Medusa Programmer's Guide Beta Draft

Notice

The information in this document reflects the current state of the product. Every effort has been made to verify the accuracy of this information; however, it is subject to change. Beta Drafts are released in this form to provide the development community with essential information in order to work on compatible products.

Contents

Figures and Tables

Preface

THIS DOCUMENT is to be used by Apple software developers who wish to develop a protocol interface to the Apple® TokenTalk™ NB card in conjunction with the Macintosh® Operating System (OS). To make use of the information presented here, you should have a working knowledge of the Macintosh OS and, depending on your application, a working knowledge of token ring networks. The information presented in this manual describes how to interface to the data link layer by way of calls to the SubNetwork Access Protocol (SNAP) interface and the 802.2 logical link control (LLC) interface.

You should be familiar with the following information:

- Macintosh II computer and NuBus™
- Macintosh Programmer's Workshop (MPW®)
- C programming
- Multiprocessor programming techniques
- Network programming techniques
- Minimal Realtime Distributed Operating System (MR-DOS™)

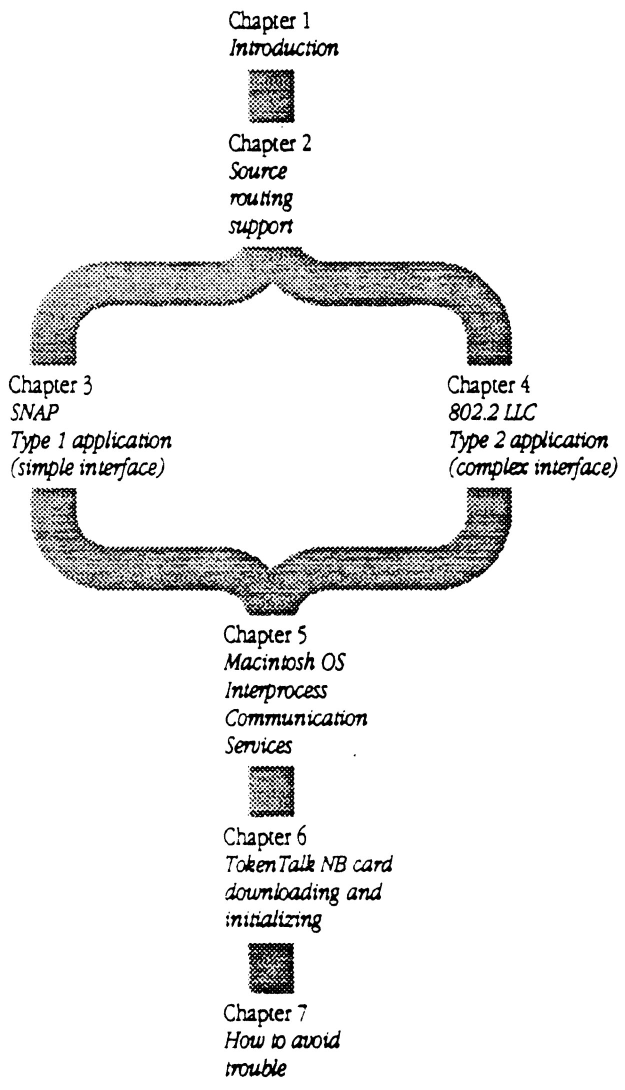
What this document contains

This document provides a description of the programming interface to the TokenTalk NB card and includes programming information on the SubNetwork Access Protocol (SNAP) interface, the logical link control (LLC) interface, and the interprocessor communication (IPC) interface provided in the Macintosh OS. The Macintosh services that initialize the TokenTalk NB card are also presented.

The intent of this document is to supply information that allows developers to develop other protocol interfaces (APPC, 3270, and so on) that run under the Macintosh OS for the TokenTalk NB card.

The following table describes the contents of this manual and shows where to find information that helps you accomplish a desired task. Not all chapters are applicable to all tasks. A roadmap that illustrates the manual organization follows the table.

What you need	Location	Content
Introductory information	Chapter 1	An introduction to token ring concepts and interface services running under the Macintosh OS
Source routing support in a multi-network environment	Chapter 2	A discussion of source routing support in TokenTalk
Developing Type 1 "connectionless" token ring services	Chapter 3	The SubNetwork Access Protocol (SNAP) interface calls to the Macintosh OS
Developing Type 2 connection-oriented token ring services	Chapter 4	The 802.2 logical link control (LLC) interface, which is useful for applications based on a specific protocol with an assigned Service Access Point (SAP) identifier
Interprocess commun-ication between the Macintosh OS and the TokenTalk NB card	Chapter 5	The interprocess communication (IPC) services provided by the Macintosh OS for passing messages between the operating system and the TokenTalk NB card. All developers need the information contained in this chapter.
Initialize the TokenTalk NB card	Chapter 6	The code and parameters in the TokenTalk Prep file used for initializing the TokenTalk NB card and an example of how to use the TokenTalk Prep file. All developers need the information contained in this chapter.
General troubleshooting guidelines	Chapter 7	Troubleshooting tips and hints for avoiding trouble with software and hardware.


Chapter 1
Introduction
Chapter 2
Source routing support
Chapter 3
SNAP
Type 1 application (simple interface)
Chapter 4
802.2 LLC
Type 2 application (complex interface)
Chapter 5
Macintosh OS Interprocess Communication Services
Chapter 6
TokenTalk NB card downloading and initializing
Chapter 7
How to avoid trouble

Suggested reading

Here is a list of reference materials that relate or apply directly to the TokenTalk NB card:

- *Macintosh Coprocessor Platform Developer's Guide* (MR-DOS IPC implementations)
- *Apple TokenTalk NB User's Guide*
- *Athena Programmer's Reference and User's Guide*
- *Texas Instruments TMS380 Adapter Chipset User's Guide*
- *Texas Instruments TMS380 Adapter Chipset User's Guide Supplement*
- *Texas Instruments Manual Update, Revision F*
- *IBM Token Ring Network Architecture Reference*
- IEEE 802.2 Standard
- IEEE 802.5 Standard

Possible applications

You may wish to develop any number of possible applications. For example, you may want to create your own 3270 protocol emulator that accesses mainframe computers by way of the token ring interface. Other applications might be to implement TCP/IP under the Macintosh OS for the TokenTalk NB card or to provide X.25 dial-up services.

The information presented in this document assumes that the token ring application you are developing runs under the Macintosh OS and is not downloaded to reside in memory on the TokenTalk NB card itself. The *Macintosh Coprocessor Platform Developer's Guide* contains information you need to develop TokenTalk NB memory-resident applications.

Conventions used in this manual

Look for these conventions throughout the manual:

◆ *Note:* Notes like this contain supplementary information.

A special typeface is used to indicate lines of code:

```
Program code looks like this
```

Chapter 1 Introduction

THIS CHAPTER INTRODUCES the topics that support programming access to the Apple TokenTalk NB card. The TokenTalk NB card provides an interface to a token ring network. By using the services provided in the Macintosh Operating System (OS), you can program a protocol interface, such as 3270 data stream protocol or TCP/IP, that supports token ring communication.

In this chapter you will find introductory information on token ring networks, SubNetwork Access Protocol (SNAP), the 802.2 Logical Link Control (LLC) interface, Macintosh OS Interprocess Communication (IPC), and the download and initialization services for the TokenTalk NB card. ■

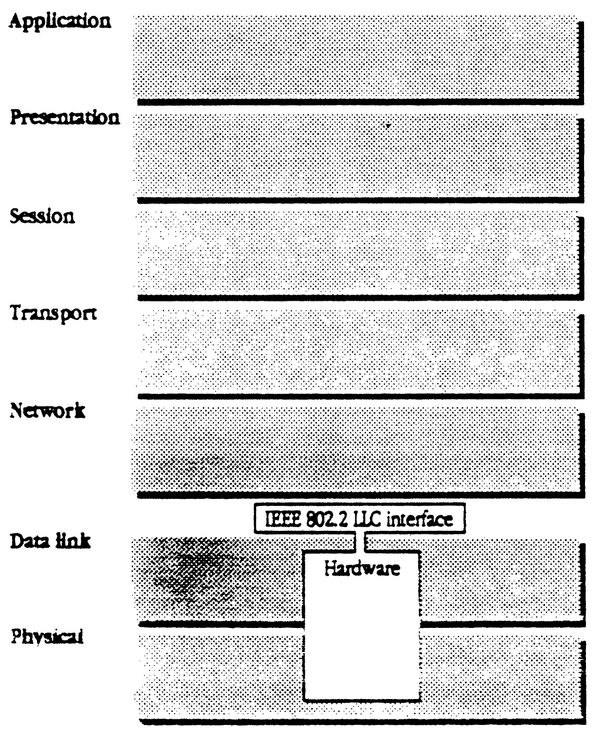
Token ring networks

A token ring network is a topology (ring) and a protocol (token-passing) defined by the IEEE 802 committee. The actual token ring access method, or how to interface with the physical media, is defined in the IEEE 802.5 standard. However, you need not be concerned with the physical access to the token ring network because the access is handled by the TokenTalk NB card itself, as are the 802.2 logical link control functions.

The network layers

The TokenTalk NB card provides an interface to the token ring network. The token ring network interface adheres to the International Standards Organization Open System Interconnection (ISO OSI) network model. The 802.2 LLC interface provided for the TokenTalk NB card corresponds to the ISO OSI model as shown in *Figure 1-1*.

■ **Figure 1-1** TokenTalk NB protocol model



A token ring network

The topology of a token ring network is shown in *Figure 1-2,* which shows the ring, the nodes, and the free token that circulates around the ring. The physical components of a token ring network consist of the TokenTalk NB cards, one or more multistation access units (MAU), and the connecting cables. The MAU and the connecting cables provide the physical "ring" for the network, in fact, the MAU acts as a wiring concentrator for the connecting cables. Multistation access units can be connected in a daisy chain to provide whatever size network is required. The TokenTalk NB card and the Macintosh II system provide the network node on the ring (*Figure 1-3*).

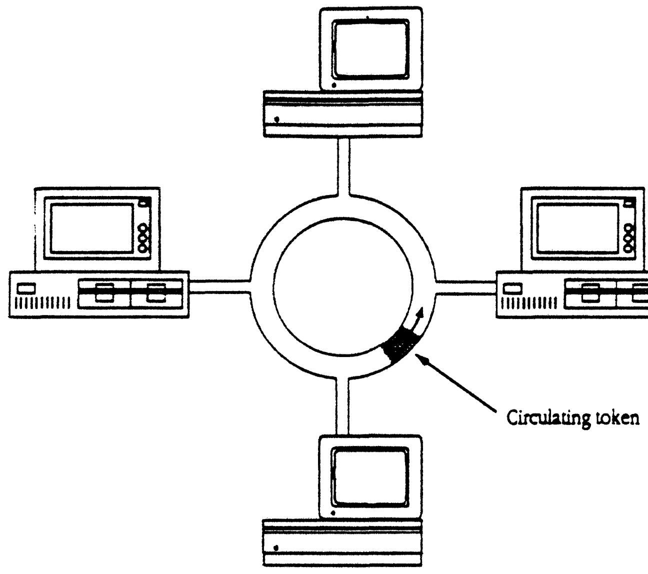
Token communication

In a token ring network, a data packet called a *free token* is passed from node to node. If a node has no data to transmit, it passes the free token to the next node. On the other hand, if a node does have data to transmit, it captures the free token, changes it to a busy token, and appends the necessary destination address, source address, data, data checks, and control bytes to ensure reliable delivery to the destination node. This busy token is called a *frame.*

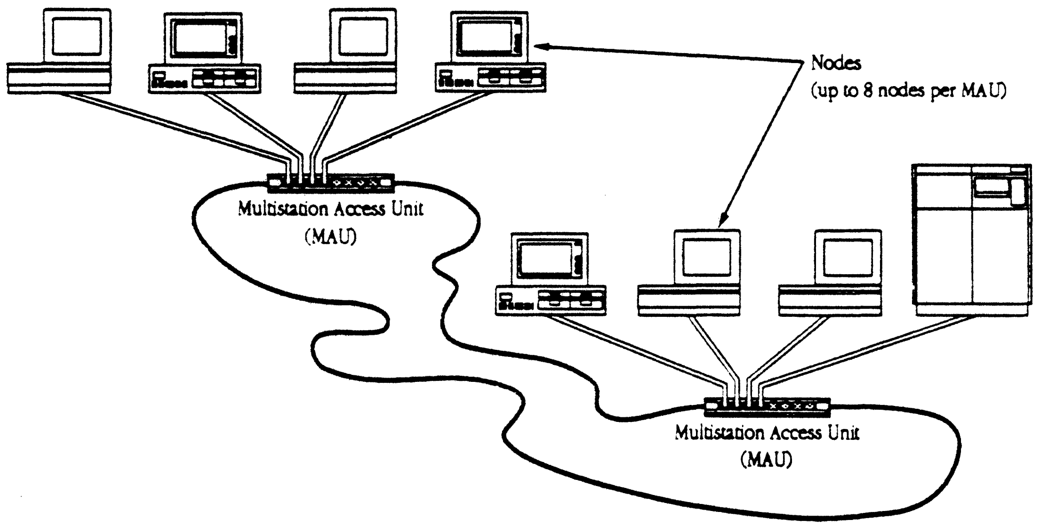
Each node between the source node and the destination node passes the frame, or data packet, onward. When the data is received at the destination node, it marks the data packet as received and sends the busy token around the ring to the source node. The source node then checks the token and verifies that the destination node received the data. The originating node removes the busy token from the ring and releases a new free token on the ring so another node can transmit (*Figure 1-4*). The originating node must wait for another free token before it can transmit again.

Any one node is allowed one transmission per free token, which limits each node's access to the network. In this manner, every node on the network is guaranteed equal access time to the network.

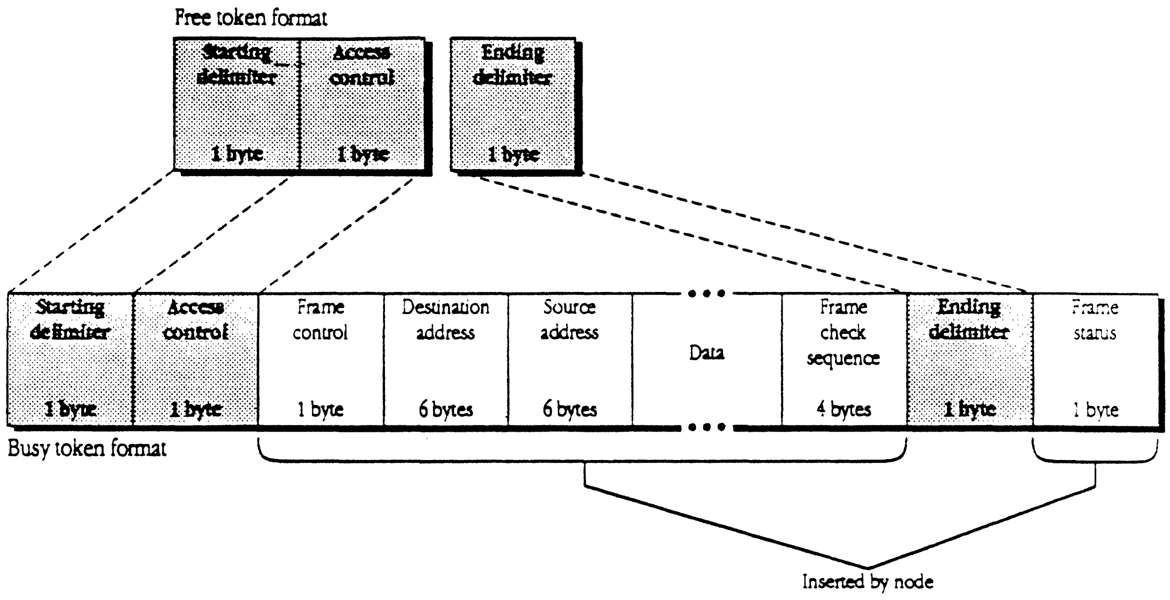
■ **Figure 1-2** Token ring topology



■ **Figure 1-3** Token ring components



■ **Figure 1-4** Frame formats: free token, busy token

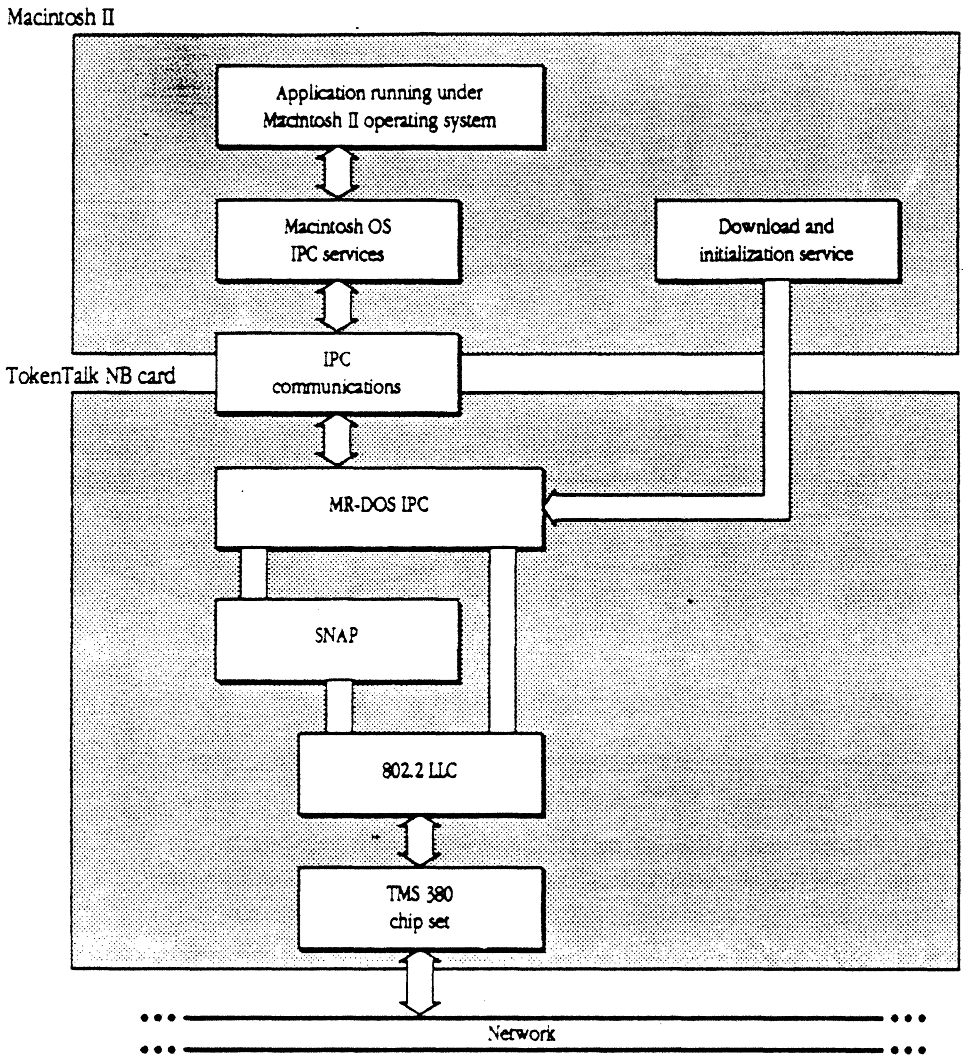


The Macintosh II token ring interface

The actual formatting and transmission of the data packets, free tokens, and busy tokens is handled by the hardware on the TokenTalk NB card and the 802.2 LLC interface software. Your task as a developer or programmer is to use the programming support tools to pass the necessary destination address and data information to the TokenTalk NB card and to deliver the data from the card to applications running under the Macintosh OS. .Source routing of packets through bridges is described in Chapter 2.

Figure 1-5 shows the Macintosh II and TokenTalk NB card interface to the token ring network.

■ **Figure 1-5** Macintosh interface to the token ring network


Macintosh II
Application running under Macintosh II operating system
Macintosh OS IPC services
Download and initialization service
TokenTalk NB card
IPC communications
MR-DOS IPC
SNAP
802.2 LLC
TMS 380 chip set
Network

As shown in Figure 1-5, the primary communication interface between the TokenTalk NB card and the Macintosh II is through the interprocess communication (IPC) services. These services are provided by the Macintosh OS on the Macintosh II and by MR-DOS on the TokenTalk NB card. A specific set of services for the 802.2 LLC and SNAP provide the interface to the chip set that handles the low-level protocol processing and physical communication with the token ring network.

The TokenTalk NB card is initialized and downloaded by way of the services provided in the TokenTalk Prep file.

SubNetwork Access Protocol (SNAP)

The IEEE 802.2 committee has implemented a SubNetwork Access Protocol (SNAP) that allows protocol multiplexing and demultiplexing among multiple users of a data link. When Ethernet was first designed, it allowed for 64 different protocol identifiers. However, with the maturation of local area network technology and the development of other network standards such as token ring and token bus, 64 different protocols identifiers were too few. Different network companies devised various schemes to expand the number of protocol identifiers so as to differentiate between, say, AppleTalk, TCP/IP, XNS, and other protocols.

To accommodate the large number of network protocols, the IEEE 802 committee has imposed the SNAP to standardize protocol access to the network and to ensure that protocol identifiers from different vendors do not conflict. SNAP is analogous to the old Ethernet protocol ID except that SNAP is a 5-byte field and the old Ethernet protocol ID is a 2-byte field. The trend now is to represent the old Ethernet protocol IDs in SNAP, which provides compliance with the current standard.

SNAP allows Type 1 (datagram) communication services only; it does not support connection and session-oriented Type 2 services. For those services you must bypass SNAP and use the 802.2 logical link control (LLC) interface directly.

The SNAP interface described in this manual is sufficient for a wide variety of network protocol applications. Source routing is supported by the SNAP interface to allow transmission of packets through bridges and multiple networks, but is not implemented in the LLC interface. The more complex LLC interface should be used primarily in Type 2 applications, such as connection-oriented 3270 data stream protocol communication.

The 802.2 Logical Link Control IPC interface

The logical link control (LLC) sublayer is the part of the data link layer that supports the media-independent data link functions, and which uses the services of the medium access control (MAC) sublayer to provide services to the network layer. The IPC interface to the 802.2 LLC communicates with either the Texas Instruments token ring chip set (the TMS380 family) that implements the 802.2 LLC, or with a software-based 802.2 LLC wherein the tasks performed by the chip set are implemented in software.

The 802.2 IPC interface functions described in this manual provide access to and communicate with the 802.2 LLC. It is important to understand that for the TokenTalk NB card applications, the 802.2 LLC itself is implemented in the chip set on the TokenTalk NB card.

Macintosh Operating System IPC services

The Macintosh II operating system supports a multitasking, multiprocessor environment. Different intelligent cards residing on the NuBus, such as the TokenTalk NB card, depend on interrupt-driven communications to transfer information and to coordinate task execution. The interprocess communication (IPC) is the mechanism that provides this communication service.

Many IPC functions are provided for the Macintosh Operating System and for the MR-DOS. MR-DOS is an operating system that resides on the smart cards in the Macintosh II and provides the IPC services for these cards. For information on the MR-DOS IPC, refer to the *Macintosh Coprocessor Platform Developer's Guide.*

Download and initialization services

A TokenTalk NB card is initialized from the Macintosh Operating System by way of a special file called TokenTalk Prep. This file contains resources that hold code images for downloading to the TokenTalk NB card. The TokenTalk Prep file provides the services that initialize the TokenTalk NB card and download MR-DOS, SNAP, 802.2 LLC/IPC interface, and default LLC parameters.

Chapter 2 Source Routing Support

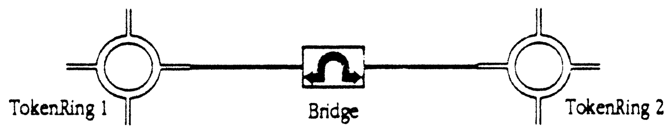
THIS CHAPTER DESCRIBES network source routing support and includes background information on network routing and bridges. This chapter also discusses source routing implementation and source routing limits. For the most part, source routing support is transparent because it is included as part of the SubNetwork Access Protocol (SNAP) services in TokenTalk. ■

What is source routing?

Chapter 1 presented the concepts associated with a single token ring network and briefly described the frame formats associated with data transmission within a token ring network. In a single token ring network, the information contained in the frame, or data packet, includes the address of the source node and the address of the destination node. Source node and destination node addresses are all that are required to send data packets in a token ring network.

The term *source routing* refers to the means by which frames between multiple networks are correctly sent, or routed, between the source and destination nodes. Source routing occurs when a bridge connects two or more token ring networks and frames pass through the bridge between the two networks (*Figure 2-1*). In essence, a bridge forwards frames from one network to another based on routing information that is inserted by the source node.

■ **Figure 2-1** Single bridge between networks



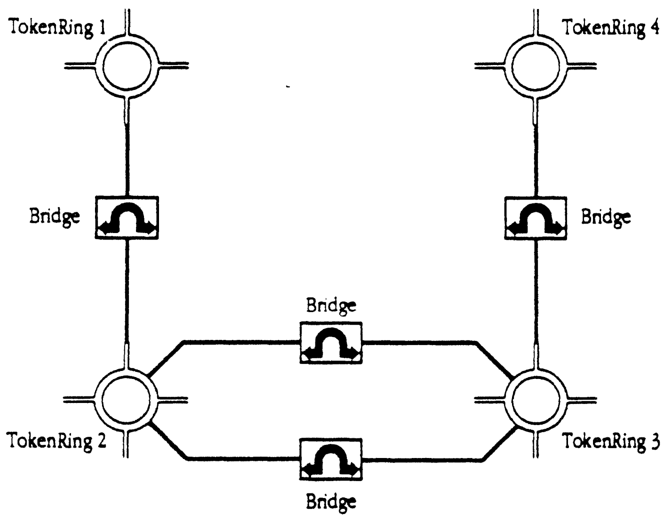
As defined by the IEEE 802 specification, a bridge is a functional unit that connects two networks using a single logical link control (LLC) procedure, which in TokenTalk is the IEEE 802.2 LLC. Several configurations are possible when more than two networks are connected by bridge, but the resultant network is either a hierarchical network or a mesh network. These two concepts are explained in the following paragraphs.

Hierarchical networks

Simply defined, a ***hierarchical network*** is one that provides only one path between the source and destination nodes, no matter the number of intermediate rings. For example, in *Figure 2-2* a frame from ring 1 must pass through intermediate ring 2 in order to reach its destination on ring 3. No other path exists.

Likewise, a frame from ring 4 destined for ring 1 must pass through intermediate rings 3 and 2. The key to a hierarchical network is that only one path, or route, is provided between source and destination nodes. As the figure shows, there is a choice of bridges between ring 2 and 3 but no choice of intermediate rings.

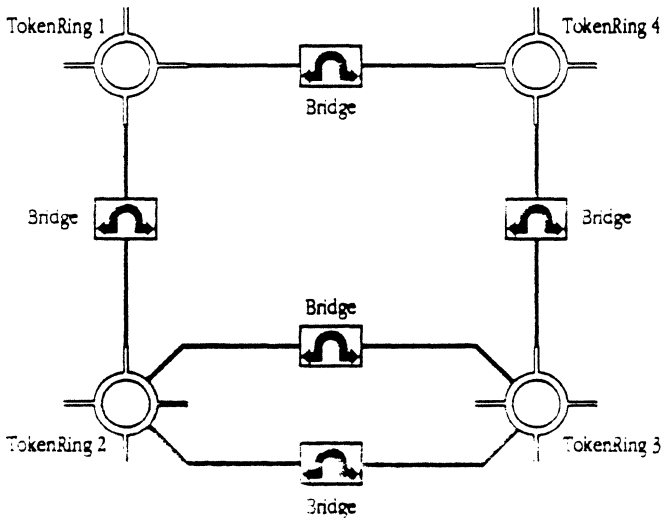
■ **Figure 2-2** Hierarchical network



Mesh networks

A *mesh network* provides multiple paths between the source and destination rings and alternative choices of bridges. *Figure 2-3* shows four rings connected in a mesh configuration.

■ **Figure 2-3** Mesh network



In the mesh network shown in Figure 2-3, a frame has two possible paths from ring 1 to ring 3. The frame can be routed through ring 4 or through ring 2.

Note that a parallel connection exists between ring 2 and 3. Parallel connections provide redundancy in situations that require high reliability. Up to 16 parallel connections can exist between any two rings.

Variations on hierarchical and mesh networks can accommodate a wide variety of network configurations. Configuration parameters and network layouts are determined during the planning and installation phase and are dependent on specific limitations enforced by the bridge manufacturer. The primary benefit of bridges is to allow more than 260 devices to be supported in the network installation.

How source routing works

For any two nodes, or stations, to communicate in a hierarchical or mesh network, routing information must exist that describes the path between the two stations. Route determination can be the responsibility of the communicating stations, the bridges, or a central management facility*. Source routing applies to the first case, where the station that is the source of the frame puts the routing information into the frame. Bridges, which operate at the data link layer of the network, support source routing. (Refer to Chapter 1 for an illustration of the network layers.)

Source routing exhibits the following features:

- Routing information is based on information about the path between two communicating stations; station addresses are not used.
- Path information is learned dynamically by a station that initiates communication with another.
- Route discovery is a two-part process that involves broadcasting a message to all of the interconnected networks.
- Bridge routing tables are not required; bridges decide whether to forward a frame by comparing a fixed, identifying value with a small portion of the routing information field in the frame.

Routing information

Routing information is contained in its own field in the frame and is separate from the destination address. The routing information is obtained in two stages. The first stage occurs when the source station broadcasts a frame to all of the connected networks. The broadcast frame contains the destination address of the target station plus information that tells the intervening bridges to forward the frame.

* Kian-Bon K. Sy, Daniel Avery Pitt, and Robert A. Donnan. "Source Routing for Local Area Networks," IBM Corporation 1985

The routing information is added to the frame during the broadcast phase. A bridge on the first network adds the identifying numbers of the two networks that it joins. Additional bridges add only the identifying number of the next network. (The network ID numbers are assigned by a network administrator when the network is initially installed and configured.) Frames are prevented from looping because no bridge will forward a frame to a network whose number already appears in the frame.

The second phase of obtaining the routing information is performed by the station that received the initial broadcast frames. Each frame is returned as soon as possible according to the route it acquired from the bridges along the way, rather than being returned by broadcast message. Because the initial broadcast frame is returned by any of several possible routes, the source station acquires frames that contain valid routing information. The source station can choose any of the valid routes returned by the destination, but the first response has usually traveled the fastest route.

Up to this point, the destination station still has no idea which route will be used for communication. The source station keeps its chosen routing information, which is learned by the destination station when nonbroadcast communication begins. Because the same route is used for communication in both directions, failed links can be easily diagnosed.

The routing information can be associated solely with the destination address, or with the combination of destination address and destination and source link service access points (SAPs). The first case limits all communication to the same route, whereas the second case allows different "conversations" to use different routes. Chapter 4 describes SAPs.

As previously mentioned, the SubNetwork Access Protocol (SNAP) interface automatically provides source routing support in a connectionless environment. Because the source routing is provided in a connectionless environment, an aging timer is used to eliminate source routing information from the routing tables, thus preventing possible errors from table overflow. By contrast, if connection-oriented source routing were supported, the routing information would be maintained only for the duration of the link connection.

Source routing implementation

Source routing is implemented in the SubNetwork Access Protocol (SNAP) interface. Supplied with the TokenTalk NB card, this protocol automatically handles the discovery and response phase for source routing addresses.

SNAP use

The SNAP interface allows Type 1 (datagram) communication services only; connection and session-oriented Type 2 services are only supported by the 802.2 logical link control (LLC). For those services you must bypass SNAP and use the LLC interface directly.

The SNAP interface described in this manual is sufficient for a wide variety of network protocol applications.

LLC use

The more complex LLC interface is used primarily in Type 2 applications, such as connection-oriented 3270 data stream protocol communication. Source routing is not directly supported in the LLC interface

Source route limits

Some limits on source routing are imposed when the networks and network bridges are installed. A network administrator is responsible for properly configuring the network and supplying workable values. The configuration parameters that can restrict frame forwarding and source routing activity include the following:

- Bridge ID number. To properly route frames, each bridge must have an ID number assigned.
- Hop count limit. The *hop count* is the number of bridges that broadcast frames have already crossed to reach the current bridge. Broadcast frames with a hop count equal to or higher than the hop count limit imposed on the bridge are not allowed to cross the bridge. If the number of hops between the source and destination station exceeds the hop count limit, the frame transmission fails.

Additional bridge configuration parameters controlled by the network administrator affect how frames are passed throughout the network.

The number of source routing addresses that any one station can keep track of is limited by the table size reserved for storing these addresses. Two tables are used: one keeps track of the address-to-ring numbers; the other keeps track of the ring-number-to-route. The tables can hold approximately 80 node addresses and 100 ring addresses.

Table overflow is prevented by a "least-used timeout" algorithm. A node address entry is dropped when it is not heard from for 40 seconds. A ring number is dropped when it is not heard from for three minutes.

Chapter 3 SubNetwork Access Protocol (SNAP) Interface

THIS CHAPTER DISCUSSES the programming interface for the 802.2 SubNetwork Access Protocol (SNAP) interface. SNAP is used to deliver Type 1 messages in a network and is a less complex interface than the 802.2 LLC interface described in Chapter 4. ■

General information

SubNetwork Access Protocol (SNAP)is defined by the IEEE 802 committee as the standard means of identifying a large number of protocols in an 802.2 environment. SNAP uses a service access point (SAP) identifier of 0xAA. By comparison, a ISO OSI SAP identifier is the hex value 0xFE.

The first five bytes of the information field of each SNAP frame contain a protocol discriminator that identifies a particular protocol. The first three bytes of the protocol discriminator are the vendor ID assigned to the creator of the protocol, that is, the same vendor ID used in globally-administered node addresses. The Ethernet bit ordering in these three bytes is retained, which means that the bytes are transmitted most-significant-byte, least-significant-bit first. This Ethernet bit ordering is the format for representing the vendor ID in SNAP on all media. The last two bytes are assigned by the vendor to identify a particular protocol. By convention, if the vendor ID is set to zero, the remaining two bytes represent an Ethernet protocol ID.

As you can see, the SNAP interface is not strictly limited to token ring applications. Because the SNAP interface is at the data link level of the network model, it is insulated from the implementation of the physical level.

In the TokenTalk NB card implementation, the SNAP interface registers itself under the type "SNAP" with the MR-DOS name manager. A name that is associated with the type is passed as a startup parameter. (Startup parameters are provided in the TokenTalk Prep file discussed in Chapter 6.) By convention, the name is "TokenTalkNB."

Client processes should limit the number of requests that they queue to the SNAP. As a general guideline, no more than ten SNAPReceive and ten SNAPTransmit requests should be queued by a single client at once. With any more queued requests MR-DOS can run out of message buffers. One method to impose this limit is to allocate a fixed number of transmit buffers, receive buffers, and data buffers when the code is initialized and to keep the buffers in a linked list. Then, by removing entries from the list and requeueing them when a request completes, there only await a limited number of requests to the SNAP interface at any given time. Queueing several receive or transmit requests improves both the throughput and reliability, but the number of queued requests must never exceed the number of available MR-DOS message buffers.

The following list presents the requests that a client can issue to the SNAP interface. In each case, mCode identifies the function and, in the reply, mStatus holds the result code for the function. As is the convention with MR-DOS IPC, all requests have an even mCode value and all replies use the corresponding mCode plus one.

mCode	**Meaning**	**See page**
SNAPAttach	Attach protocol discriminator	19
SNAPGetConfig	Return SNAP configuration information	21
SNAPGetHdr	Return media header template	22
SNAPTransmit	Send a SNAP type 1 frame	24
SNAPDetach	Detach protocol discriminator	25
SNAPReceive	Receive a frame	26
SNAPCancel	Cancel all queued receives	28
SNAPGetParms	Returns SNAP-associated parameters	29

In addition to the above messages, the SNAP interface supplies the following library of support functions:

Name	**Description**
SNAPSwapHdr	Swap node addresses in LANHdr structure for return to sender

Typical SNAP use

IPC requests support both Type 1 and Type 2 logical link control (LLC). Type 1 is connectionless and uses both the SNAP interface and, because the SNAP services are built on top of the LLC services, the 802.2 LLC interface. Type 2 is connection-oriented and is not supported by the SNAP interface. A typical application for Type 2 is 3270 terminal emulation.

Because the SNAP allows a Type 1 data link service only, it is discussed in terms of Type 1 LLC. Type 1 LLC provides a data link with a minimum protocol complexity and is used when the upper layers of the ISO model provide the error detection and recovery. Type 1 LLC is also used in an application in which it is not necessary to guarantee all data link layer transmissions.

Once a SNAP is attached, the application or protocol stack associated with that protocol discriminator can transmit and receive any of the following Type 1 frames through the SNAP:

- TEST – Test Command causes the remote node to send a Test Response.
- UI – Unnumbered Information is used to transfer data in a Type 1 environment.

The following series of actions illustrates a typical usage for a SNAP client using Type 1 services and outlines the actions necessary to transmit data by way of a TokenTalk NB card:

1. Use the TokenTalk Prep Utilities (TTGetSNAPTID) or the IPC name lookup to find the SNAP service.
2. Issue a SNAPGetParms to obtain the Task ID of LLC and the RefNum of the SNAP's SAP. This allows the SNAP client to be able to make requests directly of the 802.2 LLC IPC interface, such as LLCT1Transmit, LLCGetHdr, and LLCGetConfig.
3. Issue a SNAPAttach, which includes a 5-byte protocol discriminator.
4. Optionally obtain configuration information from LLC by way of SNAPGetConfig.
5. Obtain header template by way of SNAPGetHdr. The header can be copied after it has been obtained, but it is important initially to use SNAPGetHdr to build the LAN header with values supplied by the client (such as destination node). Different LLC implementations might assume a different header setup, so by using SNAPGetHdr you insulate yourself from unnecessary problems. In general, the offset values supplied in the header should be left alone.
6. Queue receive requests to SNAP to accept incoming frames by way of SNAPReceive.
7. Issue transmit requests to SNAP as required by way of SNAPTransmit.
8. Reissue receive requests as the receive frames are returned.
9. On completion, issue SNAPDetach. SNAPDetach automatically cancels outstanding receives. Any outstanding receives are returned as "cancelled."

IPC requests to SNAP

In all structure declarations in this chapter, the type "byte" refers to an unsigned 8-bit integer and "word" refers to an unsigned 16-bit integer. All structures and symbols used in this document are defined in the include file SNAP.h.

SNAPAttach

The SNAPAttach message is used to begin listening for packets on the specified SNAP protocol. This request also specifies various options that are associated with the particular protocol.

Example 3-1 shows the type SNAP_PD, which is used to hold protocol discriminators. This type may not be useful for representing the SNAP header in frames because the C compiler pads it to six bytes.

mDataPtr points to the five-byte PD data structure. mDataSize is equal to the size of the PD data structure.

Refer to *Example 3-2* for the structure of mOData in the SNAPAttach request. Note that mDataPtr points to the type SNAP.PD, which holds the protocol discriminator to attach to.

Result codes	*Value*	*Description*
	SNAPNoErr	Normal completion
	SNAPInUse	PD already attached
	SNAPNoMore	Insufficient resources

The "Listener" function pointer is a special hook that some clients find useful to handle incoming frames more efficiently. Most clients should not use a listener function. Any listener that is provided must be located on the same slot as the 802.2 interface. A listener function is called with parameters that pass the media header, information pointer, information length, and frame type. The listener must be completed with this buffer before returning. When a listener function is in use, SNAPReceive requests are not used to receive frames.

An example declaration for the SNAP listener function might be coded as follows:

```
void    Sample_Listener (nul, nu2, hp, bp, len, ft)
long    nul, nu2;               /*Not used, but do not alter*/
LANHdr  *hp;        /*Pointer to LANHdr of received frame*/
unsigned char *bp;              /*Pointer to I-field, includes the protocol
                                  discriminator*/
int     len;                    /*Length of I-field*/
int     ft;                     /*Frame type, 3, 8, or 9 only*/
{
        if (ft != 3)            /*If not UI frame, ignore*/
                return;
        if (bp[5] != 0)         /*If byte following protocol discriminator is not 0*/
                return;         /*ignore frame */

        /* Other code to manipulate frame data */

        return;
```

■ **Example 3-1** mOData structure for SNAP_PD request

```
typedef struct
{
        union
        {
                long    PD1;            /* Fast access to first four bytes of PD */
                char    PDc[5];         /* Access to each and every byte of PD */
        } PD;
} SNAP_PD;
```

■ **Example 3-2** mOData structure for SNAPAttach request

```
typedef struct
{
     word    PDRefNum;        /* Returns RefNum of this PD (used on SNAPReceive) */
     word    Options;         /* Options :
                                      /*      Bit 15:        Unused
                                       *      Bit 14:        Use listener function
                                       *      Bits 13-0:     Unused
                                       */
     void    (*Listener)();   /* Pointer to optional listening function */
} SNAPAttachOData;
```

SNAPGetConfig

The SNAPGetConfig message returns configuration information about SNAP. *Example 3-3* shows the structure returned at the address passed in mDataPtr.

Result codes	*Value*	*Description*
	LLCNoErr	Normal completion
	LLCTruncated	Buffer too short to receive all information

■ **Example 3-3** Structure mDataPtr points to following completion of SNAPGetConfig

```
typedef struct
{
   long   LLCVersion;     /* LLC Version ID */
   long   FAddr;             /* Functional address (token ring only) */
   long   G1Timer1;          /* Does not apply to SNAP */
   long   G2Timer1;          /* Does not apply to SNAP */
   long   G1Timer2;          /* Does not apply to SNAP */
   long   G2Timer2;          /* Does not apply to SNAP */
   long   G1ITimer;          /* Does not apply to SNAP */
   long   G2ITimer;          /* Does not apply to SNAP */
   word   MaxFrameLen;    /* Maximum frame length */
   word   ASAPs;             /* Does not apply to SNAP */
   word   AStations;      /* Does not apply to SNAP */
   word   MaxHeader;      /* Maximum header size for this media */
   byte   LLCClass;       /* Class of LLC implementation:
                                *   1 - Implements type1 only
                                *   2 - Implements type1 and type 2
                                */
   byte   Media;             /* Media indicator:
                                *   0 - Unknown        4 - 16 Mb Token Ring
                                *   1 - LocalTalk          5 - FDDI
                                *   2 - 10 Mb Ethernet     6 - Token Bus
                                *   3 - 4 Mb Token Ring
                                */
   byte   Routing;           /* Source-routing indicator:
                                *   0 - No source-routing
                                *   1 - IBM source-routing
                                */
   byte   AddrLen;           /* Length of node address in bytes */
   byte   Addr[9];           /* This node's address */
   byte   NumGAddrs;      /* The number of group addresses that follow */
   byte   GAddrBuf[1];    /* Start of group addresses (length, address pairs) */
} LLCGetConfigBuffer;
```

SNAPGetHdr

The SNAPGetHdr message creates a LANHdr structure that is used to transmit to the specified node. Options are also provided to return broadcast header templates.

Example 3-4 shows the structure of mOData in the IPC message and *Example 3-5* shows the structure of the LANHdr structure that is returned to the area pointed to by the Hdr field in mOData.

mDataPtr points to the node address and mDataSize indicates the size of that address in bytes.

Not all media support all possible options. In cases when an unsupportable option is specified, the SNAP interface builds the best header it can and returns the status LLCNotFullySupported to the client. mDataSize must either be zero to get a broadcast template or be the exact size of a node address for the underlying media.

Result codes	*Value*	*Description*
	LLCNoErr	Normal completion
	LLCNotFullySupported	Some option or type requested is not fully supported by this media
	LLCAddrError	Invalid remote address—size must be 0 or equal to the node address size for the media

■ **Example 3-4** mOData structure for SNAPGetHdr requests

```
typedef struct
{
   word     HdrType;     /* Header types:
                          *   0 - Normal header •
                          *   1 - Single-route b'cast, all-routes b'cast return
                          *   2 - Single-route b'cast, non-broadcast return
                          *   3 - All-routes broadcast header
                          */
   word     Options;     /* Header options (always zero) */
   byte     Reserved;    /* Always zero */
   byte     Reserved;    /* Always zero */
   LANHdr   *Hdr;        /* Pointer to LANHdr structure to be returned */
} LLCGetHdrOData;
```

■ **Example 3-5** LANHdr structure for SNAPGetHdr

```
typedef struct
{
       byte   Media;         /* Media indicator:
                                     *      0 - Unknown          4 - 16 Mb Token Ring
                                     *      1 - LocalTalk  5 - FDDI
                                     *      2 - 10 Mb Ethernet    6 - Token Bus
                                     *      3 - 4 Mb Token Ring
                                     */
       byte   Routing;       /* Source-routing indicator:
                                     *      0 - No source-routing
                                     *      1 - IBM source-routing
                                     */
       byte   DOff;          /* Offset to destination address in header buffer */
       byte   DLen;          /* Length of destination address in header buffer */
       byte   SOff;          /* Offset to source address in header buffer */
       byte   SLen;          /* Length of source address in header buffer */
       byte   ROff;          /* Offset to routing information in header buffer */
       byte   RLen;          /* Length of routing information in header buffer */
       byte   HOff;          /* Offset to media header in header buffer */
       byte   HLen;          /* Length of media header in header buffer */
       byte   DSAP;          /* Destination SAP value */
       byte   SSAP;          /* Source SAP value */
       byte   HBuf[40];      /* Header buffer */
} LANHdr;
```

SNAPTransmit

The SNAPTransmit message is used to send a Type 1 frame.

Refer to *Example 3-6* for the structure of mOData in the IPC message and to Example 3-5 for the description of the LANHdr structure that is pointed to by Hdr.

mDataPtr points to either a frame holding the buffer, or, if the "list-directed" bit of the Options field is set, to an array of counts and pointers to buffers, as with receive.

If mDataPtr points to a frame holding user data, the first 5 bytes must be the protocol discriminator (PD) and filled in by the user. It is possible to separate the 5 bytes of the PD from the user data by using the list-directed option. In this case the mDataPtr points to an array of counts and pointers: the first pointer points to the 5-byte PD and the second points to the user data.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadPri	Unauthorized access priority
LLCTxError	Error in frame transmit or strip
LLCUnauthMAC	Unauthorized MAC frame
LLCTxTooLong	Invalid transmit frame length
LLCBadRefNum	Invalid RefNum
LLCRoutingError	Invalid routing information length
LLCBadFrame	Invalid frame type
LLCCancelled	Transmit cancelled

■ **Example 3-6** mOData structure; SNAPTransmit requests

```
typedef struct
{
     word      Reserved;      /* Reserved - always zero */
     word      InfoLen;       /* Length of information placed in buffer */
     word      Options;       /* Options:
                               *      Bits 15-8:     Unused
                               *      Bit 7: List-directed
                               *      Bits 6-0:      Unused
                               */
     byte      FrameType;     /* Frame type
                               * Specifies frame to send:
                               *      03      UI frame       0B      Test cmd (p=1)
                               */
     byte      FS;            /* Returns frame status */
     LANHdr    *Hdr    /* Pointer to LANHdr (N/A to LLCT2Transmit) */
} SNAPTxOData;
```

SNAPDetach

The SNAPDetach message deactivates a SNAP protocol. All outstanding SNAPReceives are cancelled, and if a listener was in use on the protocol being detached, it will no longer be called.

Refer to *Example 3-7* for the structure of mOData in the IPC message.

Result codes	*Value*	*Description*
	SNAPNoErr	Normal completion
	SNAPNotAttached	Invalid RefNum

■ **Example 3-7** mOData structure for SNAPDetach and SNAPCancel requests

```
typedef struct
{
      word    PDRefNum;           /* RefNum of SNAP protocol discriminator
*/
} SNAP_PD_RefNum;
```

SNAPReceive

The SNAPReceive message is used to receive frames from an attached SNAP protocol.

Refer to *Example 3-8* for the structure of mOData in the IPC message and to LLC interface documentation for the description of the LANHdr structure that may be pointed to by Hdr.

mDataPtr points to either a buffer to receive the frame or, if the "list-directed" bit of the Options field is set, to an array of counts and pointers. See *Example 3-9* for the structure of the array of counts and lengths.

It is possible to separate the 5 bytes of the PD from the user data by using the list-directed option. In this case the mDataPtr points to an array of counts and pointers: the first pointer points to the 5-byte PD and the second points to the user data.

When list-directed, the number of elements in the list is determined by its size in bytes, given by the value of mDataSize. Note that multiple receives can be queued for any given RefNum.

Result codes

Value	*Description*
SNAPNoErr	Normal completion
SNAPNotAttached	Invalid RefNum
SNAPTruncated	Frame larger than provided buffer space
SNAPCancelled	Receive cancelled, either explicitly or by SNAPDetach

■ **Example 3-8** mOData structure for SNAPReceive requests

```
typedef struct
{
        word    PDRefNum;       /* RefNum of protocol discriminator */
        word    Options;        /* Options:
                                        *       Bits 15-8:      Unused
                                        *       Bit 7:          List-directed
                                        *       Bits 6-0:       Unused
                                        */
        word    InfoLen;        /* Number of bytes of data in the I-field */
        byte    FrameType;      /* Frame type received:
                                        *       03      UI frame
                                        *       08      Test resp (f=1)
                                        *       09      Test resp (f=0)
                                        */
        byte    Filler;                 /* Not used */
        LANHdr  *Hdr;   /* Pointer to area to receive header */
} LLCTxRxOData;
```

■ **Example 3-9** Structure for list-directed SNAPReceive requests

```
struct
{
        word    Count; /* Byte count for this transfer */
        byte    *Ptr;          /* Pointer for this transfer */
} array[];
```

SNAPCancel

The SNAPCancel message is used to cancel SNAPReceive requests outstanding on an attached SNAP protocol.

Refer to Example 3-7 for the structure of mOData in the IPC message.

Result codes

Value	*Description*
SNAPNoErr	Normal completion
SNAPNotAttached	Invalid RefNum

SNAPGetParms

The SNAPGetParms message is used to get the SNAP associated parameters. The message returns the Task ID of the associated LLC process and the RefNum of the SNAP's SAP (0xAA). The LLC process information is useful if the client process wishes to make calls directly to the LLC interface.

Example 3-10 shows the structure of the mOData in the IPC message.

Result code	*Value*	*Description*
	SNAPNoErr	Normal completion

■ **Example 3-10** Structure for SNAPGetParmsOData

```
typedef struct
{
       tid_type        LLCTID;
       word            SAPRefNum;
} SNAPGetParmsOData;
```

Functions supporting 802.2

In addition to the preceding SNAP messages, the interface supplies a library containing the following support function. Note that you must link the LLCSupportLib.O file with your code before using SNAPSwapHdr.

Name	Description
SNAPSwapHdr	Swap node addresses in LANHdr structure for return to sender

SNAPSwapHdr

The SNAPSwapHdr function is called using Pascal calling conventions. The function swaps the addresses in a LANHdr. This swapping would usually be done to respond to a Type 1 frame.

```
pascal void SNAPSwapHdr(LANHdr *Hdr);
```

Example program listing

The program listing presented below is a sample of how to invoke TokenTalk NB functions and perform a SNAPAttach. Note the use of #define statements that simplify program maintenance and insulate the code from extreme revisions in the TokenTalk interface code.

```
/*      Useful defined functions. */

#define         ODataAs(x,y)    ((x *)((y)->mOData))
#define         SDataAs(x,y)    ((x *)((y)->mSData))
#define         DPAs(x,y)       ((x *)((y)->mDataPtr))
#define         Reply(x,y)      \
{       tid_type t;\
        t = (x)->mFrom, (x)->mFrom = (x)->mTo, (x)->mTo = t;\
        (x)->mCode |= 1, (x)->mStatus = y;\
        Send(x);\
}

static tid_type        SNAP_TID;             /* TID of SNAP process */
static short   OurSNAPRefNum;                /* Our SNAP RefNum */

/*      This does a SNAPAttach */
#define         VendorID        0x000000      /* Vendor ID */
#define         ProtcolID       0x1234        /* Ethernet protocol 0x1234 */
```

```
message         *cmp;
long            Id;
SNAP_PD         pd;            /* How to get the SNAP_TID */
if ((cmp = GetMsg()) == 0)
        return;
cmp->mTo = SNAP_TID;           /* SNAP_TID acquired from previous name lookup */
cmp->mCode = SNAPAttach;
Id = cmp->mId;
pd.PD.PDc[0] = (VendorID >> 16) & 0xff;
pd.PD.PDc[1] = (VendorID >> 8) & 0xff;
pd.PD.PDc[2] = VendorID & 0xff;
pd.PD.PDc[3] = (ProtocolID >> 8) & 0xff;
pd.PD.PDc[4] = ProtocolID & 0xff;
(SNAP_PD *) cmp->mDataPtr = &pd;
cmp->mDataSize = sizeof(SNAP_PD);
Send(cmp);
cmp = Receive(Id, 0, 0, 0);
if (cmp->mStatus)
{
        FreeMsg(cmp);
        return;
}
OurSNAPRefNum = ODataAs(SNAPAttachReplyOData, cmp)->PDRefNum;
FreeMsg(cmp);
```

Chapter 4 The 802.2 LLC / IPC Interface

THIS CHAPTER DESCRIBES the programming function calls that support the 802.2 LLC / IPC interface. The 802.2 LLC / IPC interface provides the message-passing interface to the TMS380 chip set that implements the logical link control (LLC) for the token ring network. The 802.2 IPC interface described in this chapter works equally well with an LLC that is implemented in software rather than the TMS380 chip set. ■

General information

The IPC services use a name table to identify various IPC clients. Every client must register its name to use the name lookup functions provided by the IPC services. The TokenTalk NB implementation of the 802.2 LLC / IPC interface registers itself under three different types with the MR-DOS Name Manager. These types are

- LLC
- Token Ring LLC
- 4 MB Token Ring LLC

By providing several types for the LLC interface, a client can look for a generic LLC or a specific type of LLC by name.Associated with each of these types is the name that is passed in the LLCName field of the startup parameters. (Startup parameters are provided in the TokenTalk Prep file discussed in Chapter 6.) By convention, the name is "TokenTalkNB."

Client processes should limit the number of requests that they queue to the LLC. As a general guideline, no more than ten LLCReceive and ten LLCT1Transmit (or LLCT2Transmit) requests should be queued by a single client at once. With any more queued requests MR-DOS can run out of message buffers. One method to impose this limit is to allocate a fixed number of transmit buffers, receive buffers, and data buffers when the code is initialized and to keep the buffers in a linked list. Then, by removing entries from the list and returning them when a request finishes, only a limited number of requests await the LLC interface at any given time. Queueing several receive or transmit requests improves both the throughput and reliability, but the number of queued requests must never exceed the number of available MR-DOS message buffers.

The majority of functions described in this chapter support IPC messages to the 802.2 LLC. An additional function provides address swapping that swaps the source and destination addresses in the frame header.

The following list presents the requests that a client can issue to the 802.2 LLC. In the normal fashion, replies from LLC to these requests increment by one the mCode in the IPC message to indicate the reply.

mCode	**Meaning**	**See page**
LLCOpenSAP	Activate an individual or group SAP	39
LLCCloseSAP	Deactivate a SAP	42
LLCGetHdr	Return media header template	43
LLCGetConfig	Return LLC configuration information	45
LLCOpenStation	Allocate resources to support a Type 2 connection	46
LLCCloseStation	Terminate activity on a station and release the station	47
LLCConnectStation	Attempt to place local and remote stations into data transfer state	48
LLCModifyParams	Modify parameters associated with a SAP or link station	49
LLCReceive	Receive a frame from a SAP or link station	50
LLCReceiveCancel	Cancel outstanding receives on stations or SAPs	51
LLCT1Transmit	Send a Type 1 frame	52
LLCT2Transmit	Send a Type 2 frame (I frame)	53
LLCReset	Reset link stations and/or SAPs	54
LLCReturnBuffer	Return interface-owned buffer to LLC (no reply to this request)	55
LLCGetStatistics	Get link station statistics	56
LLCSetFunctionalAddr	Add/remove functional addresses	59
LLCStatus	Notifies client of status changes	58

The LLCStatus message is sent by the 802.2 LLC interface to a client. This message informs the client of status changes related to Type 2 LLC. There is no specific reply to this message.

Typical 802.2 LLC use

The IPC requests support both Type 1 and Type 2 logical link control (LLC). Type 1 is connectionless and uses the SNAP interface described in Chapter 3. Type 2 is connection-oriented and is not supported by the SNAP interface. A typical application for Type 2 is connection-oriented 3270 data stream protocol.

Refer to the *TMS380 Adapter Chipset User's Guide Supplement* for additional information.

SubNetwork Access Protocol (SNAP) is not supported for Type 2 connections; therefore, Type 2 connections depend on the 802.2 LLC interface described in this chapter. Token ring connections used by IBM, such as 3270 data stream protocols, exclusively use Type 2 data link services.

Type 2 services are connection-oriented. That is, the attached client must open further connections after opening the service access point (SAP). Type 2 services guarantee the delivery of all data link transmissions with proper sequencing, acknowledgments, and automatic retries. With Type 2 services, connections are established prior to any data transmissions between nodes wishing to communicate. These connection points between nodes are referred to as "link stations."

For example, consider a link station 1 that wishes to communicate with link station 2. Station 1 allocates a link resource and sends a connection request frame to station 2. If station 2 has the resources and is authorized to communicate with station 1, it returns a positive acknowledgment to the station 1 connection request. Assuming a positive acknowledgment is returned, a link is established and data transfer can occur in either direction. Once all data and all acknowledgments transferred, either station can send a disconnect request to close the link, which frees resources in both stations for other communications.

To establish communications for Type 2 operation, the attached client must first open a SAP, then open a link station associated with that SAP, and finally perform a connection request with the remote station. This sequence creates a link from the SAP in this node to another SAP in a different node. One link station can be associated with only one local SAP and only one remote SAP on one remote node. However, a single SAP may be associated with multiple link stations (*Figure 4-1*).

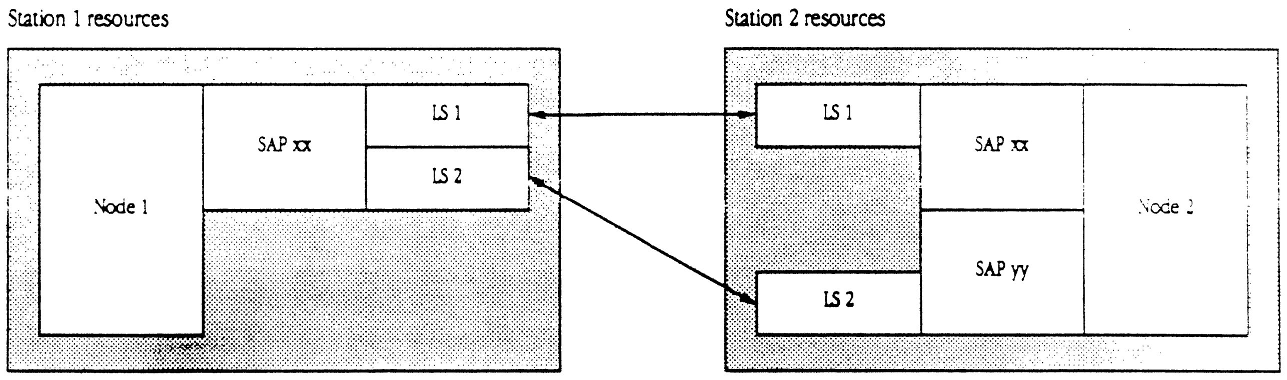
The following series of actions illustrates a typical usage for an LLC client using a Type 2 connection-oriented data link service:

1. Use the TokenTalkPrep Utilities (TTGetLLCTID) or the IPC name lookup to find the LLC service.
2. Optionally obtain configuration information by way of LLCGetConfig, which provides the maximum frame size and the physical limit for the maximum number of stations.
3. Issue LLCOpenSAP to begin LLC activity.
4. Obtain a header template by way of LLCGetHdr. The header can be copied after it is obtained, but it is important initially to use LLCGetHdr to build the LAN header with values supplied by the client (such as destination node). Different LLC implementations might assume a different header setup, so by using LLCGetHdr you insulate yourself from unnecessary problems. In general, the offset values supplied in the header should be left alone.

5. Queue receive requests to the SAP to accept incoming Type 1 frames by way of LLCReceive. Remember that XID frames necessary to establish Type 2 communication are transmitted as Type 1.
6. Obtain the address of the node that is to receive the Type 2 frame. The destination address can be obtained from a broadcast name lookup function, or it can be provided by a hard-wired table maintained on the network. A hardwired name table is site-dependent.
7. Issue LLCGetHdr with the destination node address.
8. Issue an LLCOpenStation request using a template.
9. Using a template, exchange XID frames as required with the destination node.
10. Using a template, issue LLCConnectStation to activate the Type 2 link station connection.
11. Issue LLCReceive requests to the link station to permit reception of information frames (I frames).
12. Issue transmit requests as required by way of LLCT2Transmit.
13. Reissue receive requests as the receive frames are returned.
14. When done with the link station, issue LLCCloseStation.
15. On completion, close the SAP by way of LLCCLoseSAP.

Establishing a link station requires a significant amount of resource. As a result, only a limited number of link stations can be open at any one time. The number of open link stations allowed is a parameter to LLC when it is first started. The number of available stations can be determined with LLCGetConfig.

■ **Figure 4-1** SAPs and link stations



IPC Requests to 802.2 LLC

In all structure declarations in this chapter, the "type" byte refers to an unsigned 8-bit integer, and "word" refers to an unsigned 16-bit integer.

All the structures and symbols used in this document are defined in the include file LLC.h. The include file OS.h contains the structures for the IPC messages referred to in this chapter.

In each case, mCode identifies the function. In the reply, mStatus holds the result code for the function. As is the convention with MR-DOS IPC, all requests have an even mCode value and all replies use the corresponding mCode plus one.

LLCOpenSAP

The LLCOpenSAP message activates either an individual or group SAP. This request also specifies various options and defaults associated with the particular SAP.

Example 4-1 shows the structure of mOData in the IPC message. *Example 4-2* shows the structure of the optional statton parameters that can be pointed to by mDataPtr. Station parameters can be set to default values by passing mDataPtr as zero.

The universal receive option on a SAP (Example 4-1) allows the SAP to receive all frames directed to it whether the frames are for an associated link station or for the SAP itself. In this case, a single posted receive will accept either a Type 1 or a Type 2 frame. When the universal receive option is used for Type 2 frames (I frame), the RefNum in the completed receive is replaced by the RefNum of the destination link station.

The universal receive option is provided as a convenience for some SAPs. When used, all receives should be queued to the SAP, and none to the link stations.

The listener function pointer is a special hook that for certain clients find useful to handle received frames more efficiently. Most clients should simply specify 0 for this particular bit. Any listener that is provided must be located on the same slot as the 802.2 interface. A listener function is called with parameters that pass the media header, information pointer, information length, and frame type. The listener must be finished with this buffer and header before returning.

An example declaration for the 802.2 LLC listener function might be coded as follows:

```
void    Sample_Listener (hp, bp, len, ft)
LANHdr  *hp;    /*Pointer to LANHdr of received frame*/
unsigned char *bp;          /*Pointer to I-field*/
int     len;                /*Length of I-field*/
int     ft;                 /*Frame type, 3-9*/
{
        if (bp[0] > 3)      /*If first byte of the I-field is > 3 ignore frame */
              return;

                            /* Other code to manipulate frame data*/

        return;
}
```

Result codes

Value	Description
LLCNoErr	Normal completion
LLCBadSAPOpts	Invalid SAP options
LLCBadPri	Unauthorized access priority
LLCMaxExceeded	Parameter exceeds maximum
LLCBadSAPValue	Invalid SAP value

Value	*Description*
LLCNoGroup	Requested membership in nonexistent group
LLCNoResources	The maximum number of SAPs are already open
LLCGroupLimit	The group SAP already has maximum members
LLCBadSize	mDataSize has inappropriate length

■ **Example 4-1** mOData structure for LLCOpenSAP and LLCModifyParams

```
typedef struct
{
        word    RefNum;                       /* Returns refnum assigned to SAP */
        word    Options;              /* Holds SAP options :
                                      /*      Bit 15:         "Universal" receive
                                       *      Bit 14:         "Listener" is a Listening
function
                                       *      Bits 13-9:Unused
                                       *      Bit 8: Reserved
                                       *      Bits 7-5:       Access priority
                                       *      Bit 4: Unused
                                       *      Bit 3: Handle XIDs
                                       *      Bit 2: Individual SAP
                                       *      Bit 1: Group SAP
                                       *      Bit 0: Unused
                                       */
        byte    SAPValue;             /* Holds desired SAP number (individual or group)
*/
        byte    StationCnt;           /* Max. link stations for this SAP */
        byte    FailedGSAP;           /* Returns failing GSAP */
        byte    Unused;                       /* Unused */
        void    (*Listener)();        /* Pointer to optional listening function */
} LLCSAPOData;
```

■ **Example 4-2** mODataPtr structure for LLCOpenSAP, LLCOpenStation, and LLCModifyParams

```
typedef struct
{
        byte    TimerT1;        /* Response timer value (default = 5) */
        byte    TimerT2;        /* Receive Acknowledge timer value (default = 2) */
        byte    TimerTI;        /* Inactivity timer value (default = 3) */
        byte    MaxOut;                 /* Max. no. of outstanding Tx I frames */
        byte    MaxIn;          /* Max. no. of outstanding Rx I frames */
        byte    MaxOutIncr;     /* Dynamic windowing increment */
        byte    MaxRetries;     /* Max. no. of retransmissions */
        byte    GSAPMaxMem;     /* Max. no. of members of a GSAP */
        word    MaxIField;      /* Max. length of I-field */
        byte    GCnt;           /* Number of GSAPs to join */
        byte    GSAP[8];        /* Up to eight group SAPs */
  LLCStationParms;
```

Note that the GCnt and GSAP fields are not used in LLCOpenStation requests and that GSAPMaxMem and MaxIField are not used in LLCModifyParams requests.

The timers all range in value from 0 (for default) to 10. For TimerT1, values in the range 1–5 use the corresponding group 1 timer interval which is 200 ms. Values in the range 6–10 use the group 2 timer interval which is 1 second.

For TimerT2, values in the range 1–5 use the corresponding group 1 timer interval which is 40 ms. Values in the range 6–10 use the group 2 interval which is 400 ms.

For TimerTI, values in the range 1–5 use the corresponding group 1 timer interval which is 1 second. Values in the range 6–10 use the group 2 interval which is 5 seconds.

LLCCloseSAP

The LLCCloseSAP message deactivates a SAP. An individual SAP should not be closed until all link stations associated with the SAP are closed. Likewise, a group SAP should not be closed as long as the group has any SAPs as members. A SAP can be removed from a group by LLCCloseSAP (for that SAP), or by LLCModifyParams. *Example 4-3* shows the structure of mOData in the IPC message.

Result codes	*Value*	*Description*
	LLCNoErr	Normal completion
	LLCBadRefNum	Invalid RefNum
	LLCLinkOpen	Unclosed link stations on SAP
	LLCSAPOpen	Group SAP cannot close—all member SAPs are not closed
	LLCSeqError	Sequence error

■ **Example 4-3** mOData structure for LLCCloseSAP, LLCCloseStation, LLCReceiveCancel, and LLCReset

```
typedef struct
{
	word	RefNum;			/* RefNum of SAP to close */
} LLCRefNumOData;
```

LLCGetHdr

The LLCGetHdr message creates a LANHdr structure that is used to receive, transmit, open a station, or connect a station to or from the specified node. Options are also provided to return broadcast header templates.

Example 4-4 shows the structure of mOData in the IPC message, and *Example 4-5* shows the structure of the LANHdr structure that is returned to the area pointed to by the Hdr field in mOData. mDataPtr points to the node address and mDataSize indicates the size of that address in bytes.

Not all media support all possible options. When an unsupportable option is specified, the IPC interface builds the best header it can and returns the status LLCNotFullySupported to the client. mDataSize must either be zero to get a broadcast template or be the exact size of a node address for the underlying media.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCNotFullySupported	Some option or type requested is not fully supported by this media
LLCAddrError	Invalid remote address—size must be 0 or node address size for the media

■ **Example 4-4** mOData structure for LLCGetHdr requests

```
typedef struct
{
    word      HdrType;  /* Header types:
                         *      0 - Normal header
                         *      1 - Single-route b'cast, all-routes b'cast return
                         *      2 - Single-route b'cast, non-broadcast return
                         *      3 - All-routes broadcast header
                         */
    word      Options;  /* Header options: Always zero */
    byte      SSAP;     /* Source SAP value */
    byte      DSAP;     /* Destination SAP value */
    LANHdr    *Hdr;     /* Pointer to LANHdr structure to be returned */
} LLCGetHdrOData;
```

■ **Example 4-5** LANHdr structure for LLCGetHdr used by LLCOpenStation, LLCConnectStation, LLCReceive, and LLCTransmit

```
typedef struct
{
       byte    Media;          /* Media indicator:
                                            *     0 - Unknown
                                            *     1 - LocalTalk
                                            *     2 - 10 Mb Ethernet
                                            *     3 - 4 Mb Token Ring
                                            *     4 - 16 Mb Token Ring
                                            *     5 - FDDI
                                            *     6 - Token Bus
                                            */
       byte    Routing;               /* Source-routing indicator:
                                            *     0 - No source-routing
                                            *     1 - IBM source-routing
                                            */
       byte    DOff;           /* Offset to destination address in header buffer */
       byte    DLen;           /* Length of destination address in header buffer */
       byte    SOff;           /* Offset to source address in header buffer */
       byte    SLen;           /* Length of source address in header buffer */
       byte    ROff;           /* Offset to routing information in header buffer */
       byte    RLen;           /* Length of routing information in header buffer */
       byte    HOff;           /* Offset to media header in header buffer */
       byte    HLen;           /* Length of media header in header buffer */
       byte    DSAP;           /* Destination SAP value */
       byte    SSAP;           /* Source SAP value */
       byte    HBuf[40];       /* Header buffer */
} LANHdr;
```

LLCGetConfig

The LLCGetConfig message returns configuration information about LLC. *Example 4-6* shows the structure returned to the address passed in mDataPtr.

Result codes	*Value*	*Description*
	LLCNoErr	Normal completion
	LLCTruncated	Buffer too short to receive all information

■ **Example 4-6** Structure mDataPtr points to following completion of LLCGetConfig

```
typedef struct
{
    long    LLCVersion;     /* LLC Version ID */
    long    FAddr;          /* Functional address (token ring only) */
    long    G1Timer1;       /* Group 1 timer 1 (response timer) in milliseconds */
    long    G2Timer1;       /* Group 2 timer 1 (response timer) in milliseconds */
    long    G1Timer2;       /* Group 1 timer 2 (receive ack) in milliseconds */
    long    G2Timer2;       /* Group 2 timer 2 (receive ack) in milliseconds */
    long    G1ITimer;       /* Group 1 inactivity timer in milliseconds */
    long    G2ITimer;       /* Group 2 inactivity timer in milliseconds */
    word    MaxFrameLen;    /* Maximum frame length */
    word    ASAPs;          /* Available SAPs */
    word    AStations;      /* Available stations */
    word    MaxHeader;      /* Maximum header size for this media */
    byte    LLCClass;       /* Class of LLC implementation:
                                *      1 - Implements type1 only
                                *      2 - Implements type1 and Type 2
                                */
    byte    Media;          /* Media indicator:
                                *      0 - Unknown
                                *      1 - LocalTalk
                                *      2 - 10 Mb Ethernet
                                *      3 - 4 Mb Token Ring
                                *      4 - 16 Mb Token Ring
                                *      5 - FDDI
                                *      6 - Token Bus
                                */
    byte    Routing;    /* Source-routing indicator:
                                *      0 - No source-routing
                                *      1 - IBM source-routing
                                */
    byte    AddrLen;    /* Length of node address in bytes */
    byte    Addr[9];    /* This node's address */
    byte    NumGAddrs;      /* The number of group addresses that follow */
    byte    GAddrBuf[1];    /* Start of group addresses (length, address pairs) */
} LLCGetConfigBuffer;
```

LLCOpenStation

The LLCOpenStation message allocates resources to support a connection between two stations.

Refer to *Example* 4-7 for the structure of mOData in the IPC message. Refer back to Example 4-2 for the structure of the station parameters that can be pointed to by mDataPtr. The station parameters can be set to default values by passing mDataPtr as zero. Example 4-5 shows the LANHdr structure that is pointed to by the Hdr field in mOData.

The RefNum parameter (Example 4-3) holds the RefNum of the local SAP when the request is made and returns the new station's RefNum on successful reply. The high byte of the RefNum is the reference number for the SAP, and the low byte is the reference number of the link station.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadPri	Unauthorized access priority
LLCBadRefNum	Invalid RefNum
LLCMaxExceeded	Parameter exceeded maximum
LLCBadSAPValue	Invalid SAP value or SAP value already in use
LLCNoResources	Maximum number of link stations are already open
LLCAddrError	Invalid remote address—group address invalid
LLCBadSize	mDataSize has inappropriate value

■ **Example 4-7** mOData structure for LLCOpenStation and LLCConnectStation requests

```
typedef struct
{
        word    RefNum;                 /* Returned station refnum */
        word    StaOpts;                /* Station options:
                                                *       Bits 15-8:      Unused
                                                *       Bits 7-5:       Priority
                                                *       Bits 4-0:       Unused
                                                */
        LANHdr  *Hdr;           /* Pointer to LANHdr holding
                                                        remote node address */
} LLCStationOData;
```

LLCCloseStation

The LLCCloseStation message causes a link station to go to a closed state. Refer back to Example 4-3 for the structure of mOData in the IPC message.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadRefNum	Invalid RefNum
LLCClosedNoAck	Station closed without remote acknowledgment
LLCSeqError	Sequence error—have already issued a close to this link

LLCConnectStation

The LLCConnectStation message places the local and remote link stations into the data transfer state. Example 4-7 shows the structure of mOData in the IPC message. Refer back to Example 4-5 for the LANHdr structure pointed to by the contents of mOData.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadRefNum	Invalid RefNum
LLCProtoError	Protocol error—link in invalid state for command
LLCRoutingError	Invalid routing information length
LLCConnSeqError	Connect sequence error
LLCConnFail	The remote station did not accept the connection request

LLCModifyParams

The LLCModifyParams message is used to modify either open link station parameters or default SAP parameters.

Refer back to Example 4-1 for the structure of mOData in the IPC message and to Example 4-2 for the structure of the station parameters that can be pointed to by mDataPtr. Specification of the station parameters can be set to default values by passing mDataPtr as zero. The only fields in the LLCSAPOData structure used for this request are RefNum and the access priority in SAPOptions. Any GSAP addition that failed is returned in FailedGSAP.

If the low-order bit of a specified GSAP is zero, the specified group SAP membership should be added. If the low-order bit is one, the specified group SAP membership should be cancelled.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadPri	Unauthorized access priority
LLCBadRefNum	Invalid RefNum
LLCMaxExceeded	Parameter exceeded maximum
LLCNoGroup	Requested group membership in nonexistent group SAP
LLCGroupLimit	Group SAP has maximum members
LLCNotMember	Member not found in group SAP
LLCBadSize	mDataSize has inappropriate value

LLCReceive

The LLCReceive message is used to receive frames from a link station or a SAP. *Example 4-8* shows the structure of mOData in the IPC message,and Example 4-5 shows the description of the LANHdr structure that can be pointed to by Hdr.

mDataPtr points either to a buffer that receives the frame or, if the "list-directed" bit of the Options field is set, to an array of counts and pointers. See *Example 4-9* for the structure of the array of counts and lengths.

Do not use he "list-directed" option in conjunction with a SAP listener function (see LLCOpenSAP). The number of elements in the list is determined by mDataSize, as usual. Multiple receives can be queued for any given RefNum, which allows reception of Type 1 or Type 2 frames. Normally this interface requires the receiver to provide the buffer space. If the "use interface buffer" bit of the Options field is set, the interface fills in addresses for mDataPtr and Hdr. When the "use interface buffer" feature is used, the client initially passes mDataPtr and Hdr as zero and passes back to LLC any buffer that is present on completion of the receive. After completion the buffers are returned to the interface by reissuing the receive or by issuing LLCReturnBuffer.

Result codes	*Value*	*Description*
	LLCNoErr	Normal completion
	LLCBadRefNum	Invalid RefNum
	LLCMsgReject	Unusual interface error
	LLCCancelled	Receive cancelled, either explicitly or by close operation
	LLCBadPointer	Bad pointer passed as "interface-owned"

LLCReceiveCancel

The LLCReceiveCancel message is used to cancel all outstanding receives on either a link station or a SAP. Refer back to Example 4-3 for the structure of mOData in the IPC message.

Result codes	*Value*	*Description*
	LLCNoErr	Normal completion
	LLCBadRefNum	Invalid RefNum

■ **Example 4-8** mOData structure, LLCReceive, LLCT1Transmit, and LLCT2Transmit requests

```
typedef struct
{
    word    RefNum;     /* RefNum for link station or SAP */
    word    Options;    /* Options:
                                *   Bits 15-8:  Unused
                                *   Bit 7:      List-directed
                                *   Bit 6:      Use interface buffer
                                                (LLCReceive only)
                                *   Bits 5-0:   Unused
                                */
    word    InfoLen;    /* Length of information placed in buffer */
    byte    FrameType;  /* Returns received frame type (on LLCReceive) or
                            * Specifies frame to send (on LLCT1Transmit):
                                *                       06  XID resp (f=1)
                                *   02  I frame         07  XID resp (f=0)
                                *   03  UI frame        08  Test resp (f=1)
                                *   04  XID cmd (p=1)   09  Test resp (f=0)
                                *   05  XID cmd (p=0)   0B  Test cmd (p=1)
                            */
    byte    FS;             /* Returns frame status (token-ring  Type 1 only) */
    LANHdr  *Hdr        /* Pointer to LANHdr (N/A to LLCT2Transmit) */
} LLCTxRxOData;
```

■ **Example 4-9** Structure for list-directed receives and transmits

```
struct
{
        word    Count;          /* Byte count for this transfer */
        byte    *Ptr;           /* Pointer for this transfer */
} array[];
```

LLCT1Transmit

The LLCT1Transmit message is used to send a Type 1 frame. Refer to Example 4-8 for the structure of mOData in the IPC message and to Example 4-5 for the description of the LANHdr structure that can be pointed to by Hdr.

mDataPtr points either to a frame holding the buffer or, if the "list-directed" bit of the Options field is set, to an array of counts and pointers, as with receive. See Example 4-9 for the structure of the list-directed transmit array. Typically, FrameType 3 is used for Type 1 transmissions.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadPri	Unauthorized access priority
LLCTxError	Error in frame transmit or strip
LLCUnauthMAC	Unauthorized MAC frame
LLCTxTooLong	Invalid transmit frame length
LLCBadRefNum	Invalid RefNum
LLCRoutingError	Invalid routing information length
LLCBadFrame	Invalid frame type
LLCCancelled	Transmit cancelled

LLCT2Transmit

The LLCT2Transmit message is used to send a Type 2 frame (I frame) through a link station. Refer to Example 4-8 for the structure of mOData in the IPC message and to Example 4-5 for the description of the LANHdr structure that can be pointed to by Hdr.

mDataPtr points either to a frame holding the buffer or, if the "list-directed" bit of the Options field is set, to an array of counts and pointers, as with receive. See Example 4-9 for the structure of the list-directed transmit array. Typically, FrameType 2 is used for Type 2 transmissions.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadPri	Unauthorized access priority
LLCTxError	Error in frame transmit or strip
LLCUnauthMAC	Unauthorized MAC frame
LLCNoIFrames	Link not transmitting I frames
LLCTxTooLong	Invalid transmit frame length
LLCBadRefNum	Invalid RefNum
LLCProtoError	Protocol error—I frame issued before DMA ready
LLCCancelled	Transmit cancelled

LLCReset

The LLCReset message reset san individual link station or a SAP and all of its link stations. Be certain to use the correct RefNum so that a link station is not inadvertently reset. Refer to Example 4-3 for the structure of mOData in the IPC message.

Result codes

Value	*Description*
LLCNoErr	Normal completion
LLCBadRefNum	Invalid RefNum

LLCReturnBuffer

The LLCReturnBuffer request is used to return interface-owned receive buffers to the interface. Normally this is not needed since requeuing the receive also returns the buffer. However, a race condition can occur when closing a SAP that can result in the client receiving completed receive requests and yet not being ablē to requeue the receives because an LLCCloseSAP request has already been sent. When this rare event occurs, the LLCReturnBuffer request is used to return the buffers.

To return a buffer, set mDataPtr to the buffer address and place the address of the header in mOData[0]. This message has no reply.

Result codes There are no result codes for this function because there is no reply.

LLCGetStatistics

The LLCGetStatistics message is used to get statistics for a link station. Refer to *Example 4-10* for the structure of mOData in the IPC message and to *Example 4-11* for the description of the structure returned to the area pointed to by mDataPtr. The type of statistics tracked include

- Number of I frames sent and received
- Number of I frame errors sent and received
- T1 timer expirations
- Last command/response sent or received
- Primary and secondary link states

Result codes	*Value*	*Description*
	LLCNoErr	Normal completion
	LLCTruncated	Returned data incomplete due to inadequate buffer space
	LLCBadRefNum	Invalid RefNum

- **Example 4-10** mOData structure for LLCGetStatistics requests

```
typedef struct
{
        word    RefNum;            /* RefNum of link station */
        word    Options;           /* Options:
                                          *     Bit 15:     Clear error counters
                                          *                 after returning statistics
                                          *     Bits 14-0:  Unused
                                          */
        word    ActLen;            /* Actual length of buffer returned */
} LLCGetStatisticsOData;
```

■ **Example 4-11** mDataPtr pointer to LLCGetStatistics buffer

```
typedef struct
{
       word    NumITx;               /* Number of I  frames sent */
       word    NumIRx;               /* Number of I  frames received */
       byte    NumIRxErr;      /* Number of bad I  frames received */
       byte    NumITxErr;      /* Number of I frames sent ending in error */
       word    NumT1Exp;       /* Number of times T1 expired when not */
                               /*     transferring data */
       byte    LastCmdRx;      /* Last command/response rcvd (LLC byte 0) */
       byte    LastCmdTx;      /* Last command/response sent */
       byte    PriState;       /* Link primary state:
                                       *     Bit 7: Closed
                                       *     Bit 6: Disconnected
                                       *     Bit 5: Disconnecting
                                       *     Bit 4: Opening
                                       *     Bit 3: Resetting
                                       *     Bit 2: FRMR Sent
                                       *     Bit 1: FRMR Received
                                       *     Bit 0: Opened
                                       */
       byte    SecState;       /* Link secondary state:
                                       *     Bit 7: Checkpointing
                                       *     Bit 6: Local busy (user set)
                                       *     Bit 5: Local busy (system)
                                       *     Bit 4: Remote busy
                                       *     Bit 3: Rejection
                                       *     Bit 2: Clearing
                                       *     Bit 1: Dyn. win. running
                                       *     Bit 0: reserved
                                       */
       byte    TxState;              /* Send state variable V(S) */
       byte    RxState;              /* Receive state variable V(R) */
       byte    LastNR;               /* Last received N(R) */
       byte    Unused;               /* Unused */
       LANHdr Hdr;             /* LANHdr used to send I frames */
} LLCGetStatisticsBuffer;
```

LLCStatus

The 802.2 LLC interface sends the LLCStatus message to the client of a particular link that has changed status. There is no specific reply to this message. Refer to *Example 4-12* for the structure of mSData in the IPC message and to *Example 4-13* for the structure of mOData in the IPC message.

■ **Example 4-12** mSData structure in LLCStatus messages

```
typedef struct
{
        word    RefNum;             /* RefNum of link station */
        word    Status;             /* LLC status bits:
                                     *      Bit 15:Link lost
                                     *      Bit 14:Disconnected
                                     *      Bit 13:FRMR rcv'd
                                     *      Bit 12:FRMR sent
                                     *      Bit 11:SABME rcv'd
                                     *      Bit 10:Opened link
                                     *      Bit 9:Remote busy
                                     *      Bit 8:Remote not busy
                                     *      Bit 7: TI expired
                                     *      Bit 6: Counter ovfl.
                                     *      Bit 5: Priority reduced
                                     *      Bits 4-0: Reserved
                                     */
        byte    FRMR[5];       /* Holds FRMR response (if bit 12 or 13 set) */
        byte    Priority;      /* Holds access priority (if bit 5 set) */
} LLCStatusSData;
```

■ **Example 4-13** mOData structure in LLCStatus messages

```
typedef struct
{
        byte    AddrLen;       /* Length of remote node address in bytes */
        byte    Addr[9];       /* Holds remote node address (if bit 10 set) */
        byte    RSAP;          /* Holds remote SAP value (if bit 10 set) */
} LLCStatusOData;
```

LLCSetFunctionalAddr

The LLCSetFunctionalAddr message is used to add or remove bits to the functional address. Refer to *Example 4-14* for the structure of mOData in the IPC message.

Result code	*Value*	*Description*
	LLCNoErr	Normal completion

■ **Example 4-14** mOData structure for LLCSetFunctionalAddr

```
typedef struct
{
	word	Option;			/* If zero, add otherwise remove */
	long	Addr;		/* Mask of bits to add or remove */
} LLCSetFunctionalAddrOData;
```

Functions supporting 802.2

In addition to the previous IPC messages, the 802.2 interface supplies a library containing the following support function:

Name	Description
LLCSwapHdr	Swap addresses in LANHdr structure for return to sender

LLCSwapHdr

The LLCSwapHdr function swaps the addresses in a LANHdr. This swapping usually be done to respond to a Type 1 frame. Normally the caller provides its own SAP value for the SSAP. The SSAP for the swapped header cannot be taken from the DSAP in the header because the DSAP might be a group SAP, and group SAPs cannot be SSAPs.

Call the LLCSwapHdr function by using Pascal calling conventions:

```
pascal void LLCSwapHdr(LANHdr *Hdr, byte SSAP);
```

Chapter 5 **Apple IPC Services**

THIS CHAPTER PRESENTS the Apple interprocess communication (IPC) services provided by the Macintosh Operating System on the Macintosh II. IPC services provide a means of passing messages between processors that reside on the NuBus. An Apple smart card, such as the TokenTalk NB card, has its own on-card operating system called Minimal Realtime Distributed Operating System (MR-DOS). The MR-DOS IPC interface is fully described in the *Macintosh Coprocessor Platform Developer's Guide.* This chapter summarizes the applicable Apple IPC services , which are fully described in the Developer's Guide. ■

General information

The code for MR-DOS and Apple IPC includes a collection of traps, interrupt handlers, and tasks that provide support for process naming, timing services, and intercard and intracard communications using messages. These routines enable a smart card to support a multitasking distributed operating environment for communications and other real-time services on the same card or on other smart cards installed in the Macintosh II computer.

Interprocess communication is accomplished through communication messages that are fixed-size but flexibly formatted. MR-DOS allows dynamic name-binding of tasks to support interprocess communication.

Apple IPC (InterProcess Communication) is a combination of a driver and support software found in the Apple IPC file in the Apple IPC folder on the distribution disk. Apple IPC provides message-passing and naming services for communications from the Macintosh II to tasks on smart cards such as the TokenTalk NB card. Apple IPC is similar to the InterCard Communications Manager on MR-DOS.

Apple IPC is a driver and associated interface code in the form of a library that runs under the Macintosh Operating System. The Apple IPC driver handles all message passing (interprocess communication) between processes on the Macintosh II Operating System and Macintosh Coprocessor (MCP) card tasks on the NuBus.

Periodically, Apple IPC scans for and processes incoming messages, receives calls that have timed out, activates slots that have timed out, and processes outgoing messages. The driver receives messages from and delivers messages to Macintosh II processes using calls to Apple IPC driver.

An application that uses Apple IPC must have an initial call to OpenQueue to establish its use of Apple IPC. Messages are sent and received by way of the Send and Receive calls, much like tasks under MR-DOS. Several source-language examples of applications are provided in the Apple IPC folder on the distribution disk.

Apple IPC driver

Apple IPC services are handled by the Apple IPC driver, which handles all message passing between processes on the Macintosh II operating system and smart card processes on the NuBus. The Macintosh II process sends to and receives from smart card processes by way of calls to the Apple IPC driver.

The Apple IPC file is placed in the System Folder; routines contained in the file are installed by the INIT 31 mechanism during system startup.

During initialization, the driver sets up a communication area. It then searches NuBus slots for the InterCard Communication Manager (ICCM) communication areas of smart cards installed in the Macintosh II, much as the MR-DOS ICCM does. For each valid communication area found, the driver stores the address of the Apple IPC communication area in a vector in the smart card's communication area.

Periodically, Apple IPC scans for and processes Receive operations that have timed out, incoming messages, active slots that have timed out, and outgoing messages. The driver receives messages from and delivers messages to the Macintosh II processes.

Apple IPC library

The object routines, or glue code, in the Apple IPC library provide the interface between a Macintosh application and the Apple IPC driver. These routines provide for opening and closing the message queue to the driver, getting and freeing message buffers, and sending and receiving messages.

In addition, the Apple IPC library provides access to many of the same utilities that MR-DOS provides, such as moving data, obtaining the operating environment, and registering and looking up task names through the Apple IPC Name Manager. These routines are located in the Apple IPC:IPCGlue.o file on the distribution disk. The C language calling sequence is used in all of these routines.

Apple IPC managers

The managers for Apple IPC are the Echo Manager and the Name Manager. These Apple IPC managers perform functions identical to and have the same message interface as those of their MR-DOS counterparts; minor differences are due to the slightly different interface with Apple IPC.

The Apple IPC managers are tasks that carry out higher level services on behalf of applications on the Macintosh II computer. These managers are often referred to as *slot 0 managers*, and the Macintosh itself is sometimes referred to as the *slot 0 card*.

◆ *Note:* The slot 0 card is not to be confused with the Slot Manager in the Macintosh II (part of the Macintosh Operating System).

Using Apple IPC

To establish its use of Apple IPC, an application must have an initial call to OpenQueue to establish its use of IPC. Each process that uses Apple IPC requests that a queue be opened for messages addressed to that process.

Much like tasks under MR-DOS, messages are sent and received in Apple IPC by way of the Send call and the Receive call.

- When the driver gets a Receive request and no completion routine is specified, it searches the queue for a message matching the criteria specified. If it finds a matching message, the driver returns to the process. If it finds no matching message, the driver either returns immediately or, depending on the timeout specified, lets the process wait for a matching message (indefinitely if the timeout is 0, or until the timeout is reached). Waiting is handled by placing the process in an internal timeout queue.
- The Receive request behaves differently when a completion routine is specified. Additional information on the Receive call follows in this chapter.
- If a Send request is destined for a process on the Macintosh II, the destination process is unblocked, if waiting, or the message is placed in its queue. If the message is destined for a task on a smart card, the message is transferred to the ICCM on that slot for delivery to the task.

Apple IPC services

This section describes the Apple IPC services and provides examples of how to call primitives from both C and assembly language. These services are provided to support features similar to those of MR-DOS for applications running on the Macintosh II computer. The *MCP Developer's Guide* contains additional information on both MR-DOS and Apple IPC.

◆ *Note*: As with MR-DOS, Apple IPC uses C calling conventions, and all registers are preserved except D0, D1, A0, and A1. Calls in both C and assembly language take arguments and use similar data structures. Any macros referred to in this chapter are for examples only and do not exist on the distribution disk at this time.

Table 5-1 briefly describes the services provided by Apple IPC.

■ **Table 5-1** Apple IPC services

Name	Description
CloseQueue	Closes an Apple IPC queue
CopyNuBus	Copies a block of data from the source address to the destination address
FreeMsg	Frees a message buffer
GetCard	Returns the NuBus slot number on which the calling process is running
GetETick	Returns the number of major ticks since the operating system started
GetICCTID	Returns the task identifier of the InterCard Communication Manager
GetIPCg	Returns the address of the global data area within the Apple IPC driver
GetMsg	Gets message buffer
GetNameTID	Returns the task identifier of the Name Manager
GetTickPS	Returns the number of major ticks in one second
GetTID	Returns the task identifier of the calling task
IsLocal	Returns an indication of the locality of an address
KillReceive	Cancels an outstanding receive request
Lookup_Task	Returns the task identifier of the task that matches the object and type names specified
OpenQueue	Opens an Apple IPC queue
Receive	Receives a message
Register_Task	Allows a task to register itself with the object and type names specified
Send	Sends a message
SwapTID	Swaps the mFrom and mTo fields in a message buffer

CloseQueue

CloseQueue closes the queue that was previously opened. Make this IPC call last prior to terminating an entitiy.

The C declaration for CloseQueue is

```
void    CloseQueue();
```

The following example provides an equivalent of CloseQueue in assembly language.

```
JSR     CloseQueue
```

CopyNuBus

CopyNuBus copies a block of data and does a simple move of bytes from the source to the destination, without checking for overlapping source and destination addresses. The number of bytes is specified in the count parameter. The source address and destination address may be either Macintosh main memory or memory on a smart card. This routine deals with the complexity of potential 32-bit NuBus addresses for the source and the destination, but it does not deal with the possibility of overlapping buffers.

▲ **Warning** Don't overlap the source and destination blocks. Doing so could cause partial overwriting of the destination block. ▲

The C declaration for CopyNuBus is

```
void CopyNuBus ( source, destination, count );
        char    *source;        /* Address of source buffer */
        char    *destination;   /* Address of destination buffer */
        unsigned short          count; /* Byte count */
```

The following example provides an equivalent of CopyNuBus in assembly language.

```
        MOVE.L #Count,-(A7)
        PEA             Destination
        PEA             Source
        JSR             CopyNuBus
        ADD.L           #12,A7
```

FreeMsg

FreeMsg frees a message buffer that was acquired earlier by a call to GetMsg.

The number of messages initially available depends on the number requested in the named resource Apple IPC entries of type `aipn` in the Apple IPC driver file.

The C declaration of FreeMsg is

```
        void    FreeMsg( mptr );
        message         *mptr;          /* pointer to message buffer to free */
```

The form for the FreeMsg macro is as follows, where P1 is the address of the message buffer to be freed:

```
        [Label]         FreeMsg         P1
```

To indicate the location containing the desired address, P1 can be specified as a register (A0–A6, D0–D7), or by using any 68000 addressing mode valid in an LEA instruction.

GetCard

GetCard returns the NuBus slot number on which the calling process is running. For the Macintosh II computer, the number returned is always zero.

The C declaration for GetCard is

```
        char    GetCard ();
```

The following example provides an equivalent of GetCard in assembly language. On return, D0 contains the NuBus slot number on which the calling process is running.

```
        JSR     GetCard
```

GetETick

GetETick returns the number of major ticks—that is, the elapsed time in ticks—since the operating system started.

The C declaration for GetETick is

```
    unsigned long           GetETick();
```

The following example shows how to call GetETick using assembly language. To return the number of major ticks, get the value of location `gMajorTick` in the `gCommon` data area.

```
    JSR       GetETick
```

◆ *Note*: A tick on the Macintosh II is of a different duration than that on an MCP card.

GetICCTID

GetICCTID returns the task identifier of the InterCard Communication Manager.

The C declaration for GetICCTID is

```
        tid_type      GetICCTID ();
```

An equivalent of GetICCTID in assembly language is given in the following example. On return, D0 contains the task identifier of the ICCM.

```
        JSR     GetICCTID
```

GetIPCg

GetgIPCg returns the address of the data area of the Apple IPC driver. This routine is an aid for advanced developers. Refer to the include files on your distribution disk for the structure of IPCg.

The C declaration for GetgIPCg is

```
        struct IPCg *GetIPCg();
```

The following example provides an equivalent of GetIPCg in assembly language. On return, D0 contains the address of the data area of the Apple IPC driver.

```
        JSR     GetIPCg
```

▲ **Warning** Use this call at your own risk. Subject to change with no notice. ▲

GetMsg

GetMsg requests a message buffer from the free-message pool. GetMsg returns either a pointer to the allocated message or zero. A FreeMsg call releases the message.

All fields in the message, except message ID (mID) and the From address (mFrom), are cleared before the pointer to the message is returned. Message ID is a message field set to a number that is statistically unique; the From address is a message field set to the current task identifier.

The C declaration of GetMsg is

```
        message         *GetMsg();
```

The form for the GetMsg macro is

```
        [Label]         GetMsg
```

The address of the allocated message buffer is returned in D0 unless no buffer was available. In that case, 0 is returned in D0.

GetNameTID

GetNameTID returns the task identifier of the Name Manager. The C declaration for GetNameTID is

```
        tid_type        GetNameTID ();
```

The following example gives an equivalent of GetNameTID in assembly language. On return, D0 is the task identifier of the Name Manager.

```
        JSR     GetNameTID
```

GetTickPS

GetTickPS returns the number of major ticks in one second.

The C declaration for GetTickPS is

```
        unsigned short        GetTickPS ();
```

The following example provides an equivalent of GetTickPS in assembly language. On return, D0 is the number of major ticks in one second.

```
        JSR     GetTickPS
```

GetTID

GetTID returns the task identifier of the calling task.

The C declaration for GetTID is

```
        tid_type        GetTID ();
```

The following example provides an equivalent of GetTID in assembly language. On return, D0 is the task identifier of the calling task.

```
        JSR     GetTID
```

IsLocal

IsLocal returns true or false to indicate whether an address is local.

The C declaration for IsLocal is

```
        short   IsLocal(address)
        char    *address;       /* address to test. */
```

IsLocal returns true (nonzero) if the address passed is local, false (zero) if it is a remote NuBus address.

The form for the IsLocal macro is as follows, where P1 is the address to examine.

```
        [Label]         IsLocal         P1
```

To indicate the location of a longword containing the desired value, you can specify P1 as a register (A0–A6, D0–D7), an immediate (`#<abs-expr>`), or use any 68000 addressing mode valid in an LEA instruction.

KillReceive

KillReceive cancels any outstanding Receive request for this process. Messages destined for this process are not discarded.

The C declaration for KillReceive is

```
        void    KillReceive();
```

The following example shows how to call KillReceive using assembly language:

```
        JSR     KillReceive
```

Lookup_Task

Lookup_Task returns the task identifier of the process or task that matches the Object Name and Type Name specified, or 0 if no matching process or task is found. The wildcard character "=" is allowed. Initially, set the index to 0. Subsequent calls might modify the index, which should be left unchanged.

Lookup_Task modifies the variable index. The variable index allows Lookup_Task to find any additional entries that might match the criteria in subsequent calls.

The C declaration for Lookup_Task is

```
tid_type        Lookup_Task (object, type, nm_TID, index)
        char    object [];              /* Object Name */
        char    type [];                /* Type Name */
        tid_type        nm_TID;         /* Name Manager Task Identifier */
        unsigned        short *index;   /* Index */
```

The task identifier of the Name Manager is nm_TID, and it can be obtained by using GetNameTID for name managers on the Macintosh II, or by sending an ICC_GetCards message to the ICCM for name managers on NuBus cards. Lookup_Task returns the task identifier of the first process or task that matches the criteria.

The following code shows how to look up all processes on the main logic board of the Macintosh II computer:

```
short index;
tid_type tid;

index = 0;
while ((tid = Lookup_Task ("=", "=", GetNameTID (), &index)) > 0)
       printf ("TID %x Found \n", tid);
```

The following example shows how to call Lookup_Task from assembly language:

```
MOVE.W        #0,INDEX        ; initialize index
PEA           INDEX           ; address of index
MOVE.L        TID,D0 ; value of tid on stack
MOVE.L        D0,-(A7)        ; place on stack
PEA           TYPE_NAME       ; address of type name
PEA           OBJECT_NAME     ; address of object name
JSR           Lookup_Task
ADDQ.W        #16,A7 ; pop the stack
TST.W         D0              ; check if found
BNE.S         D0,XXX ; jump if found
```

OpenQueue

OpenQueue assigns an IPC queue and returns the TID of the process that called OpenQueue. If if no queue could be assigned, it returns zero. This method allows you to set up your own procedure to determine what to do while waiting on a blocking Receive; if you do not want to use this mechanism, use a parameter of zero. This procedure also lets you decide whether to cancel the outstanding Receive request or discontinue communication with Apple IPC; that is, it lets you check for operator termination.

This function must be called before any other call to IPC can be made. You can issue either

- an AppleIPC CloseQueue request, or
- a KillReceive request

If the procedure issues an AppleIPC CloseQueue request and returns to the Apple IPC driver, then the driver returns to the outstanding Receive request with a value of 0. Issuing a KillReceive request returns 0 to the Receive request (no message).

The C declaration for OpenQueue is

```
tid_type OpenQueue(procedure)
       void (*procedure) ();         /* Procedure to execute while waiting*/
                                     /* for blocking receive to complete. */
```

◆ *Note:* This parameter is required; use 0 if you do not want to call the procedure.

The form for the OpenQueue macro is as follows, where P1 is the address of the procedure to execute while waiting for a blocking receive to complete.

```
       [Label]         OpenQueue       P1
```

To indicate the location of a longword containing the desired value, you can specify P1 as a register (A0–A6, D0–D7), an immediate (`#<abs-expr>`), or use any 68000 addressing mode valid in an LEA instruction.

Receive

Receive returns the highest priority message from the message queue of the process that matches the specified criteria.

The C declaration of Receive is

```
message              *Receive( mID, mFrom, mCode, timeout, compl )
       unsigned       long  mID;     /* Unique message ID to wait on      */
       tid_type       mFrom;         /* Sender address to wait on         */
       unsigned       short mCode;   /* Message code to wait on           */
       long    timeout;      /* Time to wait in major ticks      */
                             /* before giving up                 */
       void    compl();      /* Address of a completion routine */
```

The first three parameters (mID, mFrom, and mCode) are selection criteria used to receive a specific kind of message. These parameters can be set to match either a specific value, any value (by specifying OS_MATCH_ALL), or no value (by specifying OS_MATCH_NONE).

The fourth parameter is the timeout value. A timeout value of 0 waits forever for a satisfying message. A negative value returns either a satisfying message or 0 immediately, and a positive value waits that many ticks for a satisfying message to arrive.

◆ *Note:* If a completion routine is not specified, the IPC Receive performs in exactly the same way as the MR-DOS Receive primitive.

The fifth parameter is the address of a C completion routine. Required for Apple IPC, `compl` changes the way the Receive request performs. The `compl` parameter must be either the address of a completion routine or zero, if no completion routine is desired. When this completion routine parameter is nonzero, the call to Receive always returns immediately with a result of 0.

The completion routine is called with a parameter of type `'message *'`. If the completion routine is passed a pointer of zero, a timeout occurred.

◆ *Note:* It is possible to call the completion routine before the Receive actually returns. The purpose of the completion routine is to provide a mechanism by which the Macintosh II application can continue to execute without having to wait for a message. This is necessary because the current version of the Macintosh II operating system is not a multitasking operating system; therefore, the application cannot cease to process events. Under MR-DOS, a process can do a blocking Receive and permit other processes to execute.

Table 5-2 describes the results from various settings of the timeout parameter in major ticks for the Receive call. The results column describes what is returned to the Receive request and completion routine, as well as when the completion routine is called.

■ **Table 5-2** State table for the Receive call

Name		**Description**		
Time-out value	**Comple-tion routine**	**Message available**	**Immediate results**	**Subsequent results**
<0	No (0)	No	Returns 0 to the Receive request	None
	No (0)	Yes	Returns message to Receive request	None
	Yes	No	Apple IPC driver returns 0 to the Receive request; completion routine is not called	None
	Yes	Yes	Apple IPC driver calls the completion routine with the message; driver then returns 0 to the Receive request	None
=0	No (0)	No	Waits until it gets a message, then returns a message to the Receive request	Waits for a message; OpenQueue routine is called continuously.
	No (0)	Yes	When a message arrives, returns a message to the Receive request	None
	Yes	No	Returns 0 to the Receive request; when a message arrives, the driver calls the completion routine with the message	None
	Yes	Yes	Returns a message to the completion routine and returns 0 to the Receive request	None

Table 5-2 (continued)

Time-out value	Completion routine	Message available	Immediate results	Subsequent results
>0	No (0)	No	Waits for a message	Message does not arrive
			If the time interval that you specify expires, then it returns 0 to the Receive request	
	No (0)	Yes	Message returns to the Receive request	None
	Yes	No	Immediately returns 0 to the Receive request and the task continues executing	None
			When a message comes in, the driver calls the completion routine with the message	
			If the timeout expires, the driver calls the completion routine with 0	
	Yes	Yes	Returns a message to the completion routine; returns 0 to the Receive request	None

When using completion routine, you should observe the following guidelines:

- Never use a blocking Receive in a completion routine.
- Be cautious about starting the next asynchronous Receive within a completion routine, as recursion can be deadly.
- Remember that completion routines are sometimes called as the result of an interrupt; anticipate the unexpected!

Only one Receive may be outstanding on a given queue at a time; attempted additional Receive routines return errors. Receive returns a 0 in the event of one of the following:

- no message is available (either timeout or nonblocking)
- a negative error code is received in the case of an error
- or a positive pointer to the received message buffer is returned

◆ *Note:* Exercise caution when testing the pointer returned by Receive for a negative value to ensure that the test is valid.

The form for the Receive macro is:

```
        [Label]        Receive        P1, P2, P3, P4, P5
```

where `P1` is the message ID match code, as follows:

`P2` = sender address match code
`P3` = message code match code
`P4` = timeout code
`P5` = completion routine address

To indicate the location of a longword containing the desired value, you can specify P1 through P5 as a register (A0–A6, D0–D7), an immediate (`#<abs-expr>`), or you can use any 68000 addressing mode valid in an LEA instruction.

Results returned

Whenever you call the Receive request on Apple IPC, you get one of three results returned from the IPC driver:

- 0
- message
- negative number (indicating an error)

Table 5-3 lists the only two errors that can be returned when a Receive request is made to Apple IPC.

■ **Table 5-3** Errors returned

Error	**Number**	**Description**
NoQueueErr	-64	No more queues or bad queue
QueueBusy	-65	Receive is already outstanding on queue

Error -64 (NoQueueErr) is returned if the queue number (TID) of the task doing the Receive request is bad. A queue number is bad if it is not within the range of legal queue numbers or is not open (either OpenQueue was not done or CloseQueue was done).

Error -65 (QueueBusy) is returned if an attempt is made to do a Receive request for a particular queue number (TID) when a request is already outstanding. For more information, refer to the section earlier in this chapter on OpenQueue.

▲ **Warning** To check for an error in the message pointer returned by a Receive request in C language, you must cast the message pointer to long before checking to see if the pointer is negative. Failure to do so will result in a system crash. ▲

The following code checks the message pointer to see if an error code was returned:

```
message *msgptr;

msgptr = Receive (0, 0, 0, 0, 0);
if  ((long) msgptr < 0)
{

       /*  Process error code */
}
else
{
       /*     No error, process message */
}
```

Register_Task

Register_Task allows a process to register itself with the Object Name and Type Name specified, using the Name Manager. To make the process visible only to other processes on the Macintosh II main logic board, set local_only to nonzero. To make the process visible to tasks on other cards, then set local_only to 0. Register_Task returns a nonzero value if the process was registered; if not, 0 is returned.

The C declaration for Register_Task is

```
typedef boolean short;
       char Register_Task ( object, type, local_only);
       char    object [];              /* Object Name */
       char    type [];                /* Type Name */
       boolean        local_only;      /* If Local Visibility Only */
```

The following code provides an example of how to register a process:

```
if (!Register_Task ("my_name", "my_type", 0))
       printf("Could not Register Process");
```

The following example shows how to call Register_Task from assembly language:

```
       MOVE.L #LOCAL, -(A7)  ; value of local on stack
       PEA    TYPE_NAME      ; address of type name
       PEA    OBJECT_NAME    ; address of object name
       JSR    Register_Task
       ADDQ.W #12,A7         ; pop the stack
       TST.B  D0             ; check if register ok
       BNE.S  OK             ; jump if OK
```

Send

Send allows you to send a message to the destination address specified in the message. Send places a message in the queue of the process specified by the message field, mTo. The message is placed in the queue in priority order (from highest to lowest). This call assumes that all fields have been filled in (mFrom, mTo, mCode, and so forth).

The C declaration of Send is

```
void    Send( mptr )
message         *mptr; /* pointer to message buffer */
```

If a message is undeliverable, it is returned to the sender with the message status, mStatus, set to 0x8000 and the message code, mCode, having bit 1 << 15 set.

The assembly-language form for the Send macro is as follows, where P1 is the address of the message buffer to be sent

```
[Label]                Send            P1
```

To indicate the location containing the address of the message buffer to be sent, you can specify P1 as a register (A0–A6, D0–D7), or you can use any 68000 addressing mode valid in an LEA instruction.

SwapTID

SwapTID swaps the mFrom and mTo fields of a message buffer.

The C declaration of SwapTID is

```
void SwapTID( mptr )
message         *mptr; /* pointer to message buffer */
```

The assembly-language form for the SwapTID macro is as follows, where P1 is the address of the message buffer

```
[Label]         SwapTID         P1
```

To indicate the location containing the desired address, you can specify P1 as a register (A0–A6, D0–D7), or you can use any 68000 addressing mode valid in an LEA instruction.

P1 can be specified as a register (A0–A6, D0–D7), or can use any 68000 addressing mode valid in an LEA instruction to specify the location containing the desired address.

Chapter 6 **Download and Initialization**

THIS CHAPTER DESCRIBES the interface to and the operation of the TokenTalk Prep file. The TokenTalk Prep file provides code and parameters for initializing the TokenTalk NB card. This chapter contains an example of how to use the TokenTalk Prep file to download and initialize the TokenTalk NB card, and it describes the resources and services in the TokenTalk Prep file. ■

General information

The TokenTalk Prep file contains resources. These resources consist of code images for downloading to the TokenTalk NB card and routines that participate in the boot process and initialization of the card.

The TokenTalk Prep file is a specific type, `'ttpp'`, and the file's creator is also of the type `'ttpp'`. Specifically, the TokenTalk Prep file contains the following resources:

ResType	ID	Description
`'STR'`	–4030	The TokenTalk Prefs file
`'ttbl'`	0–n	resources containing MR-DOS, LLC, and SNAP; these boot the card
`'llcp'`	–1	Resource that contains default LLC parameters
`'ttut'`	0	Resource that contains the utility routine to support the TokenTalk NB card initialization

The TokenTalk Prefs file (whose name is contained in `'STR'`) can contain an `'llcp'` resource that overrides the default logical link control LLC parameters. See the section "LLC Resource Description" later in this chapter for information on creating your own `'llcp'` resource.

The `'ttbl'` resource contains the software that boots the card and `'llcp'` contains the default LLC parameters.

The `'ttut'` resource contains a utility routine that supports the TokenTalk NB card initialization.

TokenTalk Prep services

The `'ttut'` resource in the TokenTalk Prep file provides the following services. Definitions relating to these operations are located in the include file TTUtil.h.

Code	Meaning	See page
TTFindCards	Find all cards and return mask	82
TTFindBootedCards	Find booted cards and return mask	82
TTFindUnbootedCards	Find unbooted cards and return mask	82
TTBootCards	Boot cards	83
TTForceBoot	Force boot of cards	83
TTGetStatusAddr	Return TT status address	83
TTGetLLCTID	Return TID of LLC for given slot	84
TTGetSNAPTID	Return TID of SNAP for given slot	84
TTGetBoardID	Return board ID for given slot	84
TTDynamicDL	Perform dynamic download	85

To call on the TokenTalk Prep file to perform these services, perform the following steps:

1. Open the resource fork of the TokenTalk Prep file in the System Folder.
2. Load the `'ttut'` resource.
3. Obtain the pointer to the resource and strip it using StripAddress.
4. Use the stripped pointer to call the function in the `'ttut'` resource.

See the section "TokenTalk Prep File Example" later in this chapter for a sample program listing that performs these steps.

Call the TokenTalk Prep services using Pascal calling conventions. The function accepts two long integer parameters and returns a long integer result. The first parameter is always one of the service names listed above; it specifies which operation to perform. The second parameter and the returned result vary depending on the operation specified. In addition, the TokenTalk Prep file should be on the top of the resource file list for all operations except TTDynamicDL. When dynamically downloading a running TokenTalk NB card, the file that contains the resources to download should be at the top of the resource file list.

TTFindCards

The TTFindCards function finds all or some TokenTalk NB cards. The second parameter is a 16-bit mask of the NuBus slots to check in the low-order 16 bits. In the bit mask, bit 2^n denotes NuBus slot n. The result returned is a similar mask of the TokenTalk NB cards found.

Assume that the pointer UtilPtr is declared as follows and that it has been initialized with the stripped address of the `'ttut'` resource:

```
typedef pascal long (*TTUtilPtr)(long op, long data);

TTUtilPtr      UtilPtr;
```

A TTFindCards request to find all TokenTalk NB cards would be similar to the following:

```
result = (*UtilPtr)(TTFindCards, -1);
```

To verify that slot 1 (NuBus slot 9—the one nearest the Macintosh II power supply) contains a TokenTalk NB card, use the following call:

```
result = (*UtilPtr)(TTFindCards, 0x0200);
```

This operation uses the Slot Manager to identify TokenTalk NB cards. It makes no use of any MR-DOS IPC services.

TTFindBootedCards

The TTFindBootedCards function is similar to TTFindCards except that it only locates TokenTalk NB cards that already have MR-DOS and LLC running. Logically, this function uses TTFindCards to identify TokenTalk NB cards then checks that the card is running by using MR-DOS IPC services.

The following call finds all TokenTalk NB cards that are running:

```
result = (*UtilPtr)(TTFindBootedCards, -1);
```

TTFindUnbootedCards

The TTFindUnbootedCards operation is similar to TTFindBootedCards except that it only locates TokenTalk NB cards that do not already have MR-DOS and LLC running. Logically, this request uses TTFindCards to identify TokenTalk NB cards then checks to see whether the card is running by using MR-DOS IPC services.

The following call finds all TokenTalk NB cards that are not running:

```
result = (*UtilPtr)(TTFindUnbootedCards, -1);
```

TTBootCards

The TTBootCards function boots the TokenTalk NB card. This request only boots TokenTalk NB cards that have not yet been booted. Logically, this request uses TTFindUnbootedCards then boots those cards found. The result of this operation is a mask of the cards that were actually booted.

See the section "TokenTalk NB Card Boot Process Summary" later in this chapter for more information on the boot process.

The following call starts all TokenTalk NB cards that are not running:

```
result = (*UtilPtr)(TTBootCards, -1);
```

TTForceBoot

The TTForceBoot function is similar to the TTBootCards operation except that it forcibly restarts cards that are already running. In normal use, this function should never be used since a TokenTalk NB card may be supporting multiple concurrent applications. The result of this operation is a mask of the cards that were actually started.

See the section "TokenTalk NB Card Boot Process Summary" later in this chapter for more information on the boot process.

The following call starts the TokenTalk NB card in slot 2 (NuBus slot A):

```
result = (*UtilPtr)(TTForceBoot, 0x0400);
```

TTGetStatusAddr

The TTGetStatusAddr function returns the address of the LLC status structure for the given slot. The second parameter to this request is a mask of the slot to operate on. Unlike previous requests, this mask should have only a single bit set since only a single status address can be returned. The result of this operation is a 32-bit NuBus address. This address will be returned even if the card is not running. By inspecting the structure at this address, ring status can be monitored. Refer to the *TMS380 Adapter Chipset User's Guide Supplement* for additional information on ring status messages.

The following call returns the status address for the TokenTalk NB card in slot 3 (NuBus slot B):

```
result = (*UtilPtr)(TTGetStatusAddr, 0x0800);
```

TTGetLLCTID

The TTGetLLCTID function returns the task ID of the LLC task running on the given slot. The second parameter is a mask of the slot to operate on. This mask should only have a single bit set since only a single task ID can be returned. This task ID may be used to issue LLC requests as described in Chapter 4, "The 802.2 LLC / IPC Interface." A zero is returned if the card is missing or not running.

The following call returns the LLC task ID for the TokenTalk NB card in slot 4 (NuBus slot C):

```
result = (*UtilPtr)(TTGetLLCTID, 0x1000);
```

TTGetSNAPTID

The TTGetSNAPTID function is much like the TTGetLLCTID function except that it returns the task ID of the SNAP task running on the given slot. The second parameter is a mask of the slot to operate on. This mask should only have a single bit set since only a single task ID can be returned. This task ID may be used to issue SNAP requests as described in Chapter 3, "SubNetwork Access Protocol (SNAP) Interface." A zero is returned if the card is missing or not running.

The following call returns the SNAP task ID for the TokenTalk NB card in slot 5 (NuBus slot D):

```
result = (*UtilPtr)(TTGetSNAPTID, 0x2000);
```

TTGetBoardID

The TTGetBoardID function returns the BoardID for the TokenTalk NB card in the given slot. This is the board ID returned by the Slot Manager. The second parameter is a mask of the slot to operate on. This mask should only have a single bit set since only a single board ID can be returned. The result of this operation is board ID stored in the declaration ROM on the card. The board ID is returned even if the card is not running.

The following call returns the board ID for the TokenTalk NB card in slot 6 (NuBus slot E):

```
result = (*UtilPtr)(TTGetBoardID, 0x4000);
```

TTDynamicDL

The TTDynamicDL function downloads and starts a task onto a running TokenTalk NB card. Dynamic download requires considerable familiarity with the MR-DOS environment and is beyond the scope of this document The second parameter is the address of the structure shown in *Example 6-1*. The result of this operation is the task ID of the started task, or zero if the task could not be started.

The following call attempts to start a task on the TokenTalk NB card in slot 3 (NuBus slot B):

```
TTDDLP ttdl;

memset((char *)&ttdl, 0, sizeof(TTDDLP));  /* clearing memory */

ttdl.type = 'abcd';          /* the type of your task code file */
ttdl.SlotNo = B;                     /* the slot number to download to */
ttdl.STPB.stack = 2048;              /* the size of the task's stack */
ttdl.STPB.priority = 25;     /* the task's priority */
result = (*UtilPtr) (TTDynamicDL, (long)&ttdl);
```

■ **Example 6-1** TTDynamicDL request structure

```
typedef struct
{
        long    Type;                /* Resource type holding code to download */
        long    SlotNo;              /* Slot number to download to
                                             (not a mask - 9 - 14) */
        struct st_PB   xxx;    /* StartTask parameter structure defined in MR-DOS
                                                *     include file os.h.
                                                */
} TTDDLP;
```

TokenTalk Prep file example

The following routine returns a pointer to the TTUtil routine and a RefNum to the resource file, so that the file can be closed on completion. The pointer is returned as zero if any errors occur. See Appendix C for a complete programming example.

```
/*
 *    GetTTUtilPtr - Return pointer to TTUtil routine.
 *
 *    Inputs:
 *            resno   Resource number of string resource holding prep file name.
 *            refptr  Address of a short to receive the resource file refnum.
 *
 *    Outputs:
 *            Returns pointer to TTUtil routine, or zero if unavailable.
 *
 *    Note that no refnum is returned if the pointer returned is zero.  This
 *    routine will automatically close any resource file it may have opened
 *    in that case.
 */

TTUtilPtr     GetTTUtilPtr(resno, refptr)
short resno;
short *refptr;
{
      Handle   strhdl, utlhdl;
      short    ttrefnum;
      SysEnvRec        sysrec;

      if ((strhdl = GetResource('STR ', resno)) == 0 || SysEnvirons(1, &sysrec))
              return 0;                    /* Fail if resource missing or
                                           SysEnvirons fails */
      ttrefnum = OpenRFPerm(*strhdl, sysrec.sysVRefNum, fsRdPerm);
      ReleaseResource(strhdl);      /* File name no longer needed */
      if (ttrefnum == -1)                  /* If open failed, return 0 */
              return 0;

      if ((utlhdl = Get1Resource('ttut', 0)) == 0)        /* Error loading
                                                                 resource? */
      {
              CloseResFile(ttrefnum);
              return 0;                    /* Close file and return 0 if didn't
                                           get resource */
      }
      *refptr = ttrefnum;
      return (TTUtilPtr)StripAddress(*utlhdl);     /* Return stripped pointer */
}
```

LLC resource description

The following information summarizes the boot process and describes the LLC resource.

TokenTalk NB card boot process summary

The boot process checks for the presence of a TokenTalk Prefs file whose name comes from the `'STR'` resource. In the TokenTalk Prefs file, the boot process checks for an `'llcp'` resource with an ID that matches the slot being booted. If the matching slot ID is present, the LLC parameters in that resource are used when starting LLC on that slot. Otherwise, the default contents of the `'llcp'` resource in the TokenTalk Prep file is used. This approach allows a Macintosh II system with multiple TokenTalk NB cards to have each card initialized with different parameters based on its intended use.

Defining the LLC resource

If the TokenTalk Prefs file does not exist and you want to define special LLC parameters, you must create the TokenTalk Prefs file in the System Folder with a type of `'ttpf'` and creator `'ttpp'`. The description of the `'llcp'` resource follows:

```
/*
 *      TTInit.r - Define format of LLC parameter resource.
 *
 *      Mark D. Rustad.       8/10/88.
 *
 *      Copyright © Apple Computer, Inc.  1988.
 */

type 'llcp'
{
        longint;                     /* Initial functional address */
        longint;                     /* Initial group address */
        longint;                     /* Options (not used - should be zero) */
        longint;                     /* Address of listener, always zero
                                          in resource */
        unsigned integer;            /* Maximum frame size */
        unsigned integer;            /* Maximum number of link stations */
        unsigned integer;            /* Buffer size within tms380 */
        unsigned byte;               /* Maximum number of SAPs */
        unsigned byte;               /* Maximum number of group SAPs */
        unsigned byte;               /* Maximum number of group SAP members */
        unsigned byte;               /* Number of transmit buffer in list */
        unsigned byte;               /* Number of receive buffers in list */
        unsigned byte;               /* Number of interrupt messages to reserve */
        unsigned byte;               /* Group 1 response period (40 ms ticks) */
        unsigned byte;               /* Group 1 receive acknowledge period */
        unsigned byte;              /* Group 1 inactivity period */
        unsigned byte;              /* Group 2 response period */
        unsigned byte;              /* Group 2 receive acknowledge period */
        unsigned byte;              /* Group 2 inactivity period */
        unsigned byte;              /* Minimum transmit buffers */
        unsigned byte;              /* Maximum transmit buffers */
        hex string[6];              /* Node address, 0 uses burned-in address */
        hex string[18];             /* Product ID string (in EBCDIC?) */
        cstring[32];        /* IPC name of this LLC */
};

/*      End of llcp.r */
```

Chapter 7 **Avoiding Trouble**

THIS CHAPTER DISCUSSES some common situations that might prevent your development code from operating correctly. The object of this chapter is to provide a first-line troubleshooting guide that helps identify common but subtle errors. The troubleshooting information includes software tips and hardware tips. ■

General information

The overall complexity of programming and developing applications for a network environment provides ample opportunity for problems. Good programming technique and design can prevent some problems; other problems arise from implementing good practices in an environment that lacks full support for those tried-and-true practices.

The SNAP and LLC interfaces provide error messages. Problems with the interface itself can usually be resolved by investigating the causes of the error messages. Other problems can be more subtle, such as having the token ring chipset shut down for no apparent reason or having code that worked in a standalone environment fail when ported to a dynamic download environment. The remainder of this chapter presents guidelines for those less obvious error conditions.

Refer to the echo task program in Appendix C for a comprehensive example of a functional, dynamically downloaded TokenTalk task.

Common error causes

Potential causes of errors not easily detected include the following:

- Unchecked error codes
- Failure in the physical network connection
- Errors in programming the listener function
- Global data structures referred to incorrectly
- DMA activity that conflicts

Each of these causes is discussed in the following paragraphs.

Error codes

The mStatus message returns 0 on successful completion of an interface call. Your program should always check the mStatus message for an error condition and provide a suitable recovery routine.

By checking the error codes, you obtain a diagnostic indication of the cause of the error, which is an important program development tool. A summary of error codes is presented in *Table 7-1.*

■ **Table 7-1.** mStatus error code summary

SNAP Result codes	*Value*	*Description*
	SNAPCancelled	Receive cancelled, either explicitly or by SNAPDetach
	SNAPInUse	PD already attached
	SNAPNoErr	Normal completion
	SNAPNoErr	Normal completion
	SNAPNoErr	Normal completion
	SNAPNoMore	Insufficient resources
	SNAPNotAttached	Invalid RefNum
	SNAPNotAttached	Invalid RefNum
	SNAPTruncated	Frame larger than provided buffer space

LLC Result codes	*Value*	*Description*
	LLCAddrError	Invalid remote address—group address invalid. Also,size must be 0 or node address size for the media
	LLCBadFrame	Invalid frame type
	LLCBadPointer	Bad pointer passed as "interface-owned"
	LLCBadPri	Unauthorized access priority
	LLCBadRefNum	Invalid RefNum
	LLCBadSAPOpts	Invalid SAP options
	LLCBadSAPValue	Invalid SAP value or SAP value already in use
	LLCBadSize	mDataSize has inappropriate value
	LLCCancelled	Receive cancelled, either explicitly or by close operation
	LLCClosedNoAck	Station closed without remote acknowledgment
	LLCConnFail	The remote station did not accept the connection request
	LLCConnSeqError	Connect sequence error
	LLCGroupLimit	The group SAP already has maximum members
	LLCLinkOpen	Unclosed link stations on SAP
	LLCMaxExceeded	Parameter exceeds maximum
	LLCMsgReject	Unusual interface error
	LLCNoErr	Normal completion
	LLCNoGroup	Requested group membership in nonexistent group SAP
	LLCNoiFrames	Link not transmitting i frames

LLCNoResources	Maximum number of link stations or SAPs are already open
LLCNotFullySupported	Some option or type requested is not fully supported by this media
LLCNotMember	Member not found in group SAP
LLCProtoError	Protocol error—I frame issued before DMA ready or link in invalid state for command
LLCRoutingError	Invalid routing information length
LLCSAPOpen	Group SAP cannot close—all member SAPs are not closed
LLCSeqError	Sequence error—have already issued a close to this link
LLCTruncated	Buffer too short to receive all information
LLCTxError	Error in frame transmit or strip
LLCTxTooLong	Invalid transmit frame length
LLCUnauthMAC	Unauthorized MAC frame

Network connection failure

If a cable is disconnected on the TokenTalk NB card or the Multistation Access Unit (MAU) while the TokenTalk software is running, the error "LLC not open" occurs. The adapter card's chipset will waits for approximately 2 seconds and then shuts itself down, which closes the LLC interface. All queued messages are returned to the client and any future messages are also returned to the client with the "LLC not open" error code.

Recovery for this condition depends on the application on the TokenTalk NB card. The choices are either to download and initialize the card again, or to require a complete system reboot.

A similar condition occurs if the card is downloaded and initialized without being plugged in to the MAU.

Problems programming the listener function

Treat the listener function like an interrupt service routine, which is to say keep it simple and efficient. Avoid allocating large amounts of stack to the listener function and avoid attempting to perform a large amount of processing. All "good programming" techniques for dealing with interrupt service routines apply equally well to dealing with the listener function.

Global data structures and dynamic download

For each task, the A5 register contains the starting address of the global data structure associated with that task when it is created and linked with MR-DOS. It is normally useful to assign the common global data structure in this manner, because all tasks created and linked with MR-DOS will have the same value for A5, for example,

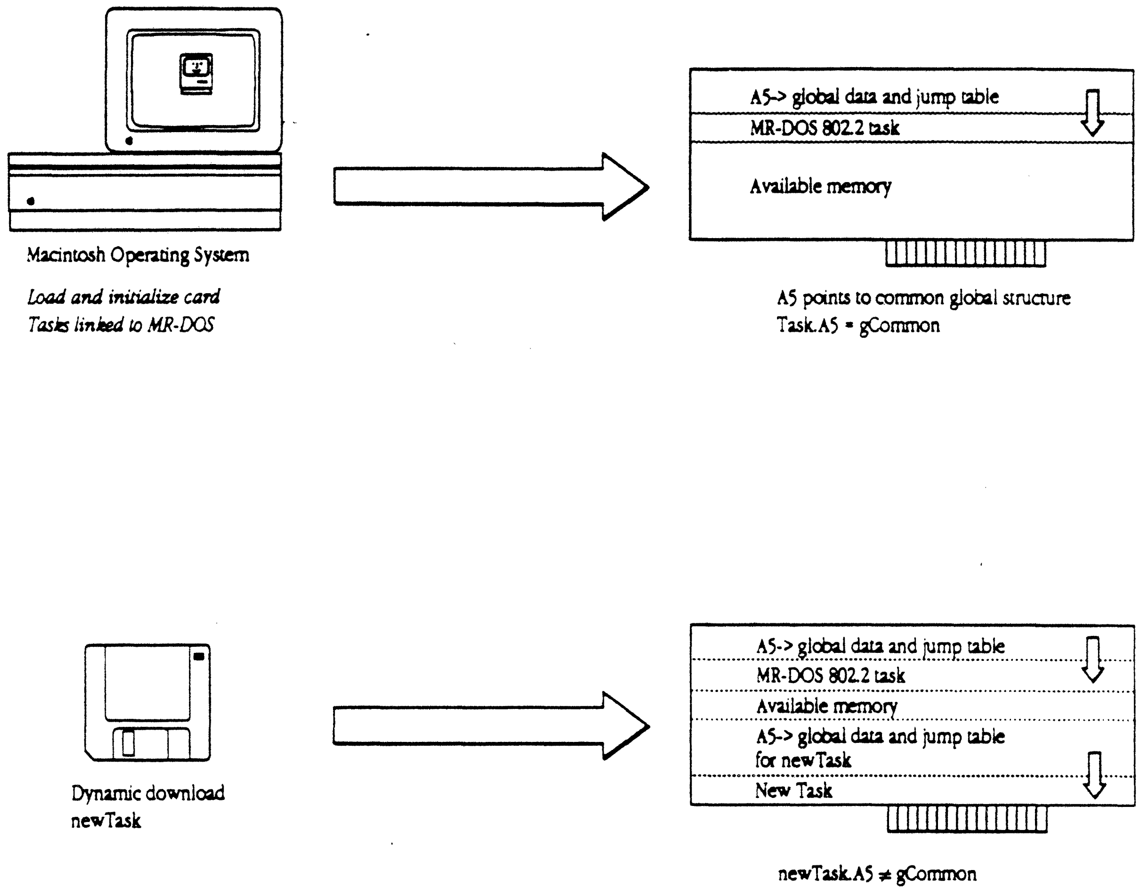
```
A5 = GetgCommon()->gInitA5
```

In a dynamic download situation, however, this assumption is wrong. A task spawned from the dynamically downloaded task has a different A5, which differs from that of the task created and linked with MR-DOS. Therefore, to spawn another task from the dynamic download task, you must set up your own A5 to ensure that the correct data structure is used.

This type of error is difficult to trace because a task developed as a standalone under MR-DOS will execute. But when the same task is dynamically downloaded it will fail, and all because the wrong data structure address is used. This is a situation in which a useful and acceptable programming practice backfires.

Figure 7-1 shows a situation in which the adapter card is loaded and initialized from the Macintosh OS. A task is linked with MR-DOS, the 802.2 interface is downloaded to he card along with MR-DOS, and all tasks begin execution. At this point, the A5 register contains the address of the common global data structure, which is set when the the tasks are linked with MR-DOS. Sometime later, a new task is dynamically downloaded to the card. As the figure shows, the new task must have its own value for A5, which is created by the dynamic download process. The echo task program in Appendix C contains an example of the code that captures and manages the dynamic download value for A5.

■ **Figure 7-1** Dynamic task download


Macintosh Operating System
Load and initialize card
Tasks linked to MR-DOS
A5-> global data and jump table
MR-DOS 802.2 task
Available memory
A5 points to common global structure
Task.A5 = gCommon
Dynamic download
newTask
A5-> global data and jump table
MR-DOS 802.2 task
Available memory
A5-> global data and jump table
for newTask
New Task
newTask.A5 ≠ gCommon

Direct Memory Access (DMA) conflicts

An error condition can occur when the TMS380 chipsets on two cards in the same system attempt a DMA transfer to one another at the same time. The chipset attempts to retry on error; but if it fails repeatedly, it shuts itself down. The simplest way to avoid this condition is to have the CPU perform the DMA transfer and have the chipset copy the data from the CPU.

This DMA conflict is possible because the token ring chipset has no connector to the CPU halt signal. It is by means of asserting the halt and bus error signals at the same time that a bus retry occurs. A bus retry occurs when the DMA request cannot complete. Because the chipset only sees the bus error signal, it acts as though a bus error has occurred in fact, rather than merely a bus retry. The retry-on-error count is set to its maximum limit of 255, meaning that 255 consecutive bus errors must be seen by the chipset before it shuts itself down.

Avoid this potential error condition in one of two ways:

- As mentioned, have the adapter card CPU perform the DMA transfer rather than the token ring chipset
- Split the DMA transmit buffers into small enough sizes that the buffer will exhaust before the bus retry count does

Appendix A Components

The release diskette for software development on the TokenTalk NB card includes the following files:

- LLCSupportLib.o
- LLC.h
- SNAP.h
- TRInit.h
- TTUtil.h
- TTInit.r

Appendix B The TokenTalk NB Card

The TokenTalk NB card is a single-board communications controller that occupies one I/O slot on the system board of the Macintosh II system. The card provides high-speed (4M bps) token ring network communications and is compatible with the IBM token ring adapter at the physical transmission level.

Designed to the Draft 2.0 NuBus specifications, the TokenTalk NB card can function as either a NuBus master or slave device. When acting as bus master, TokenTalk NB has full access to all other NuBus devices installed in the host system. As a NuBus slave, TokenTalk NB relinquishes control of all its internal resources to the designated bus master.

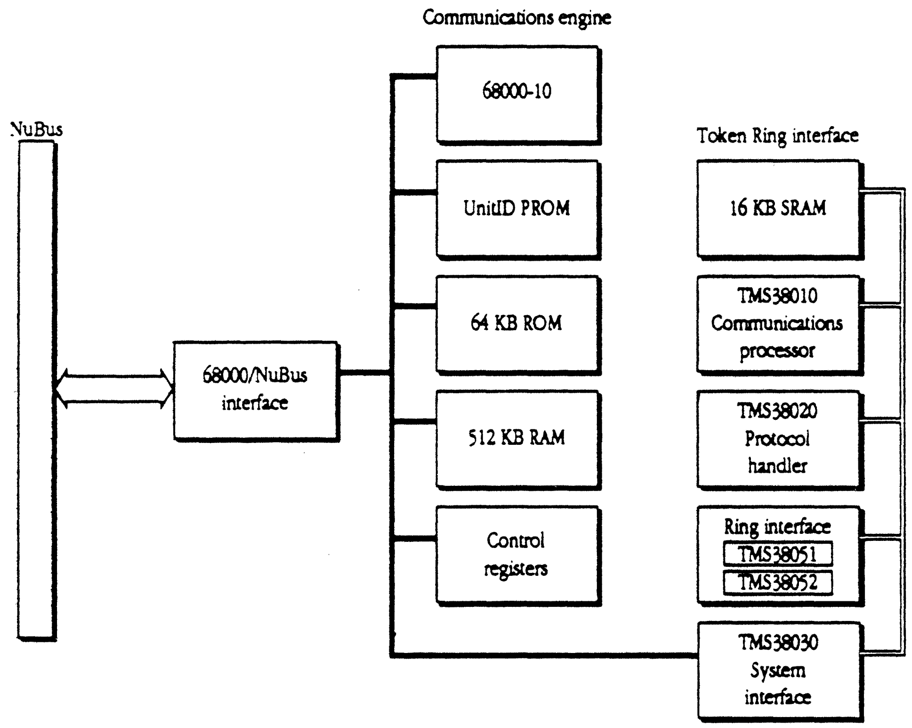
Throughout this appendix, all address reference and data values are given as hexadecimal values and refer to the 24-bit address range of the Motorola 68000 microprocessor. The high-order 8 bits that indicate the NuBus slot address are not contained in the addresses listed in this appendix.

The term "TMS380" as used in this appendix refers to the complete Texas Instruments TMS380 Token Ring Adapter Chipset as a whole rather than to a specific member of the chipset. ■

Hardware overview

The overall design of the TokenTalk NB card can be divided into two main functional blocks: the communications engine and the token ring interface. *Figure B-1* shows a functional block diagram of the TokenTalk NB card.

■ **Figure B-1** TokenTalk NB Block Diagram



Communications engine

The communications engine consists of five major components:

- The Motorola 68000 CPU
- PROM
- RAM
- Communications engine/NuBus interface
- Communications engine/token ring interface

These components are described in the following sections.

Central processor unit (CPU)

The processor employed on the TokenTalk NB card is a Motorola 68000 CPU with a clock speed of 10 Mhz. The 10-Mhz clock is derived from the 10-Mhz NuBus clock. Because TokenTalk NB can function as a NuBus Master device, the 68000 processor is capable of acquiring full access and control of all NuBus devices and resources.

Read-only memory (ROM)

TokenTalk NB provides space for 64KB of adapter ROM. This ROM contains the NuBus configuration information required to interface the card to the Macintosh II environment, the power on self-test code, the power on reset vectors, the burned-in unit ID, the version number, the copyright notice, and any additional firmware provided by Apple Computer, Inc.

The adapter ROM is mapped at adapter addresses FF0000 to FFFFFF. The on-board ROM appears as a 16-bit device to the 68000 and as a 32-bit device to the NuBus. When accessed by NuBus, circuitry on the communications engine performs two 16-bit accesses to provide a full 32 bits of data in one NuBus access. When this action is performed, the low-address word occupies the low-order bits (0-15) and the high-address word occupies the high-order bits (16-31) of a 32-bit longword.

Dynamic random access memory (DRAM)

A total of 512KB of DRAM on the TokenTalk NB is mapped at adapter addresses 000000 to 07FFFF. The CPU, TMS38030 Token Ring System Interface, and NuBus all have access to this memory. The on-board DRAM is used for TokenTalk NB system code and data space.

When it functions as a bus slave, all on-board RAM is accessible to the current system bus master. While the current NuBus master has access, both the 68000 and the TMS38030 are denied access to the on-board RAM. This RAM, like the ROM, appears as a 16-bit device to the 68000 and TMS380, and as a 32-bit device to NuBus.

When accessed by NuBus, circuitry on the communications engine performs two 16-bit accesses to provide a full 32 bits of data in one NuBus access. When this action is performed, the low-address word occupies the low-order bits (0–15) and the high-address word occupies the high-order bits (16–31) of a 32-bit longword.

Communications engine/NuBus interface

The communications engine/NuBus interface provides an 8/16/32-bit interface between the TokenTalk NB 68000 and NuBus. Because the 68000 is a 16-bit device and the NuBus allows 32-bit accesses, special circuitry is provided to transform a 32-bit NuBus access into two 16-bit 68000 accesses. If any problem occurs with the NuBus access, a bus error is reported to the 68000. Access to the communications engine/NuBus interface is accomplished through a set of control registers located at addresses C00000–C00040.

NuBus pinouts as viewed from the front edge of the card are as follows:

Pin	**Row A**	**Row B**	**Row C**
1	-12	-12	/RESET
2	GND	GND	GND
3	/SPV	GND	+5
4	/SP	+5	+5
5	/TM1	+5	/TM0
6	/AD1	+5	/AD0
7	/AD3	+5	/AD2
8	/AD5	•	/AD4
9	/AD7	•	/AD6
10	/AD9	•	/AD8
11	/AD11	•	/AD10
12	/AD13	GND	/AD12
13	/AD15	GND	/AD14
14	/AD17	GND	/AD16
15	/AD19	GND	/AD18
16	/AD21	GND	/AD20
17	/AD23	GND	/AD22
18	/AD25	GND	/AD24
19	/AD27	GND	/AD26
20	/AD29	GND	/AD28
21	/AD31	GND	/AD30
22	GND	GND	GND
23	GND	GND	/PFW
24	/ARB1	•	/ARB0
25	/ARB3	•	/ARB2
26	/ID1	•	/ID0
27	/ID3	•	/ID2
28	/ACK	+5	/START
29	+5	+5	+5
30	/RQST	GND	+5
31	/NMRQ	GND	GND
32	+12	+12	/CLK

• *These pins are connected but not supplied with the –5.2 V signal specified in the NuBus specification.*

Communications engine/token ring interface

The communications engine/token ring interface consists of the 68000, the token ring interface logic, and the direct I/O control registers and DMA controller located in the TMS38030. This interface logic provides a 16-bit interface between the 68000 and the TMS38030. Access to this interface is accomplished through the use of the TMS38030 direct I/O control registers that are mapped to the 68000 memory addresses from 800000–800006.

Token ring interface

The token ring interface section of TokenTalk NB is implemented using the TI TMS380 token ring interface controller chipset. The TMS380 configuration consists of the five TMS380 chips, 16KB of buffer RAM, and the interface logic.

The TMS380 chips are briefly described in the following sections.

TMS38010 communications processor

The TMS38010 executes the protocol firmware residing in the TMS38030 and provides intermediate buffering of ring traffic. There are 2816 bytes of internal buffer RAM that are supplemented with 16KB of external static RAM (19,200 bytes total) to provide a larger and more efficient buffer space.

TMS38020 protocol handler (PH)

The token ring Media Access Control (MAC) sublayer protocol firmware normally resident within the TMS38020 can be replaced with enhanced protocol firmware residing in external PROM. Addition of this enhanced PROM provides features required in a bridge environment. An application that needs to verify the installation of the optional protocol firmware can read the TokenTalk NB options register (address 800008) and check whether bit 0 is set to zero.

Texas Instruments has a set of two PROMs that contains an enhanced version of the TMS38020 protocol handler internal ROM. This enhanced PROM set is used in bridge applications. The TokenTalk NB card can incorporate these PROMs through the use of a piggy-back board, which is plugged into a connector located on the TokenTalk NB card.

The maximum number of TokenTalk NB cards that can be installed in a single Macintosh II system is limited by the power supply and by the number of available slots. Software access to each card is accomplished through the NuBus slot addressing conventions.

TMS38030 system interface (SIF)

The TMS38030 system interface chip controls all interface functions between the 68000 and the remainder of the token ring chipset. The TMS38030 provides a 16-bit bus between the 68000 and the TMS380 token ring interface. The TMS38030 provides a set of direct I/O registers and a direct memory access (DMA) channel for data transfers.

TMS38051 and TMS38052 ring interface

The TMS38051 and TMS38052 ring interface chips perform the actual data encoding and decoding using the differential Manchester code. The ring interface chips also perform the ring insertion and de-insertion tasks. Physical connection to the ring is by way of an IBM Token Ring Adapter DB9 nine-pin connector. The DB9 connector provides correct signal connection to the IBM Type 1 Cabling System. Pinouts for the DB9 connector are as follows:

Pin	Wire	Signal
Shield	1	Ground
1	4	Receive
2		Not Used
3		Not Used
4		Not Used
5	3	Transmit
6	5	Receive
7		Not Used
8		Not Used
9	2	Transmit

Burned-in unit ID

The unit ID/serial number is stored in a reserved location in the 68000 Declaration PROM. The unit ID is the network node address of a TokenTalk NB card and its host. Each token ring adapter card, whether a TokenTalk NB card or otherwise, has a unique 6-byte (48-bit) burned-in unit ID. The unit ID contained in this ROM is used as the default node ID when the adapter card is first opened. By supplying a locally administered node ID as a parameter to the TMS380 Open command, you can override the default unit ID. If no unit ID override is provided by the application software, the low-level protocol software must retrieve the burned-in unit ID from the Declaration PROM and pass it to the TMS380 chipset as the node ID used when opening the TokenTalk NB card (or other adapter card).

The IEEE 802 committee administers and assigns blocks of unit ID numbers to respective manufacturers.

Adapter interfaces

The following sections describe the adapter interfaces and include descriptions of the adapter memory map, control registers, options register, direct I/O interface registers, DMA, timers, resets, and interrupts.

TokenTalk NB memory map

The following list provides an address map of all resources on the TokenTalk NB card:

Address	Function
FF0000–FFFFFF	ROM (64KB)
E00000–FEFFFF	Reserved
C00040	68000 Reset
C0000A	Set Interrupt TokenTalk NB Request
C00008	Clear Interrupt TokenTalk NB Request
C00006	Set Interrupt Host Request
C00004	Clear Interrupt Host Request
C00002	Clear Timer Interrupt
C00000	NuBus Extension Register / Clear Reset
A00000–BFFFFF	NuBus
800012–9FFFFF	Reserved (I/O Interface Decode)
800010	TMS380 NuBus Extension Register
800008	TokenTalk NB Options Register
800006	TMS38030 DIO Interrupt Register
800004	TMS38030 DIO Address Register
800002	TMS38030 DIO Data Auto Increment Register
800000	TMS38030 DIO Data Register
400000–7FFFFF	Reserved (I/O Interface-No Decode)
080000–3FFFFF	Reserved
000000–07FFFF	RAM 512KB

Control registers

The communications engine provides eight control registers that assist the 68000/TMS380 software interface. The control registers are memory-mapped at addresses C00000–C00040. The eight control registers and functions are as follow:

Address	Function	R/W
C00040	68000 Reset	W
C0000A	Set Interrupt TokenTalk NB Request	R
C00008	Clear Interrupt TokenTalk NB Request	R
C00006	Set Interrupt Host Request	R
C00004	Clear Interrupt Host Request	R
C00002	Clear Timer Interrupt	R
C00000	NuBus Extension Register / Clear Reset	R

TokenTalk NB card options register

The options register at address 800008 is provided to determine what options, if any, are currently installed on the TokenTalk NB card. The only option currently planned for the card is the optional Enhanced TMS38020 PROM set. By reading the options register and testing bit 0, you can determine whether the bridge PROM set is installed when 0 = installed and 1 = not installed.

TMS38030 direct I/O interface registers

The TMS38030 provides both a direct I/O (DIO) interface and a DMA interface. The DIO is used for initializing the TokenTalk NB card, command initiation, and status reporting. The DMA interface is used for transferring commands, parameter lists, and frames between the TMS380 RAM and the 68000 RAM.

The DIO interface consists of four 16-bit registers, located in the 68000 memory space starting at address 800000. The registers are as follows:

Address	Function
800006	TMS38030 DIO Interrupt Register
800004	TMS38030 DIO Address Register
800002	TMS38030 DIO Data Auto Increment Register
800000	TMS38030 DIO Data Register

DATA register

The DATA register is the primary means of reading from or writing to the buffer RAM of the TMS38010. The data being read or written is pointed to by the address contained in the ADDRESS register.

DATA AUTO INCREMENT register

The DATA AUTO INCREMENT register functions similarly to the DATA register, except that the address contained in the ADDRESS register is automatically incremented in preparation for the next data access.

ADDRESS register

The ADDRESS register points to the address of the TMS38010 buffer memory at which the next data access will occur.

INTERRUPT register

The INTERRUPT register interrupts and reads status information from the TMS380 chipset. Bits 0–7 of the INTERRUPT register can be set to 1 only by the 68000. Only the communications processor (TMS38010) can reset these bits. Bit 8 can be set only by the TMS38010, and only the 68000 can reset this bit.

Bits 9–15 of the INTERRUPT register are read-only to the 68000. The bit definitions for the INTERRUPT register change depending on whether a read or a write operation is being performed. In read mode, the register bits have the following definitions, where bit 0 is the most significant bit:

Bit	Definition
0	Interrupt adapter (TMS380)
1	Adapter reset (TMS380)
2	System status block clear
3	Execute
4	System control block request
5	Receive continue
6	Receive valid
7	Transmit valid
8	Interrupt host system
9	Initialize
10	Test
11	Error
12	Interrupt code 0 / Error 0
13	Interrupt code 1 / Error 1
14	Interrupt code 2 / Error 2
15	Error 3

In write mode, the register bits have the following definitions:

Bit	Definition
0	Interrupt adapter (TMS380)
1	Adapter reset (TMS380)
2	System status block clear
3	Execute
4	System control block request
5	Receive continue
6	Receive valid
7	Transmit valid
8	Reset system interrupt
9	Don't care
10	Don't care
11	Don't care
12	Don't care
13	Don't care
14	Don't care
15	Don't care

TMS38030 DMA

The TMS38030 DMA channel provides 24 address bits, enabling access to a full 16 MB of memory. Since the DMA only has 16 hardware address lines, the 8 most significant address bits are separately latched onto the 68000 bus before the least significant 16 bits are latched. This action and any updating of the most significant address bits are accomplished automatically by the TMS38030. DMA access to the Macintosh II system board or any other NuBus cards installed in the system is accomplished by the communications engine through the NuBus extension register. The contents of this 12-bit register are used as the NuBus slot address (bits 20–31) to create a full 32-bit NuBus address. The DMA channel can be programmed for either burst-mode or cycle-steal modes of operation.

NuBus addressing

A special 12-bit address extension register located at address C00000 provides access to the 32-bit NuBus address space from the 68000. Access to this address space from the TMS38030 is through the TMS380 NuBus extension register at address 800010. To access the NuBus, the 12 most significant bits of the NuBus address should be written to this register prior to the NuBus access. Additionally, setting bit A20 in the address field – a bit not normally used – performs a hardware read/modify/write cycle. This bit must be set whenever executing an 68000 software test-and-reset (BSET) instruction. Address bit A20 should be false (0) for all other operations.

The contents of the NuBus extension register are appended by the communications engine as the high-order 12 bits of all addresses used by the TMS38030 to transfer DMA data across the system interface. By using this extension register, it is possible for the communications engine to route a data packet from the TMS38010 buffer to the Macintosh II or other NuBus card.

If you change the NuBus extension register to route data packets from the TMS380 to a location other than the TokenTalk NB card on-board RAM, you must restore the extension register contents to the appropriate value.

When you route packets from the TMS380 to another destination, remember that you are transferring IEEE 802.5 packets including all header information, which must be processed.

Adapter timer

An on-board timer circuit provides a Level 1 interrupt every 6.5536 milliseconds. The timer interrupt can be cleared by reading address C00002. The timer interrupt must be cleared within 3 milliseconds or the next timer tick will be lost.

68000 reset

The 68000 processor can be reset by reading address C00004. The RESET line is cleared when address C00000 is read or whenever NuBus is reset. On a power-on RESET (NuBus reset), the 68000 supervisor stack pointer and program counter are read from the on-board ROM locations FF0000 and FF0004, respectively. The power on reset vectors will point to the diagnostic and power-up code located in the ROM.

In the event of a software initiated reset (address C00004 is read), the 68000 supervisor stack pointer is loaded from address 000000 in the TokenTalk NB card RAM, and the program counter is loaded from RAM address 000004. You must make certain that valid programmed reset vectors are loaded in these locations.

TMS38030 reset

The TMS38030 can be reset in software by writing an FF to the DIO INTERRUPT register of the TMS38030, located at address 800006. From the host side, the TMS380 is reset when the 68000 is reset by reading address C00004 and it is removed from reset when the 68000 reads address C00000. When the 68000 is reset from NuBus, the TMS38030 is also reset.

Interrupts

The TokenTalk NB card provides three levels of interrupts and priorities as follows:

Interrupt	Level	Priority
Timer	1	Lowest
NuBus	2	Low
Token ring interface	3	Highest

Software overview

The following sections provide overviews of the power-on self-test, the software interface, and the TMS380 command set.

Power-on self-test

A series of power-on self-test (POST) routines are executed when the adapter is first powered up. All tests are initiated and controlled by the on-board 68000 processor. The following functions are performed:

- Write then read test of 512KB RAM
- CRC check of declaration ROM
- Initialization of the 68000 exception vector table
- TMS380 diagnostic and lobe media tests. These tests are performed under the control of the TMS380, with status and error information passed back to the 68000.
- Read/write test across NuBus between 68020 and 68000
- 68000 hardware reset test across NuBus
- Timer, NuBus, token ring interrupt test

The TMS380 diagnostic and lobe media tests include internal CRC circuitry checkout, an internal loop-back test from the TMS38010 to the TMS38020 and ring interface chips and back to the TMS38010. After successful completion of the internal loop-back and CRC tests, the TMS380 performs a lobe media test. This is the same as the internal loop-back test, except that instead of looping back to the TMS38010 from the ring interface, the test continues through the connecting cable to the multistation access unit (wiring concentrator) before looping back. For additional information on these operational tests, refer to the *Texas Instruments TMS380 Adapter Chipset User's Guide*.

Software interface

The following sections present a basic overview of the software mechanisms that control and operate the TMS380 from the 68000. In addition to direct manipulation of the DIO registers, two software constructs—the system command block (SCB) and the system status block (SSB) that reside in the TokenTalk NB card's 68000 memory—are used to pass commands to and get status from the TMS380.

System command block (SCB)

The system command block is a six-byte buffer that is used to issue commands to the TMS380. From low memory to high memory, the format of the SCB is as follows:

COMMAND	2 bytes
ADDRESS high	2 bytes
ADDRESS low	2 bytes

The COMMAND field contains the 16-bit command code of the command to be issued. The two address fields contain a 32-bit pointer to a command parameter table. (The upper 8 bits are ignored, resulting in a 24-bit address.) The format of the command parameter table varies for each command and contains parameter and address information needed to execute the command.

System status block (SSB)

The system status block is an eight-byte buffer that the TMS380 uses to return status information and completion codes on completion of an adapter chipset command. From low memory to high memory, the format of the SSB is as follows:

COMMAND	2 bytes
STATUS 0	2 bytes
STATUS 1	2 bytes
STATUS 2	2 bytes

The COMMAND field is updated by the TMS380 and identifies either RING STATUS, COMMAND REJECT STATUS, or the status of a general command. The three status fields contain actual status information for the COMMAND field. The format and meaning of the status fields vary depending on the command.

TMS380 initialization

TMS30 initialization is accomplished by allocating in the 68000 memory an SCB and an SSB. The particular application running on the TokenTalk NB card also creates a 22-byte initialization block. This block, similar to a command parameter table, contains various intialization options, interrupt vectors for the TMS380, TMS380 DMA parameters, and 24-bit pointers to the SCB and SSB.

To transfer the intialization block to the TMS380, the direct I/O registers are used. The basic procedure is as follows:

1. Software reset the TokenTalk NB card.
2. Verify that the power-up diagnostics are successfully completed.
3. Write the value 0200 to the TMS38030 address register.
4. Transfer the intialization block to the TMS38030 by writing each byte or 16-bit word to the DATA AUTO INCREMENT register. This action causes the initialization block to be written to successive TMS380 RAM locations beginning at address 0A00.
5. After transferring the entire initialization block, write the hex value 9080 to the INTERRUPT register. This value causes an adapter interrupt, instructs the adapter to execute the intialization block, and prevents resetting the system interrupt bit.
6. Loop on reading the INTERRUPT register until either the error bit is set (initialization failed), the INITIALIZE, TEST and ERROR bits are all zero (successful initialization), or 10 seconds have passed (hardware failure).
7. Verify that the SSB and SCB contents are correct, which verifies TMS38030 DMA.

TMS380 command execution

After successful initialization of the TMS380, commands can be issued to the adapter. The process of issuing a command involves the SCB and an associated command parameter table. The basic procedure is as follows:

- Allocate an appropriate command parameter table and initialize its values as required.
- Fill the SCB with the command code and pointers to the command parameter table as required.
- Set the INTERRUPT ADAPTER (bit 0), SSB CLEAR (bit 2), and EXECUTES bits to 1.

This process interrupts the TMS380 and causes it to transfer by way of the DMA the SCB and any required parameters into the TMS380 RAM, and then begins execution of the command. Once the SCB and parameters are copied into the TMS380 RAM, the TMS380 writes a zero into the COMMAND field of the SCB, indicating that another command may now be issued.

Command completion

On completion of a command or on discovering an error while executing a command, the TMS380 transfers by way of the DMA 8 bytes of status information into the SSB. The DMA result is always 8 bytes, regardless of the actual number of bytes of information supplied. After a DMA transfer of the command status information, the 68000 is interrupted by the TMS 380. At this point an application can check the SSB for successful completion. The actual status values that indicate success or failure vary depending on the command.

TMS380 commands

All TMS380 SCB commands and their associated hex codes are as follows:

Command	Code
Open	0003
Transmit	0004
Transmit Halt	0005
Receive	0006
Close	0007
Set Group Address	0008
Set Functional Address	0009
Read Error Log	000A
Read Adapter Buffer	000B

Appendix C **Echo Task Program Example**

The echo task program presented in this appendix shows all major software components needed to successfully program a downloadable task for the TokenTalk NB card.

Additionally, this echo task program is fully functional and exercises the transmit and receive functions on a single TokenTalk NB card. As such, it provides not only a good programming example but also a functional exercise of the token ring hardware.

Throughout this appendix, all address reference and data values are given as hexadecimal values and refer to the 24-bit address range of the Motorola 68000 microprocessor. The high-order 8 bits that indicate the NuBus slot address are not contained in the addresses listed in this appendix. ■

Program summary

The majority of this appendix is a C program listing that demonstrates a fully-functional echo task that is dynamically downloaded to the TokenTalk NB card in Slot A. The echo task exercises the transmit and receive functions of the SNAP interface, causing a single card to send frames to itself, effectively flooding the network with SNAP frams. This exercise is useful because it verifies overall operation of the card and also provides a template for implementing the listener function and for managing global data structures under a dynamically downloaded task. (Chapter 7 discussed some of the problems associated with managing global data structures under a dynamically downloaded task.) The program was developed using the Macintosh Programmer's Workshop (MPW).

The program does not produce any displayed output. By using a network packet analyzing tool, such as a Sniffer from Data General, you can examine the traffic created by this program.

The echo task program contains several modules:

- Header files. The C language include file giving constants that can be added together to set the options field in many of the LLC and SNAP calls.
- Make files. The make files for building the program example that show how to make the program from its various modules.
- Source files. The source files for DynDownLoadExamp.

The program listing consists of modules that support the echo task. It demonstrates many features and techniques for working with the LLC and SNAP interface:

- How to conditionally compile and use a listener function
- How to write a protocol (shown in Echo.c)
- How to start tasks from other tasks (shown in Download.c)
- How to find and use SNAP from MR-DOS.

Programming checklist

The following procedure describes step-by-step how to create the example program using MPW Version 3.0 (or later). Copies of the files are provided on the distribution diskettes supplied with TokenTalk NB development tools.

1. Copy the Apple IPC and Token Talk Prep files into your System Folder.
2. Copy the MR-DOS Includes folder into your MPW folder.
3. Copy the UserStartup•TokenRingExamp file into your MPW folder.
4. Copy the TokenRingExamp folder into the MPW folder.
5. Create a new folder in MPW and name it TokenTalk Includes. Copy the following files into this folder:
   - LLC.h
   - SNAP.h
   - TRInit.h
   - TTUtil.h
   - TTInit.r
6. Creat a new folder in MPW and name it TokenTalk Libraries. Copy the following file into this folder:
   - LLCSupportLib.o

These folders and files are required to build the example program. If Apple IPC did not already exist in the System Folder prior to copying these files, you must reboot before you can execute the example program.

The next steps are accomplished using MPW 3.0 or later. If you are not using MPW 3.0 or later, you can copy UserStartup•TokenRingExamp to your UserStartup file.

1. Launch MPW.
2. Build the Echo Task program.
3. Build the DynDownLoadExamp program.
4. Quit MPW.
5. Copy Echo Task to your System Folder.
6. Launch DynDownLoadExamp.

At this point, DynDownLoadExamp downloads the Echo Task program to the TokenTalk NB card in Slot A of your Macintosh II. The Echo Task program floods the token ring with frames.

Program listing

The remainder of this appendix is a listing of the make files, header files, and source files that create the dynamic download task and the echo task. The modules are presented as follow:

- **Dynamic download**
  - DynDownLoadExamp.make
  - DynDownLoad.c
- **Dynamic global data structure management**
  - ADT.h
  - ADT.c
  - ListenerGlue.a
- **Echo task**
  - EchoTask.make
  - Echo.h
  - General.h
  - EchoBlastTask.c
  - EchoTask.c
  - EchoTask.r
- **MR-DOS and SNAP interface**
  - Externals.h
  - SNAP-Interface.h
  - Echo-Interface.h
  - MREcho-Interface.c
  - MRSNAP-Interface.c

Dynamic download

The following program files show the make file for the dynamic download process and the source code that launches the download process.

DynDownLoadExamp.make

```
#    File:        DynDownLoadExamp.make
#    Target:      DynDownLoadExamp
#    Sources:     DynDownLoad.c
#    Created:     Monday, January 30, 1989 8:48:13 AM

DynDownLoad.c.o ƒ DynDownLoadExamp.make DynDownLoad.c
        C  DynDownLoad.c

SOURCES = DynDownLoad.c
OBJECTS = DynDownLoad.c.o

DynDownLoadExamp ƒƒ DynDownLoadExamp.make {OBJECTS}
       Link -w -t APPL -c '????' ∂
               {OBJECTS} ∂
               "{CLibraries}"CRuntime.o ∂
               "{Libraries}"Interface.o ∂
               "{CLibraries}"StdCLib.o ∂
               "{CLibraries}"CSANELib.o ∂
               "{CLibraries}"Math.o ∂
               "{CLibraries}"CInterface.o ∂
               -o DynDownLoadExamp
```

DynDownLoad.c

```
/**************************************************************************************\
*                                                                                      *
*       File: DynDownLoad.c                                                            *
*       Written by Eric M. Trehus                                                      *
*       Copyright Apple Computer, Inc. 1988-1989                                       *
*       All rights reserved                                                            *
*                                                                                      *
\**************************************************************************************/
#include <Files.h>
#include <Resources.h>
#include <TTUtil.h>
#include <Memory.h>

/*
        This is the tiny Application that downloads our EchoTask program onto a Token Talk NB
        Card in slot A.  It makes assumptions about the location and name of the file. It also
        downloads LLC onto the card if it hasn't been loaded.
*/

static TTUtilPtr GlobalUtilPtr;
static short TTRefNum;

TTUtilPtr GetTTUtilPtr(char *PrepFile,short VRefNum,short *refptr)
{
        Handle utlhdl;
        short ttrefnum;
        ttrefnum = OpenRFPerm(PrepFile,VRefNum,fsRdPerm);
        if(ttrefnum == -1)
        {
                /* Error in Opening the file */
                return(0);
        }
        if((utlhdl = Get1Resource('ttut',0)) == 0)
        {
                /* Error in getting the resource */
                CloseResFile(ttrefnum);
                return(0);
        }
        *refptr = ttrefnum;
        return((TTUtilPtr)StripAddress(*utlhdl));
}

long FindCards(long mask)
{
        return((*GlobalUtilPtr)(TTFindCards,mask));
}
```

```
long FindBootedCards(long mask)
{
        return((*GlobalUtilPtr)(TTFindBootedCards,mask));
}
long FindUnbootedCards(long mask)
{
        return((*GlobalUtilPtr)(TTFindUnbootedCards,mask));
}
long BootCards(long mask)
{
        return((*GlobalUtilPtr)(TTBootCards,mask));
}
long ForceBoot(long mask)
{
        return((*GlobalUtilPtr)(TTForceBoot,mask));
}
long DynamicDL(TTDDLP *Parameters)
{
        return((*GlobalUtilPtr)(TTDynamicDL,(long)Parameters));
}
Bootit(long How,long MyMask)        /* Assumes that a PREP file has been opened */
{
        long BootedMask;
        long Mask;
        long type = -1;
        long Cards;
        Cards = FindCards(-1);              /* Find all the cards */

        if((How == TTForceBoot) || FindUnbootedCards(MyMask))
        {
                BootedMask = (*GlobalUtilPtr)(How,MyMask); /* Perform Download here */
        }
        Mask = FindBootedCards(Cards);      /* Look through all cards to find booted ones */
}

void InitLLC(void)    /* Call this once before executing any tests */
{
        SysEnvRec sysrec;


        if(SysEnvirons(1,&sysrec))
        {
                /* Error in SysEnvirons call */
                return;
        }
        GlobalUtilPtr = GetTTUtilPtr("\pTokenTalk Prep",sysrec.sysVRefNum,&TTRefNum);
        Bootit(TTBootCards,-1);
}
```

```
void DoDynamicDownLoad(void)
{
        long type = -1;
        short ResRefNum;
        SysEnvRec sysrec;
        TTDDLP *DynamicDownLoadParms;
        long DownLoadTID;
        Handle DynamicDPHandle;
        long DynDownLoadSlot;

        DynDownLoadSlot = 0x0A;

        /*
                In this example, we expect a file in the system folder named "Echo Task", which
                contains the 'PARM' resource that indicates how to Download the code onto the
                Token Talk NB Card.  We are also assuming that the Token Talk NB Card is
                in slot A for simplicity.
        */
        if(SysEnvirons(1,&sysrec))
        {
                /* Error in SysEnvirons call */
                return;
        }
        ResRefNum = OpenRFPerm("\pEcho Task",sysrec.sysVRefNum,fsRdPerm);

        DynamicDPHandle = Get1Resource('PARM',0);
        if(DynamicDPHandle)
        {
                HLock(DynamicDPHandle);
                DynamicDownLoadParms = (TTDDLP *)(*DynamicDPHandle);
                DynamicDownLoadParms->SlotNo = DynDownLoadSlot;

                DownLoadTID = DynamicDL((TTDDLP *)StripAddress((char *)DynamicDownLoadParms));
                HUnlock(DynamicDPHandle);

                CloseResFile(ResRefNum);
        }
}

main()
{
        InitLLC();                          /* Make sure the card is initialized */
        DoDynamicDownLoad();  /* Now put my task there */
}
```

Dynamic global data structure management

The following program files show how to set up and manage the global data structure in a dynamic download environment. Chapter 7 discusses the problems that can occur when the task's pointers are managed incorrectly. An assembly language routine shows how to capture and restore the dynamic download task's A5 register so that it points to the correct global data structure.

ADT.h

```
/*****************************************************************************************\
*                                                                                         *
*       File: ADT.h                                                                       *
*       Written by Eric M. Trehus                                                         *
*       Copyright Apple Computer, Inc. 1988-1989                                          *
*       All rights reserved                                                               *
*                                                                                         *
\*****************************************************************************************/
#ifndef __ADT__
#define __ADT__

#include <Types.h>

typedef struct ELEMENT
{
        struct ELEMENT *Next;
}ELEMENT;

typedef struct QUEUE
{
        ELEMENT *Head;
        ELEMENT *Tail;
        long    Size;
        Boolean         InUse; /* Optional Flag */
}QUEUE;

void InitQueue(QUEUE *Queue);
void EnQueue(void *Element,QUEUE *Queue);
void *ServeQueue(QUEUE *Queue);

#endif
```

ADT.c

```
/*********************************************************************************************\
*                                                                                             *
*       File: ADT.c                                                                           *
*       Written by Eric M. Trehus                                                             *
*       Copyright Apple Computer, Inc. 1988-1989                                              *
*       All rights reserved                                                                   *
*                                                                                             *
\*********************************************************************************************/

#include <ADT.h>
#include <STDIO.h>
#include <strings.h>
#include <os.h>

/*
        This file provides Queue Manipulation routines.  It ensures mutual exclusion during
        critical code regions through the use of Rescheduling.
*/

void InitQueue(QUEUE *Queue)            /* Initializes a queue as empty */
{
        Queue->Head = NULL;
        Queue->Tail = NULL;
        Queue->Size = 0;
        Queue->InUse = false;
}

void EnQueue(void *Element,QUEUE *Queue)          /* Adds an element to the end of the queue
*/
{
        short   oldSchedMode;

        oldSchedMode = Reschedule(OS_BLOCK_IMMED);

        if(Queue->Size)
        {
                Queue->Tail->Next = (ELEMENT *)Element;     /* Link in the new Element */
                Queue->Tail = (ELEMENT *)Element;           /* Update Tail */
        }
        else                                                /* The Queue is empty */
        {
                Queue->Head = (ELEMENT *)Element;   /* Head and Tail is the same Element */
                Queue->Tail = (ELEMENT *)Element;
        }
        Queue->Size++;                              /* Show that the Queue has grown */
        if(oldSchedMode == OS_SLICE_MODE)
                Reschedule(OS_SLICE_MODE);
}

void *ServeQueue(QUEUE *Queue)       /* Removes the first element from the queue,
                                        and returns a pointer to it */
{
        ELEMENT *ptr;
        short   oldSchedMode;

        oldSchedMode = Reschedule(OS_BLOCK_IMMED);

        if(Queue->Size)
        {
                ptr = Queue->Head;                  /* FIFO */
                Queue->Head = ptr->Next;            /* Update Head to point to Next Element */
                Queue->Size--;                      /* show that the Queue has shrunken */
        }
        else
                ptr = NULL;                         /* No Elements in the Queue */

        if(oldSchedMode == OS_SLICE_MODE)
                Reschedule(OS_SLICE_MODE);
        return(ptr);
}
```

ListenerGlue.a

```
**      Written by Eric M. Trehus
**      Copyright (C) Apple Computer Inc., 1988.
**      All Rights Reserved.
**

**      Glue code so that A5 is set up when our EchoListener function is called.

**      SaveA5 - Save A5 in code space.
                CASE ON                                 ; Case is important to C.
                Proc
                Export          SaveA5

SaveA5          LEA             EchoA5, A0              ; Get location to keep A5
                Move.L          A5, (A0)                ; Put A5 in that location
                RtS                                     ; Return

EchoA5  DC.L    0                                       ; Keep A5 Here.


**      EchoListen - Set up to call Echo listener.

                Export          EchoListen
                Import          EchoListener

EchoListen
                Move.L          A5, -(A7)               ; Save A5 on stack
                MoveA.L         EchoA5, A5              ; Set A5
                JSR             EchoListener            ; Call listener
                MoveA.L         (A7)+, A5               ; Restore A5
                RtS                                     ; Return
                Endp
                End
```

The echo task

The following program are the major components of the echo task that is dynamically downloaded to the TokenTalk NB card. The make file, header files, and source files for the echo task are included.

EchoTask.make

```
EchoBlastTask.c.o ƒ 'Echo Task'.make EchoBlastTask.c
	C {CompilerOptions} EchoBlastTask.c

EchoTask.c.o ƒ 'Echo Task'.make EchoTask.c
	C {CompilerOptions} EchoTask.c

ADT.c.o ƒ 'Echo Task'.make ADT.c
	C {CompilerOptions} ADT.c

MRSNAP-Interface.c.o ƒ 'Echo Task'.make MRSNAP-Interface.c
	C {CompilerOptions} MRSNAP-Interface.c

MREcho-Interface.c.o ƒ 'Echo Task'.make MREcho-Interface.c
	C {CompilerOptions} MREcho-Interface.c

'Echo Task' ƒƒ 'Echo Task'.make EchoTask.r
	Rez EchoTask.r -append -o 'Echo Task'

'Listener Glue.a.o' ƒ 'Echo Task'.make 'Listener Glue.a'
	asm 'Listener Glue.a'

SOURCES =  EchoTask.r MRSNAP-Interface.c EchoTask.c ADT.c MREcho-Interface.c EchoBlastTask.c
'Listener Glue.a'
OBJECTS = MRSNAP-Interface.c.o EchoTask.c.o ADT.c.o MREcho-Interface.c.o EchoBlastTask.c.o
'Listener Glue.a.o'

'Echo Task' ƒƒ 'Echo Task'.make {OBJECTS}
	Link {LinkOptions} -t Card -c mash ∂
		{OBJECTS} ∂
		"{IPCLibraries}"osglue.o  ∂
		"{LLCLibraries}"LLCSupportLib.o ∂
		-o "{SystemFolder}"'Echo Task'
```

Echo.h

```
/*********************************************************************************************\
*                                                                                             *
*      File: Echo.h                                                                           *
*      Written by Eric M. Trehus                                                              *
*      Copyright Apple Computer, Inc. 1988-1989                                               *
*      All rights reserved                                                                    *
*                                                                                             *
\*********************************************************************************************/
#ifndef __Echo__
#define __Echo__
#include <LLC.h>
/* mCodes for Echo Protocol */
#define EchoOpen            0x0EC0
#define EchoClose           0x0EC2
#define EchoReceive         0x0EC4
#define EchoTransmit  0x0EC6

#define       EchoNoErr     0x0000 /* No Errors, good result */
#define EchoBadRefNum       0x0101 /* Bad refnum passed in */
#define EchoClosed          0x0102 /* Echo was closed */
#define EchoTooMany         0x0103 /* No resouces left */
#define EchoTruncated       0x0104 /* Buffer not large enough */

/* mOData of IPC will contain the following structure for EchoOpen, EchoClose */

typedef struct EchoRefNumOData
{
       unsigned short RefNum;             /* Given from EchoOpen */
}EchoRefNumOData;
/* mOData of IPC will contain the following structure for an EchoTransmit */
/* mDataPtr will point to the buffer to be transmitted */
/* mDataSize is the size of the information to be transmitted */
typedef struct EchoTransmitOData
{
       unsigned short RefNum;             /* Given from EchoOpen */
       LANHdr *Hdr;                       /* Hdr to use on Echo Frame */
}EchoTransmitOData;
/* mOData of IPC will contain the following structure for an EchoReceive */
/* mDataPtr will point to the buffer for information to be placed */
/* mDataSize is the size of the buffer */
typedef struct EchoReceiveOData
{
       unsigned short RefNum;
       unsigned short InfoLen;      /* Amount of information placed in the buffer */
       LANHdr *Hdr;
}EchoReceiveOData;
#endif
```

General.h

```
/*************************************************************************************\
*                                                                                     *
*       File: General.h                                                               *
*       Written by Eric M. Trehus                                                     *
*       Copyright Apple Computer, Inc. 1988-1989                                      *
*       All rights reserved                                                           *
*                                                                                     *
\*************************************************************************************/
#define Sync 0
#define Async 1
#define CannotGetMessageBufferErr    0xFE

#define byte unsigned char
#define word unsigned short

#define ODataAs(x,y)   ((x *)  ((y)->mOData))
#define SDataAs(x,y)   ((x *)  ((y)->mSData))
#define DPAs(x,y)              ((x *)  ((y)->mDataPtr))
```

EchoBlastTask.c

```
/*******************************************************************************************\
*                                                                                           *
*       File: EchoBlastTask.c                                                               *
*       Written by Eric M. Trehus                                                           *
*       Copyright Apple Computer, Inc. 1988-1989                                            *
*       All rights reserved                                                                 *
*                                                                                           *
\*******************************************************************************************/
#include <os.h>
#include <managers.h>
#include <mrdos.h>
#include <siop.h>
#include <LLC.h>
#include <types.h>
#include <Echo.h>
#include <Echo-Interface.h>
#include <Externals.h>

/*
        EchoBlastTask continueously broadcasts frames using our Echo Protocol.
*/

static void EchoBlastTask()
{
        word                            Result;
        LANHdr                          Hdr;
        message                                 *Message;
        char                            *TransmitData;
        word                            BrodcastAddr[3];
        word                            RefNum;
        /* Get a Hdr for transmitting */

        BrodcastAddr[0] = 0xC000;
        BrodcastAddr[1] = 0xFFFF;
        BrodcastAddr[2] = 0xFFFF;

        TransmitData = "Sending Echo Frames to everyone as fast as I can";
        Result = Echo_Open(Sync,&RefNum);
        Result = SNAP_GetHdr(Sync,3,0,0,0,&Hdr,6,BrodcastAddr);
        Result = Echo_Transmit(Async,RefNum,&Hdr,80,TransmitData);
        for(;;)
        {
                Message = Receive(0,0,0,0);

                SwapTID(Message);            /* Prepare to reissue the transmit request */
                Message->mCode &= 0x7FFE;    /* Fix the mCode */
                Send(Message);               /* Re-Queue the transmit */
        }
        Result = Echo_Close(Sync,RefNum);
}

void StartEchoBlastTask()    /* Create the EchoBlastTask */
{
        struct ST_PB          stpb, *pb;

        pb = &stpb;
        pb -> CodeSegment = NULL;
        pb -> DataSegment = NULL;
        pb -> StartParmSegment = NULL;
        pb -> stack = 12000;
        pb -> heap = 0;
        pb -> priority = 31;
        pb -> InitRegs.PC = EchoBlastTask;
        pb -> InitRegs.A_Registers [5] = GetMyA5();
        pb -> ParentTID = GetTID();
        if (StartTask (pb) == 0)
                illegal ();
}
```

EchoTask.c

```
/*************************************************************************************\
*                                                                                     *
*       File: EchoTask.c                                                              *
*       Written by Eric M. Trehus                                                     *
*       Copyright Apple Computer, Inc. 1988                                           *
*       All rights reserved                                                           *
*                                                                                     *
\*************************************************************************************/

/*
       The Echo protocol consists of 4 commands:

              EchoOpen:      Allocates resources for the client, and assigns a refnum so that
                             the client can accumulate the responses via EchoReceive.

              EchoReceive:   When an EchoReply is received a search is made for a matching
                             EchoReceive request.

              EchoTransmit:  Transmits a SNAP frame with our Echo Protocol Descriminator, and
                             the clients refnum, and data.

              EchoClose:     Deallocates resources allocated from EchoOpen, and cancels all
                             pending EchoReceive requests.


       The pre-processor symbol UseEchoListener can be used to create 2 different versions
       of the EchoTask.  If UseEchoListener is defined, then SNAP's listener function is
       used.  This is more efficient than using SNAPReceive, however it is slightly more
       complicated to use.  Otherwise SNAPReceive's are posted, and reposted as they
       complete.

       The EchoTask that is started here responds to EchoRequests when received.
*/


#define MCP

#include <os.h>
#include <managers.h>
#include <mrdos.h>
#include <siop.h>
#include <SNAP.h>
#include <types.h>
#include <ADT.h>
#include <Externals.h>
#include <SNAP-Interface.h>
#include <Echo.h>

#define UseEchoListener              /* Use SNAPs listener function vrs SNAPReceive */
#define MaxOpen 10                   /* Maximum number of EchoQueue's to be opened */
#define UseEchoListener
```

```
typedef struct
{
        char    PD[5];
        short   RefNum;
}       EchoHeaderStruct;           /* SNAP PD and Clients RefNum goes here */


tid_type GlobalSNAPTID;             /* Task Identifier of SNAP on this card */
tid_type GlobalEchoTID;             /* Task Identifier of Echo Protocol on this card */
long GlobalLLCMessagePriority;      /* Priority of messages used in this system */

static QUEUE EchoQueue[MaxOpen];
word    EchoPDRefNum;
int             NOpen;

static char *EchoBuffer1,*EchoBuffer2,*EchoBuffer3;
static LANHdr EchoHeader1,EchoHeader2,EchoHeader3;
word EchoPDRefNum;

LLCGetConfigBuffer    ConfigBuffer;

/*******************************************************************************************\
*                                                                                           *
*       We will use the Protocol Descriminator as the indicator for both Echo Requests,    *
*       and Echo Repy's as follows:                                                         *
*                                                                                           *
*       Echo Request PD is EE EE EE EE EE.                                                  *
*       Echo Reply PD is EE EE EE EE EF.                                                    *
*                                                                                           *
\*******************************************************************************************/


/*      We bootstrap our self by starting 2 additional tasks, the first one is to
        respond to Echo Requests on the token ring network.  The second one is to start
        our client which uses our Echo Protocol services.  Finally we fall into a loop and
        the clients of our Echo Protocol send their messages here.
*/
```

```
main ()
{
        message *Message;

        GlobalSNAPTID = FindMySNAP();
        GlobalEchoTID = GetTID();     /*Alternatly we could register a name, and let our
                clients find us using the name manager.  Then our protocol could
                be used by tasks everywhere.  You get more bussiness if
                you advertise! */
        GlobalLLCMessagePriority = 0;
        SNAP_GetConfig(false,sizeof(LLCGetConfigBuffer),&ConfigBuffer);

#ifdef UseEchoListener
        SaveA5();       /* Use Glue, EchoListen will access a variable not based on A5 to get A5
*/
#endif

        StartEchoTask();              /* Sets up the Echo Protocol */
        StartEchoBlastTask();         /* Client of the Echo Protocol */

        for(;;)
        {
                Message = Receive(0,0,0,0);
                switch(Message->mCode)
                {
                        case EchoOpen:
                                StartEchoOpen(Message);
                                break;

                        case EchoClose:
                                StartEchoClose(Message);
                                break;

                        case EchoReceive:
                                StartEchoReceive(Message);
                                break;

                        case EchoTransmit:
                                StartEchoTransmit(Message);
                                break;

#ifndef         UseEchoListener
                        case SNAPReceive | 1:        /* If it completes */
                                EchoComplete(Message);
                                break;
#endif
                }
        }
}
```

```
static tid_type FindMySNAP()
{
        short index = 0;
        return(Lookup_Task("=","SNAP",GetNameTID(),&index));
}


#ifdef UseEchoListener
        extern void EchoListen();
#endif

InitEchoProtocol()    /* Initializes the Echo Protocol */
{
        char EchoReplyDescriminator[5];
        long BufferSize;
        word Result;
        void EchoListen();

        BufferSize = ConfigBuffer.MaxFrameLen - ConfigBuffer.MaxHeader;
        EchoReplyDescriminator[0] = 0xEE;
        EchoReplyDescriminator[1] = 0xEE;
        EchoReplyDescriminator[2] = 0xEE;
        EchoReplyDescriminator[3] = 0xEE;
        EchoReplyDescriminator[4] = 0xEF;

        /* Allocate 3 buffers */
        EchoBuffer1 = GetMem(BufferSize);
        EchoBuffer2 = GetMem(BufferSize);
        EchoBuffer3 = GetMem(BufferSize);

#ifndef UseEchoListener
        Result = SNAP_Attach(Sync,&EchoPDRefNum,0,NULL,EchoReplyDescriminator);
        Result = SNAP_Receive(Async,EchoPDRefNum,0,&EchoHeader1,BufferSize,EchoBuffer1);
        Result = SNAP_Receive(Async,EchoPDRefNum,0,&EchoHeader2,BufferSize,EchoBuffer2);
        Result = SNAP_Receive(Async,EchoPDRefNum,0,&EchoHeader3,BufferSize,EchoBuffer3);
#else
        Result =
SNAP_Attach(Sync,&EchoPDRefNum,ListenerFunction,EchoListen,EchoReplyDescriminator);
#endif
}
```

```
static void EchoTask()       /* Turns around any Echo Request into an Echo Reply */
{
        word            Result;
        word            PDRefNum;
        LANHdr          Hdr[3];
        char            *Buffer1;
        char            *Buffer2;
        char            *Buffer3;
        message         *Message;
        char            ErrorCount;
        char            EchoRequestDescriminator[5];
        long            ReceiveBufferSize;
        long            ID;

        ErrorCount = 0;

        EchoRequestDescriminator[0] = 0xEE;
        EchoRequestDescriminator[1] = 0xEE;
        EchoRequestDescriminator[2] = 0xEE;
        EchoRequestDescriminator[3] = 0xEE;
        EchoRequestDescriminator[4] = 0xEE;

        ReceiveBufferSize = ConfigBuffer.MaxFrameLen - ConfigBuffer.MaxHeader;

        Buffer1 = GetMem(ReceiveBufferSize);
        Buffer2 = GetMem(ReceiveBufferSize);
        Buffer3 = GetMem(ReceiveBufferSize);
        if(Buffer1 && Buffer2 && Buffer3)    /* If we got memory in all requests */
        {
                /* Queue up 3 Receive requests */

                Result = SNAP_Attach(Sync,&PDRefNum,0,NULL,EchoRequestDescriminator);

                Result = SNAP_Receive(Async,PDRefNum,0,&Hdr[0],ReceiveBufferSize,Buffer1);

                Result = SNAP_Receive(Async,PDRefNum,0,&Hdr[1],ReceiveBufferSize,Buffer2);

                Result = SNAP_Receive(Async,PDRefNum,0,&Hdr[2],ReceiveBufferSize,Buffer3);

                for(;;)         /* Do this until told otherwise */
                {
                        Message = Receive(0,0,0,0);
                        if(Message->mStatus)
                        {
                                FreeMsg(Message);       /* After 3 errors,
                                                           we will no longer echo */
                                ErrorCount++;
                                if(ErrorCount == 3)
                                {
                                        break;          /* Thats it, 3 strikes */
                                }
                        else
                        {
                                SwapTID(Message);     /* Prepare to echo reply */
                                LLCSwapHdr(ODataAs(LLCTxRxOData,Message)->Hdr,0xAA);
                                Message->mCode = SNAPTransmit;
                                ID = Message->mId;
                                Message->mDataPtr[4] = 0xEF;          /* Make it an Echo Reply */
                                Send(Message);                /* The packet is on its way */
                                Message = Receive(ID,0,0,0);          /* Wait for transmit
                                                                         to complete */

                                SwapTID(Message);             /* Prepare to reissue SNAPReceive */
                                Message->mCode = SNAPReceive;
                                ODataAs(SNAPReceiveOData,Message)->PDRefNum = PDRefNum;   /*
Transmit Messed me up */
                                Send(Message);                /* Requeue SNAP_Receive */
                        }
                }
                Result = SNAP_Detach(Sync,PDRefNum);
                FreeMem(Buffer1);
                FreeMem(Buffer2);
                FreeMem(Buffer3);
        }
}

StartEchoTask()       /* Create the new task EchoTask */
{
        struct ST_PB          stpb, *pb;


        pb = &stpb;
        pb -> CodeSegment = NULL;
        pb -> DataSegment = NULL;
        pb -> StartParmSegment = NULL;
        pb -> stack = 12000;
        pb -> heap = 0;
        pb -> priority = 31;
        pb -> InitRegs.PC = EchoTask;
        pb -> InitRegs.A_Registers [5] = GetMyA5();
        pb -> ParentTID = GetTID();
        InitEchoQueue();
        InitEchoProtocol();
        if (StartTask (pb) == 0)
                illegal ();
}

StartEchoTransmit(message *Message)
{
        LLCList       LBuffer[2];
        EchoHeaderStruct EchoHeader;
        word RefNum;
        RefNum = ODataAs(EchoRefNumOData,Message)->RefNum;
        if(!EchoQueue[RefNum].InUse || RefNum >= MaxOpen || !RefNum)
                                                            /* Check for invalid refnum */
                Message->mStatus = EchoBadRefNum;
        else
        {
                EchoHeader.PD[0] = 0xEE;
                EchoHeader.PD[1] = 0xEE;
                EchoHeader.PD[2] = 0xEE;
                EchoHeader.PD[3] = 0xEE;
                EchoHeader.PD[4] = 0xEE;
                EchoHeader.RefNum = RefNum;

                LBuffer[0].Count = sizeof(EchoHeader);
                LBuffer[0].Ptr = (char *)&EchoHeader;
                LBuffer[1].Count = Message->mDataSize;
                LBuffer[1].Ptr = Message->mDataPtr;

                Message->mStatus = SNAP_Transmit(Sync,sizeof(EchoHeader)+Message->mDataSize,

        ListDirected,3,ODataAs(EchoTransmitOData,Message)->Hdr,
                               sizeof(LBuffer),(char *)LBuffer);
        }
        SwapTID(Message);
        Message->mCode |= 1;  /* Mark that it is a reply, Transmit complete */
        Send(Message);
}

static InitEchoQueue()
{
        int i;

        for(i=0;i<MaxOpen;i++)
                InitQueue(&EchoQueue[i]);
        EchoQueue[0].InUse = true;    /* Waste 1 queue so refnums are never 0 */
        NOpen = 1;
}

static GetFreeEchoQueueIndex()
{
        int i;

        for(i=1;i<MaxOpen;i++)
        {
                if(!EchoQueue[i].InUse)
                break;
        }
        return(i);
}
```

```
StartEchoOpen(message *Message)
{
        word RefNum;

        Message->mStatus = EchoNoErr;
        if(NOpen >= MaxOpen)
        {
                Message->mStatus = EchoTooMany;
        }
        else
        {
                NOpen++;        /* We are going to allocate the resources */
                RefNum = GetFreeEchoQueueIndex();
                EchoQueue[RefNum].InUse = true;      /* Mark the queue busy */
                ODataAs(EchoRefNumOData,Message)->RefNum = RefNum;
        }
        SwapTID(Message);
        Message->mCode |= 1;
        Send(Message);
}

/* This will cancel all of the Echo Receive Requests */
StartEchoClose(message *Message)
{
        message *mp;
        word RefNum;

        Message->mStatus = EchoNoErr;
        RefNum = ODataAs(EchoRefNumOData,Message)->RefNum;
        if(!EchoQueue[RefNum].InUse || RefNum >= MaxOpen || !RefNum)
                                                        /* Check for invalid refnum */
                Message->mStatus = EchoBadRefNum;
        else
        {
                while(mp = ServeQueue(&EchoQueue[RefNum]))
                {
                        mp->mStatus = EchoClosed;
                        SwapTID(mp);
                        mp->mCode |= 1;
                        Send(mp);
                }
                EchoQueue[RefNum].InUse = false;
                NOpen--;
        }

        SwapTID(Message);
        Message->mCode |= 1;
        Send(Message);
}
```

```
StartEchoReceive(message *Message)
{
        word RefNum;

        Message->mStatus = EchoNoErr;
        RefNum = ODataAs(EchoRefNumOData,Message)->RefNum;
        if(!EchoQueue[RefNum].InUse || RefNum >= MaxOpen || !RefNum)
                                                        /* Check for invalid refnum */
        {
                Message->mStatus = EchoBadRefNum;
                SwapTID(Message);
                Message->mCode |= 1;
                Send(Message);
        }
        else
                EnQueue(Message,&EchoQueue[RefNum]);
}

#ifndef UseEchoListener
EchoComplete(message *Message)
                        /* Handle SNAPReceive for EE EE EE EE EF Protocol Descriminator */
{
        message *mp;  /* A pointer to the users EchoReceive Message structure */
        word InfoLen; /* Length of information placed into user's buffer */
        word RefNum;
        int EchoHeaderSize;

        EchoHeaderSize = sizeof(EchoHeaderStruct);
        if(!Message->mStatus)           /* If there was an error */
        {
                /* Get Refnum from frame received */
                RefNum = ((EchoHeaderStruct *)(Message->mDataPtr))->RefNum;
                if(mp = ServeQueue(&EchoQueue[RefNum]))
                {
                        mp->mStatus = EchoNoErr;
                                        /* Assume no error until proven otherwise */

                        InfoLen = ODataAs(SNAPReceiveOData,Message)->InfoLen - EchoHeaderSize;
                        if(InfoLen > mp->mDataSize)
                                        /* Error, got more than we were asking for */
                        {
                                mp->mStatus = EchoTruncated;
                                InfoLen = mp->mDataSize;
                        }
                        ODataAs(EchoReceiveOData,mp)->InfoLen = InfoLen;

                        /* Copy the data into the user's buffer */
                        CopyNuBus(Message->mDataPtr+EchoHeaderSize,mp->mDataPtr,InfoLen);

                        /* Copy the header into the user's header */
                        CopyNuBus(ODataAs(SNAPReceiveOData,Message)
                        ->Hdr,ODataAs(EchoReceiveOData,mp)->Hdr,sizeof(LANHdr));
                        /* Send the message to the user */
                        SwapTID(mp);
                        mp->mCode |= 1;
                        Send(mp);
                }
                /* Re-issue the receive */
                SwapTID(Message);
                Message->mCode = SNAPReceive;
                Send(Message);
        }
}
#else
/* Handle SNAPReceive for EE EE EE EE EF Protocol Descriminator */
void EchoListener(long nu1,long nu2,LANHdr *Hdr,char *Buffer,int len,int FrameType)
{
        message *mp;   /* A pointer to the users EchoReceive Message structure */
        word InfoLen;  /* Length of information placed into user's buffer */
        word RefNum;
        int EchoHeaderSize;

#pragma unused(nu1)
#pragma unused(nu2)
#pragma unused(FrameType)

        EchoHeaderSize = sizeof(EchoHeaderStruct);
        /* Get Refnum from frame received */
        RefNum = ((EchoHeaderStruct *)(Buffer))->RefNum;
        if(mp = ServeQueue(&EchoQueue[RefNum]))
        {
                mp->mStatus = EchoNoErr;      /* Assume no error until proven otherwise */
                InfoLen = len - EchoHeaderSize;
                if(InfoLen > mp->mDataSize)  /* Error, got more than we were asking for */
                {
                        mp->mStatus = EchoTruncated;
                        InfoLen = mp->mDataSize;
                }
                ODataAs(EchoReceiveOData,mp)->InfoLen = InfoLen;
                /* Copy the data into the user's buffer */
                CopyNuBus(Buffer+EchoHeaderSize,mp->mDataPtr,InfoLen);
                /* Copy the header into the user's header */
                CopyNuBus(Hdr,ODataAs(EchoReceiveOData,mp)->Hdr,sizeof(LANHdr));
                /* Send the message to the user */
                SwapTID(mp);
                mp->mCode |= 1;
                Send(mp);
        }
}
#endif
```

EchoTask.r

```
/****************************************************************************************\
*                                                                                        *
*       File: EchoTask.r                                                                 *
*       Written by Eric M. Trehus                                                        *
*       Copyright Apple Computer, Inc. 1988-1989                                         *
*       All rights reserved                                                              *
*                                                                                        *
\****************************************************************************************/
type 'mash' {
        pstring;
};

resource 'mash' (0) {
        $$Format("Echo Task %s",$$Date)
};

type 'PARM'     /* Created resource type for dynamic download */
{
        longint;        /* Resource type holding code to download */
        longint;        /* SlotNo */
        longint;        /* ParamSize */
        longint;        /* CodeSegment:       memory region on card for code      */
        longint;        /* DataSegment:       memory region on card for global data*/
        longint;        /* StartParmSegment:  memory region on card for start parameters */
        longint;        /* D0 */
        longint;        /* D1 */
        longint;        /* D2 */
        longint;        /* D3 */
        longint;        /* D4 */
        longint;        /* D5 */
        longint;        /* D6 */
        longint;        /* D7 */
        longint;        /* A0 */
        longint;        /* A1 */
        longint;        /* A2 */
        longint;        /* A3 */
        longint;        /* A4 */
        longint;        /* A5 */
        longint;        /* A6 */
        longint;        /* A7 */
        longint;        /* PC;        Program Counter                                    */
        longint;        /* stack;     initial stack size (in bytes)                      */
        longint;        /* heap;      initial heap size (in bytes)                       */
        integer;        /* return_code;     error code if task not started (Tid = 0)     */
        unsigned byte; /* priority;  priority of task (in bytes)                  */
        longint;        /* ParentTID; TID of Parent on Network/Host                      */
};
```

```
resource 'PARM' (0)
{
        'CODE',         /* Resource type holding code to download */
        10,             /* SlotNo A, Assume this is the only place it will go */
        0,              /* ParamSize */
        0,              /* CodeSegment:      memory region on card for code             */
        0,              /* DataSegment:      memory region on card for global data      */
        0,              /* StartParmSegment: memory region on card for start parameters */
        0,              /* D0 */
        0,              /* D1 */
        0,              /* D2 */
        0,              /* D3 */
        0,              /* D4 */
        0,              /* D5 */
        0,              /* D6 */
        0,              /* D7 */
        0,              /* A0 */
        0,              /* A1 */
        0,              /* A2 */
        0,              /* A3 */
        0,              /* A4 */
        0,              /* A5 */
        0,              /* A6 */
        0,              /* A7 */
        0,              /* PC;         Program Counter                                  */
        32768,          /* stack;      initial stack size (in bytes)                    */
        0,              /* heap;       initial heap size (in bytes)                     */
        0,              /* return_code;      error code if task not started (Tid = 0)   */
        20,             /* priority;   priority of task (in bytes)                      */
        0               /* ParentTID;  TID of Parent on Network/Host                    */
};
```

Interface to MR-DOS and SNAP

The final set of program files show how to set up the interface to MR-DOS and SNAP by means of header files that declare the necessary parameters.

Externals.h

```
/****************************************************************************************\
*                                                                                        *
*       File: Externals.h                                                                *
*       Written by Eric M. Trehus                                                        *
*       Copyright Apple Computer, Inc. 1988-1989                                         *
*       All rights reserved                                                              *
*                                                                                        *
\****************************************************************************************/
extern tid_type GlobalSNAPTID;          /* Task Identifier of SNAP on this card */
extern tid_type GlobalEchoTID;
extern long GlobalLLCMessagePriority;   /* Priority of messages used in this system */

pascal void illegal ()
       extern 0x4afc;

unsigned long GetMyA5() = {0x200D};
```

SNAP-Interface.h

```
/*****************************************************************************************\
*                                                                                         *
*      File: SNAP-Interface.h                                                             *
*      Written by Eric M. Trehus                                                          *
*      Copyright Apple Computer, Inc. 1988-1989                                           *
*      All rights reserved                                                                *
*                                                                                         *
\*****************************************************************************************/

#ifndef __SNAPINTERFACE__
#define __SNAPINTERFACE__

#include <General.h>
#include <LLC.h>
word SNAP_Attach(int                         SyncFlag,
              word                           *PDRefNum,
              word                           Options,
              void                           (*Listener)(),
              void                           *ProtocolDescriptor);

word SNAP_Detach(int                  SyncFlag,
              word                    RefNum);

word SNAP_GetConfig(int               SyncFlag,
              long                    ConfigBufferSize,
              LLCGetConfigBuffer      *ConfigBuffer);

word SNAP_GetHdr(int                  SyncFlag,
              word                    HdrType,
              word                    Options,
              byte                    SSAP,
              byte                    DSAP,
              LANHdr                  *Hdr,
              long                    AddressSize,
              char                    *Address);
```

```
word SNAP_Transmit( int             SyncFlag,
              word                  InfoLen,
              word                  Options,
              byte                  FrameType,
              LANHdr                *Hdr,
              long                  BufferSize,
              char                  *Buffer);

word SNAP_Receive(int               SyncFlag,
              word                  PDRefNum,
              word                  Options,
              LANHdr                *Hdr,
              long                  BufferSize,
              char                  *Buffer);
#endif
```

Echo-Interface.h

```
/********************************************************************************\
*                                                                                *
*      File: Echo-Interface.h                                                    *
*      Written by Eric M. Trehus                                                 *
*      Copyright Apple Computer, Inc. 1988-1989                                  *
*      All rights reserved                                                       *
*                                                                                *
\********************************************************************************/
#ifndef __EchoInterface__
#define __EchoInterface__

#include <LLC.h>
#include <General.h>

word Echo_Open( int                     SyncFlag,
               word                    *RefNum);

word Echo_Close( int                    SyncFlag,
               word                    RefNum);

word Echo_Receive(  int                 SyncFlag,
               word                    RefNum,
               LANHdr                  *Hdr,
               long                    BufferSize,
               void                    *Buffer);

word Echo_Transmit( int                         SyncFlag,
               word                    RefNum,
               LANHdr                  *Hdr,
               long                    BufferSize,
               void                    *Buffer);
#endif
```

MREcho-Interface.c

```
/*****************************************************************************************\
*                                                                                         *
*       File: MREcho-Interface.c                                                          *
*       Written by Eric M. Trehus                                                         *
*       Copyright Apple Computer, Inc. 1988-1989                                          *
*       All rights reserved                                                               *
*                                                                                         *
\*****************************************************************************************/
#include <STDIO.h>
#include <Types.h>
#include <os.h>
#include <LLC.h>
#include <SNAP.h>
#include <Echo.h>
#include <Echo-Interface.h>
#include <Externals.h>

/*
        MREcho-Interface.c provides a procedure interface to the ECHO protocol.  This hides
        many of the details of MR-DOS.
*/

word Echo_Open( int                 SyncFlag,
                            word           *RefNum)
{
        /* Local Variables */
        message *Message;
        word Result = 0;
        long ID;

        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = EchoOpen;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalEchoTID;
                if(SyncFlag)                                /* If Async */
                {
                        Send(Message);
                }
                else                                        /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        Result = Message->mStatus;
                        *RefNum = ODataAs(EchoRefNumOData,Message)->RefNum;
                        FreeMsg(Message);
                }
        }
        else
                Result = CannotGetMessageBufferErr;
        return(Result);
}

word Echo_Close( int            SyncFlag,
                                word            RefNum)
{
        /* Local Variables */
        message *Message;
        word Result = 0;
        long ID;

        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = EchoClose;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalEchoTID;
                ODataAs(EchoRefNumOData,Message)->RefNum = RefNum;

                if(SyncFlag)                                    /* If Async */
                {
                        Send(Message);
                }
                else                                            /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        Result = Message->mStatus;
                        FreeMsg(Message);
                }
        }
        else
                Result = CannotGetMessageBufferErr;

        return(Result);
}
```

```
/*
In our example this procedure is not needed, however it is provided for completeness.
*/
word Echo_Receive(  int                 SyncFlag,
                                        word            RefNum,
                                        LANHdr          *Hdr,
                                        long            BufferSize,
                                        void            *Buffer)
{
        /* Local Variables */
        message *Message;
        word Result = 0;
        long ID;

        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = EchoReceive;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalEchoTID;
                ODataAs(EchoReceiveOData,Message)->RefNum = RefNum;
                ODataAs(EchoReceiveOData,Message)->Hdr = Hdr;
                Message->mDataSize = BufferSize;
                Message->mDataPtr = Buffer;

                if(SyncFlag)                            /* If Async */
                {
                        Send(Message);
                }
                else                                    /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        Result = Message->mStatus;
                        FreeMsg(Message);
                }
        }
        else
                Result = CannotGetMessageBufferErr;
        return(Result);
}

word Echo_Transmit( int                 SyncFlag,
                word                    RefNum,
                LANHdr                  *Hdr,
                long                    BufferSize,
                void                    *Buffer)
{
        /* Local Variables */
        message *Message;
        word Result = 0;
        long ID;
```

```
	if(Message = GetMsg())
	{
		ID = Message->mId;
		Message->mCode = EchoTransmit;
		Message->mPriority = GlobalLLCMessagePriority;
		Message->mTo = GlobalEchoTID;
		ODataAs(EchoTransmitOData,Message)->RefNum = RefNum;
		ODataAs(EchoTransmitOData,Message)->Hdr = Hdr;
		Message->mDataSize = BufferSize;
		Message->mDataPtr = Buffer;

		if(SyncFlag)                            /* If Async */
		{
			Send(Message);
		}
		else                                    /* Sync */
		{
			Send(Message);
			Message = Receive(ID,0,0,0);
			Result = Message->mStatus;
			FreeMsg(Message);
		}
	}
	else
		Result = CannotGetMessageBufferErr;

	return(Result);
}
```

MRSNAP-Interface.c

```
/************************************************************************************\
*                                                                                    *
*      File: MRSNAP-Interface.c                                                      *
*      Written by Eric M. Trehus                                                     *
*      Copyright Apple Computer, Inc. 1988-1989                                      *
*      All rights reserved                                                           *
*                                                                                    *
\************************************************************************************/

/*
       MRSNAP-Interface.c provides a procedure interface to SNAP.  This hides many of the
       details of MR-DOS.
*/

#include <os.h>             /* IPC-MRDOS interface */
#include <LLC.h>
#include <STDIO.h>
#include <Types.h>
#include <SNAP.h>
#include <General.h>
#include <Externals.h>

word SNAP_Attach(int                       SyncFlag,
              word                         *PDRefNum,
              word                         Options,
              void                         (*Listener)(),
              void                         *ProtocolDescriptor)
{
       /* Local Variables */
       message *Message;
       word Result = 0;
       long ID;

       if(Message = GetMsg())
       {
              ID = Message->mId;
              Message->mCode = SNAPAttach;
              Message->mPriority = GlobalLLCMessagePriority;
              Message->mTo = GlobalSNAPTID;

              Message->mDataPtr = ProtocolDescriptor;
              Message->mDataSize = 5;
              ODataAs(SNAPAttachOData,Message)->Options = Options;
              ODataAs(SNAPAttachOData,Message)->Listener = Listener;
```

```
                if(SyncFlag)                                    /* If Async */
                {
                        Send(Message);
                }
                else                                            /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        Result = Message->mStatus;
                        *PDRefNum = ODataAs(SNAPAttachOData,Message)->PDRefNum;

                        FreeMsg(Message);
                }
        }
        else
                Result = CannotGetMessageBufferErr;

        return(Result);

}

word SNAP_Detach(int                    SyncFlag,
                word                    RefNum)
{
        /* Local Variables */
        message *Message;
        word Result = 0;
        long ID;

        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = SNAPDetach;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalSNAPTID;

                ODataAs(SNAP_PD_RefNum,Message)->PDRefNum = (short)RefNum;

                if(SyncFlag)                                    /* If Async */
                {
                        Send(Message);
                }
                else                                            /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        Result = Message->mStatus;
                        FreeMsg(Message);
                }
        }
        else
                Result = CannotGetMessageBufferErr;

        return(Result);
}

word SNAP_GetConfig(int                     SyncFlag,
                long                        ConfigBufferSize,
                LLCGetConfigBuffer          *ConfigBuffer
                )
{
        message *Message;
        word Result = 0;
        long ID;

        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = SNAPGetConfig;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalSNAPTID;
                Message->mDataSize = ConfigBufferSize;
                Message->mDataPtr = (char *)ConfigBuffer;

                if(SyncFlag)                            /* If Async */
                {
                        Send(Message);
                }
                else                                    /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        Result = Message->mStatus;
                        FreeMsg(Message);
                }
        }
        else
                Result = CannotGetMessageBufferErr;
        return(Result);
}

word SNAP_GetHdr(int                        SyncFlag,
                        word                            HdrType,
                        word                            Options,
                        byte                            SSAP,
                        byte                            DSAP,
                        LANHdr                          *Hdr,
                        long                            AddressSize,
                        char                            *Address
                    )
{
        message *Message;
        word Result = 0;
        long ID;

        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = SNAPGetHdr;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalSNAPTID;
                ODataAs(LLCGetHdrOData,Message)->HdrType = HdrType;
                ODataAs(LLCGetHdrOData,Message)->Options = Options;
                ODataAs(LLCGetHdrOData,Message)->SSAP = SSAP;
                ODataAs(LLCGetHdrOData,Message)->DSAP = DSAP;
                ODataAs(LLCGetHdrOData,Message)->Hdr = Hdr;

                Message->mDataSize = AddressSize;
                Message->mDataPtr = Address;

                if(SyncFlag)                                    /* If Async */
                {
                        Send(Message);
                }
                else                                            /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        if(Message)
                        {
                                Result = Message->mStatus;
                                FreeMsg(Message);
                        }
                }
        }
        else
                Result = CannotGetMessageBufferErr;
        return(Result);
}

word SNAP_Transmit( int                 SyncFlag,
                word                    InfoLen,
                word                    Options,
                byte                    FrameType,
                LANHdr                  *Hdr,
                long                    BufferSize,
                char                    *Buffer
                )
{
        message *Message;
        word Result = 0;
        long ID;
        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = SNAPTransmit;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalSNAPTID;
                ODataAs(SNAPTxOData,Message)->InfoLen = InfoLen;  /* This is ignored */
                ODataAs(SNAPTxOData,Message)->Options = Options;
                ODataAs(SNAPTxOData,Message)->FrameType = FrameType;
                ODataAs(SNAPTxOData,Message)->Hdr = Hdr;

                Message->mDataSize = BufferSize;
                Message->mDataPtr = Buffer;

                if(SyncFlag)                            /* If Async */
                {
                        Send(Message);
                }
                else                                    /* Sync */
                {
                        Send(Message);
                        Message = Receive(ID,0,0,0);
                        Result = Message->mStatus;
                        FreeMsg(Message);
                }
        }
        else
                Result = CannotGetMessageBufferErr;
        return(Result);
}

word SNAP_Receive(    int             SyncFlag,
                word                  PDRefNum,
                word                  Options,
                LANHdr                *Hdr,
                long                  BufferSize,
                char                  *Buffer)
{
        message *Message;
        word Result = 0;
        long ID;

        if(Message = GetMsg())
        {
                ID = Message->mId;
                Message->mCode = SNAPReceive;
                Message->mPriority = GlobalLLCMessagePriority;
                Message->mTo = GlobalSNAPTID;
                ODataAs(SNAPReceiveOData,Message)->PDRefNum = PDRefNum;
                ODataAs(SNAPReceiveOData,Message)->Options = Options;
                ODataAs(SNAPReceiveOData,Message)->Hdr = Hdr;
```

```
            Message->mDataSize = BufferSize;
            Message->mDataPtr = Buffer;

            if(SyncFlag)                                /* If Async */
            {
                    Send(Message);
            }
    }
    else
            Result = CannotGetMessageBufferErr;

    return(Result);
}
```