

NOTE: This Technical Note has been [retired](#). Please see the [Technical Notes](#) page for current documentation.



Technical Note NW03

AppleTalk Phase 2 on the Macintosh

## CONTENTS

[Introduction](#)

[What is AppleTalk Phase 2?](#)

[Are AppleTalk Phase 2 Drivers Present?](#)

[Calls to the .ATP Driver](#)

[Name Binding Protocol \(NBP\) Change: Wildcard Lookup](#)

[Obtaining Zone Information Using the New .XPP Driver Calls](#)

[Potential Nastiness](#)

[The AppleTalk Transition Queue](#)

[Making a LAP Manager Call](#)

[The AppleTalk Transition Queue LAP Calls](#)

[The Transitions](#)

[Potential Compatibility Problems](#)

[References](#)

[Downloadables](#)

This Technical Note discusses the new features and calls available with AppleTalk Phase 2.

[Aug 01 1989]

---

## Introduction

AppleTalk Phase 2 is only available on Macintosh Plus or later Macintosh platforms, and it requires the installation of AppleTalk file V53, or greater. Both EtherTalk 2.0 and TokenTalk 2.0 automatically install this AppleTalk file. Developer Technical Support can supply the Phase 2 drivers for development use; however, if you need to include the Phase 2 drivers in your product, you must license them from Software Licensing. For more information, contact:

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

[Back to top](#)

## What is AppleTalk Phase 2?

AppleTalk Phase 2 contains enhancements to the routing and naming services of AppleTalk. Among these enhancements is the ability to create AppleTalk networks which support more than 254 nodes, and to do so in a manner that is, to the greatest extent possible, compatible with current AppleTalk implementations and applications. Multiple zones per network are now supported, and users can choose their machine's zone. Benefits include improved network traffic and better router selection. New calls and features have been implemented with this enhancement and are documented in this Note.

[Back to top](#)

## Are AppleTalk Phase 2 Drivers Present?

So you want to use these new calls and features, but can you? First, one needs to check to see if the node is running AppleTalk Phase 2. There are two ways this can be accomplished. The easiest way is to make a `_SysEnviron`s call and check the returned `atDrvVersNum` field. If this byte is greater than or equal to 53, then AppleTalk Phase 2 drivers are present. If, for some reason, a `_SysEnviron`s call is not practical or otherwise not possible, one can check 7 bytes off the device control entry for the .MPP driver for a single byte, which is the driver version (actually the low byte of the `qFlags` field of `DctlQHdr` in the DCE). Again, if this byte is 53 or greater, AppleTalk Phase 2 is present, and the calls and features outlined in this Note may be used.

## Calls to the .MPP Driver

AppleTalk Phase 2 introduces many new variables, and we highly recommend that you use the new `GetAppleTalkInfo` call instead of looking at MPP globals directly. In addition, on a Macintosh running the AppleTalk Internet Router software, there may be more than one .MPP driver present. These additional drivers can be found by walking through the unit table (`UTableBase $11C`) and looking for drivers named .MPP other than at unit slot 9. Generally, the only port of interest to you is the user port, reflected in this call as `PortID 0` with a `refnum` of `-10`.

## GetAppleTalkInfo

```
Parameter Block
--> 26  csCode    word    ; always GetAppleTalkInfo (258)
--> 28  Version   word    ; requested info version
<-- 30  VarsPtr   pointer   ; pointer to well known MPP vars
<-- 34  DCEPtr   pointer   ; pointer to MPP DCE
<-- 38  PortID   word     ; port number [0..7]
<-- 40  Configuration long  ; 32-bit configuration word
<-- 44  SelfSend word     ; non zero if SelfSend enabled
<-- 46  NetLo    word     ; low value of network range
<-- 48  NetHi    word     ; high value of network range
<-- 50  OurAddr  long     ; our 24-bit AppleTalk address
<-- 54  RouterAddr long   ; 24-bit address of (last) router
<-- 58  NumOfPHs word     ; max. number of protocol handlers
<-- 60  NumOfSkts word     ; max. number of static sockets
<-- 62  NumNBPEs word     ; max. concurrent NBP requests
<-- 64  NTQueue  pointer   ; pointer to registered name queue
<-- 68  *LALength word     ; length in bytes of data link addr
--> 70  *LinkAddr pointer   ; data link address returned
--> 74  *ZoneName pointer   ; zone name returned
```

This call is provided to simplify the task of obtaining details about the current AppleTalk network connection. The following are the parameters which this call returns:

`Version` is passed by the caller. The concept is similar to one used by `_SysEnviron`s, where a version ID is passed to the function to return a requested level of information. If the driver cannot respond because this number is too high, `paramErr` is returned. The current version number is 1.

`VarsPtr` is the pointer to AppleTalk variables. This points to the well known `sysLapAddr` and read header area or RHA.. This pointer may not be equal to `$2D8 (ABusVars)` for other than port 0.

`DCEPtr` is a pointer to the driver's device control entry. See the Device Manager chapters of *Inside Macintosh* for details.

`PortID` is the port number, and it is always zero, unless a router is active and a driver `refnum` other than `-10` is used.

`Configuration` is a 32-bit word of configuration flags. Currently only the following bits are returned:

31 (`SrvAdrBit`) is true if server node-ID was requested at open time. Note that even if server address is requested, it may be ignored by those ADEVs which do not honor it (i.e., EtherTalk, TokenTalk, etc.).

30 (`RouterBit`) is true if an AppleTalk Internet Router was loaded at system startup. Note that a router may be loaded, but not active.

7 (`BadZoneHintBit`) is true if the node's zone name hint is invalid, thus causing a default zone to be selected.

6 (`OneZoneBit`) is true if only one zone is assigned to an extended network.

`SelfSend` (the ability for a node to send packets to itself) is non-zero if this feature is currently enabled.

`NetLo` is the low value of the network range. Non-extended networks always have a range of exactly one network, if the network number is known.

`NetHi` is the high value of the network range.

`OurAddr` is the 24-bit AppleTalk network address of the node. The most significant byte is always zero.

`RouterAddr` is the 24-bit AppleTalk address of the router from which we last heard. Users should always use this address when attempting to communicate directly with a router.

`NumOfPHs`, are maximum capacities for the driver. They are number of protocol

`NumOfSkts`, and handlers, number of static sockets, and number of concurrent NBP

`NumNBPEs` requests allowed, respectively.

`NTQueue` is a pointer to the registered names table queue. See *Inside Macintosh*, Volume II, The AppleTalk Manager, for NT Queue details.

`LALength` is passed by the caller to indicate how much (if any) of the data link address is to be copied to a user-supplied buffer (pointed to by `LinkAddr`). The actual length is returned by the driver. If the caller requests more bytes than the actual number, then data in the buffer after the address is undefined. The caller is responsible for providing sufficient buffer space.

`LinkAddr` is a pointer to a user-supplied buffer into which the data link address data is copied. If the pointer is `NIL`, no data is copied.

`ZoneName` is a pointer to a user-supplied buffer into which the node's stored zone name is copied. If the pointer is `NIL`, no data is copied. The user buffer must be 33 bytes or more in size.

[Back to top](#)

## Calls to the .ATP Driver

### KillAllGetReq

```
Parameter Block
--> 26   csCode    word    ; always KillAllGetReq (259)
--> 28   atpSocket byte    ; socket on which to kill all pending
```

`KillAllGetReq` aborts all outstanding `GetRequest` calls on the specified socket and completes them with `reqAborted` errors (it does not close the specified socket, it only kills all pending `GetRequest` calls on that socket). To kill all the `GetRequest` calls, simply pass the desired socket number in the `atpSocket` field.

`cbNotFound` control block not found (-1102)

### Setting the TRel Timer in SendRequest Calls

It is now possible to set the TRel timer in `SendRequest` or `NSendRequest` calls with ATP XO (exactly once) service so as not to be locked into the pre-AppleTalk Phase 2 time of 30 seconds. This is done by setting bit 2 in the `atpFlags` field to indicate to the driver that an extended parameter block is being used. Make a standard `SendRequest` call, but add the timeout constant desired in the new `TRelTime` field byte of the parameter block. Both nodes must be running AppleTalk Phase 2 for this feature to be supported.

The timeout constants are enumerated as follows in the lower three bits of the `TRelTime` (\$32 offset) byte:

```
000 $0 TRel timer set to 30 seconds
001 $1 TRel timer set to one minute
010 $2 TRel timer set to two minutes
011 $3 TRel timer set to four minutes
100 $4 TRel timer set to eight minutes
```

All other values are reserved.

```
Parameter Block
```

[Back to top](#)

## Name Binding Protocol (NBP) Change: Wildcard Lookup

In AppleTalk Phase 2, NBP is enhanced to provide additional wildcard support. The double tilde (~), \$C5, is now reserved in the object name and type strings and used in a lookup to mean a match of zero or more characters. Thus "~cliff" matches "cliff," "the cliff," "grazing off the cliff," etc., and "123~456" matches "123456," "123zz456," etc. At most one ~ is allowed in any string. A single ~ has the same meaning as a single =, which also must continue to be accepted. The ~ has no special meaning in zone names. Clients of NBP must be aware that "old" (pre-AppleTalk Phase 2) nodes may not process this new wildcard feature correctly. This feature should probably only be used when it is known that the responding devices are running Phase 2 drivers as well.

[Back to top](#)

## Obtaining Zone Information Using the New .XPP Driver Calls

Previously, Zone Information Protocol (ZIP) functions were accomplished via direct ATP calls to the local router. It was rather nasty business, having to mess with the `ATPUserData` on subsequent calls to retain state information. We now recommend the use of the following XPP driver calls to access ZIP. Old ATP calls will continue to be supported for compatibility. It should also be noted that with Phase 2 drivers present, the .XPP driver is automatically opened by MPP.

### GetZoneList

```
Parameter Block
--> 26   csCode    word    ; always xCall (246)
--> 28   xppSubCode word    ; always zipGetZoneList (6)
--> 30   xppTimeout byte    ; retry interval (seconds)
--> 31   xppRetry   byte    ; retry count
--> 32   <unused>   word    ; word space for rent. see the super.
--> 34   zipBuffPtr pointer  ; pointer to buffer (must be 578 bytes)
<-- 38   zipNumZones word    ; no. of zone names in this response
<-- 40   zipLastFlag byte    ; non-zero if no more zones
--> 41   <unused>   byte    ; filler
--> 42   zipInfoField 70 bytes ; on initial call, set first word to zero
```

GetZoneList is used to obtain a complete list of zones on the internet. ZipBuffPtr points to a buffer that must be 578 bytes (ATPMaxData) in length. The actual number of zone names returned in the buffer is returned in zipNumZones. The fields xppTimeout and xppRetry contain the ATP retry interval (in seconds) and count, respectively.

The first time this call is made, the first word of the zipInfoField should be set to zero. When the call completes, zipLastFlag is non-zero if all the zone names fit into the buffer. If not, the call should be made again immediately, without changing zipInfoField (it contains state information needed to get the next part of the list). The call should be repeated until zipLastFlag is non-zero. The 70-byte zipInfoField must always be allocated at the end of the parameter block.

Result codes noErr No Error (0)

noBridgeErr No router is available (-93)

ReqFailed SendRequest failed; retry count exceeded (-1096)

Following are short examples of using GetZoneList.

#### Pascal

```

const
{ csCodes for new .XPP driver calls }
xCall = 246;

{ xppSubCodes }
zipGetLocalZones = 5;
zipGetZoneList = 6;
zipGetMyZone = 7;

type
{ offsets for xCall queue elements }
xCallParam = packed record
  qLink: QElemPtr;
  qType: INTEGER;
  ioTrap: INTEGER;
  ioCmdAddr: Ptr;
  ioCompletion: ProcPtr;
  ioResult: OsErr;
  ioNamePtr: StringPtr;
  ioVRefNum: INTEGER;
  ioRefNum: INTEGER;
  csCode: INTEGER;
  xppSubCode: INTEGER;
  xppTimeOut: Byte;
  xppRetry: Byte;
  filler: INTEGER;
  zipBuffPtr: Ptr;
  zipNumZones: INTEGER;
  zipLastFlag: INTEGER;
  zipInfoField: packed array[1..70] of Byte;
end;

procedure doGetZoneListPhs2;

type
  XCallParamPtr = ^XCallParam;
var
  xpb: XCallParamPtr;
  resultCode: OsErr;
  zoneBuffer, theBufferPtr: Ptr;
  totalZones: integer;
begin
  xpb := XCallParamPtr(NewPtr(sizeof(XCallParam)));

  zoneBuffer := NewPtr(33 * 100);    { size of maxstring * 100 zones }

  theBufferPtr := NewPtr(578);      { size of atpMaxData }

  xpb^.zipInfoField[1] := 0;        { ALWAYS 0 on first call.  contains state info
                                     on subsequent calls }

  xpb^.zipInfoField[2] := 0;        { ALWAYS 0 on first call.  contains state info
                                     on subsequent calls }

  xpb^.ioRefNum := XPPRefNum;       { driver refNum -41 }

  xpb^.csCode := xCall;
  xpb^.xppSubCode := zipGetZoneList;
  xpb^.xppTimeOut := 3;

```

```

xpb^.xppRetry := 4;
xpb^.zipBuffPtr := Ptr(theBufferPtr);    { this buffer will be filled with }
                                         { packed zone names }

{ initialization for loop }
xpb^.zipLastFlag := 0;
totalZones := 0;
resultCode := 0;

{ loop until zipLastFlag is non-zero or an error occurs }
while ((xpb^.zipLastFlag = 0) and (resultCode = 0)) do
begin
    resultCode := PBControl(ParmBlkPtr(xpb), false);

    if (resultCode = noErr) then
    begin
        totalZones := xpb^.zipNumZones + totalZones;
        { you can now copy the zone names into the zoneBuffer }
    end;
end;
DisposPtr(theBufferPtr);
DisposPtr(zoneBuffer);
DisposPtr(Ptr(xpb));

```

C

```

/*
csCodes for new .XPP driver calls
*/
#define xCall          246

/*
xppSubCodes
*/
#define zipGetLocalZones    5
#define zipGetZoneList     6
#define zipGetMyZone       7

/*
offsets for xCall queue elements
*/
typedef struct
{
    QElemPtr      qLink;
    short         qType;
    short         ioTrap;
    Ptr           ioCmdAddr;
    ProcPtr       ioCompletion;
    OSErr         ioResult;
    StringPtr     ioNamePtr;
    short         ioVRefNum;
    short         ioRefNum;
    short         csCode;
    short         xppSubCode;
    unsigned char xppTimeOut;
    unsigned char xppRetry;
    short         filler;
    Ptr           zipBuffPtr;
    short         zipNumZones;
    short         zipLastFlag;
    unsigned char zipInfoField[70];
} xCallParam;

doGetZoneListPhs2()
{
    xCallParam      xpb;
    OSErr           resultCode = 0;
    Ptr             zoneBuffer, theBufferPtr;
    short           totalZones = 0;

    zoneBuffer = NewPtr(33*100);    /* size of maxstring * 100 zones */

    theBufferPtr = NewPtr(578);    /* size of atpMaxData */

    xpb.zipInfoField[0] = 0;        /* ALWAYS 0 on first call.  contains
state info on subsequent calls */

```

```

    xpb.zipInfoField[1] = 0;          /* ALWAYS 0 on first call.  contains
state info on subsequent calls */

    /* initialization for loop */
    xpb.zipLastFlag = 0;

    xpb.ioCRefNum = XPPRefNum;        /* driver refNum -41 */
    xpb.csCode = xCall;
    xpb.xppSubCode = zipGetZoneList;
    xpb.xppTimeout = 3;
    xpb.xppRetry = 4;
    xpb.zipBuffPtr = (Ptr) theBufferPtr; /* this buffer will be filled with
the packed zone names */

    /* loop until zipLastFlag is non-zero or an error occurs */
    while(xpb.zipLastFlag == 0 && resultCode == 0) {

        resultCode = PBControl(&xpb, false);

        if(resultCode == noErr) {
            totalZones += xpb.zipNumZones;
            /* you can now copy the zone names into the zoneBuffer */
        }
        DisposPtr(theBufferPtr);
        DisposPtr(zoneBuffer);
    }
}

```

### GetLocalZones

```

Parameter Block
--> 26   csCode      word      ; always xCall (246)
--> 28   xppSubCode  word      ; always zipGetLocalZones (5)
--> 30   xppTimeout  byte      ; retry interval (seconds)
--> 31   xppRetry    byte      ; retry count
32      <unused>     word      ; filler
--> 34   zipBuffPtr  pointer    ; pointer to buffer (must be 578 bytes)
<-- 38   zipNumZones word      ; no. of zone names in this response
<-- 40   zipLastFlag byte      ; non-zero if no more zones
41      <unused>     byte      ; filler
--> 42   zipInfoField 70 bytes   ; on initial call, set first word to zero

```

This call has the same format and procedures as `GetZoneList`, the difference being that `GetLocalZones` returns a list of zone names currently defined only on the node's network cable rather than the entire network. The 70-byte `zipInfoField` must always be allocated at the end of the parameter block.

Result codes `noErr` No Error (0)

`noBridgeErr` No router is available (-93)

`ReqFailed` SendRequest failed; retry count exceeded (-1096)

#### Note:

The examples for `GetZoneList` will also work for `GetLocalZones` if you substitute the `xppSubCode`.

### GetMyZone

```

Parameter Block
--> 26   csCode      word      ; always xCall (246)
--> 28   xppSubCode  word      ; always zipGetMyZone (7)
--> 34   zipBuffPtr  pointer    ; pointer to buffer (must be 33 bytes)
--> 42   zipInfoField 70 bytes   ; first word must be set to zero on every
call

```

`GetMyZone` returns the node's AppleTalk zone name. This is the zone in which all of the node's network visible entities are registered. `ZipBuffPtr` points to a buffer that must be 33 bytes in length. If `noBridgeErr` is returned by the call, there is no internet, and the zone name is effectively an asterisk (\*). The 70-byte `zipInfoField` must always be allocated at the end of the parameter block.

Result codes `noErr` No Error (0)

`ReqFailed` SendRequest failed; retry count exceeded (-1096)

Following are short examples of using `GetMyZone`.

**Pascal**

```

procedure getMyZonePhs2;
var
  xpb:xCallParam;
  resultCode :OSErr;
  myZoneNameBuffer:Ptr;
begin
  myZoneNameBuffer := NewPtr(33);

  xpb.ioCRefNum := xppRefNum;
  xpb.csCode := xCall;
  xpb.xppSubCode := zipGetMyZone;
  xpb.zipBuffPtr := myZoneNameBuffer;
  xpb.zipInfoField[1] := 0;      { ALWAYS 0 }
  xpb.zipInfoField[2] := 0;      { ALWAYS 0 }
  resultCode := PBControl(@xpb, false);

```

**C**

```

getMyZonePhs2()
{
  xCallParam    xpb;
  OSErr         resultCode;
  Ptr           myZoneNameBuffer;

  myZoneNameBuffer := NewPtr(33);

  xpb.ioCRefNum = xppRefNum;
  xpb.csCode = xCall;
  xpb.xppSubCode = zipGetMyZone;
  xpb.zipBuffPtr = (Ptr) myZoneNameBuffer;
  xpb.zipInfoField[0] = 0;      /* ALWAYS 0 */
  xpb.zipInfoField[1] = 0;      /* ALWAYS 0 */
  resultCode = PBControl(&xpb, false);

```

[Back to top](#)

## Potential Nastiness

When running on a node with Phase 2 compatible drivers, we always recommend using the .XPP calls outlined in the previous section. Care was taken to keep backward compatibility with the already existing ATP ZIP calls (they are being trapped out with the Phase 2 drivers), but there are problems about which you should be aware.

- Do not rely on checking the TID (transaction ID validity bit) or other bits in the `atpFlags`, as some of you have been doing. The `atpFlags` are not guaranteed to be correct on an ATP ZIP call with a Phase 2 driver present.
- Do not repeatedly stuff the router address back into the `ATPPParamBlock` on subsequent ATP ZIP `GetZoneList` calls. There exists the possibility of concurrent `GetZoneList` calls being made by other tasks and wrong router addresses being used (a small possibility yes, but it does exist).

[Back to top](#)

## The AppleTalk Transition Queue

To keep applications and other resident processes on the Macintosh informed of AppleTalk events, such as the opening and closing of AppleTalk drivers, a new transition queue has been implemented. Processes can register themselves with the AppleTalk Transition Queue, and when a significant event occurs, they will be notified of this fact. Each transition queue element has the following MPW assembly-language format:

AeQentry	RECORD	0	
QLink	DS.L	1	; link to next record
QType	DS.W	1	; unused
CallAddr	DS.L	1	; pointer to task record

Three calls have been provided in the LAP Manager to add an entry, remove an entry, and return a pointer to the AppleTalk event queue header. The method for making calls to the LAP Manager is explained in the following section. The queue is maintained by the LAP Manager, so it can be active even when AppleTalk (MPP) is not.

[Back to top](#)

## Making a LAP Manager Call

The LAP Manager is installed in the system heap at startup time, before the AppleTalk Manager opens the .MPP driver (hence, the inclusion of the AppleTalk Transition Queue in LAP Manager rather than under .MPP). Calls are made to the LAP

Manager by jumping through a low-memory location, with register D0 equal to a dispatch code that identifies the function. The exact sequence is:

```

                MOVEQ    #Code,D0      ; D0 = ID code of wanted LAP call
                MOVE.L   LAPMgrPtr,An   ; An -> start of LAP manager (from $B18)
                JSR      LAPMgrCall(An) ; Call the LAP manager at entry point

LAPMgrPtr      EQU      $B18           ; This points to our start (more
                                        ; commonly known as ATalkHk2)
LAPMgrCall     EQU      2             ; Offset to make LAP manager

```

[Back to top](#)

## The AppleTalk Transition Queue LAP Calls

### LAddAEQ (D0=23)

Call: A0--> Entry to be added to the AppleTalk event queue.

The LAddAEQ call adds an entry, pointed to by A0, to the AppleTalk event queue.

```

                MOVEQ    #LAddAEQ,D0    ; D0 = 23 code of LAddAEQ LAP call
                MOVE.L   LAPMgrPtr,An    ; An -> start of LAP manager (from $B18)

```

### LRmvAEQ (D0=24)

Call: A0--> Entry to be removed from the AppleTalk event queue.

The LRmvAEQ call removes an entry, pointed to by A0, from the AppleTalk event queue.

```

MOVEQ #LRmvAEQ,D0 ; D0 = 24 code of LRmvAEQ LAP call
MOVE.L LAPMgrPtr,An ; An -> start of LAP manager (from $B18)
JSR LAPMgrCall(An) ; Call the LAP manager at entry point

```

### LGetAEQ (D0=25)

Return: A1--> Pointer to the AppleTalk event queue header.

The LGetAEQ call returns a pointer in A1 to the AppleTalk event queue header, previously described.

```

MOVEQ #LGetAEQ,D0 ; D0 = 25 code of LGetAEQ LAP call
MOVE.L LAPMgrPtr,An ; An -> start of LAP manager (from $B18)
JSR LAPMgrCall(An) ; Call the LAP manager at entry point

```

[Back to top](#)

## The Transitions

Each process is called at CallAddr when any significant transitions occur. A value is passed in, which indicates the nature of the event. Additional parameters may also be passed and a pointer to the task's queue element is also passed. This is provided so processes may append their own data structures (e.g., a globals pointer) at the end of the task record, which can be referenced when they are called. Processes should follow the MPW C register conventions. Registers D0, D1, D2, A0, and A1 are scratch registers that are not preserved by C functions. The arguments passed to the process should be left on the stack, since the calling routine removes them. All other registers should be preserved.

### The Open Transition

For AppleTalk open transitions, the process has the following interface:

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

```

OpenEvent RECORD 0 ReturnAddr DS.L 1 ; address of caller theEvent DS.L 1 ; = 0 ; ID of
Open transaction aqe DS.L 1 ; pointer to task record SlotDevParam DS.L 1 ; pointer to
Open parameter block ENDR

```

This routine is called **only** when the open routine for .MPP executes successfully. Every entry in the transition queue is called in the same order that the entries were added to the queue. If AppleTalk is already open and an \_Open call is made, no process is called. The process should return a function result in D0, which is currently ignored.

A pointer to the open request parameter block is passed to the open event process for information only (i.e., the event process may not prevent AppleTalk open calls). Those fields which are of interest are OpenPB->ioPermsn, passed by the caller, and OpenPB->ioMix, which is both passed by the caller and updated by the .MPP open (see *Inside Macintosh*, Volume V, The AppleTalk Manager).

### The Close Transition

For AppleTalk close transitions, the process has the following interface:

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

```

CloseEvent RECORD 0 ReturnAddr DS.L 1 ; address of caller theEvent DS.L 1 ; = 2 ; ID
of Close transaction aqe DS.L 1 ; pointer to task record ENDR

```



The process is being told that AppleTalk is closing, which gives the process an opportunity to close gracefully. Every entry in the event queue is called, one after the other, in the same order that the entries were added to the queue. The close action cannot be cancelled. The process should return a function result in D0, which is currently ignored.

### The ClosePrep and CancelClosePrep Transitions

The `AtalkClosePrep` and the `CancelAtalkClosePrep` control calls are used by various elements of the System, such as the Chooser, to inform or query AppleTalk clients of the closing of network drivers. For example, on a machine equipped to go to sleep or to wake up, the `_Sleep` trap is used by such entities as sleep timer, Finder, and Shutdown to inform AppleTalk clients that it is desirable for the network driver (.MPP) to be closed. The `_Sleep` trap may be trying to do any of the following three things: request permission for sleep, alert for impending sleep, or inform that wake up is underway. The sleep request calls the following two .MPP control calls; these calls are made before sleep queue procedures are called.

The first control call, `AtalkClosePrep`, is used to inform or query AppleTalk clients that the network driver might be closed in the very near future. The call has the following interface:

**AtalkClosePrep** (csCode = 259)

```
Parameter Block --> 26 csCode word ;always AtalkClosePrep <-- 28 clientName pointer
;-> name of client using driver
```

Result codes

`noErr` The AppleTalk network driver (.MPP) may be closed

`closeErr` The AppleTalk network driver (.MPP) may not be closed

`clientName` is a pointer to an identifying string that is returned only if the result is `closeErr`. Note that the pointer may be NIL in this case, while the pointer is always NIL if the return code is `noErr`.

All tasks in the AppleTalk Transition Queue are called with the event `ClosePrep`. The tasks can prevent driver closure with a negative response to the event call. Each task is called with the following interface:

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

```
ClosePrep RECORD 0 ;top of the stack ReturnAddr DS.L 1 ;addr of caller theEvent DS.L 1
;=3 aqe DS.L 1 ;->task rec. clientName DS.L 1 ;ptr. to ptr. to name of client ENDR
```

For this event, `theEvent` = 3, and the task is being **both** informed and asked if closing the network driver is acceptable. If driver closure is acceptable, the task need only to reply affirmative (D0 = 0), or if not acceptable, deny the request (D0 != 0). The task may use the event as an opportunity to "prepare to die" or may simply respond. For example, a task may prevent further sessions from forming while waiting for the actual close event.

`clientName` is a pointer to a field in the .MPP control call parameter block where the task may optionally store a string address. This string identifies the client who has AppleTalk in use and is denying the request to close it. This string may be used in a dialog to inform the user to take appropriate action or explain why the requested action could not be performed.

If any task responds negatively, no subsequent tasks are called. Any tasks called prior to the one that denied a query are recalled with another event, `CancelClosePrep` (described below), enabling them to "undo preparations to die," and the control call then completes with a `closeErr` error.

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

```
CancelClosePrep RECORD 0 ;top of the stack ReturnAddr DS.L 1 ;addr of caller theEvent
DS.L 1 ;=4 aqe DS.L 1 ;->task rec. ENDR
```

For this event, `theEvent` = 4, and the task is being informed that although it has recently approved a request to close the network driver, a subsequent task in the AppleTalk Transition Queue has denied permission. This event permits the task to undo any processing that may have been performed in anticipation of the network driver being closed. The process should return a function result in D0, which is currently ignored.

The second new control call, `CancelAtalkClosePrep`, is used to undo the effects of a successful `AtalkClosePrep` control call. Even though all queried tasks in the AppleTalk Transition Queue approved of network driver closure, other conditions may exist after making the `AtalkClosePrep` control call which prohibit network driver closure. In this case, it is necessary to recall all tasks to undo any processing that may have been performed in anticipation of the network driver being closed. The control call to do this has the following interface:

**CancelAtalkClosePrep** (csCode = 260)

```
Parameter Block --> 26 csCode word ;always CancelAtalkClosePrep
```

Result codes `noErr` Nothing could possibly go wrong

All tasks in the AppleTalk Transition Queue are called with the event `CancelClosePrep` as described above.

#### Note:

The use of the low-memory global `ChooserBits` (\$946) is no longer an acceptable means of preventing AppleTalk from closing when AppleTalk Phase 2 is present. Transitions other than defined above must be ignored and are reserved for future implementation. In the future transitions may be defined for notifying processes when a change in zone name occurs.

[Back to top](#)

## Potential Compatibility Problems

### Using DDP and Talking to Routers

If, for some reason, you need to talk to any router via DDP, always use the `GetAppleTalkInfo` call outlined in this Note to get the router's actual 24-bit address.

The `WriteLAP` function (`csCode` = 243) to the `.MPP` driver is no longer supported, since a node is no longer identified only by its eight-bit (LAP) node ID.

On a Macintosh running the AppleTalk Internet Router software, the `SelfSend` flag is always set, so if you try to clear this flag using the `PSetSelfSend` call (Inside Macintosh, Volume V-514), you will get an error.

[Back to top](#)

## References

Inside AppleTalk

Inside Macintosh, Volume II, The AppleTalk Manager

Inside Macintosh, Volume V, The AppleTalk Manager

EtherTalk and Alternate AppleTalk Connections Reference, May 5, 1989--Draft (DTS)

AppleTalk Phase 2 Protocol Specification (DTS)

Macintosh Portable Developer Notes (DTS)

[Back to top](#)

## Change History

01-August-1989	Incorporated the <code>ClosePrep</code> and <code>CancelClosePrep</code> transitions and the new control calls to the <code>.MPP</code> driver.
----------------	-------------------------------------------------------------------------------------------------------------------------------------------------

## Downloadables



Acrobat version of this Note (K)

[Download](#)[Back to top](#)

---

Technical Notes by [Date](#) | [Number](#) | [Technology](#) | [Title](#)  
[Developer Documentation](#) | [Technical Q&As](#) | [Development Kits](#) | [Sample Code](#)