AppleTalk Remote Access Developer's Toolkit

R0128LL/A

Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014
(408) 996-1010
TLX 171-576

To reorder products, please call:
1-800-282-2732 (in the United States)
1-800-637-0029 (in Canada)
1-408-562-3910 (International)

AppleTalk Remote Access Application Programming Interface (API) External Reference Specifications

AppleTalk® Remote Access Application Programming Interface (API) External Reference Specifications

The AppleTalk Remote Access **API** provides an application programming interface to the Remote Access Manager. It supports calls to load and unload the Remote Access Manager (RAM), create and terminate connections, retrieve the current RAM status, and to determine if a specified network address is local or remote. Optionally, the AppleTalk Remote Access **API** can display a user interface showing the process of a connection. The parameters for the connect call can be a connection document, created with the Remote Access application, or all the parameters of the connection can be specified out right. These specifications also include how to tell if AppleTalk Remote Access is installed, and how to deal with the serial port when AppleTalk Remote Access is in answer mode.

***NOTE**: Although every attempt has been made to verify the accuracy of the information presented, this document may contain errors and is subject to change.*

Gestalts

When fully installed, Remote Access defines several new gestalt selectors. Below are the new defines and explanation of their use.

To see if serial port arbitration is installed, call _Gestalt using these defines.

```
#define gestaltArbitorAttr                          'arb '
#define gestaltSerialArbitrationExists              0
```

For example:

```
OSErr io;
// check to see if serial port arbitrating is installed...
long attribs;
io = Gestalt(gestaltArbitorAttr, &attribs);
if ( ((io == noErr) && (attribs & (1 << gestaltSerialArbitrationExists))) )
   {
   // have serial port arbitration
   }
else
   {
   // no serial port arbitration
   }
```

More information on serial port arbitration is discussed in a section below.

To see if Remote Access Connection Interface is available use these defines:

```
#define gestaltRemoteAccessAttr                     'strm'
#define gestaltRemoteAccessExists                   0
```

To check if Remote Access support is available in the Alias Manager use these defines:

```
#define gestaltAliasMgrAttr                         'alis'      // as defined in GestaltEqu.h
#define gestaltAliasMgrSupportsRemoteAppletalk      1
```

Serial Port Arbitration

When installed Remote Access provides serial port arbitration throught the Serial Port Arbitrator tool. All serial drivers registered with the Communications Resource Manager are arbitrated by the Serial Port Arbitrator.

To check to see if the Serial Port Arbitrator is installed, check the `gestaltSerialArbitrationExists` flag of the `gestaltArbitorAttr` Gestalt selector (see "Gestalts" section for these defines and an example of this call).

If serial port arbitration is present, call OpenDriver when you want to use a serial port. If the driver requested is not open, OpenDriver will return a result of noErr and the reference number of the driver. If the driver requested is open (in use), OpenDriver will return the error `portInUse`. When you are finished using the driver, call CloseDriver. OpenDriver and CloseDriver calls should always be balanced, although the Serial Port Arbitrator will protect against multiple open/close calls.

If serial port arbitration is not present, do not use OpenDriver to determine if the driver is open. It will return you a result of noErr and the reference number, allowing you access to a driver that another application is using. To determine if the serial driver requested is open by another application, you must walk the DCE (Device Control Entry) unit table (see Device Driver chapter of Inside Macintosh vol II).

It is important to use the new method if serial port arbitration is available. Remote Access, when set up for answering calls, is passively using a particular serial driver. If you use the new method, the Serial Port Arbitrator will give up the passive claim and allow your OpenDriver call to return noErr. Later, when you call CloseDriver, Remote Access will again passively claim the port and setup the modem for answering.

Below is a code example in C illustrating how to use serial port drivers under the new and old methods:

```
Boolean HaveSerialPortArbitration()
{
    /*  check to see if serial port arbitrating is installed,
        return true if so.
    */

    OSErr io;
    long attribs;
    io = Gestalt(gestaltArbitorAttr, &attribs);
    return ((io == noErr) && (attribs & (1 << gestaltSerialArbitrationExists)) );
} // HaveSerialPortArbitration
```

```
Boolean CanOpenDriver(unsigned char *driverName)
{
    /*  walks the unit table looking for a match for driverName,
        if found, check to see if the driver is open. return
        false if so.
    */

    Boolean canOpen = false;
    Boolean match = false;
    short index = 0;
    short count;
    DCtlHandle dceHndl;
    unsigned char *namePtr;

    // number of entries in unit table...
    count = *(short*)UnitNtryCnt;

    while ( !match && index < count )
        {
        // get handle to this DCE...
        dceHndl = (DCtlHandle)(*(Handle)((*(Handle)UTableBase) + (index*4) ));

        if ( dceHndl )
            {
            // see if ram based (test bit 6)...
            if ((*dceHndl)->dCtlFlags & 0x40)
                {
                // in ram, so we have a handle...
                namePtr = (*(Handle)((*dceHndl)->dCtlDriver)) + 18;
                }
            else
                {
                // in rom, so we have a pointer...
                namePtr = ((*dceHndl)->dCtlDriver) + 18;
                }

            // compare name: case insensitive, diacritical sensitive...
            if ( RelString((const Str255)driverName,(const Str255)namePtr,false,true) == 0 )
                {
                match = true;
                // see if drvr is open (test bit 5)...
                canOpen != ((*dceHndl)->dCtlFlags & 0x20);
                }
            }
        ++index;    // look at next drvr in unit table
        }

        return canOpen;
} // CanOpenDriver
```

```
void UseSerialPort()
{
    /*  illustrate the new way and old way of testing whether
        a serial driver is open or closed.
    */

    short refNum;
    OSErr io;

    if ( HaveSerialPortArbitration() )
        {
        // have serial port arbitration, use new method...

        io = OpenDriver("\p.aout",&refNum);

        if ( io == portInUse )
            {
            // port is in use by another application/etc, we can't use it!
            }
        else
            {
            // use the serial driver

            // close the driver when through...
            io = CloseDriver(refNum);
            }
        }
    else
        {
        // no serial port arbitration, use old method...

        if ( CanOpenDriver("\p.aout") )
            {
            io = OpenDriver("\p.aout",&refNum);

            // use the serial driver

            // close the driver when through...
            io = CloseDriver(refNum);
            }
        }

} // UseSerialPort
```

AppleTalk Remote Access API

Common Parameters

The TRemoteAccessParamBlock is a union of all of the available AppleTalk Remote Access API commands. The TRemoteAccessParmHeader is a struct which consists of a DControlParamHeader followed by a DExtendedParam which is followed by a DRemoteAccessParmHeader. The extendedCode is used to specify the AppleTalk Remote Access API command wanted. The resultStrPtr field returns a Pascal string to indicate what error occurred. If you are not interested in the string, set this field to nil. If you do pass a pointer however, it must point to a buffer of at least 256 bytes in length. If the result of the call happens to be noErr, then the length byte of the string will be zero. Since this version of Remote Access only deals with the user port, the parameter portGlobalsPtr should always be set to zero. The csCode field should normally set to RAM_EXTENDED_CALL and the extendedType is set to REMOTEACCESSNAME. These constants are defined in RemoteAccessInterface.h.

```
#define DControlParamHeader \
      QElem      *qLink;           // next queue entry \
      short      qType;            // queue type \
      short      ioTrap;           // routine trap \
      Ptr        ioCmdAddr;        // routine address \
      ProcPtr    ioCompletion;     // completion routine \
      OSErr      ioResult;         // result code \
      long       userData;         // for use by the user \
      short      unused;           // unused field \
      short      ioRefNum;         // driver reference number \
      short      csCode;           // normally set to RAM_EXTENDED_CALL
                                   // for AppleTalk Remote Access API  calls

#define DExtendedParam \
      DControlParamHeader \
      Ptr        hReserved1; \
      Ptr        hReserved2; \
      Ptr        resultStrPtr; \   // set to zero if result string is unwanted
      Ptr        extendedType;     // pointer to identifier string, normally set to
                                   // REMOTEACCESSNAME for AppleTalk Remote Access API  calls

#define DRemoteAccessParmHeader \
      DExtendedParam \
      short      extendedCode;     // for use by extended call proc \
      Ptr        portGlobalsPtr;   // pointer to globals for this port (0=userport) \

struct TRemoteAccessParmHeader
{
      DRemoteAccessParmHeader
};
typedef struct TRemoteAccessParmHeader TRemoteAccessParmHeader;

union TRemoteAccessParamBlock
{
      TRemoteAccessParmHeader       HDR;          // header pb
      TRemoteAccessParmHeader       LOAD;         // load pb
      TRemoteAccessParmHeader        UNLOAD;        // unload pb
      TRemoteAccessConnectParam      CONNECT;       // connect pb
      TRemoteAccessDisconnectParam   DISCONNECT;    // disconnect pb
      TRemoteAccessStatusParam       STATUS;        // get current status
      TRemoteAccessIsRemoteParms     ISREMOTE;      // check network address location
      TRemoteAccessPasswordMunger    MUNGEPW;       // run password through munger
      TRemoteAccessGetCodeHooks      CODEHOOKS;     // get internal code hooks
};
typedef union TRemoteAccessParamBlock TRemoteAccessParamBlock;
typedef TRemoteAccessParamBlock *TPRemoteAccessParamBlock;
```

Load

The load command is used ensure that the Remote Access Manager is loaded into memory and must be used before making a Connect call . It uses the standard TRemoteAccessParmHeader as the parameter block. The example code below shows how it works. To use the MungePW command, it is not necessary to use the load command first.

```
#include "RemoteAccessInterface.h"
void LoadRemoteAccess()
{
  TRemoteAccessParmHeader loadPB;

  loadPB.LOAD.csCode = RAM_EXTENDED_CALL;             // extended call
  loadPB.LOAD.resultStrPtr = nil;                     // result string
  loadPB.LOAD.extendedType = REMOTEACCESSNAME;        // to remote access
  loadPB.LOAD.extendedCode = CmdRemoteAccess_Load;    // try to load
  PBRemoteAccess(&loadPB, false);                     // issue sync call
  if (loadPB.LOAD.ioResult)
    ShowError(loadPB.LOAD.ioResult);
}
```

Unload

The unload command is used release the Remote Access Manager and free its memory. It uses the standard TRemoteAccessParmHeader as the parameter block and can be issued immediately after making a Connect call. This allows the Remote Access Manager to be unloaded as soon as an active connection is terminated with a disconnect, if no other clients have loaded Remote Access. Below is an example unload call.

```
#include "RemoteAccessInterface.h"
void UnloadRemoteAccess()
{
  TRemoteAccessParmHeader loadPB;

  // unload the code (will not actually go away till this connection is done)
  loadPB.UNLOAD.csCode = RAM_EXTENDED_CALL;             // extended call
  loadPB.UNLOAD.resultStrPtr = nil;                     // result string
  loadPB.UNLOAD.extendedType = REMOTEACCESSNAME;        // to remote access
  loadPB.UNLOAD.extendedCode = CmdRemoteAccess_Unload;  // try to unload
  PBRemoteAccess(&loadPB, false);                       // issue sync call
  if (loadPB.UNLOAD.ioResult)
    ShowError(loadPB.UNLOAD.ioResult);
}
```

Connect

This call is used to initiate an outgoing connection. When you are connected in this mode you will still retain access to your current network. Network numbers are re-mapped in a limited way in order to solve problems of network number conflicts between the two machines being directly connected, thus ensuring they will always be accessible to each other. Unfortunately, it is not possible to solve all of the other possible conflicts due to the limited number of network numbers available. In order to provide a method of ensuring access to all networks on the destination network a guaranteedAccess method is available. When connected in this mode, you will lose access to all services beyond those on the same single network number that the calling machine belongs to. In order to notify clients of the AppleTalk stack that a network will no longer be reachable we have created a new AppleTalk Transition Queue event. (See Network Transition Events later in this document.) When connecting the client passes in a TRAConnectInfoTemplate, or the FSSpec of a document which contains the connect parameters. The connect parameter block contains the optionFlags field which specifies the connect options. The flags are shown below:

```
// connect option flags
#define kNSCanInteract       0x00000001   // User interaction (password prompt) is OK
#define kNSShowStatus        0x00000002   // show the status of the connect or disconnect call
#define kNSConnectDocument   0x00000004   // connect using the specified document using FSSpec
#define kNSPassWordSet       0x00000010   // use the specified password field when connecting
                                          // by document
```

The kNSCanInteract flag allows interaction with the user to get the password if necessary. The kNSShowStatus flag enables the connection status display. The display is modal dialog which updates with new messages as the connection progresses. When the kNSConnectDocument flag is set, the AppleTalk Remote Access API will use the specified document for the connect parameters of the TRAConnectInfoTemplate. The document is specified by the FSSpec record which contains vRefNum (volume reference number), parID (directory ID), and name (pointer to Pascal style string containing the document name). No other parameters need to be supplied. The kNSPassWordSet flag overrides the saved password when connecting by document or by PB and forces AppleTalk Remote Access API to use the passWord field in cleartext. If the kNSPassWordSet flag is clear and the passwordSaved flag is set, then the client must supply a munged password.

In the case that a client is connecting by PB, the fields in the connectInfo record need to be supplied. Within this record is a version which is used to check for compatibility (currently set to 1). The ltType parameter specifies the type of the link tool that will be used in this connection. You specify the length and point to the address used in connecting by setting up addressInfoLength and addressInfoPtr. The ltSpecificTemplatePtr is expected to point to the template of the link tool specific parms. An example of link tool specific parms might be items such as the serial port reference that is used in the MNP Link Tool. A userName is passed in that indicates the name of the user logging in. A passWord is specified if the user is not logging in as a guest. If the user wants to be a guest, the guestLogin flag is set. The connectReminderTimer is used if the caller wants to be reminded that a connection is in progress. This field is set to the number of seconds between reminders, and can be set to zero if no reminders are wanted. If a connectReminderTimer is set you must set the connectOKWaitTimer that indicates how long the reminder dialog will wait for OK to be hit before disconnecting.

```
struct TRAConnectInfoTemplate
{
  unsigned long version;                      // version of this format
  unsigned long ltType;                       // Link Tool type
  long addressInfoLength;                     // length of the address information
  Ptr addressInfoPtr;                         // pointer to connect address info
  long ltSpecificTemplateLength;              // length of the ltspecific information
  Ptr ltSpecificTemplatePtr;                  // pointer to link tool specific params
  unsigned char passWord[PASSWORDBUFSIZE];    // user password
  unsigned char userName[USERNAMESIZE];       // user name
  unsigned long connectReminderTimer;         // value for connection reminder in seconds
  unsigned long connectOKWaitTimer;           // how long to wait for OK on reminder timer
  Boolean guestLogin;                         // try to log in as a guest
  Boolean passwordSaved;                      // set if password is saved
  Boolean guaranteedAccess;                   // flag to guarantee access to servers internet
};
typedef struct TRAConnectInfoTemplate TRAConnectInfoTemplate;
typedef TRAConnectInfoTemplate *TPRAConnectInfoTemplate;

struct TRemoteAccessConnectParam
{
  DRemoteAccessParmHeader
  TRAConnectInfoTemplate connectInfo;         // The connection information template
  unsigned long optionFlags;                  // bit mapped connect option flags
  FSSpec fileInfo;                            // file info for connect document
};
typedef struct TRemoteAccessConnectParam TRemoteAccessConnectParam;
```

The following is an example connection procedure:

```
#include "RemoteAccessInterface.h"
void DoConnect()
{
  TRemoteAccessConnectParam pb;
  Str255 PathName = "MyHardDisk:Remote Access:Connect Document";

  LoadRemoteAccess();                                   // Get the Remote Access Manager
                                                        // loaded
  pb.CONNECT.csCode = RAM_EXTENDED_CALL;                // extended call
  pb.CONNECT.resultStrPtr = nil;                        // don't want result strings
  pb.CONNECT.extendedType = REMOTEACCESSNAME;           // to Remote Access
  pb.CONNECT.extendedCode = CmdRemoteAccess_DoConnect;  // connect command
  pb.CONNECT.portGlobalsPtr = nil;                      // use the user port
  pb.CONNECT.fileInfo.vRefNum = 0;                      // Use the full pathname
  pb.CONNECT.fileInfo.parID = 0;
  CopyPStr(&PathName,&pb.CONNECT.fileInfo.name);        // copy the string to fileInfo.name

  // Ask for password if needed, use connection document, & show connection status
  pb.CONNECT.optionFlags = kNSCanInteract | kNSConnectDocument | kNSShowStatus;
  PBRemoteAccess(&pb, false);                           // issue sync call
  if (pb.CONNECT.ioResult)
    ShowError(pb.CONNECT.ioResult);                     // Do Error reporting and recovery
UnloadRemoteAccess();                                   // Unload when disconnected.
}
```

Disconnect

The disconnect command is used to terminate an existing session or cancel one that is being created. If you only want to disconnect a session that was connected with a specific parameter block you can do so by setting a pointer to the parameter block used to issue the connect in abortOnlyThisPB. If you want to disconnect a connection created by anyone, you set the abortOnlyThisPB field to zero. If you are disconnecting an outgoing call, you pass zero in portGlobalsPtr. Disconnecting ports other than the userport, is not supported in this version. You should always set disconnectin to zero. The option kNSShowStatus will cause the AppleTalk Remote Access API to display the status dialog during the disconnect.

```
#define kNumWarnEntriesMax  5                   // number of entries in warn array
struct TRemoteAccessDisconnectParam
{
  DRemoteAccessParmHeader
  unsigned long disconnectin;                   // Note: Set this parameter to 0
  TPRemoteAccessParamBlock abortOnlyThisPB;     // only abort a connection opened by this pb
  unsigned long warnArr[kNumWarnEntriesMax];    // set warn times here in seconds (zero all if
                                                // no warnings)
  unsigned long optionFlags;                    // bit mapped connect option flags
};
typedef struct TRemoteAccessDisconnectParam TRemoteAccessDisconnectParam;
```

The following is an example of a simple disconnect procedure. It will disconnect any existing active connection.

```
#include "RemoteAccessInterface.h"
void DoDisconnect()
{
  TRemoteAccessDisconnectParam pb;

  // set up the Remote Access PB
  pb.DISCONNECT.csCode = RAM_EXTENDED_CALL;               // extended call
  pb.DISCONNECT.resultStrPtr = nil;                       // don't want result strings
  pb.DISCONNECT.extendedType = REMOTEACCESSNAME;          // to Remote Access
  pb.DISCONNECT.extendedCode = CmdRemoteAccess_Disconnect;  // disconnect command
  pb.DISCONNECT.portGlobalsPtr = nil;                     // user port
  pb.DISCONNECT.abortOnlyThisPB = nil;                    // don't get tied to any specific pb
  pb.DISCONNECT.optionFlags = 0| kNSShowStatus;           // show status while disconnecting
  PBRemoteAccess(&pb, false);                             // issue sync call
  if (pb.DISCONNECT.ioResult)
    ShowError(pb.DISCONNECT.ioResult);                    // Do Error reporting and recovery
}
```

IsRemote

The "IsRemote" command is used to determine if a network address is remote or local. If the network is remote, the call will optionally return the information necessary to make the connection to the remote network. The parameter theAddress contains the network address to be checked. The format of theAddress is the same as for the struct AddrBlock as defined in AppleTalk.h:

Bytes 3 & 2 (High Word): Network Number
Byte 1: Node Number
Byte 0 (Low Byte): Socket Number

The optionFlags parameter is used for getting, or disposing, connection information. The following flags are defined:

```
#define ctlir_getConnectInfo 0x01          // will get connect info if address remote
#define ctlir_disposeConnectInfo 0x02      // will dispose info in connectInfoPtr properly
```

If the ctlir_getConnectInfo flag is set, and the network address is remote, the information necessary to create the remote connection is returned. If the ctlir_disposeConnectInfo flag is set, the connect information structure pointed to by connectInfoPtr, is disposed of properly.

The locationIsRemoteFlag parameter is a flag that is returned true if the network address is remote. The connectInfoLength parameter indicates the length of the connect information. The connectInfoPtr is a pointer to the connection information for connecting with the remote address. These values are returned when the network address is remote, and the ctlir_getConnectInfo flag is set.

```
struct TRemoteAccessIsRemoteParms
{
  DRemoteAccessParmHeader
  long theAddress;                          // address that is to be checked
  unsigned long optionFlags;                // Set to ctlir_getConnectInfo or
                                            // ctlir_disposeConnectInfo, if zero only checks
                                            // theAddress
  Boolean locationIsRemoteFlag;             // returns true if address is remote
  long connectInfoLength;                   // length of the following data
  TPRAConnectInfoTemplate connectInfoPtr;   // The connection information template pointer
};
typedef struct TRemoteAccessIsRemoteParms TRemoteAccessIsRemoteParms;
```

Status

The status command is used to obtain information about Remote Access. The information you can obtain is how long a connection has been active, how much time remains in the connection, the name of the user that made an answering connection, the name of the computer you are connected to on a calling connection, and the last message that was posted. The statusBits parameter is used to determine if a connection is active, starting up, in the process of tearing down, if the connection is an answering or calling connection, if the computer is enabled to receive answer calls or if a disconnect is in progress. The following flags are defined:

```
// bits passed back in statusBits
#define CctlConnected           0x00000001    // set when connected
#define CctlAnswerEnable        0x00000004    // set when we are set to answer calls
#define CctlServerMode          0x00000008    // set for answer mode, clear for call mode
#define CctlConnectionAborting  0x00000010    // connection is being torn down
#define CctlConnectInProg       0x00000020    // set when connection in progress or fully
                                              // connected
#define CctlDisconnectInStarted 0x00008000    // somebody has started a disconnectIn
#define CctlMultiNodeReady      0x80000000    // shows if we currently have a multinode
                                              // address to enable answer mode.
```

The following struct is used when making a status call:

```
struct TRemoteAccessStatusParam
{
  DRemoteAccessParmHeader
  unsigned long statusBits;               // bits for current status
  unsigned long timeConnected;            // number of seconds we have been connected
  unsigned long  timeLeft;                // number of seconds remaining in connection
                                          // (0xffffffff infinite)
  unsigned char *userNamePtr;             // returns user name, expects pointer to buffer
                                          // of USERNAMESIZE if non nil
  unsigned char  *connectedToNamePtr;     // returns name of where we connected to,
                                          // expects pointer to buffer of USERNAMESIZE if
                                          // non nil
  TPRemoteAccessParamBlock connectedByParamPtr; // a pointer to the parameter block
                                          // "initiating" the connection if we are
                                          // connected
  TPRemoteAccessParamBlock statusConnectedByParamPtr; // a pointer to the parameter block
                                          // "initiating" the connection when status was
                                          // posted
  unsigned char  *theLastStatusMsgPtr;    // expects pointer to buffer of size
                                          // MAXSTATUSMSGSIZE
  unsigned char  *statusUserNamePtr;      // pointer to buffer of size USERNAMESIZE
  long  statuslttype;                     // link tool type
  long  statusmsgOptionFlags;             // classification of message type
  long  statusMsgNum;                     // specific message number
  long  statusMsgSeqNum;                  // pass in zero if always want status, otherwise
                                          // use last value, if status is new, new number
                                          // is returned
  unsigned long  userSignature;           // signature of port creator
  unsigned long  userRefCon;              // refcon of port creator
};
typedef struct TRemoteAccessStatusParam TRemoteAccessStatusParam;
```

An example status call that determines the state Remote Access is in.

```
#include "RemoteAccessInterface.h"
void GetStatus()
{
  TRemoteAccessStatusParam pb;
  Str255 UserName,connectedTo,lastMessage;
  long lastSeqNum,statusBits;

  pb.STATUS.csCode = RAM_EXTENDED_CALL;                    // extended call
  pb.STATUS.resultStrPtr = nil;                            // put results here
  pb.STATUS.portGlobalsPtr = nil;                          // do UserPort
  pb.STATUS.extendedType = REMOTEACCESSNAME;               // to Netshare
  pb.STATUS.extendedCode = CmdRemoteAccess_Status;         // status command
  pb.STATUS.userNamePtr = &UserName;
  pb.STATUS.connectedToNamePtr = &connectedTo;
  pb.STATUS.theLastStatusMsgPtr = &lastMessage;
  pb.STATUS.statusUserNamePtr = nil;
  pb.STATUS.statusMsgSeqNum = 0;
  PBRemoteAccess(&pb, false);
  if (pb.STATUS.ioResult)
    ShowError(pb.STATUS.ioResult);                         // Do Error reporting and recovery
  else
  {
    // now decode the flag bits into words
    statusBits = pb.STATUS.statusBits;
    if (statusBits & CctlServerMode)
      printf("Answer connection\n");
    if (statusBits & CctlConnected)
      printf("Calling connection\n");
    if (statusBits & CctlConnectionAborting)
      printf("Cancel in progress\n");
    if (statusBits & CctlAnswerEnable)
      printf("Waiting for incoming call\n");
    if (statusBits & CctlConnectInProg)
      printf("Connection in progress\n");
  }
}
```

MungePW

The MungePW command is used to encrypt a password to be stored in a document. Normally, when connecting by document, it is not necessary to use this command, since the password in a document is stored in encrypted format. It uses a struct TRemoteAccessPasswordMunger with the inputs username pointer and password pointer. The reserved field should always be set to zero. The munged password is return in the data buffer pointed to by passWordPtr. It is not necessary to call the Load command before using MungePW. The maximum username and password lengths are defined in the RemoteAccessInterface.h header file.

```
struct TRemoteAccessPasswordMunger
{
  DRemoteAccessParmHeader
  unsigned char *userNamePtr;                          // pointer to username string
  unsigned char *passWordPtr;                          // user password
  unsigned short reserved;                             // must set to zero
};
typedef struct TRemoteAccessPasswordMunger TRemoteAccessPasswordMunger;
```

Below is an example routine that calls and gets the password in *passWordPtr munged.

```
#include "RemoteAccessInterface.h"
void MungePassword()
{
  TRemoteAccessPasswordMunger MungePB;
  Str255 UserName = "John Doe";
  Str255 password = "thispass";

  MungePB.MUNGEPW.csCode = RAM_EXTENDED_CALL;             // extended call
  MungePB.MUNGEPW.resultStrPtr = nil;                     // result string
  MungePB.MUNGEPW.extendedType = REMOTEACCESSNAME;        // to remote access
  MungePB.MUNGEPW.extendedCode = CmdRemoteAccess_PassWordMunger;
  MungePB.MUNGEPW.userNamePtr = &UserName;
  MungePB.MUNGEPW.passWordPtr = &password;
  PBRemoteAccess(&MungePB, false);                        // issue sync call
                                                          // and encrypted the eight bytes in
                                                          // password


  if (MungePB.MUNGEPW.ioResult)
    ShowError(MungePB.MUNGEPW.ioResult);
  }
```

GetCodeHooks

The GetCodeHooks command is used to return a pointer to the remapper procedure. This routine can then be called to do special remappings for applications are passing network addresses as part of their data. The call can be made with the clients network number and the node number and this routine will return the remapped equivalents.

```
struct TRemoteAccessGetCodeHooks
{
  DRemoteAccessParmHeader
  RemmaperProcPtr remapperProc;                           // quick vector to remapper code
};
typedef struct TRemoteAccessGetCodeHooks TRemoteAccessGetCodeHooks;
```

The routine returned by this call is defined as follows:

```
pascal void DoRemapper(unsigned long whereNet, unsigned long incomingFlag, unsigned long
sourceSwapFlag, unsigned short *theNet, unsigned char *theNode)
```

whereNet-> This value has the net where this packet just came from or is going to, it is needed to determine if any remapping should even take place.

incomingFlag-> Set to true if data is incoming, false if data is outgoing.

sourceSwapFlag-> Set to true if a source style swap is to be used. A source style swap means that we do remappings based on the address being a source address. If this flag is false, destination style swaps are done.

theNet-> Pointer to unsigned short containing the net to be remapped.

theNode-> Pointer to unsigned char containing the node to be remapped.

Network Transition Events

Network transition events are generated by Remote Access to inform interested clients that network connectivity has changed. The type of change is indicated by the newConnectivity flag. If this flag is true, new connectivity is being added (i.e. a connection to a new internet has taken place). In this case, all network addresses will be returned as reachable. If the newConnectivity flag is false, certain networks are no longer reachable. Since Remote Access is connection based and internally functions much like a router it has knowledge of where a specific network exists. Remote Access can take advantage of that knowledge during a disconnect to inform AppleTalk clients that a network is no longer reachable. This information can be used by the AppleTalk client to age out connections immediately rather than waiting a potentially long period of time before discovering that the other end is no longer reachable.

When Remote Access is disconnecting, it will generate a "Network Transition Event (theEvent=5)" through the AppleTalk transition queue. A client upon receiving such a message can ask Remote Access (through a network validate hook passed to the client) if a specific network is still reachable. If the network is still reachable, true will be returned. A client can then continue to check other networks he is interested in until he has learned the status of each of them. After a client is finished checking his networks he returns to Remote Access where the next AppleTalk transition queue client is called.

Since the "Network Transition Event" is transitional, it is important to realize that the information that the network validate hook returns is only valid if a client has just been called as a result of a transition. In other words, a client can only validate networks when it has been called to handle a "Network Transition Event". It is also important to realize that the "Network Transition Event" can be called as the result of an interrupt, so a client should obey all of the normal conventions involved at being called at this time (i.e. don't ask for memory from the memory manager, etc.).

The following information assumes you have already installed yourself into the AppleTalk transition queue.

ATTransNetworkTransition

The ATTransNetworkTransition event will be generated whenever a network transition occurs. You will be passed the following information using C calling conventions:

```
ClientTransitionHandler(long theEvent, Ptr aqe, TNetworkTransition *thetrans);

theEvent        <---        will be set to ATTransNetworkTransition
aqe             <---        points to transition task struct
thetrans        <---        points to the TNetworkTransition struct
```

The TNetworkTransition struct passed to you is defined as:

```
typedef struct TNetworkTransition
{
  uPtr private;                  // pointer used internally by 976
  ProcPtr netValidProc;          // pointer to the network valid proc
  Boolean newConnectivity;       // true=new connectivity, false=loss of connectivity
} TNetworkTransition;
```

To check a network number for validity the client uses the netValidProc to call Remote Access. This call is defined as follows:

```
long netValidProc(TNetworkTransition *thetrans, unsigned long theAddress);

thetrans        --->        pass in the TNetworkTransition struct given to you when your
                            transition handler was called.
theAddress      --->        this is the network address you want checked.  The format of
                            theAddress is the same as for the struct AddrBlock as defined in
                            AppleTalk.h:
```

Bytes 3 & 2 (High Word): Network Number
Byte 1: Node Number
Byte 0 (Low Byte): Socket Number

Return codes

TRUE	network is still reachable
FALSE	network is no longer reachable

Error Codes

```
// ------------------------------------------------------------------------------
// MNP Error Codes - MNPInterface.h
// ------------------------------------------------------------------------------

#define MNP_ERR_BASE                    -6050               // base for MNP driver errors

#define ERR_MNP_NEGOTIATION_FAILURE     (MNP_ERR_BASE-1)    // Connection parameter negotiation
                                                            // failure
#define  ERR_MNP_CONNECT_TIME_OUT       (MNP_ERR_BASE-2)    // Connect request (acceptor mode)
                                                            // timed out
#define  ERR_MNP_NOT_CONNECTED          (MNP_ERR_BASE-3)    // Not connected
#define  ERR_MNP_ABORTED                (MNP_ERR_BASE-4)    // Request aborted by disconnect
                                                            // request
#define  ERR_MNP_ATTENTION_DISABLED     (MNP_ERR_BASE-5)    // Link attention service is not
                                                            // enabled
#define  ERR_MNP_CONNECT_RETRY_LIMIT    (MNP_ERR_BASE-6)    // Connect (initiator mode) request
                                                            // retry limit reached.
#define  ERR_MNP_COMMAND_IN_PROGRESS    (MNP_ERR_BASE-7)    // Command already in progress.
#define  ERR_MNP_ALREADY_CONNECTED      (MNP_ERR_BASE-8)    // Connection already established.
#define  ERR_MNP_INCOMPATIBLE_PROT_LVL  (MNP_ERR_BASE-9)    // Connection failed due to
                                                            // incompatible protocol levels
#define  ERR_MNP_HANDSHAKE_FAILURE      (MNP_ERR_BASE-10)   // Connection handshake failed.
```

```
// ---------------------------------------------------------------------------------
// Netshare Error Codes - RemoteAccessInterface.h
// ---------------------------------------------------------------------------------

#define ERR_BASE                        -5800

#define ERR_NOTCONNECTED                (ERR_BASE-0)
#define ERR_CONNECTIONABORTED           (ERR_BASE-1)
#define ERR_ALREADYCONNECTED            (ERR_BASE-2)
#define ERR_COMMANDALREADYINPROGRESS    (ERR_BASE-3)
#define ERR_BADVERSION                  (ERR_BASE-4)
#define ERR_INSHUTDOWN                  (ERR_BASE-5)
#define ERR_CONNECTIONABORTING          (ERR_BASE-6)
#define ERR_ALREADYENABLED              (ERR_BASE-7)
#define ERR_ZONEBUFBADSIZE              (ERR_BASE-8)
#define ERR_CONNECTTIMEDOUT             (ERR_BASE-9)
#define ERR_CONNECTUSERTIMEDOUT         (ERR_BASE-10)
#define ERR_BADPARAMETER                (ERR_BASE-11)
#define ERR_NOMULTINODE                 (ERR_BASE-12)
#define ERR_ATALKNOTACTIVE              (ERR_BASE-13)
#define ERR_NOCALLBACKSUPPORT           (ERR_BASE-14)
#define ERR_NOTOPENEDBYTHISPB           (ERR_BASE-15)
#define ERR_NOGLOBALS                   (ERR_BASE-16)
#define ERR_NOSMARTBUFFER               (ERR_BASE-17)
#define ERR_BADATALKVERS                (ERR_BASE-18)
#define ERR_VLD8_CONNECT                0
#define ERR_VLD8_CALLBACK               (ERR_BASE-19)
#define ERR_VLD8_BADVERSION             (ERR_BASE-20)
#define ERR_VLD8_BADUSER                (ERR_BASE-21)
#define ERR_VLD8_BADPASSWORD            (ERR_BASE-22)
#define ERR_VLD8_BADLINK                (ERR_BASE-23)
#define ERR_VLD8_NOCALLBACKALLOWED      (ERR_BASE-24)
#define ERR_VLD8_ALLCBSERVERSBUSY       (ERR_BASE-25)
#define ERR_VLD8_GUESTNOTALLOWED        (ERR_BASE-26)
#define ERR_VLD8_SERVERISIMPOSTER       (ERR_BASE-27)
#define ERR_VLD8_LOGINNOTENABLED        (ERR_BASE-28)
#define ERR_REMOTEPORTALREADYEXISTS     (ERR_BASE-29)
#define ERR_OPENNOTALLOWED              (ERR_BASE-30)
#define ERR_NOUSERSANDGROUPS            (ERR_BASE-31)
#define ERR_PORTSHUTDOWN                (ERR_BASE-32)
#define ERR_PORTDOESNOTEXIST            (ERR_BASE-33)
#define ERR_PWNEEDEDFORENABLE           (ERR_BASE-34)
#define ERR_DAMAGED                     (ERR_BASE-35)
#define ERR_NETCONFIGCHANGED            (ERR_BASE-36)


// ---------------------------------------------------------------------------------
// Connection Control Language Error Codes - CCL.h
// ---------------------------------------------------------------------------------

#define  cclErr_BaseCode                -6000
#define  cclErr_AbortMatchRead          cclErr_BaseCode          // internal error used to
abort match read
```

```
#define  cclErr                       (cclErr_BaseCode - 6)     // CCL error base
#define cclErr_CloseError             (cclErr_BaseCode - 7)     // There is at least one
script open
#define cclErr_ScriptCancelled        (cclErr_BaseCode - 8)     // Script Canceled
#define  cclErr_TooManyLines          (cclErr_BaseCode - 9)     // Script contains too many
                                                                // lines
#define cclErr_ScriptTooBig           (cclErr_BaseCode - 10)    // Script contains too many
                                                                // characters
#define  cclErr_NotInitialized        (cclErr_BaseCode - 11)    // CCL has not been
                                                                // initialized
#define  cclErr_CancelInProgress      (cclErr_BaseCode - 12)    // Cancel in progress.
#define  cclErr_PlayInProgress        (cclErr_BaseCode - 13)    // Play command already in
                                                                // progress.
#define  cclErr_ExitOK                (cclErr_BaseCode - 14)    // Exit with no error.
#define  cclErr_BadLabel              (cclErr_BaseCode - 15)    // Label out of range.
#define cclErr_BadCommand             (cclErr_BaseCode - 16)    // Bad command.
#define  cclErr_EndOfScriptErr        (cclErr_BaseCode - 17)    // End of script reached,
                                                                // expecting Exit.
#define  cclErr_MatchStrIndxErr       (cclErr_BaseCode - 18)    // Match string index is out
                                                                // of bounds.
#define  cclErr_ModemErr              (cclErr_BaseCode - 19)    // Modem error, modem not
                                                                // responding.
#define  cclErr_NoDialTone            (cclErr_BaseCode - 20)    // No dial tone.
#define  cclErr_NoCarrierErr          (cclErr_BaseCode - 21)    // No carrier.
#define  cclErr_LineBusyErr           (cclErr_BaseCode - 22)    // Line busy.
#define  cclErr_NoAnswerErr           (cclErr_BaseCode - 23)    // No answer.
#define  cclErr_NoOriginateLabel      (cclErr_BaseCode - 24)    // No @ORIGINATE
#define  cclErr_NoAnswerLabel         (cclErr_BaseCode - 25)    // No @ANSWER
#define  cclErr_NoHangUpLabel         (cclErr_BaseCode - 26)    // No @HANGUP


// -------------------------------------------------------------------------------
// Link Tool Manager Error Codes - LTMInterface.h
// -------------------------------------------------------------------------------

#define  ERR_LTM_BASE                       -5900               // base of errors for LTM
#define ERR_LTM_LISTENER_ID_IN_USE          (ERR_LTM_BASE-1)    // Specified  Listener identifier is
                                                                // in use
#define ERR_LTM_NO_LISTENER                 (ERR_LTM_BASE-2)    // Listener of specified type is not
                                                                // available
#define ERR_LTM_RESOURCE_NOT_REGISTERED (ERR_LTM_BASE-3)        // Listener of specified type is not
                                                                // available
#define ERR_LTM_PORT_NOT_CLAIMED            (ERR_LTM_BASE-4)    // claim request failed because to
                                                                // port is busy
#define ERR_LTM_COMMAND_NOT_ALLOWED         (ERR_LTM_BASE-5)    // LTM command not allowed on
                                                                // specified port
#define ERR_LTM_BAD_VERSION                 (ERR_LTM_BASE-6)    // connect failed due to
                                                                // incompatible LTM versions
#define ERR_LTM_ARBITRATION_TIMEOUT         (ERR_LTM_BASE-7)    // Connection failed due to a time
                                                                // out during the listener
                                                                // arbitration
#define ERR_LTM_KODE_NOT_FOUND              (ERR_LTM_BASE-8)    // kode resource not found in
                                                                // specified file
```

```
#define ERR_LTM_PORT_DISPOSED           (ERR_LTM_BASE-9)   // A Dispose Port call caused the
                                                           // request to fail.
#define ERR_LTM_RESOURCE_CLAIMED        (ERR_LTM_BASE-10)  // call failed because resource is
                                                           // already claimed
#define ERR_LTM_PORT_RESOURCES_CLAIMED (ERR_LTM_BASE-11)  // call failed because the port's
                                                           // resources are already claimed
#define ERR_LTM_RESOURCE_NOT_CLAIMED    (ERR_LTM_BASE-12)  // call failed because the resource
                                                           // was unclaimed
#define ERR_LTM_PORT_RESOURCES_NOT_CLAIMED (ERR_LTM_BASE-13) // The LTM port's resources are
                                                           // NOT claimed.
#define ERR_LTM_PORT_UNCLAIMED          (ERR_LTM_BASE-14)  // LTM listen port unclaimed
#define ERR_LTM_CONNECTION_REFUSED      (ERR_LTM_BASE-15)  // LTM listener refused connect
                                                           // request
#define ERR_LTM_CLAIM_ABORTED           (ERR_LTM_BASE-16)  // LTM claim call aborted due to
                                                           // LTM_ARB_CLAIM_CANCEL call
#define ERR_LTM_END_OF_PORT_LIST        (ERR_LTM_BASE-17)  // End of open port list reached in
                                                           // current port status session

#define ERR_LTM_NOT_CONNECTED           (ERR_LTM_BASE-18)  // Not connected.
#define ERR_LTM_CONNECTION_ABORTED      (ERR_LTM_BASE-19)  // Connection request aborted
#define ERR_LTM_BAD_LENGTH              (ERR_LTM_BASE-20)  // Length of write request exceeds
                                                           // maximum.
#define ERR_LTM_BAD_PARAMETER           (ERR_LTM_BASE-21)  // Bad parameter.
#define ERR_LTM_COMMAND_IN_PROGRESS     (ERR_LTM_BASE-22)  // Command already in progress.
#define ERR_LTM_CONNECTED               (ERR_LTM_BASE-23)  // Connection established.
#define  ERR_LTM_CONNECT_CANCELED       (ERR_LTM_BASE-24)  // Connection request canceled.
#define  ERR_LTM_CONNECT_TIMEDOUT       (ERR_LTM_BASE-25)  // Connection time out.
#define  ERR_LTM_NO_DEFAULTS            (ERR_LTM_BASE-26)  // Could not get default info...
#define  ERR_LTM_GOOD_BYE               (ERR_LTM_BASE-27)  // The driver is going away in a
                                                           // rude fashion
```

AppleTalk Remote Access Protocol (ARAP) External Reference Specifications

AppleTalk® Remote Access Protocol (ARAP) External Reference Specifications

These are the specifications for the underlying protocols currently being used in the AppleTalk Remote Access product version 1.0.

***NOTE**: Although every attempt has been made to verify the accuracy of the information presented, this document may contain errors and is subject to change.*

ARAP POINT TO POINT LINK

The AppleTalk Remote Access Protocol (ARAP) runs on top of a Point to Point Link (PPL). This document describes the characteristics of the PPL , the link arbitration protocol, and the Modem Link Tool . The Modem Link Tool provides a PPL for ARAP over telephone lines. Many of the details of the PPL are dependent on the particular implementation.

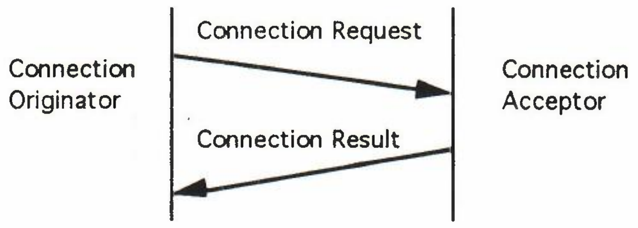
Point to Point Link Characteristics

In order to work with ARAP, each PPL must support the following basic set of characteristics:

- A connection for ARAP to send and receive "packets". ARAP expects a packet (i.e. block) type interface to the PPL. In other words, the data contained in one write request to the link, must be delivered to ARAP as the result of one read request. The framing of the packets is implementation dependent, but must relay the packet size information .
- A packet size of at least 604 bytes.
- Best effort, point to point data delivery. ARAP depends on the PPL to detect data errors in the packets, and to discard packets that contain errors. All data delivered to ARAP must be valid data sent by the other side of the link.
- Optionally, the PPL can provide reliable link. A reliable link guarantees that packets are delivered in the same order they are sent, and are free of duplicates. If the link is reliable, ARAP provides some enhancements which can greatly reduce the amount of redundant traffic sent across the link.

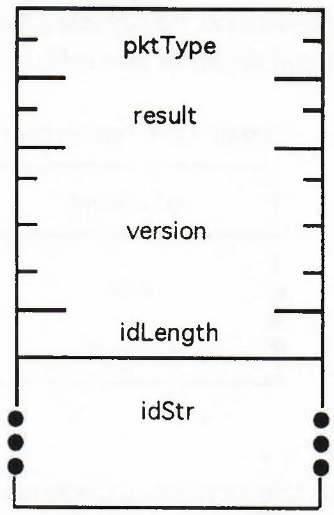
Link Arbitration Protocol

It is expected that the PPL used by ARAP may be shared other services. For example, a phone line and modem may be set up to receive ARAP connections as well as electronic mail connections. For this reason, when ARAP is establishing a connection, the side originating the connection arbitrates for the use of the link with the answering side of the connection. The arbitration dialog is as follows:



After the PPL is established, the originating (or calling) side of the connection sends a Connection Request (CR) packet to the accepting (or answering) side of the connection. The acceptor directs the CR to the appropriate service based upon the ID string contained in the CR. The answer side of the connection can accept or refuse the connect attempt, and indicates the result by sending the Connection Request Result packet to the Originator. The format of the CR and CRR packets is shown below.

Link Arbitration Packet



The pktType field indicates the type of the packet. The value 1 indicates the packet is a Connect Request, a 2 indicates the packet is a Connect Request Result. The result field is zero in the CR, and indicates the result of the connection request in the CRR packet. A zero value for the result in the CRR packet indicates that the Connection Request was successful and ARAP can use the link. A non-zero value indicates the CR failed. Possible error codes include the following:

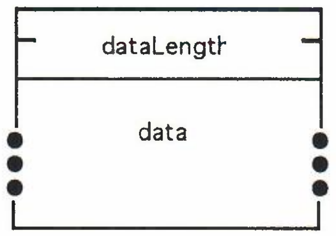
-5902	Service requested is not available
-5906	Incompatible arbitration packet version
-5915	Connection refused by acceptor

The version field indicates the version of the arbitration packet format. The version is a four byte version defined by Apple two have the following parts (packed into a long in order): 1) First part of the version number in BCD. 2) Second and third parts. 3) Release type (development=0x20, alpha=0x40, beta=0x60, release=0x80). 4) Stage of prerelease version. If the acceptor does not support the callers version, a -5906 result is returned. The idStr is a Pascal style string which indicates the service the caller wants to connect to. The ARAP idStr value is "Remote Access". The idLength is the length of the idStr, including the length byte.

The Modem Link Tool

The Modem Link Tool (MLT) provides a **reliable** PPL over asynchronous serial connections. The MLT has two major components. The first component, the CCL, establishes the physical connection (e.g. connects the modems). After the physical connection is established, the second component, the reliable protocol, provides a reliable PPL. The reliable protocol used is the V.42 Alternative Procedure, with V.42bis data compression. The protocol is described in ANNEX A of the V.42 specification. On top of the reliable protocol, the MLT uses a simple packet format to ensure packet delivery for ARAP. The MLT packet format used on top of the reliable protocol is a follows:

Modem Link Tool Packet



The dataLength field is the length of the data in the packet, and does not included the length bytes. The maximum dataLength supported by the MLT is 618 bytes.

APPLETALK REMOTE ACCESS PROTOCOL

The AppleTalk Remote Access Protocol (ARAP) provides efficient AppleTalk services on a per client basis over slow links. It defines the login and authentication sequence as well as the Appletalk data format. There are two broad types of ARAP information: 1) Internal messages (authentication packets, tickle packets, etc.) that define information related to establishing and maintaining the link. 2) AppleTalk packets which contain the higher level (DDP and above) data that is to be sent. This document describes version 1 of ARAP.

If the point to point link that we are operating on can ensure reliable, in order, delivery of data we can provide some optional enchancements to the AppleTalk packet delivery mechanism. These enhancements are very desirable since they can greatly reduce the amount of redundant traffic sent across the link.

Internal Messages

In order to operate over a wide range of point to point links we must not require that the underlying link provides reliable packet delivery for proper operation of this protocol. Therefore, for internal messages we have defined a very simple protocol to deliver the information sent in a reliable fashion. This protocol is not tailored to any specific link and is limited to sending one message at a time. This is sufficient for our needs since we send a very low volume of internal messages relative to the amount of AppleTalk information. It is assumed that the underlying link provides the framing of the packet and has a way to relay the size of the data sent. We also assume that the underlying link can support packets of at least 604 bytes in size. NOTE: All numbers passed in internal message packets are stored big endian.

DataGroup Flag:

This flag always precedes a datagroup and is used to describe the information that follows. It is a one byte value and is broken down into the following fields:

Bit 7: Reserved -
This bit must be set to zero.

Bit 6: Packet Data -
This bit is set if the data in this packet is data. It is clear if it is an out of band message.

Bit 5: Tokenized -
This bit is set if the data after this flag is a token. If it is a token, the sequence number of the token is in the word that follows (always a word, not a compactnum). If this bit is not set, it is assumed that raw data follows. The raw data is preceded by a compacted length value unless it is the last group of data in the packet. The last group of raw data in a packet is a special case in which the length can be derived from the amount of data remaining in the packet.

Bit 4:Last Group -
This bit is set if the data that follows is the last group of data in the packet. This flag must be present in the last datagroup. If raw data is being described and this flag is set, the size of the data is calculated as specified above.

Bit 3: Range -
This bit indicates whether or not the sequence number that follows describes a range or not. If the bit is set, the sequence number is describing part of a sequenced packet, not the entire packet. The bytes that are wanted from that packet are described by the two compact numbers that follow the sequence number word.

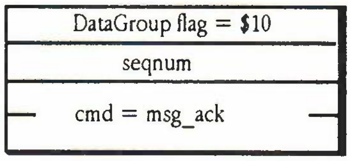
Bit 2: Want Sequenced -
This bit indicates whether or not the group of data described by this flag should be sequenced and cached by the receiver. If the bit is set, the receiver must cache and sequence the data described in this datagroup. The data described can be either raw data or data described by a sequence. If it is data described by a sequence it must be decoded before the entry is made into the cache. If the bit is clear, no entry is made for this datagroup in the receivers cache.

Bit 1: Want Packet Sequenced -
This bit only applies to the first flag in the packet. If it is set, the entire packet should first be decoded and then entered into the receivers cache. It is important that the packet be scanned and any entries requested within the packet be created before the packet itself is sequenced.

Bits 0: Reserved
Must be zero.

Simple Reliable Protocol (SRP)

All packets sent are preceded by a flag byte that indicates whether this packet is an internal message or an AppleTalk packet. If you are sending an internal message set bit 6 of the flag byte to zero. All internal message packets are then followed by a one byte packet sequence number and a two byte command. The sequence number is used to ensure the ordering of packets that have been sent. The command defines the type of data that this packet represents. The ack command (cmd=0) has a special meaning. It does not define any data but is sent to acknowledge the arrival of a message from the other end of the link.



Initializing SRP

Both sides of the connection must start off by initializing their outgoing and incoming (expected) sequence numbers to zero. This allows both sides to start off in sync with each other.

Writing data

In order to write a packet using SRP, you set bit 6 of the flag byte to zero, the command byte to the command wanted, and the sequence number to the current outgoing sequence number. The rest of the packet is made up of the data that is represented by the command we are sending.

After the packet is sent, we must receive an ack that matches the sequence number we sent to indicate successful delivery of the packet. If after a period of n seconds (where n=6 for 1200 bps, n=3 for 2400 bps, n=2 for 4800+ bps) no ack has been received, we send the packet again. This process will take place until the ack is received or a period of 60 seconds elapses in which case the connection is torn down with a timeout error.

Once the ack is received successfully we increment our current outgoing sequence number by one for the next write. Since this is a one byte value we wrap back to zero after we add 1 to a sequence number of 255.

Reading data

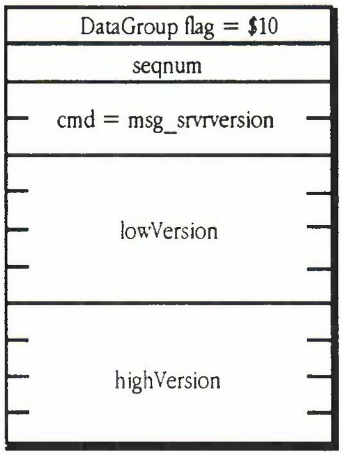
In order to read data we must first determine if an incoming packet is an internal message. We do this by checking the bit 6 of the flag byte to see if it is zero. If it is, we have an internal message packet.

The incoming sequence number is then checked to see if it equals the expected incoming sequence number. If it is not the sequence number expected, an ack should be sent back if the sequence number is one less than expected. The ack should be returned with the received sequence number. This could happen if we receive a duplicate of a message we have already accepted. If the packet matches the expected sequence number an ack is sent, the expected sequence number is bumped, and the data is delivered.

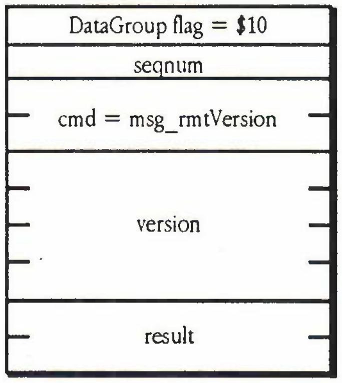
ARAP Connection Establishment

After the low level link has been established we begin the process of establishing the ARAP link. In this discussion the client represents the originator, and the server represents the answering or accepting side of the connection. Before reading or writing any data both sides initialize the SRP. The sequence of events in connection establishment is critical and must be followed exactly.

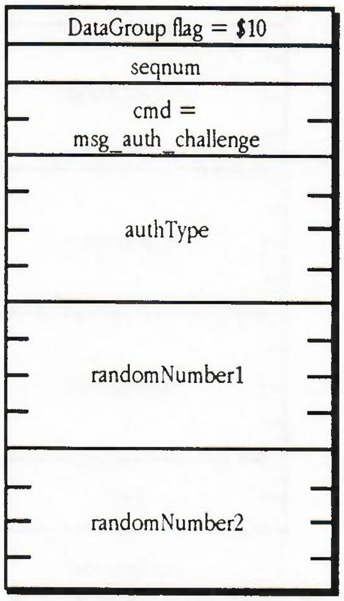
Server-
Writes a message (cmd=msg_srvrversion) that has the lowest and highest versions that the server can accept. The version is a four byte version defined by Apple to have the following parts (packed into a long in order): 1) First part of the version number in BCD. 2) Second and third parts. 3) Release type (development=0x20, alpha=0x40, beta=0x60, release=0x80). 4) Stage of prerelease version.



Client-
Reads message and confirms that it is msg_srvrversion. If it is not a msg_srvrversion, a reply is sent (cmd=msg_rmtversion) with a result code of ERR_VLD8_BADVERSION, and the session is torn down. If it is msg_srvrversion, we attempt to find a suitable version by taking the minimum of the highVersion from the server and the highest version we support. This gives us the maximum version acceptable to both sides. We then check to see if this version is less than our minimum version supported or less than the lowVersion from the server. If it is, we do not have an acceptable version match with the server and must send a reply (cmd=msg_rmtversion) with an result code of ERR_VLD8_BADVERSION, and tear down the session. If the version is acceptable we set version=acceptableversion and send it (cmd=msg_rmtversion) to the server with an result code of zero.

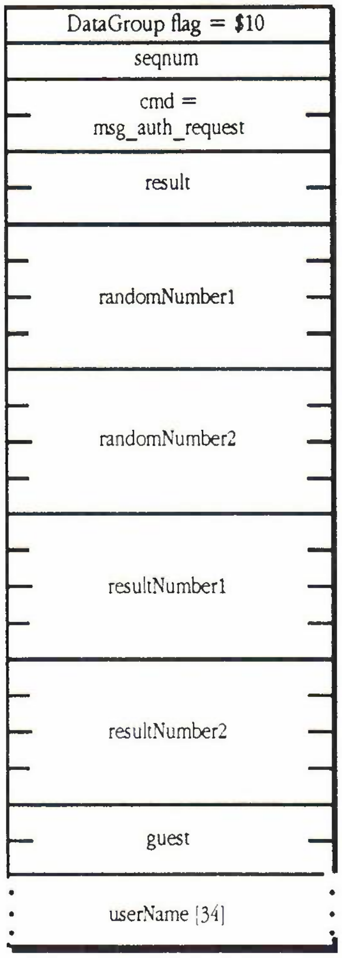


Server-
Reads message and confirms that it is msg_rmtversion and that the result is zero. If it is not, the session is torn down. If it is, we check the version sent to us to make sure that it falls within our acceptable range of versions. If it does not we tear the session down. If it does, we know which version of the protocol to use and continue by creating an authentication challenge packet. This packet contains a authType field which represents the type of authentication being done. Currently the only authtype used is two way DES (authtype=kAuth_TwoWayDES). In the two way DES packet, two 32 bit random numbers are generated and put into randomNumber1 and randomNumber2 The packet is then sent (cmd=msg_auth_challenge).

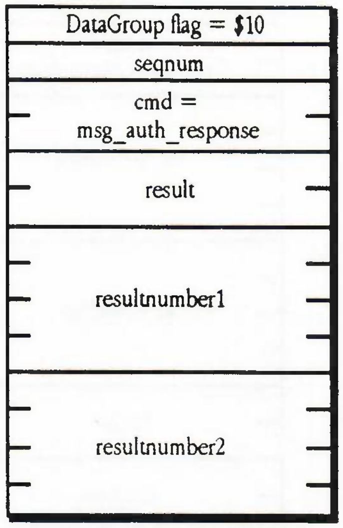


Client-
Reads message and confirms that it is msg_auth_challenge and that authType equals kAuth_TwoWayDES. If not, result is set to ERR_VLD8_BADVERSION and a reply is sent (cmd=msg_auth_request). The session is then torn down. If the message is ok, we form an authentication request packet. If we are logging in as a guest, we set the guest flag and set the username to "<Guest>". If we are logging in as an authenticated user, we set the userName to our user. The field userName is an array of 34 characters. If the string you are returning in userName is less than 34 characters the remaining space needs to be padded. If we are authenticating ourselves to the server we must use the randomNumber1 & randomNumber2 sent by the server produce a resulting number. The technique used is to copy the password (excluding the len byte) into a space of 8 bytes. Any unused bytes are set to zero. A key is then generated from the password. This key is then used to encode the 64 bits of random number information passed to us by the server. The resulting number is put into our outgoing packet in resultNumber1 and resultNumber2. We then generate our own random number to challenge the server (we want to be sure it really knows our password) and store it in randomNumber1 and randomNumber2. Finally the result is set to zero and the packet is sent

(cmd=msg_auth_request).

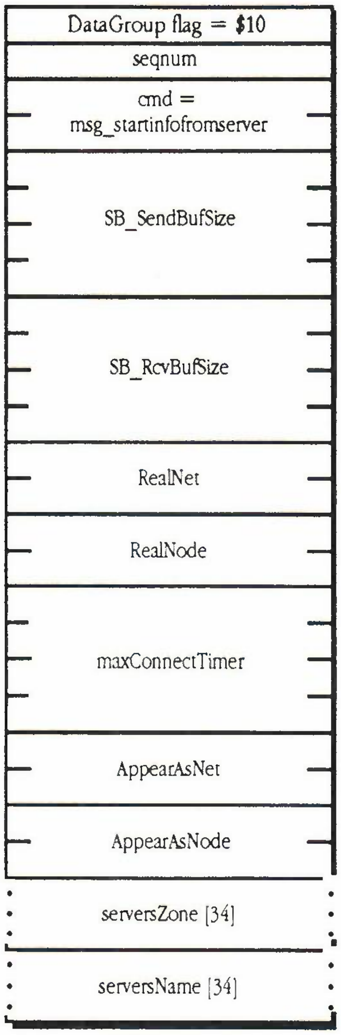


Server-
Reads message and confirms that it is msg_auth_request and that the result is zero. If it is not, the session is torn down. If it is, we confirm that the userName and resultNumber returned to us is valid. If they are trying to log in as a guest and we don't support guest the error is ERR_VLD8_GUESTNOTALLOWED. If it is a bad user we set the error to ERR_VLD8_BADUSER. If the resultNumber does not match we set the error to ERR_VLD8_BADPASSWORD. If we got an error we send it back (cmd=msg_auth_response) in result. If everything is ok and we are not doing guest login, we generate a resultNumber the same way the client did above to authenticate to it that we really know the password. At this stage it is possible that this user should be called back. If this is the case we set a result code of ERR_VLD8_CALLBACK to indicate that we will be doing callback. If the user was authenticated and no callback was wanted, we set the result code to zero. We then send the message back to the client (cmd=msg_auth_response). If we are doing callback, we disconnect, initialize SRP, and attempt to connect to the client through callback.

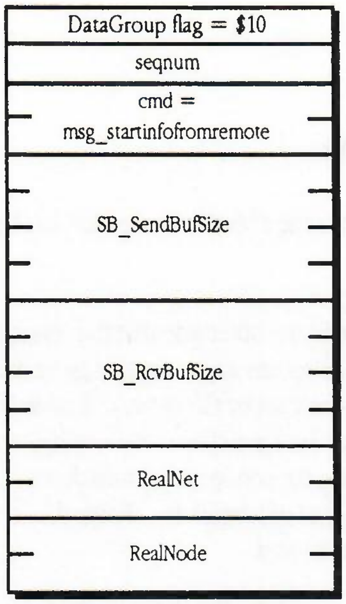


Client-
Reads message and confirms that it is msg_auth_response. If the result is not ERR_VLD8_CALLBACK or zero, we tear down the connection. If we are not doing guest login we check the resultNumbers to make sure they are expected. If not, the connection is torn down. If the result is zero we continue . If the result is ERR_VLD8_CALLBACK, we disconnect, initialize SRP and wait to answer.

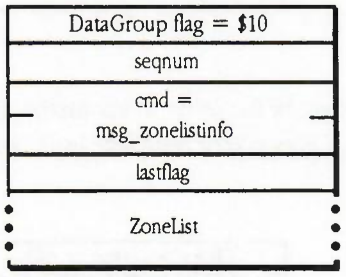
Server-
The server puts together a packet that has information about itself and how the client should appear. RealNet and RealNode are set to the address of the server itself. This information is used in clients that use remapping to ensure that they will be able to communicate with the server under all conditions. AppearAsNet and AppearAsNode defines the ddp network address that the client should use when creating AppleTalk packets. Since the AppearAsNode field is defined as an unsigned short, the high byte of this field will always be zeroed. SB_SendBufSize and SB_RcvBufSize define the amount of space allocated to SmartBuffering for sending and receiving respectively. If SmartBuffering is not used you should set both of these to zero. SmartBuffering can only be used if the underlying link is known to provide reliable delivery of packets. MaxConnectTimer defines the maximum amount of time in seconds that this client is allowed to stay connected. This value will be -1 if the time is unlimited. ServersZone defines the zone that the client will appear in. ServersName should be set to the name of this server. This string appears to the user as the name of the computer that they are connected to. The fields serverZone and serversName are arrays of 34 characters. If the string you are returning in serverZone and serversName are less than 34 characters the remaining space needs to be padded. After these fields are filled in, the packet is sent (cmd=msg_startinfofromserver).



Client-
Reads message and confirms that it is msg_startinfofromserver. If not, the connection is torn down. In order to determine the proper size of the buffers for SmartBuffering, the minimum of the servers buffer sizes and our own is obtained. If SmartBuffering is not being used we set our SB_SendBufSize and SB_RcvBufSize to zero. SmartBuffering can only be used if the underlying link is known to provide reliable delivery of packets. As a courtesy to the server we indicate our real address (this will be different than the appearAs address if we are using remapping) in RealNet and RealNode. The packet is then sent (cmd=msg_startinfofromremote).



Server-
Reads message and confirms that it is msg_startinfofromremote. If not the connection is torn down. In order to determine the proper size of the buffers for SmartBuffering, the minimum of the clients buffer sizes and our own is obtained. Next, we begin sending the list of allowable zones to the client (cmd=msg_zonelistinfo). This list of pascal strings must be sorted in ascending order. Since this list may be larger than what can fit into one packet we break it into multiple packets. Only the last packet should set the lastflag to true. Care must be taken to ensure that only complete zone names are fit into a packet (we do not allow part of a zone name in one packet and the other in the next). After all zones are sent the connection is established and we can now accept and send AppleTalk packets.

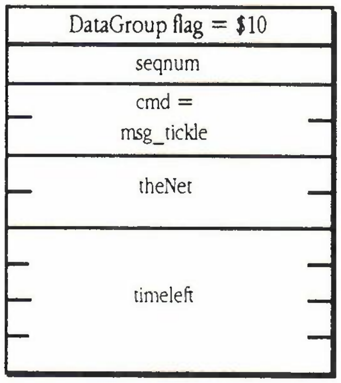


Client-
Reads message and confirms that it is msg_zonelistinfo. If not, connection is torn down. Packets are read until the lastflag is true. The connection is established and we can now send and receive AppleTalk packets.

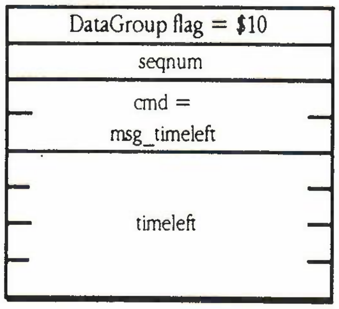
Session maintenance packets-

The following messages can be sent any time after the connection has been established.

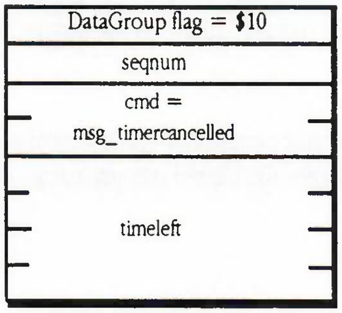
Tickle-
This message (cmd=msg_tickle) informs the other side that the connection is still intact. It contains within it the network number of the sender (so the client can adapt to a change from net zero on nonextended nets). It also contains the time left in this session in seconds (only set by the server). A value of -1 indicates that there is unlimited time remaining in the connection. A tickle packet should be sent every 20 seconds and if no valid packets (data or internal message) are received within 60 seconds, the connection is torn down. The reception of any valid packet will cause us to reset our teardown timer. If neither theNet or timeleft has changed, and a valid packet has been received in the last 20 seconds, then no tickle packet needs to be sent.



Time Left-
This message (cmd=msg_timeleft) is sent by the server to inform the client that it only has the number of seconds in timeleft remaining. This message should generally be dealt with by informing the clients user that his connection will be torn down in timeleft seconds.



Timer Cancelled-
This message (cmd=msg_timercancelled) indicates that a time left message is being cancelled and the connection is no longer being shut down. The time left indicates the new time left in the connection (generally -1 to indicate unlimited).



AppleTalk Packets

Once the connection has been established we can start to send AppleTalk packets across the link. The format of the packets depends on whether they are being encapsulated in SmartBuffering or not. If SmartBuffering is not being used, the flag byte that precedes the Appletalk data has the bits sflag_PktData and sflag_LastGroup set. This indicates that the packet represents Appletalk data and that the Appletalk data is the last group of data in this packet.

To provide a consistent state (and to improve SmartBuffering) we set certain fields to a known value before transmitting a packet. All packets are sent using the long ddp form. Both the lap source and lap destination bytes are set to zero. If a checksum has been set in the packet, the lap source and lap destination bytes are set to 1. This indicates to the receiver that it must recalculate the checksum. The length byte of ddp is also set to zero. This must be recalculated by the receiver before delivering a packet or forwarding it onto the net.

Client responsibilities
When the client needs to send an NBP lookup it should set the nbp function to nbpBrRq, even if there is no router. This allows the server to distinguish whether or not this packet represents a NBP lookup or an NBP confirm. Confirms will always have their nbp function set to nbpLkUp.

Server responsibilities
The server is responsible for properly forwarding packets sent by the client. If an NBP packet arrives with a function code set to nbpBrRq, it treats the lookup as if it had originated from its own stack as described in Inside AppleTalk. This means it must determine whether to forward this lookup to a router if there is one or to send it on its own net. If it is on an extended net, it must also check to see if this lookup is for zone '*'. If it is, it must expand it to its zone if it has one before sending onto the net.

The server is also responsible for keeping packets originated by the client from being returned to the client. This includes both packets that have a source address equal to the client's address in the DDP header as well as NBP packets that have a source equal to the client. For example, if the client sends a packet and the server forwards this packet to a router, the router may generate a lookup request. It is then the reponsibility of the server to keep the lookup request from returning to the client.

The server is also responsible for eliminating broadcast RTMP information from being forwarded to the client. This information is not needed by the client since the client is simply just another node on the servers net.

Error codes:

ERR_VLD8_CALLBACK	-5819
ERR_VLD8_BADVERSION	-5820
ERR_VLD8_BADUSER	-5821
ERR_VLD8_BADPASSWORD	-5822
ERR_VLD8_BADLINK	-5823
ERR_VLD8_NOCALLBACKALLOWED	-5824
ERR_VLD8_ALLCBSERVERSBUSY	-5825
ERR_VLD8_GUESTNOTALLOWED	-5826
ERR_VLD8_SERVERISIMPOSTER	-5827
ERR_VLD8_LOGINNOTENABLED	-5828

Message numbers:

msg_ack	0
msg_srvrversion	1
msg_rmtversion	2
msg_auth_challenge	3
msg_auth_request	4
msg_auth_response	5
msg_startinfofromserver	6
msg_startinfofromremote	7
msg_zonelistinfo	8
msg_tickle	9
msg_timeleft	10
msg_timercancelled	11

Flag bit masks:

sflag_Fixup	0x80
sflag_PktData	0x40
sflag_Tokenized	0x20
sflag_LastGroup	0x10
sflag_Range	0x08
sflag_WantSeqd	0x04
sflag_WantPktSeqd	0x02
sflag_Reserved	0x01

Authentication types:

kAuth_TwoWayDES	1

SMARTBUFFERING

SmartBuffering is an algorithm that is used to reduce repetitive traffic such as packets that are sent on a network. It can be tailored to recognize entire or partial packets of information. It works by caching a specified amount of information and checking future packets against that information for matches. If matches are found, a token representing that data is put into the data stream instead of the actual data. Smartbuffering provides the ability to mix both tokens and raw data in the data stream.

Smartbuffering is a state driven algorithm. It requires that both the sending and the receiving side always be in sync. Therefore, the medium used to transmit its data must provide a reliable point to point link. It also requires that the sender and receiver use the same size buffers in order to remain synced up when new entries items begin overflowing into earlier ones. Because of this it is necessary to exchange buffer sizes before any data is processed. The buffer size that is used is the smaller of the two buffers to ensure that both sides end up with the same size buffers since the side with the larger buffer can always temporarily reduce the amount of buffer it provides.

The choice of what data to cache and how to recognize it when matching subsequent packets is entirely that of the sender. Smartbuffering describes the format of the packets used to transmit the data and how the actual data is stored. It does not explicitly describe how matches are found on the sending side or how any indexes into the send side data are stored. Determining what to match and how to match the data is implementation independent and can be tailored for the needs at hand.

Implementation Details

At the heart of Smartbuffering lies the techniques used to ensure that the sender and receiver always remain in sync.

The first step in ensuring this is to provide a method in which both sides can determine the proper size buffer to use. In order to do this they exchange their preferred buffer sizes. After getting the other's buffer size each chooses an actual buffer size that is the smaller of the two. Thus, if the sender had 10000 bytes of storage and the receiver had 8000 bytes of storage the resulting size would be the smaller of the two which is 8000 bytes. This ensures that every operation done to the sender's buffer will result in exactly the same buffer state on the receiving side after the same operation is carried out.

The second step to ensuring that both sides remain in sync is for them to provide a consistent way of adding to and removing items from the data cache. The algorithm used in Smartbuffering uses a ring buffer. The method of storing data in this buffer must be exactly the same on both sides. Each packet inserted into the buffer has the following format:

```
typedef struct TPacketData
{
	long			right;		/* link for receive, cksumhead for send */
	long			left;		/* link for receive */
	unsigned short seqnum;	/* sequence number of this entry */
	unsigned short datalen;		/* length of data that follows */
	unsigned char		databuf[];	/* data that follows is tacked on here */
} TPacketData, *TPPacketData;
```

Any block of data added to the ring buffer must always fit without wrapping. In other words, if there is not enough room at the high end of the buffer to store the full block of data, the buffer must be wrapped back to the low end. This is done to simplify the mapping of data structures onto the data within the buffer (at the expense of slightly under utilizing the available space). It would be very cumbersome to deal with the data if part of it was in one place and the other part somewhere else. If there is not enough room to insert an entry, items are removed until enough space exists. It is important to remember that both the sender and the receiver must use exactly the same approach to manage their buffers because they must always stay in sync.

In our implementation, when used for sending, the right field is used to link together the checksum records that describe this block of data. It is not a requirement that this field be used in this way, it is only a requirement that the fields specified in the header exist.

The size of the entire record which includes the actual data must always be an even number of bytes. This is to ensure that subsequent entries will always have their headers aligned on even boundaries. Therefore, if the actual size of the record happens to be an odd number, one byte is added to the amount of space used in the cache.

In order to describe the data in this buffer in a tokenized format, a sequence number is stored with the data. This sequence number will be duplicated on the receive side when it stores this data. It is crucial that the sender and receiver use the same technique to generate sequence numbers to ensure that both sides stay in sync. In order to do this, both sides always start at a sequence number of zero before any data is stored. For each new item inserted into the data buffer, a new sequence number is generated by adding one to the previous one.

When adding data to the buffer, the following technique must be followed by both sides in order for them to stay in sync. The packet is scanned and all operations are executed in order. After all of the partial data packets are added, the entire packet may be added if the appropriate bit has been set in the first datagroup flag byte. Adding the entire packet last ensures that the receiver will be able to reconstruct the packet based only on previous information. If the packet was added first, it would be possible to have entries in the data that referenced the packet itself. If this were allowed to happen, the receiver would have no way of reconstructing the packet since the packet itself would be required!

Each packet always contains at least one datagroup. It may contain more than one datagroup if required. Each datagroup is made up of a flag followed by data whose meaning can be discovered by decoding the flag. A datagroup is a subunit of the packet that can be used to mix different types of data. For example, it might be desirable to describe a packet by using both matched data using sequence numbers as well as new data that is sent in its raw format.

In order to minimize the amount of data required to describe the information that follows the datagroup flag we have defined a compact representation of an unsigned short number. This compact representation is known as a compactnum and has the following characteristics: 1) If the high bit of the first byte (first is described as reading left to right within the packet) is set, then we mask off the high order bit to obtain the number. This means that if the value wanted is less than 128 it can be described in 1 byte. If the bit is not set, then the number is a two byte number, and the low order value of the number is obtained from the next byte. This numbering scheme limits the maximum value of a compactnum to 32767.

Redundant traffic reduction

If the link that we are running over ensures reliable packet delivery and the two ends of the connection have negotiated for SmartBuffering, we can take steps to reduce much of the redundant traffic between the two points. SmartBuffering interprets AppleTalk packets based on protocol type, and then creates checksums that define the parts of packets that are most likely to be repetitive.

The receive side of our implementation uses the standard SmartBuffering techniques. It simply obeys the requests of the send side to reproduce the data. Therefore, we will not discuss any details of the receive side.

The effectiveness of SmartBuffering depends on a number of factors. One of the most important factors is how often recognizable packet components match. A good example of highly repetitive data is NBP. Typically the same (or very similar) packets are sent a number of times. Therefore NBP data achieves very good compression since it is so repetitive (typically better than 5 to 1). Another factor in the effectiveness of SmartBuffering is how large the actual history buffers are. Obviously, the larger the buffers the more likely we are to find a match with some information that was previously sent. In our current implementation we use a send an receive history buffer size of 11200.

When a packet is ready to be sent, specific checks are made depending on the type of packet.

First, all packets are checked to see if the entire packet matches (the only exception to this are echo packets which we always want represented in their full form). If we get a full packet match, then the token that describes that match is sent to the receive side. No more interpretation is done, since the partial interpretations of that packet would already have been done and are probably still in the data cache.

If an entire match of the packet cannot be made we take special action depending on the type of data within the packet.

DDP Header -
ARAP always creates long headers when sending packets. Therefore, we only need to interpret long headers. The first time we see a header, a checksum entry is created that describes the data that includes all of the lap and ddp part of the header. If a subsequent packet has a header that matches one that has already been checksummed we substitute the raw data with that of a datagroup token range. Since a range takes between 2 and 4 more bytes to describe than a complete

token we instruct the receiver to enter the data described by this range into its cache. This allows us to provide a full match on the header if we get another packet with the same header. Once we have told the receiver to buffer the full header, we can describe it without using a range thus saving the extra bytes that would have been needed. Since ddp headers are quite similar (most transactions end up going to the same places) we have typically seen the entire header of any packet reduced down to 3 bytes. This honing down technique is used in most of the other header interpretations.

NBP data -
NBP packets tend to be very similar. The most likely items to change are the function and nbpid fields. Fortunately they are situated next to each other. Since it is very likely that the data that follows will produce a match in a later packet we cache it using the honing down technique described in the DDP Header. This produces a high likelihood of a match and allows us to describe it in a small number of bytes. We have seen between 5 and 20 to 1 compression of the nbp traffic.

ATP data-
The entire ATP header is skipped and only the data is searched for a match. Since the ATP Header is only 8 bytes we would see minimal return for special casing that part of the packet. The data however could produce substantial reduction if retransmissions with different headers were occurring.

ADSP data-
Since the ADSP header is quite likely to change from one packet to the next, it is ignored. Again, the data beyond the header is matched since a retransmission could produce a large savings even if the header had changed.

Other data-
For types of packets that we do not special case we simply take all of the data following the DDP header and attempt to match it. As new data types become known, it is possible that they will be added to our list of special cases.

In the future it is also possible that we may take advantage of the sflag_Fixup bit to special case even more details of the packet. This mechanism has not been completely defined by SmartBuffering at this time.

After the special casing of the packet is done we create an entry for the entire packet. This gives us the potential of finding entire packet matches in the future. It is possible through this mechanism to achieve 200 to 1 compression on packets that are exact duplicates of some other packet that preceded it.

AppleTalk Remote Access Smartbuffering Example:

We will show what happens to an NBP Lookup packet as it is retransmitted and it's id changes. NOTE: AppleTalk Remote Access expects the LAP source and destination to be zeroed. It also expects the length byte to be zeroed on transmission (this is filled in by the other side after being received). We will assume that no packets have yet been transmitted.

```
First Lookup Packet:

LAP/DDP Header     -> 00 00 02 00 00 00 00 11 11 22 22 11 22 02 FE 02
NBP Func & ID      -> 21 74
NBP Tuple          -> 22 22 22 FE 00 01 = 09 AFPServer 01 *
```

Since this is the first packet being sent we do not already have an entry for it or any of it's parts in our transmit cache. Therefore, we will create an entry for the entire packet (sequence number 0) and set the datagroup flag to indicate that the receiver should also create an entry. We also create an entry in our checksum table to the interesting parts of the packet. In this particular case they are the LAP/DDP Header (combined), the NBP Tuple, and of course the entire packet. These checksum entries will give us the possibility to match all or part of a subsequent packet. The resulting packet to be sent is simply a flag followed by the raw data in this case:

```
DataGroup Flag     -> 52 (PktData, LastGroup, WantPktSeqd)
LAP/DDP Header     -> 00 00 02 00 00 00 00 11 11 22 22 11 22 02 FE 02
NBP Func & ID      -> 21 74
NBP Tuple          -> 22 22 22 FE 00 01 = 09 AFPServer 01 *
```

We now have one packet sequenced and 3 checksums that describe it. Now, if the lookup packet moved on to a new ID, we would be presented with the following packet (before Smartbuffering):

```
Second Lookup Packet:

LAP/DDP Header     -> 00 00 02 00 00 00 00 11 11 22 22 11 22 02 FE 02
NBP Func & ID      -> 21 75 (Note the ID change from 74)
NBP Tuple          -> 22 22 22 FE 00 01 = 09 AFPServer 01 *
```

This time, the transmitter checks to see if it already has the entire packet in it's cache and discovers that it does not (remember the ID byte changed). It then checks to see if the LAP/DDP Header matches. It discovers that it does have a match for that part of the packet in sequence number 0, bytes 0 through 0x0f. It outputs the appropriate datagroup flag indicating that a token is being sent for that part of the packet. It also creates a new entry into the cache for this part of the packet (so it will not have to be described as a range in the future) and sets the WantSeqd bit in the datagroup flag (this will be sequence number 1). When it gets to the NBP Func & ID bytes, it simply outputs a datagroup flag that indicates they are raw data. It does not try to checksum or sequence them since they change quite often and it would not save any space to tokenize two bytes. Next, it tries to match the NBP tuple and finds that it has a match for that part of the packet in sequence number 0, bytes 0x12 through 0x24. It also creates a sequence for this portion of the packet and emits the datagroup flag that describes the range found and also indicates to the receiver that it should sequence this part of the packet (sequence number 2). Finally, the entire packet is entered as sequence number 3. The following data results:

```
DataGroup Flag      -> 6E (PktData, Tokenized, Range, WantSeqd, WantPktSeqd)
Sequence Number     -> 00 00
Range               -> 80 8f (compact numbers for 00 0f)
DataGroup Flag      -> 40 (PktData)
Raw Data Len        -> 82 (compact number for 02)
NBP Func & ID       -> 21 75 (the 2 bytes of raw data)
DataGroup Flag      -> 7C (PktData, Tokenized, LastGroup, Range, WantSeqd)
Sequence Number     -> 00 00
Range               -> 92 A4 (compact numbers for 12 24)
```

As you can see, the resulting packet is 14 bytes long, and the original packet was 37 bytes long, for a savings of 23 bytes. If yet another NBP Lookup is sent and the NBP ID byte changes once again we get the following packet to process:

```
Third Lookup Packet:

LAP/DDP Header      -> 00 00 02 00 00 00 00 11 11 22 22 11 22 02 FE 02
NBP Func & ID       -> 21 76 (Note the ID change from 75)
NBP Tuple           -> 22 22 22 FE 00 01 = 09 AFPServer 01 *
```

The transmitter will once again go through the process of attempting to match the packet. It will not find the entire packet match since the NBP ID changed again. It will find a match for the LAP/DDP Header, but this time it will find the match in sequence number 1 (the sequence it created for this portion of the packet in the above transmission). The advantage this time is that this sequence exactly describes the part of the packet it is looking for, and will be able to describe it without the range bytes. We then output a raw data description for the NBP Function and ID bytes. Then we come to the NBP tuple which is matched with sequence number 2. Again, this is a direct match so we don't need the range bytes. Finally, the entire packet is sequenced (number 4). The resulting data is:

```
DataGroup Flag      -> 62 (PktData, Tokenized, WantPktSeqd)
Sequence Number     -> 00 01
DataGroup Flag      -> 40 (PktData)
Raw Data Len        -> 82 (compact number for 02)
NBP Func & ID       -> 21 76 (the 2 bytes of raw data)
DataGroup Flag      -> 70 (PktData, Tokenized, LastGroup)
Sequence Number     -> 00 02
```

This time we were able to describe the packet in only 10 bytes. Finally, lets assume that the transmitter needs to resend the above packet with no change. We get the following packet to be processed:

```
Fourth Lookup Packet:

LAP/DDP Header     -> 00 00 02 00 00 00 00 11 11 22 22 11 22 02 FE 02
NBP Func & ID      -> 21 76 (Note same as above)
NBP Tuple          -> 22 22 22 FE 00 01 = 09 AFPServer 01 *
```

This time the transmitter will check to see if it can match the entire packet and discover that it does have a match in sequence number 4. Therefore, it can describe the packet as being made up of only one sequence, and the following data results:

```
DataGroup Flag     -> 70 (PktData, Tokenized, LastGroup)
Sequence Number    -> 00 04
```

As you can see we were able to describe the original 37 byte packet in only 3 bytes. Typically with NBP traffic, we would reach this stage after only 2 packets because the NBP ID does not normally change with each packet sent.

APPLE COMPUTER, INC. SOFTWARE LICENSE

PLEASE READ THIS LICENSE CAREFULLY BEFORE USING THE SOFTWARE. BY USING THE SOFTWARE, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. **License.** The application, demonstration, system and other software accompanying this License, whether on disk, in read only memory, or on any other media (the "Apple Software") and related documentation are licensed to you by Apple. You own the disk on which the Apple Software is recorded but Apple and/or Apple's Licensor(s) retain title to the Apple Software and related documentation. This License allows you to use the Apple Software on a single Apple computer and make one copy of the Apple Software in machine-readable form for backup purposes only. You must reproduce on such copy the Apple copyright notice and any other proprietary legends that were on the original copy of the Apple Software. You may also transfer all your license rights in the Apple Software, the backup copy of the Apple Software, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. **Restrictions.** The Apple Software contains copyrighted material, trade secrets and other proprietary material and in order to protect them you may not decompile, reverse engineer, disassemble or otherwise reduce the Apple Software to a human-perceivable form. You may not modify, network, rent, lease, loan, distribute or create derivative works based upon the Apple Software in whole or in part. You may not electronically transmit the Apple Software from one computer to another or over a network.

3. **Support.** You acknowledge and agree that Apple may not offer any technical support in the use of the Software.

4. **Termination.** This License is effective until terminated. You may terminate this License at any time by destroying the Apple Software and related documentation and all copies thereof. This License will terminate immediately without notice from Apple if you fail to comply with any provision of this License. Upon termination you must destroy the Apple Software and related documentation and all copies thereof.

5. **Export Law Assurances.** You agree and certify that neither the Apple Software nor any other technical data received from Apple, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States.

6. **Government End Users.** If you are acquiring the Apple Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:

(i) if the Apple Software is supplied to the Department of Defense (DoD), the Apple Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Apple Software and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and

(ii) if the Apple Software is supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Apple Software and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

7. **Limited Warranty on Media.** Apple warrants the disks on which the Apple Software is recorded to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase as evidenced by a copy of the receipt. Apple's entire liability and your exclusive remedy will be replacement of the disk not meeting Apple's limited warranty and which is returned to Apple or an Apple authorized representative with a copy of the receipt. Apple will have no responsibility to replace a disk damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE DISKS, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

8. **Disclaimer of Warranty on Apple Software.** You expressly acknowledge and agree that use of the Apple Software is at your sole risk. The Apple Software and related documentation are provided "AS IS" and without warranty of any kind and Apple and Apple's Licensor(s) (for the purposes of provisions 8 and 9, Apple and Apple's Licensor(s) shall be collectively referred to as "Apple") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. APPLE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE APPLE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE APPLE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE APPLE SOFTWARE WILL BE CORRECTED. FURTHERMORE, APPLE DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE APPLE SOFTWARE OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE APPLE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

9. **Limitation of Liability.** UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL APPLE BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE APPLE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.
In no event shall Apple's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Apple Software.

10. **Controlling Law and Severability.** This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this License shall continue in full force and effect.

11. **Complete Agreement.** This License constitutes the entire agreement between the parties with respect to the use of the Apple Software and related documentation, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Apple.

7/15/91
001-0158-A