* epi)
{
    OSStatus err;

    err = OTRcvDisconnect(epi->erf, NULL);
    if (epi == gListener)
    {
        //
        //  We can get a disconnect on the listener if an inbound connection was
        //  being disconnected (sent a RST) while we were in the process of refusing
        //  it because we had no idle endpoints).   In this case, we don't really
        //  want to do anything other than receive the disconnect and move on.
        //
        if (err != kOTNoError)
            DBAlert1("DoRcvDisconnect: OTRcvDisconnect on listener error %d", err);
        return;
    }
    if (err != kOTNoError)
    {
        if (err != kOTNoDisconnectErr)
            DBAlert1("DoRcvDisconnect: OTRcvDisconnect error %d", err);
        return;
    }

    //
    //  Don't start the unbind yet if the endpoint is on the waiting list
    //  and is scheduled for an orderly release (which can no longer happen).
    //  Instead, if it is scheduled, just clear the bit so we know later
    //  to do the unbind instead of the orderly relase.
    //
    if ((OTAtomicClearBit(&epi->stateFlags, kWaitingBit)) == 0)
        CheckUnbind(epi, OTUnbind(epi->erf), kDontQueueIt);
}


//
//  DoSndOrderlyDisconnect
//
```

```
//  This routine is a front end to OTSndOrderlyDisconnect().
//  In OT 1.1.2 and earlier releases, there is a problem in OT/TCP which can cause
//  OT/TCP to forget to send the orderly release indication upstream if the system
//  is running so fast the event loop doesn't get time.  To work around this problem,
//  we defer sending the orderly release until the event loop runs.   In OT 1.1.3 and
//  later the routine is called from the notifier instead.  The cost of this workaround
//  is about 18% in terms of connections per second, but the workaround appears to
//  be 100% reliable.
//
static void DoSndOrderlyDisconnect(EPInfo* epi)
{
    OSStatus err;
    OTResult epState;

    err = OTSndOrderlyDisconnect(epi->erf);
    epState = OTGetEndpointState(epi->erf);
    if (err != kOTNoError)
    {
        DBAlert2("DoSndOrderlyDisconnect: OTSndOrderlyDisconnect error %d state %d", err, epStat
        return;
    }

    //
    //  Check the endpoint state to see if we are in T_IDLE.  If so,
    //  the connection is fully broken down and we can unbind are requeue
    //  the endpoint for reuse.   If not, then wait until we have also received
    //  an orderly release from the other side, at which time we will also check
    //  the state of the endpoint and unbind there if required.
    //
    epState = OTGetEndpointState(epi->erf);
    if (epState == T_IDLE)
    {
        CheckUnbind(epi, OTUnbind(epi->erf), kDontQueueIt);
    }
}

//
//  DoWaitList
//
//  This routine is only used when running on OT 1.1.2 or earlier releases.
//  Check the comments in DoSndOrderlyDisconnect for an explanation.
//  We always check the kWaitingBit to make sure we still need to do the
//  orderly release.   If it has been cleared, then the endpoint has already
//  been disconnected and we can just toss it back into the idle list.
//
static void DoWaitList()
{
    OTLink*     list = OTLIFOStealList(gWaitingEPs);
    OTLink*     link;
    EPInfo*     epi;

    while ( (link = list) != NULL )
    {
        list = link->fNext;
        epi = OTGetLinkObject(link, EPInfo, link);
        if ((OTAtomicClearBit(&epi->stateFlags, kWaitingBit)) != 0)
            DoSndOrderlyDisconnect(epi);
        else
            CheckUnbind(epi, OTUnbind(epi->erf), kDontQueueIt);
    }
}


//
//  EPClose
//
//  This routine is a front end to OTCloseProvider.   Centralizing closing of
```



```
//  endpoints makes debugging and instrumentation easier.   Also, since this
//  program uses Ack Sends to avoid data copies when doing OTSnd(), some special
//  care is required at close time.
//
static Boolean EPClose(EPInfo* epi)
{
    OSStatus err;

    //
    //  If an endpoint is still being opened, we can't close it yet.
    //  There is no way to cancel an OTAsyncOpenEndpoint, so we just
    //  have to wait for the T_OPENCOMPLETE event at the notifier.
    //
    if (OTAtomicTestBit(&epi->stateFlags, kOpenInProgressBit) != 0)
        return false;

    //
    //  If the OTAsyncOpenEndpoint failed, the endpoint ref will be NULL,
    //  and we don't need to close it now.
    //
    if (epi->erf == NULL)
        return true;

    if (epi->outstandingSends == 0)
    {
        err = OTCloseProvider(epi->erf);
        epi->erf = NULL;
        if (err != kOTNoError)
            DBAlert1("EPClose: OTCloseProvider error %d", err);
        if (epi != gListener)
            gCntrEndpts--;
        return true;
    }

    //
    //  If we get to this point, the endpoint did an OTSnd() with AckSends,
    //  and the T_MEMORYRELEASED event hasn't been returned yet.  In order
    //  to make sure we get the event, we flush the stream and then do an
    //  OTDisconnect().   This should get the memory freed so we can close
    //  the endpoint safely.   Note, we set a flag so we don't do this
    //  more than once on an endpoint.
    //
    if ( OTAtomicSetBit(&epi->stateFlags, kFlushDisconnectInProgressBit) == 0 )
    {
        err = OTIoctl(epi->erf, I_FLUSH, (void *)FLUSHRW);
        if (err != kOTNoError)
            DBAlert1("EPClose: I_FLUSH error %d", err);
    }
    return false;
}

//
//  EPOpen:
//
//  A front end to OTAsyncOpenEndpoint.
//  A status bit is set so we know there is an open in progress.
//  It is cleared when the notifier gets a T_OPENCOMPLETE where the context
//  pointer is this EPInfo.  Until that happens, this EPInfo can't be cleaned
//  up and released.
//
static Boolean EPOpen(EPInfo* epi, OTConfiguration* cfg)
{
    OSStatus err;

    //
    //  Clear all old state bits and set the open in progress bit.
    //  This doesn't need to be done atomicly because we are
```

```
    //  single threaded on this endpoint at this point.
    //
    epi->erf = NULL;
    epi->stateFlags = 1 << kOpenInProgressBit;
    err = OTAsyncOpenEndpoint(cfg, 0, NULL, &Notifier, epi);
    if (err != kOTNoError)
    {
        OTAtomicClearBit(&epi->stateFlags, kOpenInProgressBit);
        DBAlert1("EPOpen: OTAsyncOpenEndpoint error %d", err);
        return false;
    }
    return true;
}

//
//  NetEventLoop
//
//  This routine is called once during each pass through the program's event loop.
//  If the program is running on OT 1.1.2 or an earlier release, this is where
//  outbound orderly releases are started (see comments in DoSndOrderlyRelease
//  for more information on that).   This is also where endpoints are "fixed" by
//  closing them and opening a new one to replace them.   This is rarely necessary,
//  but works around some timing issues in OTUnbind().  Having passed through the
//  event loop once, we assume it is safe to turn off throttle-back.  And, finally,
//  if we have deferred handing of a T_LISTEN, here we start it up again.
//
static void NetEventLoop()
{
    if (gOTVersion < kOTVersion113)
        DoWaitList();
    Recycle();
    gWaitForEventLoop = false;
    if (gListenPending)
        EnterListenAccept();
}

//
//  NetInit:
//
//  This routine does various networking related startup tasks:
//
//  (1) it does InitOpenTransport
//  (2) it records the OT version for us.
//  (3) it starts our timer interrupt running.
//
static void NetInit()
{
    OSStatus err;

    err = InitOpenTransport();
    if (err)
    {
        DBAlert1("NetInit: InitOpenTransport error %d", err);
        return;
    }
    err = Gestalt(gOTVersionSelector, (long*) &gOTVersion);
    if (err || (gOTVersion < kOTVersion111))
    {
        DoAlert("Please install Open Transport 1.1.1 or later");
        return;
    }
    TimerInit();
}

//
//  NetShutdown:
//
```



```
//  This routine does various networking related shutdown tasks:
//
static void NetShutdown()
{
    TimerDestroy();
    CloseOpenTransport();
}

//
//  Notifier:
//
//  Most of the interesting networking code in this program resides inside
//  this notifier.   In order to run asynchronously and as fast as possible,
//  things are done inside the notifier whenever possible.  Since almost
//  everything is done inside the notifier, there was little need for specical
//  synchronization code.
//
//  In the next iteration of this program, when information to be sent is
//  actually retreived from the disk, the synchronization, particularly for
//  doing sends and handling flow control, will become more complicated.
//
//  IMPORTANT NOTE:  Normal events defined by XTI (T_LISTEN, T_CONNECT, etc)
//  and OT completion events (T_OPENCOMPLETE, T_BINDCOMPLETE, etc.) are not
//  reentrant.  That is, whenever our notifier is invoked with such an event,
//  the notifier will not be called again by OT for another normal or completion
//  event until we have returned out of the notifier - even if we make OT calls
//  from inside the notifier.   This is a useful synchronization tool.
//  However, there are two kinds of events which will cause the notifier to
//  be reentered.   One is T_MEMORYRELEASED, which always happens instantly.
//  The other are state change events like kOTProviderWillClose.
//
static pascal void Notifier(void* context, OTEventCode event, OTResult result, void* cookie)
{
    OSStatus err;
    OTResult epState;
    EPInfo* epi = (EPInfo*) context;

    //
    //  Once the program is shutting down, most events would be uninteresting.
    //  However, we still need T_OPENCOMPLETE and T_MEMORYRELEASED events since
    //  we can't call CloseOpenTransport until all OTAsyncOpenEndpoints and
    //  OTSends with AckSends have completed.   So those specific events
    //  are still accepted.
    //
    if (gProgramState != kProgramRunning)
    {
        if ((event != T_OPENCOMPLETE) && (event != T_MEMORYRELEASED))
        {
            return;
        }
    }

    //
    //  This really isn't necessary, it's just a sanity check which should be removed
    //  once a program is debugged.   It's just making sure we don't get event notifications
    //  after all of our endpoints have been closed.
    //
    if (gServerState == kServerStopped)
    {
        DBAlert1("Notifier: got event %d when server not running!", event);
        return;
    }

    //
    //  Within the notifier, all action is based on the event code.
    //  In this notifier, fatal errors all break out of the switch to the bottom.
    //  As long as everything goes as expected, the case returns rather than breaks.
```

```
    //
    switch (event)
    {
        //
        //  kStreamIoctlEvent:
        //
        //  This event is returned when an I_FLUSH ioctl has completed.
        //  The flush was done in an attempt to get back all T_MEMORYRELEASED events
        //  for outstanding OTSnd() calls with Ack Sends.   For good measure, we
        //  send a disconnect now.   Errors are ignored at this point since it is
        //  possible that the connection will already be gone, etc.
        //
        case kStreamIoctlEvent:
        {
            if (OTAtomicTestBit(&epi->stateFlags, kOpenInProgressBit) != 0)
                (void) OTSndDisconnect(epi->erf, NULL);
            return;
        }

        //
        //  T_ACCEPTCOMPLETE:
        //
        //  This event is received by the listener endpoint only.
        //  The acceptor endpoint will get a T_PASSCON event instead.
        //
        case T_ACCEPTCOMPLETE:
        {
            if (result != kOTNoError)
                DBAlert1("Notifier: T_ACCEPTCOMPLETE - result %d", result);
            return;
        }

        //
        //  T_BINDCOMPLETE:
        //
        //  We only bind the listener endpoint, and bind failure is a fatal error.
        //  Acceptor endpoints are bound within the OTAccept() call when they get a connection.
        //
        case T_BINDCOMPLETE:
        {
            if (result != kOTNoError)
                DoAlert("Unable to set up listening endpoint, exiting");
            return;
        }

        //
        //  T_DATA:
        //
        //  The main rule for processing T_DATA's is to remember that once you have
        //  a T_DATA, you won't get another one until you have read to a kOTNoDataErr.
        //  The advanced rule is to remember that you could get another T_DATA
        //  during an OTRcv() which will eventually return kOTNoDataErr, presenting
        //  the application with a synchronization issue to be most careful about.
        //
        //  In this application, since an OTRcv() calls are made from inside the notifier,
        //  this particular synchronization issue doesn't become a problem.
        //
        case T_DATA:
        {
            //
            //  Here we work around a small OpenTransport bug.
            //  It turns out, since this program does almost everything from inside the notifier
            //  that during a T_UNBINDCOMPLETE we can put an EPInfo back into the idle list.
            //  If that notification is interrupted by a T_LISTEN at the notifier, we could
            //  end up starting a new connection on the endpoint before OT unwinds the stack
            //  out of the code which delivered the T_UNBINDCOMPLETE.   OT has some specific
            //  code to protect against a T_DATA arriving before the T_PASSCON, but in this
```



```
            //  case it gets confused and the events arrive out of order.   If we try to
            //  do an OTRcv() at this point we will get a kOTStateChangeErr because the endpoint
            //  is still locked by the earlier OTAccept call until the T_PASSCON is delivered
            //  to us.   This is fairly benign and can be worked around easily.  What we do
            //  is note that the T_PASSCON hasn't arrived yet and defer the call to ReadData()
            //  until it does.
            //
            if ( OTAtomicSetBit(&epi->stateFlags, kPassconBit) != 0 )
            {
                //
                //  Because are are running completely inside notifiers,
                //  it is possible for a T_DATA to beat a T_PASSCON to us.
                //  We need to help OT out when this occurs and defer the
                //  data read until the T_PASSCON arrives.
                //
                ReadData(epi);
            }
            return;
        }

        //
        //  T_DISCONNECT:
        //
        //  An inbound T_DISCONNECT event usually indicates that the other side of the
        //  connection did an abortive disconnect (as opposed to an orderly release).
        //  It also can be generated by the transport provider on the system (e.g. tcp)
        //  when it decides that a connection is no longer in existance.
        //
        //  We receive the disconnect, but this program ignores the associated reason (NULL para
        //  It is possible to get back a kOTNoDisconnectErr from the OTRcvDisconnect call.
        //  This can happen when either (1) the disconnect on the stream is hidden by a
        //  higher priority message, or (2) something has flushed or reset the disconnect
        //  event in the meantime.   This is not fatal, and the appropriate thing to do is
        //  to pretend the T_DISCONNECT event never happened.   Any other error is unexpected
        //  and needs to be reported so we can fix it.  Next, unbind the endpoint so we can
        //  reuse it for a new inbound connection.
        //
        //  It is possible to get an error on the unbind due to a bug in OT 1.1.1 and earlier.
        //  The best thing to do for that is close the endpoint and open a new one to replace it
        //  We do this back in the main thread so we don't have to deal with synchronization pro
        //
        case T_DISCONNECT:
        {
            DoRcvDisconnect(epi);
            return;
        }

        //
        //  T_DISCONNECTCOMPLETE:
        //
        //  Sometimes this is called as a result of the
        //  I_FLUSH / OTSndDisconenct() combo in StopServer to relaim
        //  all memory via T_MEMORYRELEASED events so we can close down.
        //  We don't actually release any memory or remove the EPInfo
        //  from a list so we don't have to synchronize with the main
        //  thread.   It will get cleaned up on the next call to StopServer().
        //
        //  Note, this is where we would normally clear the stateFlags
        //  for kFlushDisconnectInProgress, but since there is no point in
        //  doing the flush/disconnect more than once, we never clear it.
        //
        //
        case T_DISCONNECTCOMPLETE:
        {
            if (result != kOTNoError)
                DBAlert1("Notifier: T_DISCONNECT_COMPLETE result %d", result);
            return;
```

```
        }

        //
        //  T_GODATA:
        //
        //  This event is received when flow control is lifted.   We are under flow control
        //  whenever OTSnd() returns a kOTFlowErr or accepted less bytes than we attempted
        //  to send.  Since SendData() is only called from inside the notifier, we don't
        //  have to worry about interrupting another call to SendData() at this point.
        //
        //  Note, it is also possible to get a T_GODATA without having invoke flow control.
        //  Be safe and prepare for this.
        //
        case T_GODATA:
        {
            SendData(epi);
            return;
        }

        //
        //  T_LISTEN:
        //
        //  Call DoListenAccept() to do all the work.
        //
        case T_LISTEN:
        {
            DoListenAccept();
            return;
        }

        //
        //  T_OPENCOMPLETE:
        //
        //  This event occurs when an OTAsyncOpenEndpoint() completes.   Note that this event,
        //  just like any other async call made from outside the notifier, can occur during
        //  the call to OTAsyncOpenEndpoint().  That is, in the main thread the program did
        //  the OTAsyncOpenEndpoint(), and the notifier is invoked before control is returned
        //  to the line of code following the call to OTAsyncOpenEndpoint().   This is one
        //  event we need to keep track of even if we are shutting down the program since there
        //  is no way to cancel outstanding OTAsyncOpenEndpoint() calls.
        //
        case T_OPENCOMPLETE:
        {
            TOptMgmt        optReq;
            TOption         opt;

            OTAtomicClearBit(&epi->stateFlags, kOpenInProgressBit);
            if (result == kOTNoError)
                epi->erf = (EndpointRef) cookie;
            else
            {
                DBAlert1("Notifier: T_OPENCOMPLETE result %d", result);
                return;
            }

            if (gProgramState != kProgramRunning)
                return;

            if (epi != gListener)
                gCntrEndpts++;

            //
            //  Set to blocking mode so we don't have to deal with kEAGAIN errors.
            //  Async/blocking is the best mode to write an OpenTransport application in (imho).
            //
            err = OTSetBlocking(epi->erf);
            if (err != kOTNoError)
```



```
            {
                DBAlert1("Notifier: T_OPENCOMPLETE - OTSetBlocking error %d", err);
                return;
            }

            //
            //  Set to AckSends so OT doesn't slow down to copy data sent out.
            //  However, this requires special care when closing endpoints, so don't use
            //  AckSends unless you are prepared for this.   Never, ever, close an endpoint
            //  when a send has been done but the T_MEMORYRELEASED event hasn't been returned ye
            //
            err = OTAckSends(epi->erf);
            if (err != kOTNoError)
            {
                DBAlert1("Notifier: T_OPENCOMPLETE - OTAckSends error %d", err);
                return;
            }

            //
            //  Option Management
            //
            //  Turn on ip_reuseaddr so we don't have port conflicts in general.
            //  We use local stack structures here since the memory for the
            //  option request structure is free upon return.   If we were to request
            //  the option return value, we would have to use static memory for it.
            //
            optReq.flags            = T_NEGOTIATE;
            optReq.opt.len          = kOTFourByteOptionSize;
            optReq.opt.buf          = (unsigned char *) &opt;

            opt.len                 = sizeof(TOption);
            opt.level               = INET_IP;
            opt.name                = IP_REUSEADDR;
            opt.status              = 0;
            opt.value[0]            = 1;

            err = OTOptionManagement(epi->erf, &optReq, NULL);
            if (err != kOTNoError)
                DBAlert1("Notifier: T_OPENCOMPLETE - OTOptionManagement err %d", err);

            //
            //  Code path resumes at T_OPTMGMTCOMPLETE
            //
            return;
        }

        //
        //  T_OPTMGMTCOMPLETE:
        //
        //  An OTOptionManagement() call has completed.  These are used on all
        //  endpoints to set IP_REUSEADDR.   It is also used for all endpoints
        //  other than the listener to set TCP_KEEPALIVE which helps recover
        //  server resources if the other side crashes or is unreachable.
        //
        case T_OPTMGMTCOMPLETE:
        {
            TBind               bindReq;
            InetAddress         inAddr;
            TOptMgmt            optReq;
            TKeepAliveOpt       opt;

            if (result != kOTNoError)
            {
                DBAlert1("Notifier: T_OPTMGMTCOMPLETE result %d", result);
                return;
            }
```

```
        if (epi != gListener)
        {
            if ( OTAtomicSetBit(&epi->stateFlags, kIPReuseAddrBit) == 0 )
            {
                //
                //  Turn on TCP_KEEPALIVE so we can recover from connections which have
                //  gone away which we don't know about.   The keepalive value is set
                //  very low here, probably too low for a real server.
                //
                optReq.flags            = T_NEGOTIATE;
                optReq.opt.len          = sizeof(TKeepAliveOpt);
                optReq.opt.buf          = (unsigned char *) &opt;

                opt.len                 = sizeof(TKeepAliveOpt);
                opt.level               = INET_TCP;
                opt.name                = TCP_KEEPALIVE;
                opt.status              = 0;
                opt.tcpKeepAliveOn      = 1;
                opt.tcpKeepAliveTimer   = kTCPKeepAliveInSecs;

                err = OTOptionManagement(epi->erf, &optReq, NULL);
                if (err != kOTNoError)
                {
                    DBAlert1("Notifier: T_OPTMGMTCOMPLETE - OTOptionManagement err %d", err)
                    return;
                }
            }
            else
            {
                //
                //  The endpoint now has both IP_REUSEADDR and TCP_KEEPALIVE set.
                //  It is ready to go on the free list to accept an inbound connection.
                //
                OTLIFOEnqueue(gIdleEPs, &epi->link);
                OTAtomicAdd32(1, &gCntrIdleEPs);
                if (gListenPending)
                    EnterListenAccept();
            }
            return;
        }

        //
        //  Must be listener endpoint, do the bind.  Again, we use stack memory for
        //  the bind request structure and NULL for the bind return structure.
        //
        inAddr.fAddressType     = AF_INET;
        inAddr.fPort            = gListenerPort;
        inAddr.fHost            = 0;                // allow inbound connections from any in

        bindReq.addr.len        = sizeof(InetAddress);
        bindReq.addr.buf        = (unsigned char*) &inAddr;
        bindReq.qlen            = gListenerQueueDepth;

        err = OTBind(epi->erf, &bindReq, NULL);
        if (err != kOTNoError)
            DBAlert1("Notifier: T_OPTMGMTCOMPLETE - OTBind error %d", err);

        return;         // now wait for a T_LISTEN notification
    }

    //
    //  T_MEMORYRELEASED:
    //
    //  This event occurs when OpenTransport is done with the buffer passed in via
    //  an OTSnd() call with AckSends turned on.   The memory is free and we can reuse it.
    //
    //  IMPORTANT NOTE:  This event is reentrant.   That is, this event will interrupt
```



```
    //  our notifier in progress, even interrupting a T_MEMORYRELEASED in progress, so
    //  it must be coded more carefully than most other events.
    //
    case T_MEMORYRELEASED:
    {
        OTAtomicAdd32(-1, &epi->outstandingSends);
        return;
    }

    //
    //  T_ORDREL:
    //
    //  This event occurs when an orderly release has been received on the stream.
    //
    case T_ORDREL:
    {
        err = OTRcvOrderlyDisconnect(epi->erf);
        if (err != kOTNoError)
        {
            //
            //  It is possible for several reasons for the T_ORDREL to have disappeared,
            //  or be temporarily hidden, when we attempt the OTRcvOrderlyDisconnect().
            //  The best thing to do when this happens is pretend that the event never
            //  occured.   We will get another notification of T_ORDREL if the event
            //  becomes unhidden later.  Any other form of error is unexpected and
            //  is reported back so we can correct it.
            //
            if (err == kOTNoReleaseErr)
                return;

            DBAlert1("Notifier: T_ORDREL - OTRcvOrderlyDisconnect error %d", err);
            return;
        }

        //
        //  Sometimes our data sends get stopped with a kOTLookErr
        //  because of a T_ORDREL from the other side (which doesn't close
        //  the connection, it just means they are done sending data).
        //  If so, we still end up in the notifier with the T_ORDREL event,
        //  but we won't resume sending data unless we explictly check
        //  here whether or not we need to do so.
        //
        if (epi->sendBytes > 0)
        {
            SendData(epi);
            return;
        }

        //
        //  Check the endpoint state to see if we are in T_IDLE.  If so,
        //  the connection is fully broken down and we can unbind and requeue
        //  the endpoint for reuse.   If not, then wait until we have also done
        //  an OTSndOrderlyDisconnect, at which time we will also check the state of
        //  of the endpoint and unbind there if required.
        //
        epState = OTGetEndpointState(epi->erf);
        if (epState == T_IDLE)
            CheckUnbind(epi, OTUnbind(epi->erf), kDontQueueIt);

        return;
    }

    //
    //  T_PASSCON:
    //
    //  This event happens on the accepting endpoint, not the listening endpoint.
    //  At this point the connection is fully established and we can begin the
```

```
        //  process of downloading data.  Note that due to a problem in OT it is
        //  possible for a T_DATA to beat a T_PASSCON to the notifier.  When this
        //  happens we note it in the T_DATA case and then start processing the
        //  data here.
        //
        case T_PASSCON:
        {
            if (result != kOTNoError)
            {
                DBAlert1("Notifier: T_PASSCON result %d", result);
                return;
            }
            OTAtomicAdd32(1, &gCntrConnections);
            OTAtomicAdd32(1, &gCntrTotalConnections);
            OTAtomicAdd32(1, &gCntrIntervalConnects);
            if ( OTAtomicSetBit(&epi->stateFlags, kPassconBit) != 0 )
            {
                //
                //  A T_DATA previously beat the T_PASSCON to our notifier.
                //  Here we help OT out by having deferred data processing until now.
                //
                ReadData(epi);
            }
            return;
        }

        //
        //  T_UNBINDCOMPLETE:
        //
        //  This event occurs on completion of an OTUnbind().
        //  The endpoint is ready for reuse on a new inbound connection.
        //  Put it back into the queue of idle endpoints.
        //  Note that the OTLIFO structure has atomic queue and dequeue,
        //  which can be helpful for synchronization protection.
        //
        case T_UNBINDCOMPLETE:
        {
            CheckUnbind(epi, result, kQueueIt);
            return;
        }

        //
        //  default:
        //
        //  There are events which we don't handle, but we don't expect to see
        //  any of them.   When running in debugging mode while developing a program,
        //  we exit with an informational alert.   Later, in the production version
        //  of the program, we ignore the event and try to keep running.
        //
        default:
        {
            DBAlert1("Notifier: unknown event <%x>", event);
            return;
        }
    }
}

//
//  ReadData:
//
//  This routine attempts to read all available data from an endpoint.
//  Since this routine is only called from inside the notifier in the current
//  version of OTVirtualServer, it is not necessary to program to handle
//  getting back a T_DATA notification DURING an OTRcv() call, as would be
//  the case if we read from outside the notifier.   We must read until we
//  get a kOTNoDataErr in order to clear the T_DATA event so we will get
//  another notification of T_DATA in the future.
```



```
//
//  Currently this application uses no-copy receives to get data.  This obligates
//  the program to return the buffers to OT asap.  Since this program does nothing
//  with data other than count it, that's easy.  Future, more complex versions
//  of this program will do more interesting things with regards to that.
//
static void ReadData(EPInfo* epi)
{
    OTBuffer*   bp;
    OTResult    res;
    OTFlags     flags;
    OTResult    epState;
    Boolean     gotRequest = false;

    while (true)
    {
        res = OTRcv(epi->erf, &bp, kOTNetbufDataIsOTBufferStar, &flags);

        //
        //  Note, check for 0 because can get a real 0 length recive
        //  in some protocols (not in TCP), which is different from
        //  getting back a kOTNoDataErr.
        //
        if  (res >= 0 )
        {
            OTAtomicAdd32(res, &epi->rcvdBytes);
            OTAtomicAdd32(res, &gCntrIntervalBytes);
            OTReleaseBuffer(bp);
            if (epi->rcvdBytes >= kRequestSize)
            {
                if (OTAtomicSetBit(&epi->stateFlags, kGotRequestBit) == 0)
                {
                    //
                    //  We have gotten our 128 byte data request, so prepare to respond.
                    //  By setting the bit, we make sure that we can handle requests
                    //  which are bigger than expected without going weird.
                    //
                    epi->sendPtr    = gDataBuf;
                    epi->sendBytes  = gReturnDataLength;
                }
            }
            continue;
        }

        if (res == kOTNoDataErr)
        {
            //
            //  Since ReadData is only called from inside the notifier we don't
            //  have to worry about having missed a T_DATA during an OTRcv.
            //
            break;
        }

        if (res == kOTLookErr)
        {
            res = OTLook(epi->erf);
            if (res == T_ORDREL)
            {
                //
                //  If we got the T_ORDREL, we won't get any more inbound data.
                //  We return and wait for the notifier to get the T_ORDREL notification.
                //  Upon getting it, we will notice we still need to send data and do so.
                //  The T_ORDREL has to be cleared before we can send.
                //
                return;
            }
            if (res == T_GODATA)
```

```
                continue;

            DBAlert1("ReadData: OTRcv got OTLookErr 0x%08x", res);
        }
        else
        {
            epState = OTGetEndpointState(epi->erf);
            if (res == kOTOutStateErr && epState == T_INREL)
            {
                //
                //  Occasionally this problem will happen due to what appears
                //  to be an OpenTransport notifier reentrancy problem.
                //  What has occured is that a T_ORDREL event happened and
                //  was processed during ReadData().   This is proven by being
                //  in the T_INREL state without having done a call to
                //  OTRcvOrderlyDisconnect() here.   It appears to be a benign
                //  situation, so the way to handle it is to understand that no
                //  more data is going to arrive and go ahead and being our response
                //  to the client.
                //
                break;
            }

            DBAlert2("ReadData: OTRcv error %d state %d", res, epState);
        }
        return;
    }

    SendData(epi);
}

//
//  Recycle:
//
//  This routine shouldn't be necessary, but it is helpful to work around both
//  problems in OpenTransport and bugs in this program.   Basically, whenever an
//  unexpected error occurs which shouldn't be fatal to the program, the EPInfo
//  is queued on the BrokenEP queue.  When recycle is called, once per pass around
//  the event loop, it will attempt to close the associated endpoint and open
//  a new one to replace it using the same EPInfo structure.   This process of
//  closing an errant endpoint and opening a replacement is probably the most
//  reliable way to make sure that this program and OpenTransport can recover
//  from unexpected happenings in a clean manner.
//
static void Recycle()
{
    OTLink*     list = OTLIFOStealList(gBrokenEPs);
    OTLink*     link;
    EPInfo*     epi;

    while ( (link = list) != NULL )
    {
        list = link->fNext;
        epi = OTGetLinkObject(link, EPInfo, link);
        if (!EPClose(epi))
        {
            OTLIFOEnqueue(gBrokenEPs, &epi->link);
            continue;
        }
        OTAtomicClearBit(&epi->stateFlags, kBrokenBit);
        OTAtomicAdd32(-1, &gCntrBrokenEPs);
        EPOpen(epi, OTCloneConfiguration(gCfgMaster));
    }
}

//
//  SendData:
```



```
//
//  For this first, simple version of the OT Virtual Server, we just send
//  a predefined number of bytes from a RAM buffer and then start an orderly
//  release sequence.  The assumption here is that we can send the entire buffer
//  in one send.  Obviously future versions of this sample will have to do
//  the OTSnd() in a more sophisticated way.
//
static void SendData(EPInfo* epi)
{
    OTResult res;

    if (epi->sendBytes == 0)
        return;

    //
    //  Make sure we record that we are starting a send so we don't try to close
    //  the endpoint before a T_MEMORYRELEASED event is returned.
    //
    OTAtomicAdd32(1, &epi->outstandingSends);

    //
    //  In OT 1.1.2 and previous versions, there is a bug with AckSends
    //  which occurs when the same buffer is sent more than once.   In an attempt
    //  to go fast and not allocate memory, TCP may write an IP and TCP header
    //  into the data buffer which is sent.   If the buffer is sent more than once
    //  without being refreshed, the data may be corrupted.   To work around this,
    //  send the data via an OTData structure, using the gather-write mechanism.
    //  The problem does not occur in this code path, and this will not hinder performance.
    //  The problem will be fixed in the next Open Transport release following 1.1.2.
    //
    if (gOTVersion < kOTVersion113)
    {
        struct OTData data;

        data.fNext = NULL;
        data.fData = epi->sendPtr;
        data.fLen  = epi->sendBytes;
        res = OTSnd(epi->erf, &data, kNetbufDataIsOTData, 0);
    }
    else
    {
        res = OTSnd(epi->erf, epi->sendPtr, epi->sendBytes, 0);
    }
    if (res == gReturnDataLength)
    {
        //
        //  The entire buffer was accepted and we can begin the orderly release process.
        //
        OTAtomicAdd32(res, &gCntrTotalBytesSent);
        OTAtomicAdd32(res, &gCntrIntervalBytes);
        epi->sendPtr    = NULL;
        epi->sendBytes  = 0;

        if (gOTVersion < kOTVersion113)
        {
            //
            //  OT 1.1.2 and earlier versions have a bug in OT/TCP where
            //  OT/TCP can lose an inbound orderly release if the orderly releases
            //  cross AND there is no time for the STREAMS service routines to fire.
            //  The workaround is to force the system back to system task time,
            //  and the event loop, before doing the orderly release.  This costs
            //  about 18% in connections/second performance in my testing, but
            //  the workaround is 100% reliable.   Here we set a stateFlag bit
            //  just in case the connection is disconnected while it is waiting
            //  for the orderly release to occur.
            //
            OTAtomicSetBit(&epi->stateFlags, kWaitingBit);
```

```
            OTLIFOEnqueue(gWaitingEPs, &epi->link);
        }
        else
            DoSndOrderlyDisconnect(epi);

        return;
    }

    if (res > 0)
    {
        //
        //  Implied kOTFlowErr since not all data was accepted.
        //  Currently SendData is only invoked from inside the notifier.
        //  If it was called from outside the notifier, it would need race
        //  protection against the T_GODATA happening before the OTSnd returned.
        //
        OTAtomicAdd32(res, &gCntrTotalBytesSent);
        OTAtomicAdd32(res, &gCntrIntervalBytes);
        epi->sendPtr    += res;
        epi->sendBytes  -= res;
    }

    else        // res =< 0
    {
        OTAtomicAdd32(-1, &epi->outstandingSends);
        if (res == kOTFlowErr)
            return;
        if (res == kOTLookErr)
        {
            res = OTLook(epi->erf);
            if (res == T_ORDREL)
            {
                //
                //  Wait to get the T_ORDREL at the notifier and handle it there.
                //  Then we will resume sending.
                //
                return;
            }
            else
            {
                DBAlert1("SendData: OTSnd LOOK error %d", res);
            }
        }
        else
        {
            DBAlert1("SendData OTSnd error %d", res);
        }
    }
}

//
//  StartServer:
//
//  This routine gets memory for EPInfo structures.  It gets one for the listener
//  endpoint and one for each of the acceptor endpoints.
//
static void StartServer()
{
    int     i;
    EPInfo* epi;
    size_t  bytes;

    gCntrEndpts             = 0;
    gCntrIdleEPs            = 0;
    gCntrTotalBrokenEPs     = 0;
    gCntrBrokenEPs          = 0;
    gCntrTotalBrokenEPs     = 0;
```



```
    gCntrTotalConnections   = 0;
    gCntrTotalBytesSent     = 0;
    gIdleEPs->fHead         = NULL;
    gBrokenEPs->fHead       = NULL;
    gWaitingEPs->fHead      = NULL;
    gServerState            = kServerRunning;

    //
    //  Save the current setting of max connections so we don't lose
    //  track of how much memory we will get if someone changes the
    //  dialog while the server is running.
    //
    gMaxConnectionsAllowed  = gMaxConnections;

    //
    //  Get a block of memory to hold all the EPInfo structures.
    //  We use the first one for the listener.
    //  The rest are treated as an array of acceptors.
    //
    bytes = (gMaxConnectionsAllowed + 1) * sizeof(EPInfo);
    epi = (EPInfo*) NewPtr(bytes);
    if (epi == NULL)
    {
        DoAlert("Cannot get enough memory to allocate endpoints, exiting");
        return;
    }
    OTMemzero(epi, bytes);
    gListener   = epi++;
    gAcceptors  = epi;

    //
    //  Open listener, using the tilisten module to make
    //  listen/accept/disconnect processing much simpler.
    //
    if (!EPOpen(gListener, OTCreateConfiguration("tilisten, tcp")))
        return;

    //
    //  Open endpoints to accept inbound connections.
    //  Note that any configuration passed in to OTOpenEndpoint is destroyed,
    //  so we create a master configuration, clone it once for each connection,
    //  which saves a lot of OT processing, and then destroy the master
    //  configuration at the end.
    //
    gCfgMaster = OTCreateConfiguration("tcp");
    if (gCfgMaster == NULL)
    {
        DBAlert("StartServer: OTCreateConfiguration returned NULL");
        return;
    }
    for (epi = gAcceptors, i = 0; i < gMaxConnectionsAllowed; epi++, i++)
    {
        if (!EPOpen(epi, OTCloneConfiguration(gCfgMaster)))
            break;
    }
}

//
//  StopServer:
//
//  This is where the server is shut down, either because the user clicked
//  the stop button, or because the program is exiting (error or quit).
//  The two tricky parts are (1) we can't quit while there are outstanding
//  OTAsyncOpenEndpoint calls (which can't be cancelled, by the way), and
//  (2) we can't close endpoints until that have received all expected
//  T_MEMORYRELEASED events.
//
```

```
static void StopServer()
{
    int         i;
    EPInfo      *epi;
    Boolean     allClosed = true;

    gServerState = kServerShuttingDown;

    //
    //  Since the LIFOs shouldn't be used any longer, we clear them here.
    //
    (void) OTLIFOStealList(gBrokenEPs);
    (void) OTLIFOStealList(gIdleEPs);
    (void) OTLIFOStealList(gWaitingEPs);

    //
    //  Attempt to close all endpoints.
    //  EPClose doesn't mind being called again with epi->erf == NULL.
    //
    for (epi = gListener, i = 0; i < (gMaxConnectionsAllowed + 1); epi++, i++)
    {
        if (!EPClose(epi))
            allClosed = false;
    }

    //
    //  If we successfully deleted all of the endpoints, we can release
    //  the memory and head home for Christmas now...
    //
    if (allClosed)
    {
        DisposPtr((char*)gListener);
        OTDestroyConfiguration(gCfgMaster);
        gListener           = NULL;
        gAcceptors          = NULL;
        gCntrIdleEPs        = 0;
        gCntrBrokenEPs      = 0;
        gCntrConnections    = 0;
        gServerState        = kServerStopped;
    }
}

//
//  TimerInit
//
//  Start up a regular timer to do housekeeping.   Strictly speaking,
//  this isn't necessary, but having a regular heartbeat allows us to
//  detect if we are so busy with network notifier processing that the
//  program's event loop isn't ever firing.   We want to know this so
//  we can at least allow the user to quit the program if they want to.
//
static void TimerInit()
{
    gTimerTask = OTCreateTimerTask(&TimerRun, 0);
    if (gTimerTask == 0)
    {
        DBAlert("TimerInit: OTCreateTimerTask returned 0");
        return;
    }
    OTScheduleTimerTask(gTimerTask, kTimerInterval);
}

//
//  TimerDestroy
//
static void TimerDestroy()
{
```



```
    if (gTimerTask != 0)
    {
        OTCancelTimerTask(gTimerTask);
        OTDestroyTimerTask(gTimerTask);
        gTimerTask = 0;
    }
}

//
//  TimerRun
//
//  Fires every N seconds, no matter how busy the system is.
//  We use this to detect if the program's main event loop is getting no time,
//  in which case we can slow the server down by doing a throttle-back until
//  the event loop can run at least once.  It also is a convenient statistics
//  gathering point.
//
static pascal void TimerRun(void*)
{
    gConnectsPerSecond = (gCntrIntervalConnects / kTimerIntervalInSeconds);
    gKBytesPerSecond = (gCntrIntervalBytes / (kTimerIntervalInSeconds * 1024));
    gEventsPerSecond = (gCntrIntervalEventLoop / kTimerIntervalInSeconds);
    if (gCntrIntervalEventLoop == 0)
        gWaitForEventLoop = true;

    if (gEventsPerSecond > gEventsPerSecondMax)
        gEventsPerSecondMax = gEventsPerSecond;

    if (gAllowNewMax == 0)
    {
        //
        //  Avoid bytes/second data skewing from early buffering by not allowing
        //  the first non-zero measurement to be saved as a max.  We could use an
        //  exponential weighted average instead, but since our timer doesn't fire
        //  very often, the stats take too long to become valid that way.
        //
        if (gConnectsPerSecond > gConnectsPerSecondMax)
            gConnectsPerSecondMax = gConnectsPerSecond;
        if (gKBytesPerSecond > gKBytesPerSecondMax)
            gKBytesPerSecondMax = gKBytesPerSecond;
    }

    if (gConnectsPerSecond > 0)
    {
        if (gAllowNewMax > 0)
            gAllowNewMax--;
    }
    else
        gAllowNewMax = kTimerHitsBeforeAcceptMax;

    gCntrIntervalConnects   = 0;
    gCntrIntervalBytes      = 0;
    gCntrIntervalEventLoop  = 0;
    gDoWindowUpdate         = true;
    gCntrConnections        = gCntrEndpts - gCntrIdleEPs - gCntrBrokenEPs;

    OTScheduleTimerTask(gTimerTask, kTimerInterval);
}


////////////////////////////////////////////////////////////////////////////////
//
//  Macintosh Program Wrapper
//
//  The code from here down deals with the Macintosh environment, events,
//  menus, command keys, etc.   Networking code is in the section above.
//  Since this code is fairly basic, and since this isn't really intended
```

```
//  to be a "sample Macintosh application" (just a sample OpenTransport application)
//  this section isn't heavily commented.   There are much better Macintosh
//  application samples for handling mouse, keyboard, event loops, etc.
//
////////////////////////////////////////////////////////////////////////////////

static void AboutBox()
{
    Alert(kAboutBoxResID, NULL);
}

static Boolean EventDialog(EventRecord* event)
{
    DialogPtr   dp;
    short       item;
    short       itemType;
    Handle      itemHandle;
    Rect        itemRect;

    if (event->modifiers & cmdKey)
    {
        EventKeyDown(event);        // this allows menu commands while dialog is active window
        return false;               // note if I add cut/paste I will have to rework this.
    }
    if ((DialogSelect(event, &dp, &item)) && (dp == gDialogPtr))
    {
        GetDItem(gDialogPtr, item, &itemType, &itemHandle, &itemRect);
        switch (item)
        {
            case kListenerPortDItem:
                GetIText(itemHandle, gListenerPortStr);
                return true;

            case kListenerQueueDepthDItem:
                GetIText(itemHandle, gListenerQueueDepthStr);
                return true;

            case kMaxConnectionsDItem:
                GetIText(itemHandle, gMaxConnectionsStr);
                return true;

            case kReturnDataLengthDItem:
                GetIText(itemHandle, gReturnDataLengthStr);
                return true;

            case kStartStopDItem:
                GetDItem(gDialogPtr, kStartStopDItem, &itemType, &itemHandle, &itemRect);
                if (gServerRunning)
                {
                    StopServer();
                    SetCTitle((ControlHandle)itemHandle, gStartStr);
                    gServerRunning = false;
                }
                else
                {
                    TCPPrefsReset();
                    StartServer();
                    SetCTitle((ControlHandle)itemHandle, gStopStr);
                    gServerRunning = true;
                }
                DrawDialog(gDialogPtr);
                return true;
        }
    }
    return false;
}
```



```
static void TCPPrefsReset()
{
    StringToNum(gListenerPortStr, &gListenerPort);
    StringToNum(gListenerQueueDepthStr, &gListenerQueueDepth);
    StringToNum(gMaxConnectionsStr, &gMaxConnections);
    StringToNum(gReturnDataLengthStr, &gReturnDataLength);
    if (gReturnDataLength > kDataBufSize)
        gReturnDataLength = kDataBufSize;
}

static void TCPPrefsDialog()
{
    short   itemType;
    Handle  itemHandle;
    Rect    itemRect;

    gDialogPtr = GetNewDialog(kTCPPrefsDlogResID, NULL, kInFront);
    SetWTitle(gDialogPtr, "\pTCP Preferences");
    GetDItem(gDialogPtr, kListenerPortDItem, &itemType, &itemHandle, &itemRect);
    SetIText(itemHandle, gListenerPortStr);
    GetDItem(gDialogPtr, kListenerQueueDepthDItem, &itemType, &itemHandle, &itemRect);
    SetIText(itemHandle, gListenerQueueDepthStr);
    GetDItem(gDialogPtr, kMaxConnectionsDItem, &itemType, &itemHandle, &itemRect);
    SetIText(itemHandle, gMaxConnectionsStr);
    GetDItem(gDialogPtr, kReturnDataLengthDItem, &itemType, &itemHandle, &itemRect);
    SetIText(itemHandle, gReturnDataLengthStr);
    GetDItem(gDialogPtr, kStartStopDItem, &itemType, &itemHandle, &itemRect);
    if (gServerRunning)
        SetCTitle((ControlHandle)itemHandle, gStopStr);
    else
        SetCTitle((ControlHandle)itemHandle, gStartStr);
    DrawDialog(gDialogPtr);
}

static void DialogClose()
{
    DisposDialog(gDialogPtr);
    gDialogPtr = NULL;
    TCPPrefsReset();
}

static void MenuDispatch(long menu)
{
    short menuID;
    short cmdID;

    menuID = HiWord(menu);
    cmdID  = LoWord(menu);
    switch(menuID)
    {
        case kAppleMenuResID:
        {
            switch (cmdID)
            {
                case kAppleMenuAbout:
                    AboutBox();
                    break;

                default:
                    break;
            }
            break;
        }

        case kFileMenuResID:
        {
            switch (cmdID)
```

```
            {
                case kFileMenuQuit:
                    gProgramState = kProgramDone;
                    break;

                case kFileMenuOpen:
                    WindowOpen();
                    break;

                case kFileMenuClose:
                    WindowClose();
                    break;

                default:
                    break;
            }
            break;
        }

        case kEditMenuResID:
            break;

        case kServerMenuResID:
        {
            switch (cmdID)
            {
                case kServerMenuTCPPrefs:
                    TCPPrefsDialog();
                    break;

                default:
                    break;
            }
            break;
        }
    }

}

static void EventDrag(WindowPtr wp, Point loc)
{
    Rect dragBounds;

    dragBounds = qd.screenBits.bounds;
    DragWindow(wp, loc, &dragBounds);
}

static void EventGoAway(WindowPtr wp, Point loc)
{
    if (TrackGoAway(wp, loc))
    {
        if (wp == gWindowPtr)
            WindowClose();
        else if (wp == gDialogPtr)
            DialogClose();
    }
}

static void EventMouseDown(EventRecord* event)
{
    short       part;
    WindowPtr   wp;
    long        menu;

    part = FindWindow(event->where, &wp);
    switch (part)
    {
```



```
        case inMenuBar:
            menu = MenuSelect(event->where);
            HiliteMenu(0);
            MenuDispatch(menu);
            break;

        case inDrag:
            EventDrag(wp, event->where);
            break;

        case inGoAway:
            EventGoAway(wp, event->where);
            break;

        case inContent:
            SelectWindow(wp);
            break;

        case inGrow:        // no grow box
        case inZoomIn:      // no zoom box
        case inZoomOut:     // no zoom box
        case inSysWindow:
        case inDesk:
        default:
            break;
    }
}

static void EventKeyDown(EventRecord* event)
{
    char        c;
    long        menu;

    c = event->message & charCodeMask;
    if (event->modifiers & cmdKey)
    {
        // cmd key
        menu = MenuKey(c);
        HiliteMenu(0);
        if (menu != 0)
            MenuDispatch(menu);
    }
    else
    {
        // normal keystroke
    }
}

static void EventLoop()
{
    EventRecord event;

    while ((gProgramState == kProgramRunning) || (gServerState != kServerStopped))
    {
        OTAtomicAdd32(1, &gCntrIntervalEventLoop);
        if (WaitNextEvent(everyEvent, &event, gSleepTicks, 0))
        {
            if ((gDialogPtr != NULL) && (IsDialogEvent(&event)))
            {
                if (EventDialog(&event))
                    continue;
            }
            switch (event.what)
            {
                case keyDown:
                    EventKeyDown(&event);
                    break;
```

```
                case mouseDown:
                    EventMouseDown(&event);
                    break;

                case updateEvt:
                    // redraw window now
                    break;

                case activateEvt:
                    // activate or deactivate window controls
                    break;

                case mouseUp:
                case keyUp:
                case autoKey:
                case diskEvt:
                case app4Evt:
                default:
                    break;
            }
        }

        if ((gProgramState == kProgramRunning) && (gServerState == kServerRunning))
        {
            NetEventLoop();
        }
        else if (((gProgramState == kProgramRunning) && (gServerState == kServerShuttingDown)) |
                 ((gProgramState != kProgramRunning) && (gServerState != kServerStopped)))
        {
            StopServer();
        }
        WindowUpdate();
    }
}

static void WindowClose()
{
    if (gWindowPtr == NULL)
        return;
    DisposeWindow(gWindowPtr);
    gWindowPtr = NULL;
}

static void WindowOpen()
{
    if (gWindowPtr != NULL)
        return;
    gWindowPtr = GetNewWindow(kWindowResID, NULL, kInFront);
    SetWTitle(gWindowPtr, "\pOTVirtualServer");
}

static void WindowUpdate()
{
    char gStrBuf[128];
    int len;

    if (gWindowPtr == NULL)
        return;

    if (gDoWindowUpdate == false)
        return;
    gDoWindowUpdate = false;

    gCntrConnections = gCntrEndpts - gCntrIdleEPs - gCntrBrokenEPs;

    SetPort(gWindowPtr);
```



```
    EraseRgn(gWindowPtr->visRgn);

    MoveTo(20, 20);
    sprintf(gStrBuf, "EPs: total %d idle %d", gCntrEndpts, gCntrIdleEPs);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 40);
    sprintf(gStrBuf, "Connects: current %d total %d", gCntrConnections, gCntrTotalConnections);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 60);
    sprintf(gStrBuf, "KBytes sent %d", (gCntrTotalBytesSent / 1024));
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 80);
    sprintf(gStrBuf, "Conn/sec: current %d max %d", gConnectsPerSecond, gConnectsPerSecondMax);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 100);
    sprintf(gStrBuf, "KBy/sec: current %d max %d", gKBytesPerSecond, gKBytesPerSecondMax);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 120);
    sprintf(gStrBuf, "Events/sec: %d/%d", gEventsPerSecond, gEventsPerSecondMax);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 140);
    sprintf(gStrBuf, "Running at %d%% of capacity.",
            (100 - ((100 * gEventsPerSecond)/gEventsPerSecondMax)));
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 160);
    sprintf(gStrBuf, "Broken EPs: %d total: %d.", gCntrBrokenEPs, gCntrTotalBrokenEPs);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 180);
    sprintf(gStrBuf, "OTVersion 0x%08x", gOTVersion);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

}

static void SetupMenus()
{
    MenuHandle mh;
    mh = GetMenu(kAppleMenuResID);
    AddResMenu( mh, 'DRVR' );           /* Add DA list */
    InsertMenu(mh, 0);
    mh = GetMenu(kFileMenuResID);
    InsertMenu(mh, 0);
    mh = GetMenu(kEditMenuResID);
    InsertMenu(mh, 0);
    mh = GetMenu(kServerMenuResID);
    InsertMenu(mh, 0);
    DrawMenuBar();
}

static void C2PStr(char* cstr, Str255 pstr)
{
```

```
    //
    //  Converts a C string to a Pascal string.
    //  Truncates the string if longer than 254 bytes.
    //
    int i, j;

    i = strlen(cstr);
    if (i > 254)
        i = 254;
    pstr[0] = i;
    for (j = 1; j <= i; j++)
        pstr[j] = cstr[j-1];
}

static void P2CStr(Str255 pstr, char* cstr)
{
    int i;

    for (i = 0; i < pstr[0]; i++)
        cstr[i] = pstr[i+1];
    cstr[i] = 0;
}

static void AlertExit(char* err)
{
    Str255 pErr;

    C2PStr(err, pErr);
    ParamText(pErr, NULL, NULL, NULL);
    Alert(kAlertExitResID, NULL);
    ExitToShell();
}

static void MacInitROM()
{
    MaxApplZone();
    MoreMasters();
    InitGraf(&qd.thePort);
    InitCursor();
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);
    FlushEvents(everyEvent, 0);
}

static void MacInit()
{
    MacInitROM();
    WindowOpen();
    SetupMenus();
}

static void MiscInit()
{
    //  Initialize the temporary data buffer so it isn't all zeros.

    int i;
    unsigned char x = 0;

    for (i = 0; i < kDataBufSize; i++)
        gDataBuf[i] = x++;
}

void main()
{
```



```
    MacInit();
    NetInit();
    MiscInit();
    EventLoop();
    NetShutdown();
    if (gProgramState == kProgramError)
        AlertExit(gProgramErr);
}
```