Macintosh
Technical Notes

Developer Technical Support

#311: What's New With AppleTalk Phase 2

Updated by: Rich Kubota April 1992
Written by: Rich Kubota & Scott Kuechle February 1992

This Technical Note discusses the new features of AppleTalk available for System 7.0 and AppleTalk version 57. The new features include support for the Flagship Naming Service and the AppleTalk Multiple Node Architecture. We present the Multiple Node Architecture and discuss the new calls available to applications. We also discuss the impact of the new architecture on AppleTalk Device files (ADEVs), and the changes necessary to make them multinode compatible. Finally, we discuss the Flagship Naming Service, along with the new AppleTalk Transitions. The new transitions notify a process of changes to the Flagship name, network cable range, router status, and processor speed.
Changes since February 1992: Provided additional detail on the implementation to the AAddNode, ADelNode, and AGetNodeRef calls including parameter offsets. Added sample code to check for existence of LAP Manager. Added Pascal source to determine whether the LAP Manager exists. Added warning to check the result from the LAPAddATQ function since the System 7 Tuner extension may not load AppleTalk resources. Corrected typographical errors. Added information on the discussion on the Speed Change AppleTalk Transition event. Added discussion regarding the 'atkv' gestalt selector. Sidebars highlight changes or additions to this document.

AppleTalk Multiple Node Architecture

Supporting multiple node addresses on a single machine connected to AppleTalk is a feature that has been created to support software applications such as AppleTalk Remote Access. Its implementation is general enough to be used by other applications as well.

Note: AppleTalk version 57 is required to support the AppleTalk Multiple Node Architecture. Version 57 is compatible with System 6.0.4 and greater. If you implement multinode functionality into your program you should also plan to include AppleTalk version 57 with your product. Contact Apple's Software Licensing department for information on licensing version 57. For testing purposes, AppleTalk version 57 can be installed by using the Network Software Installer v1.1, now available on the latest Developer CD, on AppleLink in the Developer Services Bulletin Board, and on the Internet through anonymous FTP to ftp.apple.com (130.43.2.3).

Software Licensing can be reached as follows:

Software Licensing
Apple Computer, Inc.
20525 Mariani Avenue, M/S 38-I
Cupertino, CA 95014
MCI: 312-5360
AppleLink: SW.LICENSE
Internet: SW.LICENSE@AppleLink.Apple.com
(408) 974-4667

What Is It?

Multiple Node AppleTalk provides network node addresses that are in addition to the normal (user node) DDP address assigned when AppleTalk is opened. These additional addresses have different characteristics from those of the user node address. They are not connected to the protocol stack above the data link layer. When an application acquires a multinode, the application has to supply a receive routine through which AppleTalk will deliver broadcasts and packets directed to that multinode address.

The number of multinode addresses that can be supported on one single machine is determined by a static limit imposed by the AppleTalk ADEV itself (for example, EtherTalk). The limit is currently 253 nodes for Apple's implementation of EtherTalk ($0, $FF, and $FE being invalid node addresses) and 254 for LocalTalk ($0 and $FF being invalid node addresses). The number of receive routines that .MPP supports is determined by the static limit of 256. If all of the multiple nodes acquired need to have unique receive routines, then only a maximum of 256 nodes can be acquired, even if the ADEV provides support for more than 256 nodes. .MPP will support the lesser of either the maximum of 256 receive routines, or the node limit imposed by the ADEV.

Outbound DDP packets can be created with a user-specified **source network, node,** and **socket** (normally equal to a multinode address) with the new Network Write call. With this capability and the packet reception rules described above, a single machine can effectively become several nodes on a network. The **user** node continues to function as it always has.

Things You Need to Know When Writing a Multinode Application

Two new .MPP driver control calls have been added to allow multinode applications to add and remove multinodes.

AddNode (csCode=262)

A user can request an extra node using a control call to the .MPP driver after it has opened. Only one node is acquired through each call.

```
Parameter Block:
      -->    24     ioRefNum     short        ; driver ref. number
      -->    26     csCode       short        ; always = AddNode (262)
      -->    36     reqNodeAddr  AddrBlock    ; the preferred address requested
                                              ; by the user.
      <--    40     actNodeAddr  AddrBlock    ; actual node address acquired.
      -->    44     recvRoutine  long         ; address of the receive routine for MPP
                                        ; to call during packet delivery
      -->    48     reqCableLo   short        ; the preferred range for the
      -->    50     reqCableHi   short        ; node being acquired.
      -->    52     reserved[70] char         ; 70 reserved bytes

AddrBlock:
      aNet         short                      ; network #
      aNode        unsigned char              ; node #
      aSocket      unsigned char              ; should be zero for this call.
```

This `.MPP AddNode` call must be made as an IMMEDIATE control call at system task time. The result code will be returned in the `ioResult` field in the parameter block. The result code –1021 indicates that the .MPP driver was unable to continue with the `AddNode` call because of the current state of .MPP. The caller should retry the `AddNode` call (the retry can be issued immediately after the AddNode call failed with –1021) until a node address is successfully attained or another error is returned. The .MPP driver will try to acquire the requested node address.

If the requested node address is zero, invalid, or already taken by another machine on the network, a random node address will be generated by the .MPP driver. The parameters `reqCableLo` and `reqCableHi` will be used only if there is no router on the network and all the node addresses in the network number specified in `NetHint` (the last used network number stored in parameter RAM) are taken up.

In this case, the .MPP driver will try to acquire a node address from the network range as specified by `reqCableLo` and `reqCableHi`. The network range is defined by the seed router on a network. If a specific cable range is not important to the application, set the `reqCableLo` and `reqCableHi` fields to zero. The `recvRoutine` is an address of a routine in the application to receive broadcasts and directed packets for the corresponding multinode.

Possible Error Codes:

```
       noErr                0            ; success
       tryAddNodeAgainErr  -1021         ; .MPP was not able to add node, try again.
       MNNotSupported      -1022         ; Multinode is not supported by
                                         ; the current ADEV
       noMoreMultiNodes    -1023         ; no node address is available on
                                         ; the network
```

RemoveNode (csCode=263)

This call removes a multinode address and must be made at system task time. Removal of the user node is not allowed.

```
Parameter Block:
       -->    24     ioRefNum      word          ; driver ref. number
       -->    26     csCode        word          ; always = RemoveNode (263)
       -->    36     NodeAddr      AddrBlock     ; node address to be deleted.
```

Possible Error Codes:

```
       noErr         0             ; success
       paramErr      -50           ; bad parameter passed
```

Receiving Packets

Broadcast packets are delivered to both the user's node and the multinodes on every machine. If several multinodes are acquired with the same `recvRoutine` address, the `recvRoutine`, listening for these multinodes, will only be called once in the case of a broadcast packet.

Multinode receive handlers should determine the number of bytes already read into the Read Header Area (RHA) by subtracting the beginning address of the RHA from the value in `A3` (see *Inside Macintosh* Volume II, page 326, for a description of the Read Header Area). `A3` points past the last byte read in the RHA. The offset of RHA from the top of the .MPP variables is defined by the equate `ToRHA` in the MPW include file ATalkEqu.a. The receive handler is expected to call `ReadRest` to read in the rest of the packet. In the case of LocalTalk, `ReadRest` should be done as soon as possible to avoid loss of the packet. Register `A4` contains the pointer to the `ReadPacket` and `ReadRest` routines in the ADEV.

To read in the rest of the packet:

```
JSR 2(A4)
```

On entry:

A3	pointer to a buffer to hold the bytes
D3	size of the buffer (word), which can be zero to throw away packet

On exit:

D0	modified
D1	modified
D2	preserved
D3	Equals zero if requested number of bytes were read; is less than zero if packet was –D3 bytes too large to fit in buffer and was truncated; is greater than zero if D3 bytes were not read (packet is smaller than buffer)
A0	preserved
A1	preserved
A2	preserved
A3	pointer to 1 byte after the last byte read

For more information about `ReadPacket` and `ReadRest`, refer to the *Inside Macintosh* Volume II, page 327.

A user can determine if a link is extended by using the `GetAppleTalkInfo` control call. The `Configuration` field returned by this call is a 32-bit word that describes the AppleTalk configuration. Bit number 15 (0 is LSB) is on if the link in use is extended. Refer to *Inside Macintosh* Volume VI, page 32-15.

Sending Datagrams Through Multinodes

To send packets through multinodes, use the new .MPP control call, `NetWrite`. `NetWrite` allows the owner of the multinode to specify a **source network, node,** and **socket** from which to send a datagram.

NetWrite (csCode=261)

```
Parameter Block:
      -->    26     csCode       word         ; always NetWrite (261)
      -->    29     checkSumFlag byte         ; checksum flag
      -->    30     wdsPointer   pointer      ; write data structure
```

Possible Error Codes:

```
      noErr         0             ; success
      ddpLenErr     -92           ; datagram length too big
      noBridgeErr   -93           ; no router found
      excessCollsns -95           ; excessive collisions on write
```

This call is very similar to the `WriteDDP` call. The key differences are as follows:

- The source socket is not specified in the parameter block. Instead it is specified along with the source network number and source node address in the DDP header pointed to by the write data structure. Furthermore, the socket need not be opened. Refer to *Inside Macintosh* Volume II, page 310, for a description of the Write Data Structure, WDS. It is important to note that the caller needs to fill in the WDS with the source network, source node, and source socket values. .MPP does not set these values for the NetWrite call.

- The `checkSumFlag` field has a slightly different meaning. If true (nonzero), then the checksum for the datagram will be calculated prior to transmission and placed into the DDP header of the packet. If false (zero), then the **checksum field is left alone** in the DDP header portion of the packet. Thus if a checksum is already present, it is passed along unmodified. For example, the AppleTalk Remote Access program sets this field to zero, since remote packets that it passes to the .MPP driver already have valid checksum fields. Finally, if the application desires no checksum, the checksum field in the DDP header in the WDS header must be set to zero.

Datagrams sent with this call are *always sent using a long DDP header*. Refer to *Inside AppleTalk*, Second Edition, page 4-16, for a description of the DDP header. Even if the destination node is on the same LocalTalk network, a long DDP datagram is used so that the source information can be specified. The LAP header source node field will always be equal to the user node address (`sysLapAddr`), regardless of the source node address in the DDP header.

AppleTalk Remote Access Network Number Remapping

Network applications should be careful not to pass network numbers as data in a network transaction. AppleTalk Remote Access performs limited network number remapping. If network numbers are passed as data, they will not get remapped. AppleTalk Remote Access recognizes network numbers in the DDP header and among the various standard protocol packets, NBP, ZIP, RTMP, and so on.

Is There a Router on the Network?

Do not assume that there are no routers on the network if your network number is zero. With AppleTalk Remote Access, you can be on network zero and be connected to a remote network. Network applications should look at the `SysABridge` low-memory global or use the `GetZoneList` or the `GetBridgeAddress` calls to determine if there is a router on the network.

New for AppleTalk ADEVs

Several new calls have been implemented into the .MPP driver for AppleTalk version 57. Two calls, `AOpen` and `AClose`, were built into AppleTalk version 54 and greater, and are also documented here. These calls notified the ADEV of changes in the status of the .MPP driver. For AppleTalk version 57, three new calls, `AAddNode, ADelNode, and AGetNodeRef,` plus a change to the `AGetInfo` call, were implemented to support the Multiple Node Architecture.

EtherTalk Phase 2, version 2.3, and TokenTalk phase 2, version 2.4, drivers support the new Multiple Node Architecture. Both drivers and AppleTalk version 57, are available through the Network Software Installer, version 1.1. As mentioned previously, AppleTalk version 57 and these drivers, are compatible with System 6.0.4 and greater. Note that the AppleTalk Remote Access product includes the EtherTalk Phase 2, version 2.3 driver, but *not* the multinode-compatible TokenTalk Phase 2, version 2.4, driver. Token Ring developers, who license

TokenTalk Phase 2, version 2.2 and earlier, should contact Apple's Software Licensing department .

The following information describes changes to the ADEV that are required for multinode compatibility. This information is of specific importance to developers of custom ADEVs. The ADEV can be expected to function under System 6.0.4 and greater. A version 3 ADEV **must** be used with AppleTalk version 57 or greater. Developers of custom ADEVs will want to contact Software Licensing to license AppleTalk version 57.

For compatibility with Multinode AppleTalk, the `'atlk'` resource of an ADEV must be modified to respond to these calls as described below. To determine whether an ADEV is multinode compatible, the .MPP driver makes an `AGetInfo` call to determine whether the ADEV version is 3 or greater. Any ADEVs responding with a version of 3 or greater must be prepared to respond to the new calls: `AAddNode, ADelNode, and AGetNodeRef.` See the *Macintosh AppleTalk Connections Programmer's Guide* for more information about writing an AppleTalk ADEV.

The desired architecture for a multinode-compatible ADEV is such that it delivers incoming packets to the LAP Manager along with an address reference number, `AddrRefNum.` The LAP Manager uses the `AddrRefNum` to locate the correct receive routine to process the packet. For broadcast packets, the LAP Manager handles multiple deliveries of the packet to each multinode receive routine.

The .MPP driver for AppleTalk version 57 supports the new control call to add and remove multinodes, along with the network write call which allows the specification of the source address. .MPP includes a modification in its write function to check for one multinode sending to another. .MPP supports inter-multinode transmission within the same machine. For example, the user node may want to send a packet to a multinode within the same system.

AGetInfo (D0=3)

The `AGetInfo` call should be modified to return the maximum number of **AppleTalk** nodes that can be provided by the atlk. This limit will be used by .MPP to control the number of multinodes that can be added on a single machine. The new interface is as follows:

Call:	`D1` (word)	length (in bytes) of reply buffer
	`A1 ->`	Ptr to GetInfo record buffer
Return:	`A1 ->`	Ptr to GetInfo record
	`D0`	nonzero if error (buffer is too small)

```
      AGetInfoRec = RECORD
<--   version:            INTEGER;     { version of ADEV, set to three (3) }
<--   length:             INTEGER;     { length of this record in bytes }
<--   speed:              LongInt;     { speed of link in bits/sec }
<--   BandWidth:          Byte;        { link speed weight factor }
<--   reserved:           Byte;        { set to zero }
<--   reserved:           Byte;        { set to zero }
<--   reserved:           Byte;        { set to zero }
<--   flags:              Byte;        { see below }
<--   linkAddrSize:       Byte;        { of link addr in bytes }
<--   linkAddress:        ARRAY[0..5] OF Byte;
<--   maxnodes:           INTEGER;
      END;

      flags: bit 7 = 1 if this is an extended AppleTalk, else 0
             bit 6 = 1 if the link is used for a router-only connection (reserved
                   for half-routing)
             bit 5 through 0 reserved, = 0
```

`maxnodes` is the total number of nodes (user node and multinodes) the ADEV supports. If a version 3 ADEV does not support multinodes, it must return 0 or 1 in the `maxnodes` field in `AGetInfoRec` and the ADEV will not be called to acquire multinodes. The version 3 ADEV will be called by .MPP in one of the following two ways to acquire the user node:

• if the ADEV returns a value of 0 in `maxnodes`, .MPP will issue Lap Write calls to the ADEV with D0 set to $FF indicating that ENQs should be sent to acquire the user node. .MPP is responsible for retries of ENQs to make sure no other nodes on the network already have this address. This was the method .MPP used to acquire the user node before multinodes were introduced. This method of sending ENQs must be available, even though the new `AAddNode` call is provided, to allow older versions of AppleTalk to function properly with a version 3 ADEV.

- if the ADEV returns a value of 1 in `maxnodes`, the new `AAddNode` function will be called by .MPP to acquire the user node.

For values of maxnode greater than 1, the new `AAddNode` function will be called by .MPP to acquire the additional multinodes.

AAddNode (D0=9)

This is a **new** call which is used to request the acquisition of an AppleTalk node address. It is called by the .MPP driver during the execution of the `AddNode` control call mentioned earlier. The ADEV is responsible for retrying enough ENQs to make sure no other nodes on the network already have the address. .MPP will make this call only during system task time.

Call:	`A0->`	parameter block
Return:	`D0`	= zero if address was acquired successfully
		≠ zero if no more addresses can be acquired

```
       atlkPBRec            Record csParam
-->    NetAddr              DS.L   1      ; offset 0x1C  24-bit node address to acquire
-->    NumTrys              DS.W   1      ; offset 0x20  # of tries for address
-->    DRVRPtr              DS.L   1      ; offset 0x22  ptr to .MPP vars
-->    PortUsePtr           DS.L   1      ; offset 0x26  ptr to port use byte
-->    AddrRefNum           DS.W   1      ; offset 0x2A  address ref number used by .MPP
                            EndR
```

The offset values describe the location of the fields from the beginning of the parameter block pointed to by `A0`. `atlkPBRec` is the standard parameter block record header for a _Control call. The field `NetAddr` is the 24-bit AppleTalk node address that should be acquired. The node number is in the least significant byte 0 of `NetAddr`. The network number is in bytes 1 and 2 of `NetAddr`; byte 3 is unused. `NumTrys` is the number of tries the atlk should send AARP probes on non-LocalTalk networks to verify that the address is not in use by another entity. On LocalTalk networks, `NumTrys` x 32 number of ENQs will be sent to verify an address.

`DRVRPtr` and `PortUsePtr` are normally passed when the atlk is called to perform a write function. For ADEVs that support multinodes, AppleTalk calls the new `AAddNode` function rather than the write function in the ADEV to send ENQs to acquire nodes. However, the values `DRVRPtr` and `PortUsePtr` are still required for the ADEV to function properly and are passed to the `AAddNode` call. `AddrRefNum` is a reference number passed in by .MPP. The ADEV must store each reference number with its corresponding multinode address. The use of the reference number is described in the following two sections.

For multinode-compatible ADEVs, .MPP will issue the first `AAddNode` call to acquire the user node. The `AddrRefNum` associated with the user node must be 0xFFFF. It is important to assign 0xFFFF as the `AddrRefNum` of the user node, and to disregard the `AddrRefNum` passed by .MPP for the user node. See the discussion at the end of the `ADelNode` description.

ADelNode (D0=10)

This is a **new** call which is used to remove an AppleTalk node address. It may be called by the .MPP driver to process the `RemoveNode` control call mentioned earlier.

Call: `A0->` parameter block

`NetAddr` contains the node address to be deleted
Return: `D0` = zero if address is removed successfully
≠ zero if address does not exist
`atlkPBRec.AddrRefNum = AddrRefNum` to be used by .MPP if the operation is successful

```
       atlkPBRec           Record csParam
-->    NetAddr             DS.L   1      ; offset 0x1C  24-bit node address to remove
<--    AddrRefNum          DS.W   1      ; offset 0x2A  AddrRefNum passed in by AAddNode
                                         ;              on return
                           EndR
```

The field `NetAddr` is the 24-bit AppleTalk node address that should be removed. As with the `AAddNode` selector, the node number is in the least significant byte 0 of `NetAddr`. The network number is in bytes 1 and 2 of `NetAddr`; byte 3 is unused. The address reference number, `AddrRefNum`, associated with the `NetAddr,` must be returned to .MPP in order for .MPP to clean up its data structures for the removed node address.

As mentioned above, a value of 0xFFFF must be returned to .MPP after deleting the user node. When the AppleTalk connection is started up for the first time on an extended network, the ADEV can expect to process an AAddNode request followed shortly by an `ADelNode` request. This results from the implementation of the provisional node address for the purpose of talking with the router to determine the valid network number range to which the node is connected. After obtaining the network range, .MPP issues the `ADelNode` call to delete the provisional node. The next `AAddNode` call will be to acquire the unique node ID for the user node. As mentioned previously, .MPP may pass a value different than 0XFFFF for the user node. The user node is acquired before any multi-node. The ADEV needs to keep track of the number of `AAddNode` and `ADelNode` calls issued to determine whether the user node is being acquired. Refer to *Inside AppleTalk*, Second Edition, page 4-8, for additional information.

AGetNodeRef (D0=11)

This is a **new** call which is used by .MPP to find out if a multinode address exists on the current ADEV. This call is currently used by .MPP to check if a write should be looped back to one of the other nodes on the machine (the packet does not actually need to be sent through the network) or should be sent to the ADEV for transmission.

Call: `A0->` parameter block
Return: `D0->` = zero if address does not exist on this machine
≠ zero if address exists on this machine
`atlkPBRec.AddrRefNum = AddrRefNum` (corresponding to the node address) if the operation is successful

```
       atlkPBRec           Record csParam
-->    NetAddr             DS.L   1      ; offset 0x1C  24-bit node address to remove
<--    AddrRefNum          DS.W   1      ; offset 0x2A  AddrRefNum passed in by AAddNode
                                         ;              on return
                           EndR
```

The field `NetAddr` is the 24-bit AppleTalk node address whose `AddrRefNum` is requested. The node number is in the least significant byte 0 of `NetAddr`. The network number is in bytes 1 and 2 of `NetAddr`; byte 3 is unused. The address reference number, `AddrRefNum`, associated with the `NetAddr,` must be returned to .MPP. Remember to return 0xFFFF as the `AddrRefNum` for the user node.

AOpen (D0=7)

Call:

```
        -->    D4.B         current port number
```

ADEVs should expect the AOpen call whenever the .MPP driver is being opened. This is a good time for the ADEV to register multicast addresses with the link layer. After this call is completed, .MPP is ready to receive packets. If the ADEV does not process this message, simply return, `RTN`.

Note that AOpen is not specific to the Multinode Architecture.

AClose (D0=8)

AClose is called only when .MPP is being closed (for example, .MPP is closed when the “inactive” option is selected in the Chooser or when the user switches links in Network CDEV). The ADEV should deregister any multicast addresses with the link layer at this time. After this `AClose` call is completed, the ADEV should not defend for any node addresses until .MPP reopens and acquires new node addresses. If the ADEV does not process this message, simply return, `RTN`.

Note that `AClose` is not specific to the Multinode Architecture.

For comparison, descriptions of `AInstall` and `AShutDown` are documented as follows:

AInstall (D0=1)

Call:

```
        -->    D1.L = value from PRAM (slot, ID, unused, atlk resource ID)
        <--    D1.L = high 3 bytes for parameter RAM returned by the ADEV,
        if no error
        <--    D0.W = error code
```

The `AInstall` call is made before .MPP is opened either during boot time or when the user switches links in Network CDEV. This call is made during system task time so that the ADEV is allowed to allocate memory, make file system calls, or load resources and so on. Note: `AOpen` call will be made during .MPP opens.

AShutDown (D0=2)

ADEVs should expect the `AShutDown` call to be made when the user switches links in the Network CDEV. The Network CDEV closes .MPP, which causes the `AClose` call to be made before the CDEV issues the `AShutDown` call. Note: the `AShutDown` call is always made during system task time; therefore, deleting memory, unloading resources, and file system calls can be done at this time.

Receiving Packets

The address reference number (`AddrRefNum`) associated with each node address must be passed to .MPP when delivering packets upward. When making the LAP Manager call `LReadDispatch` to deliver packets to AppleTalk, the ADEV must fill the high word of `D2` in with the address reference number, corresponding to the packet’s destination address (LAP node address in the LocalTalk case and DDP address in the non-LocalTalk case). There are a few special cases:

- In the case of broadcasts and packets directed to the user node, $FFFF (word) should be used as the address reference number.
- On non-LocalTalk networks, packets with DDP destination addresses matching neither the user node address nor any of the multinode addresses should still be delivered to the LAP Manager so that the router can forward the packet on to the appropriate network. In this case, high word of `D2` should be filled in with the address reference number, $FFFE, to indicate to MPP that this packet

is not for any of the nodes on the machine in the case of a router running on a machine on an extended network.

- On LocalTalk networks, the ADEV looks only at the LAP address; therefore, if the LAP address is not the user node, one of the multinodes, or a broadcast, the packet should be thrown away.

Defending Multinode Addresses

Both LocalTalk (RTS and CTS) and non-LocalTalk (AARP) ADEVs have to be modified to defend not only for the user node address but also for any active multinode addresses.

The 'atkv' Gestalt Selector

The `'atkv'` Gestalt selector is available beginning with AppleTalk version 56 to provide more complete version information regarding AppleTalk, and as an alternative to the existing `'atlk'` gestalt selector. Beginning with AppleTalk version 54, the `'atlk'` Gestalt selector was available to provide basic version information. The `'atlk'` selector is not available when AppleTalk is turned off in the Chooser. It is important to note that the information between the two resources are provided in a different manner. Calling Gestalt with the `'atlk'` selector provides the major revision version information in the low order byte of the function result. Calling Gestalt with the `'atkv'` selector provides the version information in a manner similar to the `'vers'` resource. The format of the `LONGINT` result is as follows:

```
byte;                               /* Major revision */
byte;                               /* Minor revision */
byte        development = 0x20,     /* Release stage  */
            alpha = 0x40,
            beta = 0x60,
            final = 0x80, /* or */ release = 0x80;
byte;                               /* Non-final release #  */
```

For example, passing the `'atkv'` selector in a Gestalt call under AppleTalk 57, gives the following `LONGINT` result: `0x39000800`.

With the release of the System 7 Tuner product, AppleTalk may not be loaded at startup, if prior to the previous shutdown, AppleTalk was turned off in the Chooser. Under this circumstance, the `'atkv'` selector is not available. If the `'atkv'` selector is not available under System 7, this is an indicator that AppleTalk cannot be turned without doing so in the Chooser and rebooting the system.

The AppleTalk Transition Queue

The AppleTalk transition queue keeps applications and other resident processes on the Macintosh informed of AppleTalk events, such as the opening and closing of AppleTalk drivers, or changes to the Flagship name (to be discussed later in this Note). A comprehensive discussion of the AppleTalk Transition Queue is presented in *Inside Macintosh* Volume VI, Chapter 32. New to the AppleTalk Transition Queue are messages regarding the Flagship Naming Service, the AppleTalk Multiple Node Architecture, changes to processor speed that may affect LocalTalk timers, and a transition to indicate change of the network cable range. At the end of this Technical Note is a sample Transition Queue procedure in both C and Pascal which includes the known transition selectors.

In addition, there is a sample Pascal source for determining whether the LAP Manager version 53 or greater exists. Calling LAPAddATQ for AppleTalk versions 52 and prior will result in a system crash since the LAP Manager is not implemented prior to AppleTalk version 53. The

Boolean function, LAPMgrExists, should be used instead of checking the low memory global LAPMgrPtr, $0B18.

Calling the AppleTalk Transition Queue

System 7.0 requires the use of the MPW version 3.2 interface files and libraries. The AppleTalk interface presents two new routines for calling all processes in the AppleTalk Transition Queue. Rather than use parameter block control calls as described in Technical Note #250, “AppleTalk Phase 2,” use the ATEvent procedure or the ATPreFlightEvent function to send transition notification to all queue elements. These procedures are discussed in *Inside Macintosh* Volume VI, Chapter 32.

> **Note**: You can call the ATEvent and ATPreFlightEvent routines only at virtual-memory safe time. See the Memory Management chapter of *Inside Macintosh* Volume VI, Chapter 28, for information on virtual memory.

Standard AppleTalk Transition Constants

You should use the following constants for the standard AppleTalk transitions:

```
CONST   ATTransOpen                 = 0; {open transition }
        ATTransClose                = 2; {prepare-to-close transition }
        ATTransClosePrep            = 3; {permission-to-close transition }
        ATTransCancelClose          = 4; {cancel-close transition }
        ATTransNetworkTransition    = 5; {.MPP Network ADEV Transition }
        ATTransNameChangeTellTask   = 6; {change-Flagship-name transition }
        ATTransNameChangeAskTask    = 7; {permission-to-change-Flagship-name transition }
        ATTransCancelNameChange     = 8; {cancel-change-Flagship-name transition }
        ATTransCableChange          = 'rnge' {cable range change transition }
        ATTransSpeedChange          = 'sped' {change in cpu speed }
```

The following information concerns the new transitions from `ATTransNetworkTransition` through `ATTransSpeedChange`.

The Flagship Naming Service

System 7.0 allows the user to enter a personalized name by which their system will be published when connected to an AppleTalk network. The System 'STR ' resource ID –16413 is used to hold this name. The name (listed as Macintosh Name) can be up to 31 characters in length and can be set using the File Sharing Control Panel Device. This resource is different from the Chooser name, System 'STR ' resource ID –16096. When providing network services for a workstation, the Flagship name should be used so that the user can personalize their workstation name while maintaining the use of the Chooser name for server connection identification. It’s important to note that the Flagship name resource is available only from System 7.0. **DTS recommends that applications do not change either of these 'STR ' resources.**

Applications taking advantage of this feature should implement an AppleTalk transition queue to stay informed as to changes to this name. Three new transitions have been defined to communicate Flagship name changes between applications and other resident processes. Support for the Flagship Naming Service Transitions is provided starting from AppleTalk version 56. Note that AppleTalk version 56 can be installed on pre-7.0 systems; however, the Flagship Naming Service is available only from System 7.0 and later.

The ATTransNameChangeAskTask Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent            RECORD   0
ReturnAddr          DS.L     1      ; address of caller
theEvent            DS.L     1      ; = 7; ID of ATTransNameChangeAskTask transaction
aqe                 DS.L     1      ; pointer to task record
infoPtr             DS.L     1      ; pointer to NameChangeInfo parameter block
```

```
                  ENDR
```

The `NameChangeInfo` record block is as follows:

```
NameChangeInfoPtr: ^NameChangeInfo;
NameChangeInfo     = RECORD
                  newObjStr:     Str32;       {new Flagship name to change to }
                  name           StringPtr;   {ptr to location to place ptr to process }
                                              {name }
                  END;
```

The `ATTransChangeNameAskTask` is issued under System 7.0 to inform Flagship clients that a process wants to change the Flagship name. Each AppleTalk Transition Queue element that processes the `ATTransChangeNameAskTask` can inspect the `NameChangeInfoPtr^.newObjStr` to determine the new Flagship name. If you deny the request, you must set the `NameChangeInfoPtr^.name` pointer with a pointer to a Pascal String buffer containing the name of your application **or** to the nil pointer. The AppleTalk Transition Queue process returns this pointer. The requesting application can display a dialog notifying the user of the name of the application that refused the process.

While processing this event, you may make synchronous calls to the Name Binding Protocol (NBP) to attempt to register your entity under the new name. It is recommended that you register an entity using the new Flagship name while handling the `ATTransChangeNameAskTask` event. You should not deregister an older entity at this point. Your routine must return a function result of 0 in the D0 register, indicating that it accepts the request to change the Flagship name, or a nonzero value, indicating that it denies the request.

DTS does not recommended that you change the Flagship name. The Sharing Setup CDEV does not handle this event and the Macintosh name will not be updated to reflect this change if the CDEV is open.

The ATTransNameChangeTellTask Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent          RECORD   0
ReturnAddr        DS.L     1      ; address of caller
theEvent          DS.L     1      ; = 6; ID of ATTransNameChangeTellTask transaction
aqe               DS.L     1      ; pointer to task record
infoPtr           DS.L     1      ; pointer to the new Flagship name
                  ENDR
```

A process uses ATEvent to send the `ATTransNameChangeTellTask` to notify AppleTalk Transition Queue clients that the Flagship name is being changed. The LAP Manager then calls every routine in the AppleTalk Transition Queue that the Flagship name is being changed.

When the AppleTalk Manager calls your routine with a `ATTransNameChangeTellTask` transition, the third item on the stack is a pointer to a Pascal string of the new Flagship name to be registered. Your process should deregister any entities under the old Flagship name at this time. You may make synchronous calls to NBP to deregister an entity. Return a result of 0 in the D0 register.

Note: When the AppleTalk Manager calls your process with a TellTask transition (that is, with a routine selector of `ATTransNameChangeTellTask`), you cannot prevent the Flagship name from being changed.

To send notification that your process intends to change the Flagship name, use the ATEvent function described above. Pass `ATTransNameChangeTellTask` as the event parameter and a pointer to the new Flagship name (Pascal string) as the infoPtr parameter.

The ATTransCancelNameChange Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent            RECORD   0
ReturnAddr          DS.L     1      ; address of caller
theEvent            DS.L     1      ; = 8; ID of ATTransCancelNameChange transaction
aqe                 DS.L     1      ; pointer to task record
                    ENDR
```

The `ATTransCancelNameChange` transition complements the `ATTransNameChangeAskTask` transition. Processes that acknowledged an `ATTransNameChangeAskTask` transition will be sent the `ATTransCancelNameChange` transition if a later process disallows the change of Flagship name. Your process should deregister any NBP entities registered during the `ATTransNameChangeAskTask` transition. You may make synchronous calls to NBP to deregister an entity. Return a result of 0 in the D0 register.

System 7.0 Sharing Setup cdev and Flagship Naming Service Interaction

The Flagship Naming Service is a new system service built into System 7. It is used to publish the workstation using the Flagship name. The Flagship Naming Service implements an AppleTalk Transition Queue element to respond to changes in the Flagship name. For example, the Sharing Setup cdev can be used to reset the Flagship name. When a new Macintosh (Flagship) name is entered in Sharing Setup, Sharing Setup sends an `ATTransNameChangeAskTask` message to the AppleTalk Transition Queue to request permission to change the Flagship name. The Flagship Naming Service receives the `ATTransNameChangeAskTask` transition and registers the new name under the type "Workstation" on the local network. Sharing Setup follows with the `ATTransNameChangeTellTask` to notify AppleTalk Transition Queue clients that a change in Flagship name will occur. The Flagship Naming Service responds by deregistering the workstation under the old Flagship name.

If an error occurs from the NBPRegister call, Flagship Naming Service returns a nonzero error (the error returned from NBPRegister) and a pointer to its name in the `NameChangeInfoPtr^.Name` field. Note that the Workstation name is still registered under the previous Flagship name at this point.

AppleTalk Remote Access Network Transition Event

AppleTalk Remote Access allows you to establish an AppleTalk connection between two Macintosh computers over standard telephone lines. If the Macintosh you dial-in to is on an AppleTalk network, such as LocalTalk or Ethernet, your Macintosh becomes, effectively, a node on that network. You are then able to use all the services on the new network. Given this new capability, it is important that services running on your Macintosh are notified when new AppleTalk connections are established and broken. For this reason, the ATTransNetworkTransition event has been added to AppleTalk version 57. This event can be expected under System 6.0.4 and greater.

Internally, both the AppleTalk Session Protocol (ASP) and the AppleTalk Data Stream Protocol (ADSP) have been modified to respond to this transition event. When a disconnect transition event is detected, these drivers close down sessions on the remote side of the connection.

The ATTransNetworkTransition Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent           RECORD   0
ReturnAddr         DS.L     1      ; address of caller
theEvent           DS.L     1      ; = 5; ID of ATTransNetworkTransition
aqe                DS.L     1      ; pointer to task record
infoPtr            DS.L     1      ; pointer to the TNetworkTransition record
                   ENDR
```

The `TNetworkTransition` record block is passed as follows:

```
TNetworkTransition RECORD   0
private            DS.L     1      ;   pointer   used   internally   by   AppleTalk   Remote   Access
netValidProc       DS.L     1      ; pointer to the network validate procedure
newConnectivity    DS.B     1      ; true = new connectivity, false = loss of connectivity
                   ENDR
```

Network Transition Event for AppleTalk Remote Access

Network transition events are generated by AppleTalk Remote Access to inform AppleTalk Transition Queue applications and resident processes that network connectivity has changed. The type of change is indicated by the `NetTransPtr^.newConnectivity` flag. If this flag is true, 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 AppleTalk Remote Access is connection based, it has knowledge of where a specific network exists. AppleTalk Remote Access can take advantage of that knowledge during a disconnect to inform AppleTalk Transition Queue clients that a network is no longer reachable. This information can be used by clients to age out connections immediately rather than waiting a potentially long period of time before discovering that the other end is no longer responding.

When AppleTalk Remote Access is disconnecting, it passes a network validation hook in the TNetworkTransition record, `NetTransPtr^.netValidProc`. A client can use the validation hook to ask AppleTalk Remote Access whether a specific network is still reachable. If the network is still reachable, the validate function will return true. A client can then continue to check other networks of interest until the status of each one has been determined. After a client is finished checking networks, control returns to AppleTalk Remote Access where the next AppleTalk Transition Queue client is called.

The information the network validate hook returns is valid only if a client has just been called as a result of a transition. A client can validate networks only when it has been called to handle a Network Transition Event. Note 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 with being called at this time (for example, don't make calls that move memory and don't make *synchronous* Preferred AppleTalk calls).

To check a network number for validity the client uses the network validate procedure to call AppleTalk Remote Access. This call is defined using C calling conventions as follows:

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

    thetrans    -->  pass in the TNetworkTransition record 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 AddrBlock as defined in Inside
                     Macintosh II, page 281:

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

Result codes	true	network is still reachable
	false	network is no longer reachable

AppleTalk Transition Queue handlers written in Pascal must implement glue code to use the netValidProc.

Cable Range Change Transition Event

The Cable Range Transition, `ATTransCableChange,` event informs AppleTalk Transition Queue processes that the cable range for the current network has changed. This can occur when a router is first seen, when the last router ages out, or when an RTMP broadcast packet is first received with a cable range that is different from the current range. The `ATTransCableChange` event is implemented beginning with AppleTalk version 57. Most applications should have no need to process this event. This event can be expected under System 6.0.4 and greater.

The ATTransCableChange Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent            RECORD   0
ReturnAddr          DS.L     1      ; address of caller
theEvent            DS.L     1      ; = 'rnge'; ID of ATTransCableChange
aqe                 DS.L     1      ; pointer to task record
infoPtr             DS.L     1      ; pointer to the TNetworkTransition record
                    ENDR
```

The `TNewCRTrans` record block is passed as follows:

```
TNewCRTrans         RECORD   0
newCableLo          DS.W     1      ; the new Cable Lo received from RTMP
newCableHi          DS.W     1      ; the new Cable Hi received from RTMP
                    ENDR
```

The cable range is a range of network numbers starting from the lowest network number through the highest network number defined by a seed router for a network. All node addresses acquired on a network must have a network number within the defined cable range. For non-extended networks, the lowest and the highest network numbers are the same. If the cable range on the network changes, for example, if the router on the network goes down, the Cable Range Change event will be issued with the parameters described earlier in this Technical Note.

After receiving the event, a multinode application should use the new cable range to check if all the multinodes it obtained prior to the event are still valid. For the invalid multinodes, the application should issue the .MPP `RemoveNode` control call to get rid of invalid nodes. The .MPP `AddNode` control call can be issued immediately after removing invalid nodes to obtain new valid multinodes in the new cable range. This cable range change transition event will be issued only during system task time.

The Speed Change Transition Event

The `ATTransSpeedChange` transition event is defined for applications that change CPU speed without rebooting, to notify time dependent processes that such change has taken place. Such speed change occurs when altering the cache states on the 68030 or 68040 CPU's, or with third party accelerator cards which allow speed changes on the fly via a Control Panel device. Any process which alters the effective CPU speed should use the ATEvent to notify processes of this change. Issue the `ATTransSpeedChange` event **only** at `SystemTask` time! Any process which is dependent on changes to the CPU speed should watch for this event. The speed change transition event will be issued only during system task time.

One time dependent routine is LocalTalk, whose low-level timer values must be recalculated when the CPU speed changes. Note that altering the cache state on the 68030 does not affect LocalTalk, however doing so on the 68040 does affect LocalTalk timers. This event must be sent by any application that toggles caching on the 68040 processor on the fly. If the cache is toggled and LocalTalk is not notified, a loss of network connection will result if LocalTalk is the current network connection. Note that only LocalTalk implemented in AppleTalk version 57 or greater recognizes the speed change transition event. Contact Apple Software Licensing for licensing AppleTalk version 57.

Regarding LocalTalk on the Mac Plus, the timing values are hard-coded in ROM regardless of the CPU speed. Vendors of accelerators for Mac Pluses should contact DTS for information on how to make LocalTalk work for you.

The ATTransSpeedChange Transition

From Assembly language, the stack upon calling looks as follows:

```
ATQEvent           RECORD   0
ReturnAddr         DS.L     1      ; address of caller
theEvent           DS.L     1      ; = 'sped'; ID of ATTransSpeedChange
aqe                DS.L     1      ; pointer to task record
                   ENDR
```

To notify LocalTalk that a change in processor speed has taken place, use the `ATEvent` procedure. Pass `ATTransSpeedChange` as the `event` parameter and a nil pointer as the `infoPtr` parameter. This event must be issued only at System Task time.

Sample Pascal Source to LAPMgrExists Function

It is important to check whether the LAP Manager exists before making LAP Manager calls like LAPAddATQ. The LAP Manager is implemented beginning with AppleTalk version 53. Rather than check the low memory global LAPMgrPtr, it is preferable to check for it's existence from a higher level. The following Pascal source demonstrates this technique.

```
FUNCTION GestaltAvailable: Boolean;
CONST
    _Gestalt = $A1AD;
BEGIN
    GestaltAvailable := TrapAvailable(_Gestalt);
    { TrapAvailable is documented in Inside Macintosh Volume VI, page 3-8 }
END;


FUNCTION AppleTalkVersion: Integer;
CONST
    versionRequested = 1; { version of SysEnvRec }
VAR
    refNum: INTEGER;
    world: SysEnvRec;
    attrib: LONGINT;
BEGIN
    AppleTalkVersion := 0; { default to no AppleTalk }
    IF OpenDriver('.MPP', refNum) = noErr THEN { open the AppleTalk driver }
        IF GestaltAvailable THEN
        BEGIN
            IF (Gestalt(gestaltAppleTalkVersion, attrib) = noErr) THEN
```

```
             AppleTalkVersion := BAND(attrib, $000000FF);
      END
      ELSE   { Gestalt or gestaltAppleTalkVersion selector isn't available }
         IF SysEnvirons(versionRequested, world) = noErr THEN
             AppleTalkVersion := world.atDrvrVersNum;
END;


FUNCTION LAPMgrExists: Boolean;
BEGIN
   { AppleTalk phase 2 is in AppleTalk version 53 and greater }
   LAPMgrExists := (AppleTalkVersion >= 53);
END;
```

Sample AppleTalk Transition Queue Function

A sample AppleTalk Transition Queue function has been implemented in both C and Pascal. These samples have been submitted as snippet code to appear on the Developer CD. Since Transition Queue handlers are called with a C style stack frame, the Pascal sample includes the necessary C glue.

Sample AppleTalk Transition Queue Function in C

The following is a sample AppleTalk Transition Queue handler for C programmers. To place the handler in the AppleTalk Transition Queue, define a structure of type myATQEntry in the main body of the application. Assign the SampleTransQueue function to the myATQEntry.CallAddr field. Use the LAPAddATQ function to add the handler to the AppleTalk Transition Queue. Remember to remove the handler with the LAPRmvATQ function before quitting the application.

Warning: The System 7 Tuner extension will not load AppleTalk resources if it detects that AppleTalk is off at boot time. Remember to check the result from the `LAPAddATQ` function to determine whether the handler was installed successfully.

The following code was written with MPW C v3.2.

```
/*-----------------------------------------------------------------------
 file: TransQueue.h
-----------------------------------------------------------------------*/

#include <AppleTalk.h>

/*
*  Transition queue routines are designed with C calling conventions in mind.
*  They are passed parameters with a C style stack and return values are expected
*  to be in register D0.
*/

#define ATTransOpen               0      /* .MPP just opened */
#define ATTransClose                 2         /* .MPP is closing */
#define ATTransClosePrep             3         /* OK for .MPP to close? */
#define ATTransCancelClose        4      /* .MPP close was canceled */
#define ATTransNetworkTransition  5      /* .MPP Network ADEV transition */
#define ATTransNameChangeTellTask 6      /* Flagship name is changing */
#define ATTransNameChangeAskTask  7      /* OK to change Flagship name */
```

```
#define ATTransCancelNameChange      8         /* Flagship name change was canceled */
#define ATTransCableChange        'rnge' /* Cable Range Change has occurred */
#define ATTransSpeedChange        'sped' /* Change in processor speed has occurred */

/*---------------------------------------------------------------------
       NBP Name Change Info record
---------------------------------------------------------------------*/
typedef struct NameChangeInfo {
   Str32  newObjStr;   /* new NBP name */
   Ptr name;        /* Ptr to location to place a pointer to Pascal string of */
                        /* name of process that NAK'd the event */
```

```
}
   NameChangeInfo, *NameChangePtr, **NameChangeHdl;

/*-----------------------------------------------------------------------
      Network Transition Info Record
-----------------------------------------------------------------------*/

typedef struct TNetworkTransition {
   Ptr    private;         /* pointer to private structure */
   ProcPtr   netValidProc;    /* pointer to network validation procedure */
   Boolean   newConnectivity; /* true = new connection, */
                              /* false = loss of connection */

}
   TNetworkTransition , *TNetworkTransitionPtr, **TNetworkTransitionHdl;

typedef   pascal long   (*NetworkTransitionProcPtr)(TNetworkTransitionPtr netTrans, \
                 unsigned long theNet);

/*-----------------------------------------------------------------------
      Cable Range Transition Info Record
-----------------------------------------------------------------------*/
typedef struct TNewCRTrans {
   short  newCableLo;   /* the new Cable Lo received from RTMP */
   short  newCableHi;   /* the new Cable Hi received from RTMP */
}
   TNewCRTrans , *TNewCRTransPtr, **TNewCRTransHdl;

/*-----------------------------------------------------------------------
      AppleTalk Transition Queue Element
-----------------------------------------------------------------------*/
typedef struct   myATQEntry {
   Ptr    qLink;    /* -> next queue element */
   short     qType;    /* unused */
   ProcPtr   CallAddr; /* -> transition procedure */
   Ptr    globs;    /* -> to user defined globals */
}
   myATQEntry, *myATQEntryPtr, **myATQEntryHdl;


/*-----------------------------------------------------------------------
 file: TransQueue.c
-----------------------------------------------------------------------*/

#include <Memory.h>
#include <AppleTalk.h>
#include "TransQueue.h"

long SampleTransQueue(long selector, myATQEntry *q, void *p)
{
   long                     returnVal = 0; /* return 0 for unrecognized events */
   NameChangePtr            myNameChangePtr;
   TNewCRTransPtr           myTNewCRTransPtr;
   TNetworkTransitionPtr    myTNetworkTransitionPtr;
   NetworkTransitionProcPtr   myNTProcPtr;
   StringPtr                newNamePtr;
   long                     checkThisNet;
   char                     **t;
   short                    myCableLo, myCableHi;


   /*
    * This is the dispatch part of the routine. We'll check the selector passed into
    * the task; its location is 4 bytes off the stack (selector).
    */
   switch(selector) {
      case ATTransOpen:
          /*
           *  Someone has opened the .MPP driver. This is where one would reset the
           *  application to its usual network state (i.e., you could register your
           *  NBP name here). Always return 0.
           */
          break;
```

```
        case ATTransClose:
            /*
             *  When this routine is called, .MPP is going to shut down no matter what we
             *  do. Handle that type of situation here (i.e., one could remove an NBP
             *  name and close down all sessions); 'p' will be nil. Return 0
             *  to indicate no error.
             */
            break;

        case ATTransClosePrep:
            /*
             *  This event gives us the chance to deny the closing of AppleTalk if we
             *  want. Returning a value of 0 means it's OK to close; nonzero
             *  indicates we'd rather not close at this time.
             *
             *  With this event, the parameter 'p' actually means something. 'p' in
             *  this event is a pointer to an address that can hold a pointer to a
             *  string of our choosing. This string indicates to the user which task
             *  would rather not close. If you don't want AppleTalk to close, but you
             *  don't have a name to stick in there, you MUST place a nil value in
             *  there instead.
             *
             *  (We're doing this all locally to this case because it's C and we can, so
             *  there.)
             */
            newNamePtr = (StringPtr)NewPtr(sizeof(Str32));

            /*
             *  Assume Ptr allocation successful.
             */

            newNamePtr = "\pBanana Mail"; /* this will either be an Ax reference or PC
             *  relative depending on compiler and options */

            /*
             *  get a new reference to the address we were passed (in a form we can use)
             */
            t = (char **) p;
            /*
             *  place the address of our string into the address we were passed
             */
            *t = (char *)newNamePtr;

            /*
             *  return a nonzero value so that AppleTalk knows we'd rather not close
             */
            returnVal = 1;
            break;

        case ATTransCancelClose:
            /*
             *  Just kidding, we didn't really want to cancel that AppleTalk closing
             *  after all. Reset all your network activities that you have disabled
     *  here (if any). In our case, we'll just fall through. 'p' will be nil.
             */
            break;

        case ATTransNetworkTransition:
            /*
             *  A Remote AppleTalk connection has been made or broken.
             *  'p' is a pointer to a TNetworkTransition record.
             *   Always return 0.
             */
            myTNetworkTransitionPtr = (TNetworkTransitionPtr)p;
            /*
             *  Check newConnectivity element to determine whether
             *  Remote Access is coming up or going down
             */
            if (myTNetworkTransitionPtr->newConnectivity) {
                /*
                 * Have a new connection
                 */
            }
            else {
```

```
            /*
             * Determine which network addresses need to be validated
             * and assign the value to checkThisNet.
             */
            checkThisNet = 0x1234FD80;  /* network 0x1234, node 0xFD, socket 0x80 */
            myNTProcPtr = (NetworkTransitionProcPtr)myTNetworkTransitionPtr->netValidProc;
            if ((*myNTProcPtr)(myTNetworkTransitionPtr, checkThisNet)) {
                /*
                 * Network is still valid
                 */
            }
            else {
                /*
                 * Network is no longer valid
                 */
            }
        }
        break;

    case ATTransNameChangeTellTask:
        /*
         *  Someone is changing the Flagship name and there is nothing we can do.
         *  The parameter 'p' is a pointer to a Pascal style string which holds new
         *  Flagship name.
         */
        newNamePtr = (StringPtr) p;

        /*
         *  You should deregister any previously registered NBP entries under the
         *  'old' Flagship name. Always return 0.
         */
        break;

    case ATTransNameChangeAskTask:
        /*
         *  Someone is messing with the Flagship name.
         *  With this event, the parameter 'p' actually means something. 'p' is
         *  a pointer to a NameChangeInfo record. The newObjStr field contains the
         *  new Flagship name. Try to register a new entity using the new Flagship name.
         *  Returning a value of 0 means it's OK to change the Flagship name.
         */
        myNameChangePtr = (NameChangePtr)p;

        /*
         *  If the NBPRegister is unsuccessful, return the error. You must also set
         *  p->name pointer with a pointer to a Pascal style string of the process
         *  name.
         */
        break;

    case ATTransCancelNameChange:
        /*
         *  Just kidding, we didn't really want to change that name after
         *  all. Remove new NBP entry registered under the ATTransNameChangeAskTask
         *  Transition. In our case,  we'll just fall through. 'p' will be nil. Remember
         *  to return 0.
         */
        break;

    case ATTransCableChange:
        /*
         *  The cable range for the network has changed. The pointer 'p' points
         *  to a structure with the new network range. (TNewCRTransPtr)p->newCableLo
         *  is the lowest value of the new network range. (TNewCRTransPtr)p->newCableHi
         *  is the highest value of the new network range. After handling this event,
         *  always return 0.
         */
```

```
            myTNewCRTransPtr = (TNewCRTransPtr)p;
            myCableLo = myTNewCRTransPtr->newCableLo;
            myCableHi = myTNewCRTransPtr->newCableHi;
            break;

        case ATTransSpeedChange:
            /*
```

```
           *  The processor speed has changed. Only LocalTalk responds to this event.
           *  We demonstrate this event for completeness only.
           *  Always return 0.
           */
          break;

      default:
          /*
           *  For future transition queue events. (and yes, Virginia, there will be more)
           */
          break;
  } /* end of switch */

  /*
   *  return value in register D0
   */
  return returnVal;
}
```

Sample AppleTalk Transition Queue Function in Pascal

The following is a sample AppleTalk Transition Queue handler for Pascal programmers. AppleTalk Transition Queue handlers are passed parameters using the C parameter passing convention. In addition, the 4 byte function result must be returned in register D0. To meet this requirement, a C procedure is used to call the handler, then to place the 4 byte result into register D0. The stub procedure listing follows the handler.

To place the handler in the AppleTalk Transition Queue, define a structure of type myATQEntry in the main body of the application. Assign the CallTransQueue C procedure to the myATQEntry.CallAddr field. Use the LAPAddATQ function to add the handler to the AppleTalk Transition Queue. Remember to remove the handler with the LAPRmvATQ function before quitting the application.

Warning: The System 7 Tuner extension will not load AppleTalk resources if it detects that AppleTalk is off at boot time. Remember to check the result from the `LAPAddATQ` function to determine whether the handler was installed successfully.

The following code was written with MPW Pascal and C v3.2.

```
{*****************************************************************************
 file: TransQueue.p
*****************************************************************************}

UNIT TransQueue;

INTERFACE

USES MemTypes, QuickDraw, OSIntF, AppleTalk;

CONST
(*  Comment the following 4 constants since they are already defined in the AppleTalk unit
   ATTransOpen                =  0; { .MPP is opening }
   ATTransClose               =  2; { .MPP is closing }
   ATTransClosePrep           =  3; { OK for .MPP to close? }
   ATTransCancelClose         =  4; { .MPP close was canceled }
*)
   ATTransNetworkTransition      =  5; { .MPP Network ADEV transition }
   ATTransNameChangeTellTask  =  6; { Flagship name is changing }
   ATTransNameChangeAskTask      =  7; { OK to change Flagship name }
   ATTransCancelNameChange    =  8; { Flagship name change was canceled }
   ATTransCableChange         =  'rnge'; { Cable Range Change has occurred }
   ATTransSpeedChange         =  'sped'; { Change in processor speed has occurred }

{---------------------------------------------------------------------
       NBP Name Change Info record
```

```
-----------------------------------------------------------------------}

TYPE

NameChangeInfo = RECORD
   newObjStr : Str32;      { new NBP name }
   name      : Ptr;        { Ptr to location to place a pointer to Pascal string of }
                           { name of process that NAK'd the event }
   END;
NameChangePtr = ^NameChangeInfo;
NameChangeHdl = ^NameChangePtr;

{-----------------------------------------------------------------------
       Network Transition Info Record
-----------------------------------------------------------------------}

TNetworkTransition = RECORD
   private       : Ptr;        { pointer to private structure }
   netValidProc  : ProcPtr;    { pointer to network validation procedure }
   newConnectivity : Boolean; { true = new connection, }
                               { false = loss of connection }
   END;

TNetworkTransitionPtr = ^TNetworkTransition;
TNetworkTransitionHdl = ^TNetworkTransitionPtr;

{ The netValidProc procedure has the following C interface. Note the }
{ CallNetValidProc C function, which follows. The C Glue routine allows the Pascal }
{ handler to make the call to the netValidProc function. }

{
typedef pascal long (*NetworkTransitionProcPtr)(TNetworkTransitionPtr netTrans, \
                   unsigned long theNet);
}
{-----------------------------------------------------------------------
       Cable Range Transition Info Record
-----------------------------------------------------------------------}
TNewCRTrans = RECORD
   newCableLo    : INTEGER;    { the new Cable Lo received from RTMP }
   newCableHi    : INTEGER;    { the new Cable Hi received from RTMP }
   END;
TNewCRTransPtr = ^TNewCRTrans;
TNewCRTransHdl = ^TNewCRTransPtr;

{-----------------------------------------------------------------------
       AppleTalk Transition Queue Element
-----------------------------------------------------------------------}
myATQEntry = RECORD
   qlink     : Ptr;        { -> next queue element }
   qType     : INTEGER;    { unused }
   CallAddr  : ProcPtr;    { -> transition procedure }
   globs     : Ptr;        { -> to user defined globals }
   END;
myATQEntryPtr = ^myATQEntry;
myATQEntryHdl = ^myATQEntryPtr;


{---------------- Prototypes --------------------}

FUNCTION SampleTransQueue (selector :LONGINT; q :myATQEntryPtr;  p :Ptr) : LONGINT;
{
*  Transition queue routines are designed with C calling conventions in mind.
*  They are passed parameters with a C style stack and return values are expected
*  to be in register D0. Note that the CallTransQueue C glue routine is used
*  to reverse the C style stack to Pascal style before calling the handler. The
*  procedure CallTransQueue follows this listing. To install this Trans Queue
*  handler, assign CallTransQueue to the CallAddr field, NOT SampleTransQueue.
```

```
}

FUNCTION CallNetValidProc(p : ProcPtr; netTrans : TNetworkTransitionPtr;
                        theNet : LONGINT) : LONGINT;
{
*  CallNetValidProc is used to call the netValidProc passed in the TNetworkTransition
*  record. Since Pascal cannot call the ProcPtr directly, a C glue routine is
*  used. This routine is defined following this listing.
}


IMPLEMENTATION

FUNCTION SampleTransQueue (selector :LONGINT; q :myATQEntryPtr;  p :Ptr) : LONGINT;

VAR
   returnVal                : LONGINT;
   myNameChgPtr             : NameChangePtr;
   myTNewCRTransPtr         : TNewCRTransPtr;
   myTNetworkTransitionPtr  : TNetworkTransitionPtr;
   newNamePtr               : StringPtr;
   processNameHdl           : StringHandle;
   myCableLo, myCableHi       : INTEGER;
   shortSelector            : INTEGER;
   checkThisNet             : LONGINT;


BEGIN
   returnVal := 0;                { return 0 for unrecognized events )
   {
    *  This is the dispatch part of the routine. We'll check the selector passed into
    *  the task; its location is 4 bytes off the stack (selector).
    }
   IF ((selector <= ATTransCancelNameChange) AND (selector >= ATTransOpen)) THEN
   {
    *  Check whether a numeric selector is being used whose known values are between
    *  8 and 0 so that we can implement a CASE statement with an INTEGER var.
    }
   BEGIN
      shortSelector := selector;
      CASE shortSelector OF
         ATTransOpen:
         BEGIN
            {
             *  Someone has opened the .MPP driver. This is where one would reset the
             *  application to its usual network state. (i.e., you could register your
             *  NBP name here). Always return 0.
             }
         END;

         ATTransClose:
         BEGIN
            {
             *  When this routine is called .MPP is going to shut down no matter what we
             *  do. Handle that type of situation here (i.e., one could remove an NBP
             *  name and close down all sessions). 'p' will be nil. Return 0 to
             *  indicate no error.
             }
         END;

         ATTransClosePrep:
         BEGIN
            {
             *  This event gives us the chance to deny the closing of AppleTalk IF we
             *  want. Returning a value of 0 means it's OK to close; nonzero
             *  indicates we'd rather not close at this time.
             *
             *  With this event, the parameter 'p' actually means something. 'p' in
             *  this event is a pointer to an address which can hold a pointer to a
             *  string of our choosing. This string indicates to the user which task
             *  would rather not close. If you don't want AppleTalk to close, but you
             *  don't have a name to stick in there,  you MUST place a nil value in
             *  there instead.
             }
```

```
{
 *  Get a new reference to the address we were passed (in a form we can use)
 *  (We're doing this all locally to this case because we can, so
 *  there.)
 }
processNameHdl := StringHandle(NewHandle(sizeof(Str32)));
```

```
        {
         *  place the address of our string into the address we were passed
         }
         := 'Banana Mail';
        Ptr(p) := Ptr(processNameHdl);

        {
         *  return a nonzero value so AppleTalk knows we'd rather not close
         }
        returnVal := 1;
    END;

    ATTransCancelClose:
    BEGIN
        {
         *  Just kidding, we didn't really want to cancel that AppleTalk closing
         *  after all. Reset all your network activities that you have disabled here
         *  (IF any). In our case, we'll just fall through. 'p' will be nil.
         }
    END;

    ATTransNetworkTransition:
    BEGIN
        {
         *  A Remote AppleTalk connection has been made or broken.
         *  'p' is a pointer to a TNetworkTransition record.
         *   Always return 0.
         }
        myTNetworkTransitionPtr := TNetworkTransitionPtr(p);
        {
         *  Check newConnectivity element to determine whether
         *  Remote Access is coming up or going down
         }
        if (myTNetworkTransitionPtr^.newConnectivity) THEN
        BEGIN
            {
             * Have a new connection
             }
        END
        ELSE
        BEGIN
            {
             * Determine which network addresses need to be validated
             * and assign the value to checkThisNet.
             }
            checkThisNet = $1234FD80;  /* network $1234, node $FD, socket $80 */
            if (CallNetValidProc(myTNetworkTransitionPtr^.netValidProc,
                    myTNetworkTransitionPtr, checkThisNet) <> 0) THEN
            BEGIN
                {
                 * Network is still valid
                 }
            END
            ELSE
            BEGIN
                {
                 * Network is no longer valid
                 }
            END;
        END;
    END;

    ATTransNameChangeTellTask:
    BEGIN
        {
         *  Someone is changing the Flagship name and there is nothing we can do.
```

```
 *  The parameter 'p' is a pointer to a Pascal style string which holds new
 *  Flagship name.
}
newNamePtr := StringPtr (p);

{
 *  You should deregister any previously registered NBP entries under the
```

```
              *  'old' Flagship name. Always return 0.
              }
          END;

          ATTransNameChangeAskTask:
          BEGIN
             {
              *  Someone is messing with the Flagship name.
              *  With this event, the parameter 'p' actually means something. 'p' is
              *  a pointer to a NameChangeInfo record. The newObjStr field contains the
              *  new Flagship name. Try to register a new entity using the new Flagship
              *  name. Returning a value of 0 means it's OK to change the Flagship name.
              }
             myNameChgPtr := NameChangePtr (p);

             {
              *  If the NBPRegister is unsuccessful, return the error. You must also set
              *  p->name pointer with a pointer to a string of the process name.
              }
          END;

          ATTransCancelNameChange:
          BEGIN
             {
              *  Just kidding, we didn't really want to cancel that name change after
              *  all. Remove new NBP entry registered under the
              *  ATTransNameChangeAskTask Transition. 'p' will be nil.
              *  Remember to return 0.
              }
          END;

          OTHERWISE
             returnVal := 0;
             {
              *  Just in case some other numeric selector is implemented.
              }
       END; { CASE }
    END
    ELSE IF (ResType(selector) = ATTransCableChange) THEN
    BEGIN
       {
        *  The cable range for the network has changed. The pointer 'p' points
        *  to a structure with the new network range. (TNewCRTransPtr)p->newCableLo
        *  is the lowest value of the new network range. (TNewCRTransPtr)p->newCableHi
        *  is the highest value of the new network range. After handling this event,
        *  always return 0.
        }
       myTNewCRTransPtr := TNewCRTransPtr(p);
       myCableLo := myTNewCRTransPtr^.newCableLo;
       myCableHi := myTNewCRTransPtr^.newCableHi;
       returnVal := 0;
    END
    ELSE IF (ResType(selector) = ATTransSpeedChange) THEN
    BEGIN
       {
        *  The processor speed has changed. Only LocalTalk responds to this event.
        *  We demonstrate this event for completeness only.
        *  Always return 0.
        }
        returnVal := 0;
    END; { IF }

    SampleTransQueue := returnVal;
END;

FUNCTION CallNetValidProc(p : ProcPtr; netTrans : TNetworkTransitionPtr;
                        theNet : LONGINT) : LONGINT; EXTERNAL;
```

```
END. { of UNIT }

/*****************************************************************************
 file: CGlue.c
*****************************************************************************/
```

```
#include <AppleTalk.h>

/*----------------------------------------------------------------------
       Network Transition Info Record
----------------------------------------------------------------------*/

typedef struct TNetworkTransition {
   Ptr    private;         /* pointer to private structure */
   ProcPtr   netValidProc;    /* pointer to network validation procedure */
   Boolean   newConnectivity; /* true = new connection, */
                              /* false = loss of connection */

}
   TNetworkTransition , *TNetworkTransitionPtr, **TNetworkTransitionHdl;

typedef   pascal long   (*NetworkTransitionProcPtr)(TNetworkTransitionPtr netTrans, \
                   unsigned long theNet);

/*----------------------------------------------------------------------
       AppleTalk Transition Queue Element
----------------------------------------------------------------------*/
typedef struct   myATQEntry {
   Ptr    qLink;    /* -> next queue element */
   short     qType;    /* unused */
   ProcPtr   CallAddr; /* -> transition procedure */
   Ptr    globs;    /* -> to user defined globals */
}
   myATQEntry, *myATQEntryPtr, **myATQEntryHdl;

/*----------------------------------------------------------------------
       Prototypes
----------------------------------------------------------------------*/
pascal long  SampleTransQueue (long selector, myATQEntry *q, void *p);
long CALLTRANSQUEUE(long selector, myATQEntry *q, void *p);
/* Capitalize CALLTRANSQUEUE so that linker can match this entry with */
/* the pascal call */
pascal long CallNetValidProc(ProcPtr p, TNetworkTransitionPtr netTrans, long theNet);


long CALLTRANSQUEUE(long selector, myATQEntry *q, void *p)
/* CallTransQueue sets up the pascal stack for the SampleTransQueue handler */
/* then puts the result into D0 */
{
   return(SampleTransQueue(selector, q, p));
}

pascal long CallNetValidProc(ProcPtr p, TNetworkTransitionPtr netTrans, long theNet)
/* CallNetValidProc is used to call the netValidProc pointed to by ProcPtr p. */
{
   NetworkTransitionProcPtr   myNTProcPtr;

   myNTProcPtr = (NetworkTransitionProcPtr)p;
   return ((*myNTProcPtr)(netTrans, theNet));
}
```

Further Reference:

- *Inside AppleTalk,* Second Edition, Addison-Wesley
- *Inside Macintosh*, Volume II, The AppleTalk Manager, Addison-Wesley
- *Inside Macintosh*, Volume V, The AppleTalk Manager, Addison-Wesley
- *Inside Macintosh*, Volume VI, The AppleTalk Manager, Addison-Wesley
- *Macintosh AppleTalk Connections Programmer's Guide*, Final Draft 2, Apple Computer, Inc. (M7056/A)
- *AppleTalk Phase 2 Protocol Specification*, Apple Computer, Inc. (C0144LL/A)
- Macintosh Portable Developer Notes (DTS)
- *AppleTalk Remote Access Developer's Toolkit,* Apple Computer, Inc. (R0128LL/A)
- Technical Note #250, AppleTalk Phase 2, (DTS)
- *Alternate AppleTalk for System 7.0*, (DTS)