

AppleBus:

An Introduction

Apple Computer's mission is to provide people with inexpensive, easy-to-use computers. Apple now introduces AppleBus, a simple interconnect system that delivers all the benefits of multi-user communication and shared resources at a fraction of the cost usually associated with these features.

AppleBus adds a powerful new dimension to Apple's computer products by allowing them to communicate freely for cooperative purposes without in any way diminishing the independence of the individual user.

AppleBus functions in three major configurations: as a small area interconnect system, as a tributary to a larger network, and in its most elemental form, as a peripheral bus between an Apple computer and its dedicated attached devices.

As a stand-alone work area network, AppleBus provides communications and resource sharing among up to 32 computers, disks, printers, modems, and other peripherals. More important, AppleBus supports a wide variety of forthcoming services such as shared files, electronic mail, and communication with computers and resources on other networks. Further, AppleBus has been designed to allow its incorporation, through bridging devices, into larger networks.

When functioning as a peripheral bus, Apple Bus in effect implements the concept of slots by providing a single serial bus for the connection of a variety of peripheral devices.

To achieve this range of flexibility, Apple has designed highly reliable, inexpensive hardware, and a sophisticated set of communications protocols. This hardware/software package, together with the computers, cables and connectors, shared resource managers (servers), and specialized applications software, will grow into a complete line of products offered both by Apple and third parties. (see figure 1)

Design Goals

Applebus is designed to fit the needs of the individual user who wishes to extend his maximum number of directly-attached peripherals, and provide for the sharing requirements of a small cluster of computers. Careful attention has also been paid to businesses, where work area networks feed large backbone networks and corporate communication implies a mixture of networks and technologies.

Work Stations

Servers

Peripherals

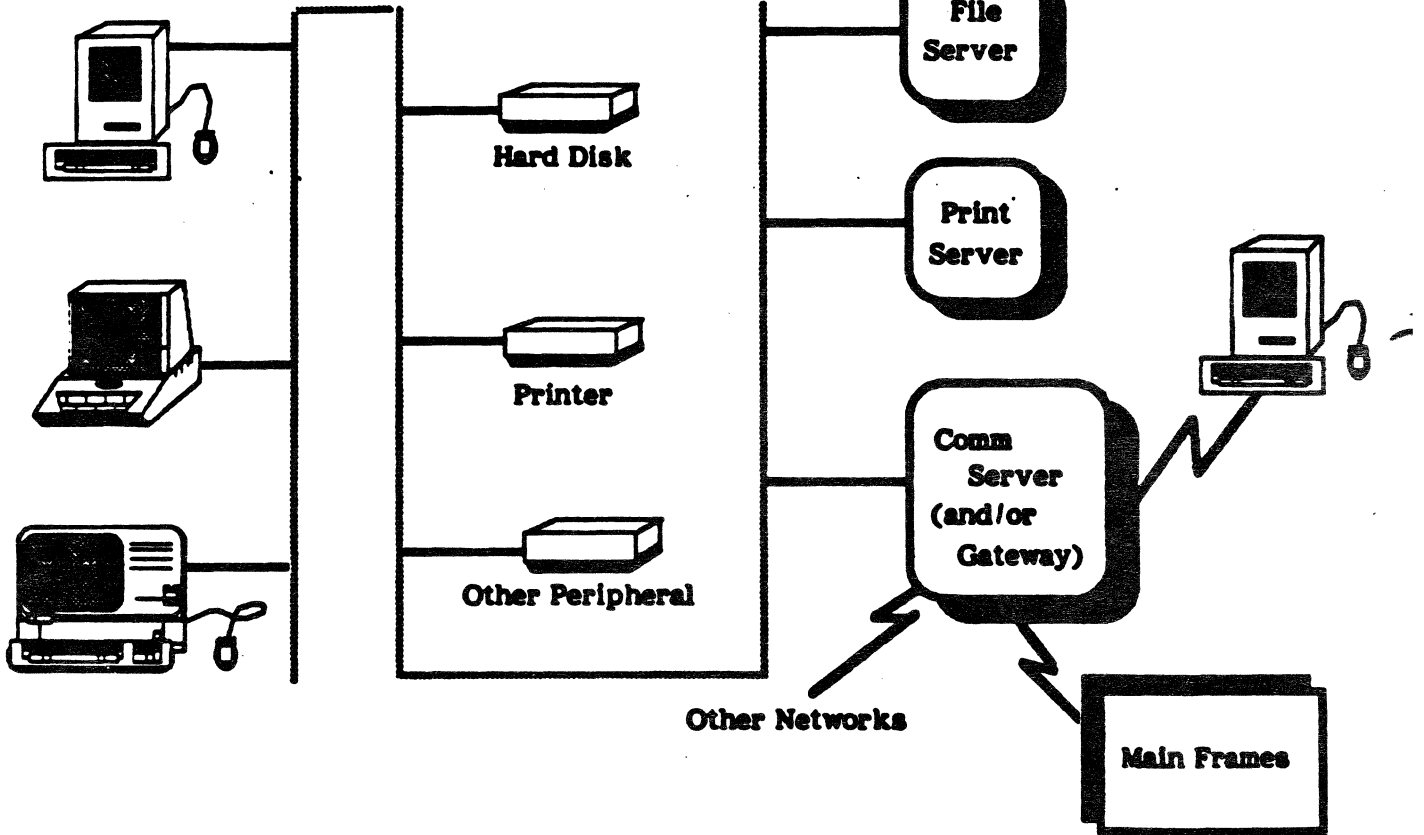


Figure 1. AppleBus System

1. Low-cost

AppleBus is built into Macintosh and Lisa. The shielded, twisted-pair cable used to form the bus, and its associated cables and connectors, bring the actual cost to only about \$25 per connect for any configuration.

2. Easy to Install

Electrically, AppleBus is designed with passive (transformer) coupling of devices to a trunk cable, with simple, self-terminating miniature DIN connectors. No special installation is necessary, and computers, peripherals, or servers may be easily relocated or added. The actual connection of a device to AppleBus is simpler than hooking stereo speakers to a "hi-fi" system.

3. Easy to Extend

The AppleBus Protocols Package supports a range of configurations from simple dedicated device attachment through internetworking. With the same software, AppleBus may be extended through AppleBus bridges to other AppleBuses, and through gateways to communications services and other networks (local or long-haul).

4. Open Systems Architecture

Apple has developed a suite of protocols for AppleBus which provide functionality corresponding to the various layers of the ISO (International Standards Organization) OSI (Open Systems Interconnection) reference model. Protocols at the ISO-OSI layers 1 through 5 (Physical, Data Link, Network, Transport, and Session) form the core of the AppleBus Protocol Architecture. The use of the ISO-OSI layering allows developers to design new functions and applications for AppleBus. Throughout the development of AppleBus, Apple will make all technical documentation completely public.

Technical Specifications

Hardware

AppleBus at the Physical level has a bus topology consisting of a linear trunk cable with intervening connection modules to which nodes attach via a short drop cable. Electrically and mechanically, AppleBus is a multi-drop, balanced, transformer isolated serial communications system for up to 32 nodes. The raw data rate is 230.4K baud over a distance of 300 meters.

The serial hardware driver in Macintosh and Lisa (Zilog 8530 Serial Communications Controller) is programmed to use SDLC frame format, and FM 0 modulation. FM 0 is a bit encoding technique that provides self-clocking. Balanced signalling is achieved using RS-422 driver and receiver ICs in each of the attached devices. The transformer provides ground isolation as well as protection from static discharge. Since a node is passively connected to the trunk cable via a drop cable, a device may fail without disturbing communications. Nodes may be added and removed from the bus with only minor disruption of service.

Since end-user installation is assumed, assembled cables will be sold in standard 2, 6, and 15 meter lengths with molded miniature DIN connectors. The trunk cable connects to an AppleBus connection module. The connection module is a small plastic case (3" x 2" x 1") containing a transformer, resistive and capacitive circuits for noise immunity, and two 3-pin miniature DIN connectors with terminating switches to a 100 ohm terminating resistor. Attached to this plastic case is an 18 inch "drop cable" which terminates at the node with either a DB-9 or DB-25 connector. The trunk cable is shielded, twisted pair cable (Belden 9272 or equivalent).

For detailed hardware information, refer to AppleBus Electrical/Mechanical Specification, March, 1984 (Apple Doc No. 062-0190-A).

Protocols and Software

Underlying all use of AppleBus is a specific set of protocols, or communication "rules". These protocols correspond to the ISO-OSI layers 1 through 5 (Physical, Data Link, Network, Transport, and Session).

While Apple recommends the use of these protocols, communication over AppleBus is not dependent on their exclusive use. All protocol implementations are layered, functionally distinct entities, which allow easy access and addition of alternative protocols. This implies that a software developer could, for instance, leverage upon the physical and data link layers to build a different protocol architecture.

For further details refer to the document, The AppleBus Protocol Architecture, An Introduction, and individual documents of each of the AppleBus protocols. Individual protocol specifications will be made available as they are finished. (see figure 2)

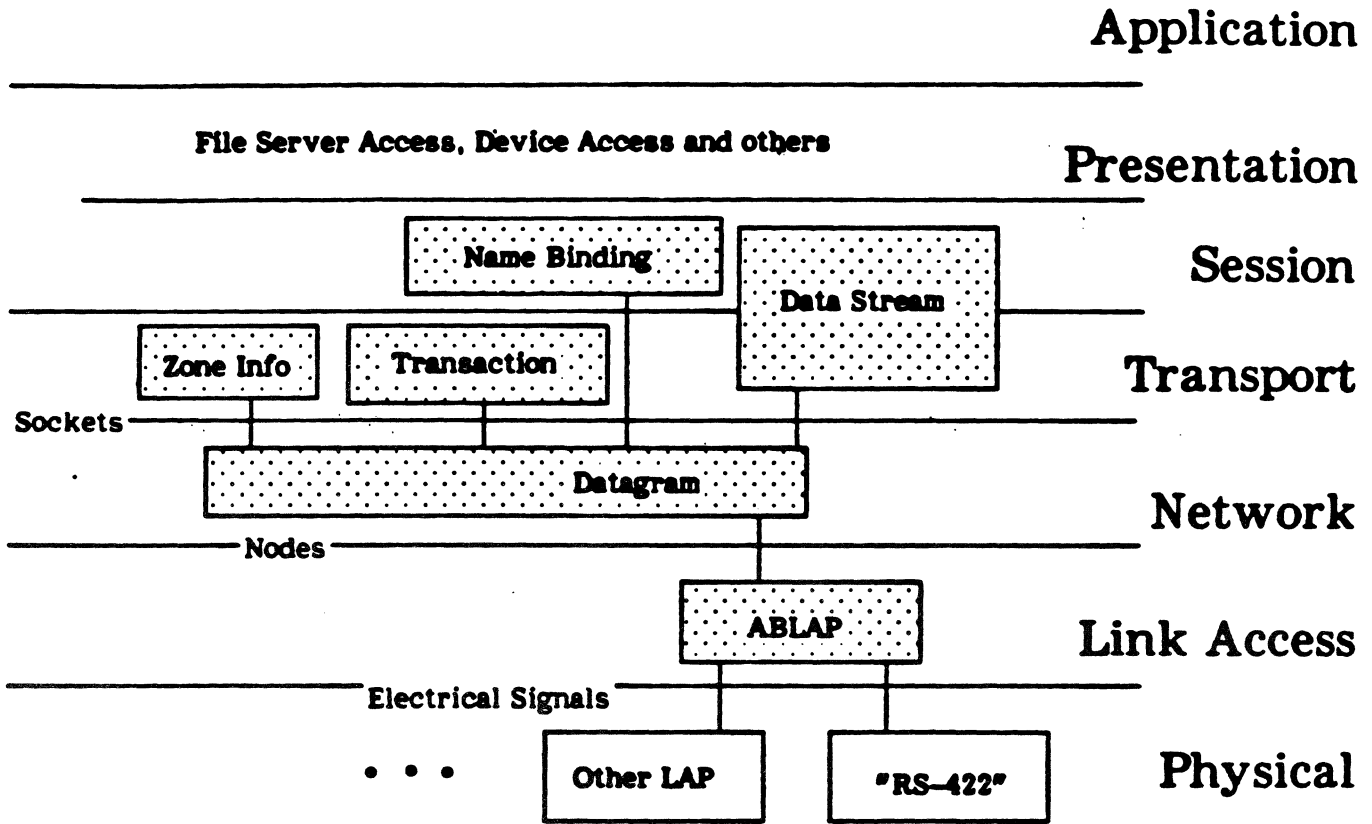


Figure 2. AppleBus Protocol Architecture

Briefly, the architecture includes :

Physical Layer

The Physical layer encompasses the physical and electrical characteristics of AppleBus as described earlier in the Hardware section.

Link Access Protocol (ABLAP)

The AppleBus Link Access Protocol corresponds to the Data Link layer of the ISO-OSI reference model. The protocol, which must be common to all systems on the bus, provides "best effort" delivery of information between nodes. The ABLAP procedure manages the encapsulation and decapsulation of data in a *frame* and then provides access to the bus for transmission and reception of frames. The format and interpretation of the data encapsulated by the frame is left to higher level protocols. Specifically, the LAP manages :

1. Bus Access Control
2. Node ID Assignment
3. Node-to-Node Addressing Mechanism
4. Packet Length and Integrity

1. Bus Access Control

All AppleBus nodes compete for use of the bus. It is the function of the LAP to resolve this contention, and provide fair access for all nodes. AppleBus uses an access discipline that is termed *Carrier Sense, Multiple Access with Collision Avoidance (CSMA/CA)*. Carrier Sense means that a sending node first "senses" the line, and defers to an ongoing transmission. *Collision avoidance* means that the protocol attempts to minimize *collisions*. A collision occurs when two (or more) stations transmit frames at the same time. In the AppleBus CSMA/CA technique, all transmitters wait until the line is idle for a minimum time plus an additional time as determined by a generation of a (pseudo-) random number whose range is adjusted based upon perceived bus traffic. AppleBus hardware does not allow the actual detection of collisions.

2. Node ID Assignment

AppleBus uses an 8-bit identification number for node addressing on the bus. AppleBus nodes have no permanent address or identification number configured into them. When a node is activated on an AppleBus, it makes a guess at its node number, either by extracting its number from some form of long-term (parameter/disk) memory, or by generating a random number if it does not have non-volatile memory. The node then sends a special AppleBus ABLAP frame to its own address, and waits for an acknowledgement. If it receives an acknowledgement, it knows that the chosen node number is already in use, and repeats the process with a different guess until it succeeds.

3. Node-to-Node Addressing Mechanism

ABLAP is ultimately responsible for the destination and source node addresses encoded in the header portion of the outgoing frame. In addition to the ability to direct packets to a specific node on the AppleBus, ABLAP allows the broadcasting of packets to all nodes on the bus using address "Hexadecimal FF".

4. Frame Length and Integrity

Frame length may vary arbitrarily with a stipulated maximum. Individual messages are assembled into frames with an appended 16-bit Frame Check Sequence (FCS). The FCS is calculated using the standard CRC-16 polynomial. Packets received with an invalid FCS are discarded by the receiving node. Furthermore, ABLAP may exploit a higher level length field to verify packet length.

Datagram Delivery Protocol (DDP)

At the next higher level of the architecture (the Network layer of the ISO-OSI model) is the Datagram Delivery Protocol (DDP). While the ABLAP protocol provides "best effort" node-to-node delivery of packets on a single AppleBus, the Datagram Delivery Protocol extends this mechanism to socket-to-socket delivery over an entire AppleBus internet. *Sockets* are logical entities within the nodes of a network. *Datagrams*, packets that are delivered as single, independent entities, are exchanged between sockets. Therefore, sockets may be visualized as simply the sources and destinations of datagrams. Sockets are said to be owned by *socket clients*, which are typically processes

(or functions within processes), implemented in software in the node.

AppleBus *internets* are formed by interconnecting up to 254 AppleBuses with intelligent nodes referred to as *bridges* or *internet routers*. [Bridges should not be confused with gateways. A *gateway* refers to a node separating and managing communication between different types of networks.]

DDP uses the concept of network and socket numbers to provide an internet-wide addressing mechanism adequate for the delivery of datagrams among all sockets of an internet.

AppleBus Transaction Protocol (ATP)

The AppleBus architecture includes several protocols at the next higher level above DDP (the transport layer of the ISO-OSI model). These protocols add different levels or types of functionality to the underlying datagram delivery service. An important example is the AppleBus Transaction Protocol which adds a measure of reliability by providing a loss-free transaction service between sockets. This allows exchanges between two socket clients (a requestor and a responder) in which one client requests the other to perform a particular task and report the result. The interaction, consisting of a request and a response, is termed a *transaction*. ATP allows for a transaction response that may be as large as 12 datagrams. ATP delivers each response message, correctly reassembling its component packets without gaps in the proper sequential order.

This model fulfills the transport requirements for a wide variety of higher level services from simple devices through servers and other general networking services.

Data Stream Protocol (DSP)

This protocol is being proposed by Apple to provide a reliable two-way data stream service (also known as byte streams, virtual circuits, etc.) between a pair of sockets on an internet. The envisioned service provides for flow control and recovery from the familiar situations of packet loss, and duplicate or out-of-sequence packet reception. Such a service, layered on top of the DDP, would satisfy the transport service needs of applications exchanging sequences of data such as screen contents, keyboard input, etc.

Name Binding Protocol (NBP)

This session level protocol allows network users to employ character string names for socket clients and network services. The basic function of NBP is the translation of a character string name into the internet socket address of the corresponding network entity. The protocol does not mandate the use of name servers.

Routing Table Maintenance Protocol (RTMP)

This transport level protocol allows bridges/internet routers to dynamically discover routes to the different networks (AppleBuses) in an internet. This information is maintained only in bridges. Ordinary AppleBus nodes employ a simple subset of RTMP for the purpose of discovering the number of the AppleBus to which they are connected and the node identification number of a bridge on that AppleBus.

Higher Level Protocols

Apple is developing other higher level protocols that exploit the services of the core protocols to build specific services. An important instance of this is the description of a general model of providing host to peripheral device access through AppleBus.

Other examples are protocols corresponding to the various servers under development at Apple for file sharing, network printing, etc.

These protocols will be published in due course for unrestricted use by developers.

Summary



The basic AppleBus product, incorporating hardware, protocols, and software described here, exceeds the requirements for its design center. In test configurations, AppleBus has performed with stability at offered loads of up to 100% channel capacity. It is without question the low-cost, highly extendable interconnect scheme Apple intended to design.

The extremely low cost of AppleBus opens the path to experimentation by users and developers. Apple expects to recreate with AppleBus an explosion of useful new products, as already seen with the Apple II and Macintosh. In addition to foreseeable, well-known network services, this activity will lead to unforeseen uses of the evolving technology of networking. It is, after all, the promise of increased human potential and productivity that underlies the concepts of personal computers and the networking of personal computers.

REV	ZONE	ECO #	REVISION	APPD	DATE
		J186	INITIAL RELEASE		

AppleBus Electrical/Mechanical Specification

DRAWING NUMBER

 METRIC <small>DIMENSIONS ARE IN MILLIMETERS TOLERANCES</small> <small>X _____ ANGLES _____</small>		 apple computer inc.	
<small>MATERIAL</small> <small>FINISH</small>		NOTICE OF PROPRIETARY PROPERTY <small>THE INFORMATION CONTAINED HEREIN IS THE PROPRIETARY PROPERTY OF APPLE COMPUTER, INC. THE POSSESSOR AGREES TO THE FOLLOWING</small> <small>1. TO MAINTAIN THIS DOCUMENT IN CONFIDENCE</small> <small>2. NOT TO REPRODUCE OR COPY IT</small> <small>3. NOT TO REVEAL OR PUBLISH IT IN WHOLE OR PART</small>	
<small>DRPT</small> <small>ENG. APPVL</small> <small>RELEASE</small>	<small>DRPT CH</small> <small>MFG. APPVL</small> <small>MFG. DIV</small>	<small>TITLE</small> Specification, AppleBus Electrical/Mechanical	
<small>DRAWING NO.</small>	<small>SCALE</small>	<small>SIZE</small> A	<small>DRAWING NUMBER</small> 062-0190-B
		<small>INT</small> 1/10	

1.0 Introduction

AppleBus is a serial interconnection system for all Apple Computers and several future peripheral products. It provides a common, convenient, inexpensive, expansion capability for computer products. In summary, AppleBus is a multi-drop, balanced, transformer isolated, serial communication system designed to connect 32 devices at 230.4 Kbaud over a total distance of up to 300 meters.

2.0 References

AppleBus Link Access Protocol, specification number 062-0214.

EIA Standard RS-422. Electrical Characteristics of Balanced Voltage Digital Interface Circuits.

Fairchild Semiconductor "Interface, Line Drivers and Receivers", 1975.

3.0 Summary of Features and Performance

- 32 total devices
- Shielded, twisted pair, connectorized interconnection; easy user configuration, maximum total cable length of 300 meters
- 230.4 Kbaud communication, SDLC frame format, FMO modulation
- Balanced signalling using standard RS-422 driver (26LS30) and receiver (26LS32) I.C.'s
- Transformer isolation for excellent noise and static discharge immunity
- Self-configuring, no user switches or action to identify devices
- Passive drops, devices may be disconnected, and one may fail without disturbing communication



apple computer inc.

SIZE
A

DRAWING NUMBER
062-0190-B

SCALE:

SHEET 2 OF 10

4.0 Signalling

- 4.1 AppleBus devices send and receive data over a single pair of wires connected to each device. Two connectors on each device allow the user to easily connect devices together by means of a simple cable (see Section 5.0 Interconnection). Balanced, transformer coupled signalling is used to reduce both RFI and noise susceptibility.
- 4.2 Each device has a single driver and receiver. The receivers are always connected to the system and pass all AppleBus data to the controller. Only one driver at a time is enabled. Software controls which device may transmit data (see Protocol Specification). The driver and receiver descriptions and specifications are given in the electrical section, 6.0.
- 4.3 The signal on AppleBus is encoded with FMO (bi-phase space). This insures that clock information can be recovered at the receivers. In FMO, a transition occurs at the beginning of every bit cell. A "0" is represented by an additional transition at the center of the bit cell, and a "1" is represented by no transition at the center of the cell.

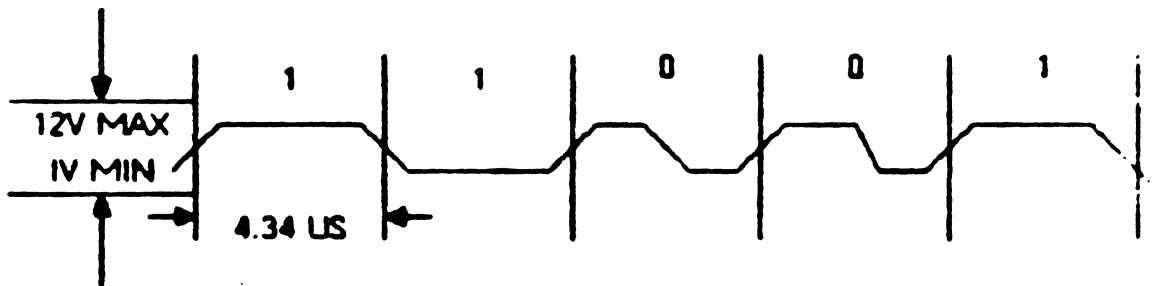


Figure 4.1 FMO Coding (230.4 K baud +/- 1%)

- 4.4 Synchronization time for the clock recovery system is provided by the transmission of 14 consecutive 1 bits directly preceding the data frame. Frame format is covered in the Protocol Specification.



apple computer inc.

SIZE
A

DRAWING NUMBER
062-0190-B

SCALE:

SHEET 3 OF 10

5.0 Interconnection

5.1 Applebus devices are connected to the AppleBus by a connection module which contains a transformer, DB-9 connector at the end of an 460 millimeter cable, and two 3-pin miniature DIN connectors as shown in Figure 5.1. Each 3-pin connector has a coupled switch. If both connectors are used, the switches are open, but if one of the connectors is not used, a 100 ohm termination resistor (R2) is connected across the line. The use of the connection module allows the AppleBus device to be removed from the system by disconnecting it from the module without disturbing the operation of the bus. R3 and R4 increase the noise immunity of the receivers, while R5 and C1 isolate the frame grounds of the AppleBus devices and prevent ground loop currents. The resistor (R1) provides static drain for the cable shield to ground.

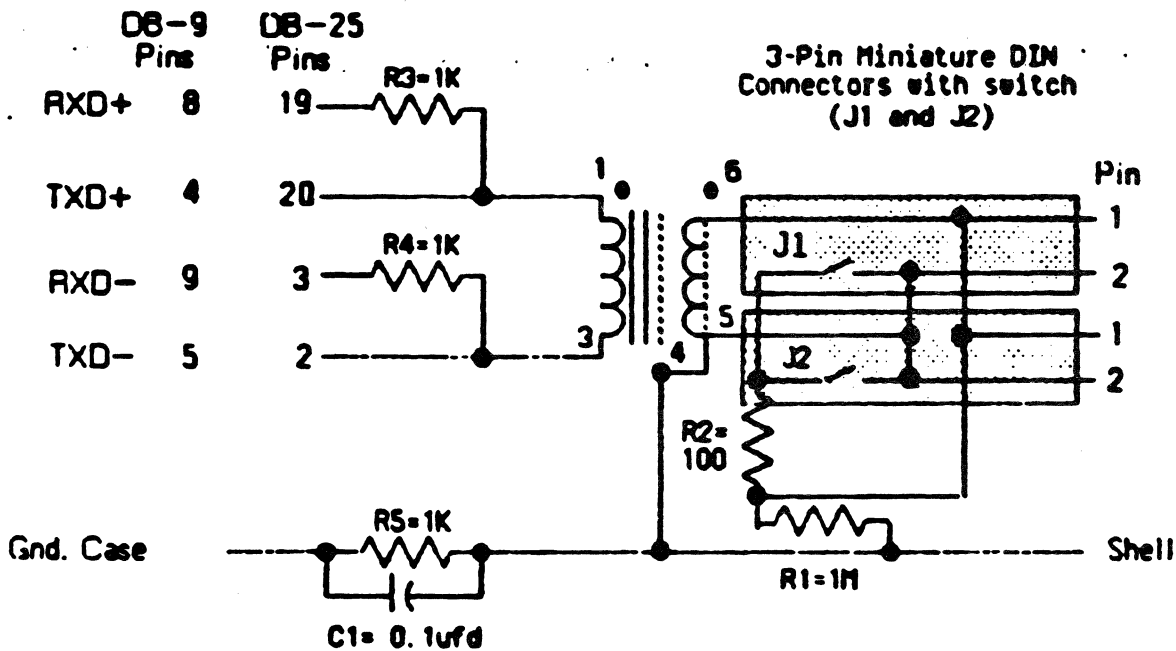


Figure 5.1 AppleBus Connection Module

 apple computer inc.

SIZE
A

DRAWING NUMBER
062-0190-B

SCALE:

SHEET 4 OF 10

5.2 Individual AppleBus devices are connected together by a twisted, shielded pair cable. The cable is available in standard lengths with connectors on each end, or in 150 meter bulk rolls. The maximum length of cable for the bus is limited to a total of 300 meters.

Cable Specification

Conductors:	22 AWG stranded 17 ohm per 300 meters
Shield:	85% coverage braid
Impedance:	78 ohm
Capacitance:	68 pF per meter
Rise Time:	175 ns 0 to 50% at 300 meters
Diameter:	4.7 mm (0.185 inches) maximum



apple computer inc.

SIZE
A

DRAWING NUMBER
062-0190-B

SCALE:

SHEET 5 OF 10

5.3 The AppleBus connector is a miniature 3 pin circular connector similar to Hosiden connector number TCP8030-01-010.

Pin 1 AppleBus - Plus
Pin 2 AppleBus - Minus
Pin 3 Unused
Shell Shield



Figure 5.2 Connector Pin Assignment (looking into connector)

5.4 The interconnecting cable is wired "one-to-one" as shown below.

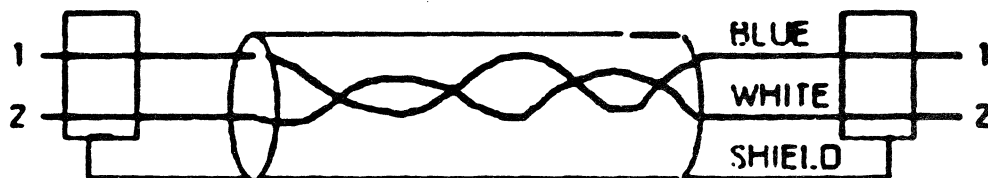


Figure 5.3 Interconnecting Cable Connection

6.0 Electrical Specification

6.1 The recommended driver is the 26LS30 used with both +5V and -5V as power supplies, and the mode control connected to give differential outputs (Mode voltage low).

The outputs of the driver are coupled to the connector through a "deglitch network" which consists of a T-network of two 20 to 30 ohm resistors and a 150 to 300 picofarad capacitor. Use of the network gives two major advantages. The first is that the high frequency components of the signal are attenuated both going onto and off of the cable thus reducing RFI and also static susceptibility. The second advantage is that at least one driver on the network can fail without causing the network to fail (as long as it fails in one state and doesn't broadcast trash).

Those who wish to use standard RS-422 specified components (power supplies of +5 volts and ground) may do so if the deglitch circuits are not used, but should be aware that the driver must drive a cable impedance of 39 ohms (middle of a 78 ohm cable).

6.2 The receiver is the 26LS32. Both inputs of the receiver are connected through deglitch networks.

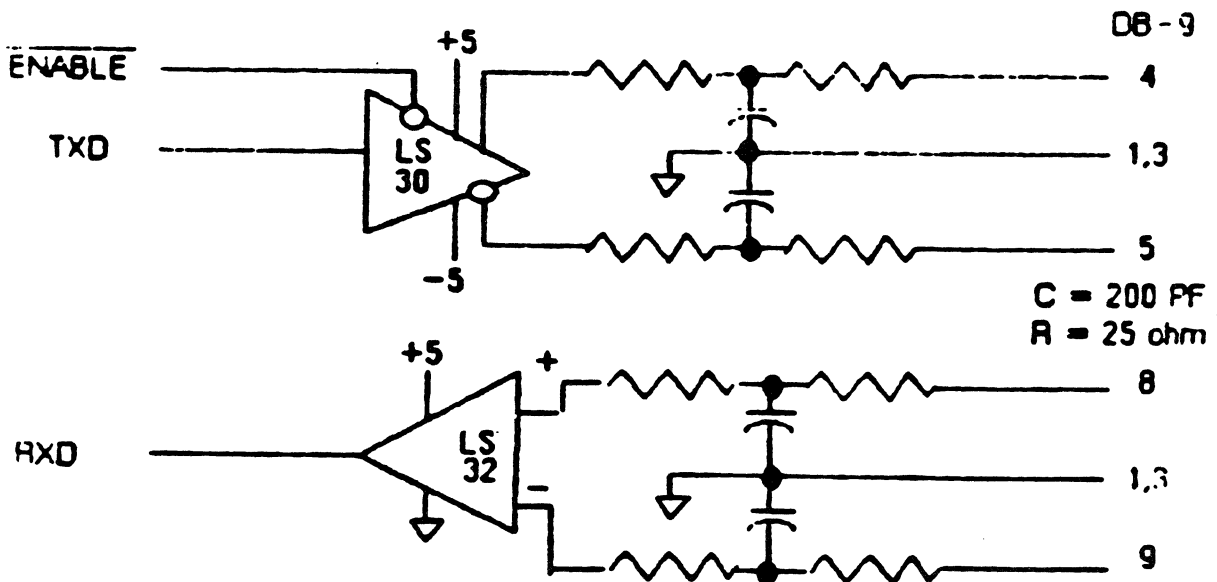


Figure 6.1 Driver and Receiver Connection



apple computer inc.

SIZE
A

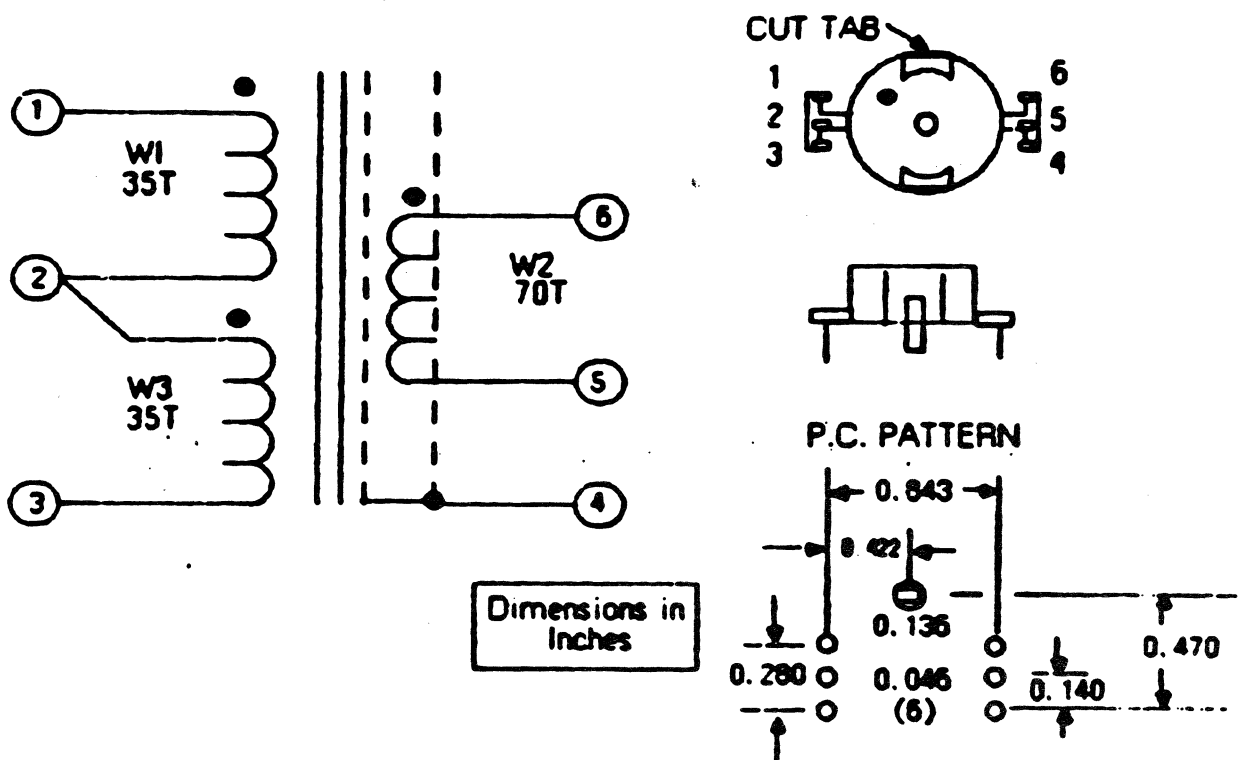
DRAWING NUMBER
062-0190-B

SCALE:

SHEET 7 OF 10

6.3 The transformer is a 1:1 turns ratio transformer with tight coupling between primary and secondary, and electrostatic shielding to give excellent common mode isolation.

The primary is wound as two windings of #32 AWG wire in series with one wound below the secondary and one above it.



Dimensions in Inches

Specifications:

- Core Material:** Siemens B65651-K000-R030 (or equivalent)
- Bobbin:** Siemens B65652-PC1,L (or equivalent)
- Retaining Clip:** Siemens B65653-T (or equivalent)
- Magnetizing Inductance:** 20 mH minimum
- Leakage Inductance:** 15 uH max
- Capacitance:** 5 pF max (primary to secondary with electrostatic shield and core guarded)

Figure 6.2 Transformer Specification

- 6.4 Each end of the cable must be terminated. The AppleBus connection module is designed such that a 100 ohm resistor is connected across the line if one of the two connectors is not used. A 100 ohm resistor is used even though the characteristic impedance of the line is 78 ohms because it gives adequate termination and minimizes resistive losses.
- 6.5 The cable shield should be grounded to Earth ground (frame ground) at each AppleBus device. This ground is necessary to prevent excessive RFI. A resistor and capacitor are included in each connection module to isolate the cable shield from the ground connection at 60 Hz while offering a low impedance connection to high frequency noise. This connection scheme allows ground connection for RFI purposes without risking high currents flowing in the cable due to differing ground potentials. In the U.S., the green wire available in most outlets is suitable for frame grounding.



apple computer inc.

SIZE
A

DRAWING NUMBER
062-0190-B

SCALE:

SHEET 9 OF 10

7.0 Documentation

The following is a list of all AppleBus hardware drawings.

Drawing Description	Drawing Number
AppleBus Connection Module Cover (Top)	815-0839
AppleBus Connection Module Cover (Bottom)	815-0838
AppleBus Cable Extender (Coupler)	519-0300
AppleBus Cable with Molded Plugs	590-025X
DB-9 Connection Cable Assembly	590-0254
DB-25 Connection Cable Assembly	590-0253
AppleBus Board Film Artwork	820-0135
AppleBus Connection Module FCC Label	825-1008



apple computer inc.

SIZE
A

DRAWING NUMBER
062-0190-B

SCALE:

SHEET 10 OF 10

NOTES!

1. MATERIAL:
CYCOLAC KJW
#85452

2. COLOR:
- APPLE FOG PER COLOR
CONTROL PANEL, APPLE
#912-0030 (REF.
SPEC. #080-0009).

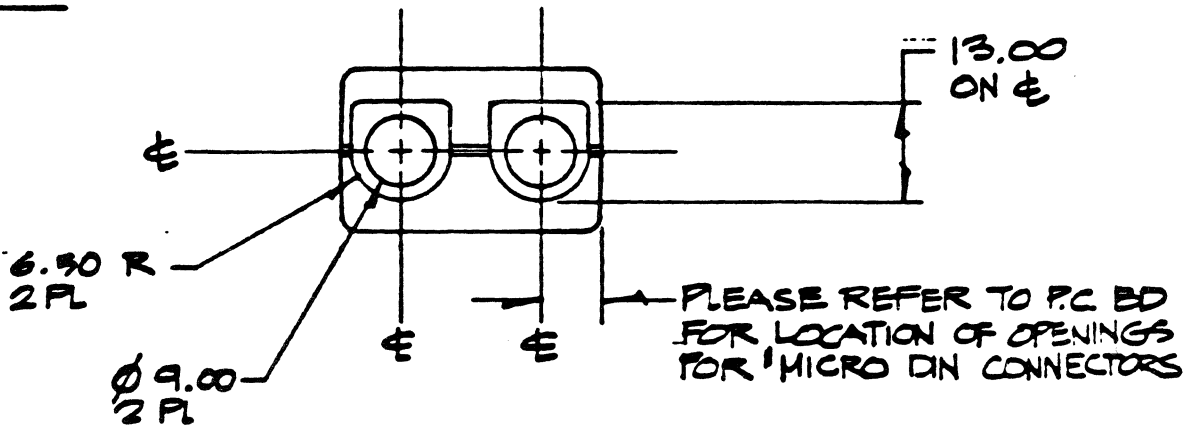
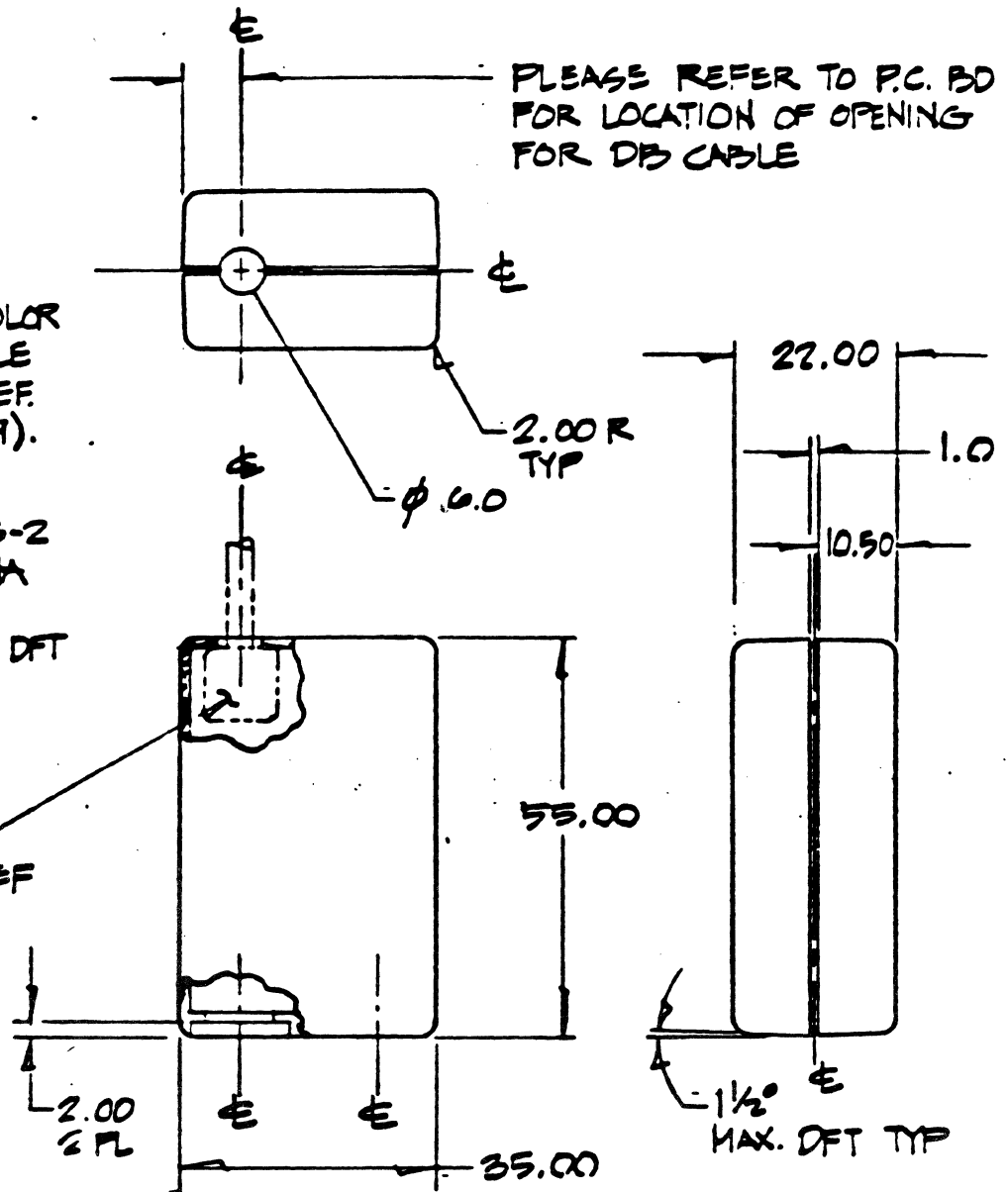
3. TEXTURE:
APPLE COMPUTER 6-2
- TANABAWA HAKKOSHA
CO. LTD., TOKYO.
.008" DEPTH, 1 1/2" DFT
20-25 u


PLEASE REFER TO P.C. BD
FOR LOCATION OF OPENING
FOR DB CABLE

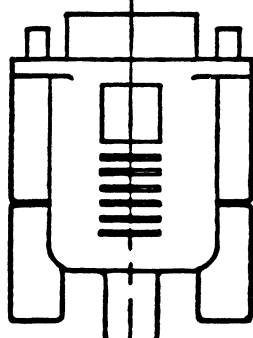
INTERNAL
STRAIN RELIEF

APPLE BUS
HOUSING

FIG. A2



 apple computer inc.	SIZE A	DRAWING NUMBER 062-0190-A
	SCALE: FULL	SHEET 10 OF 12

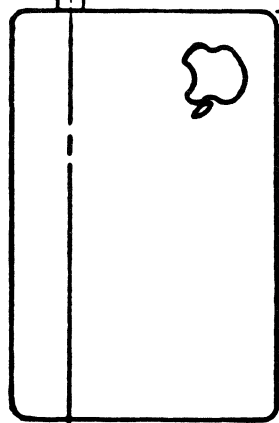


DB-9 CONNECTOR
REFER TO DWG
519-0280

304.80

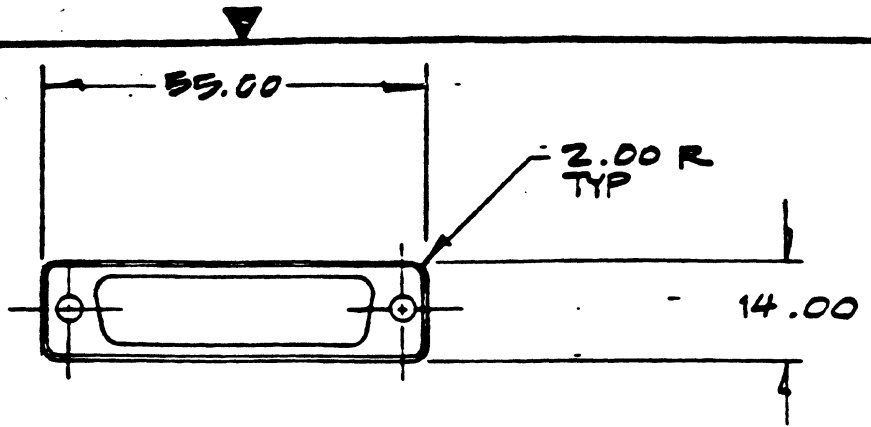
ϕ 3.81

FIG. A1



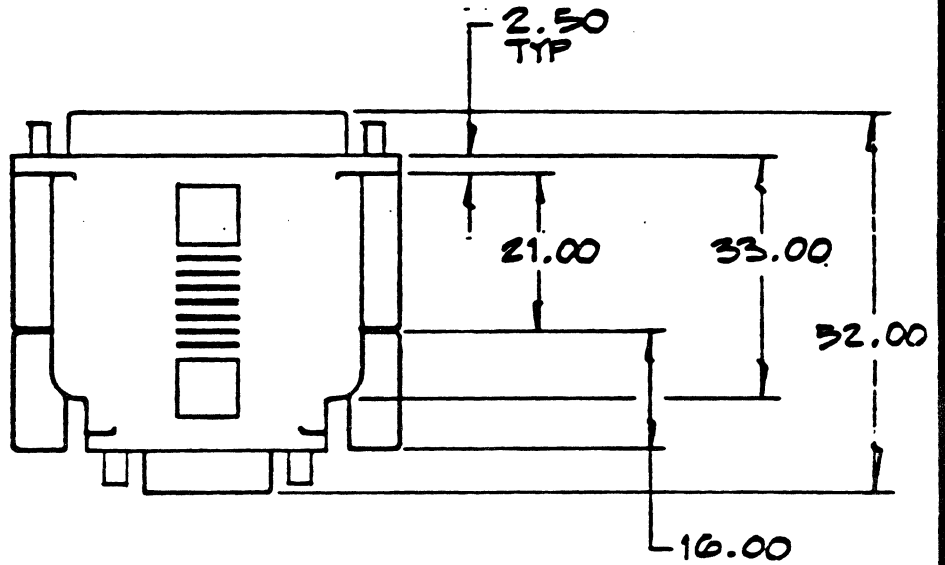
APPLE BUS HOUSING
REFER TO FIG. A2

DB 25

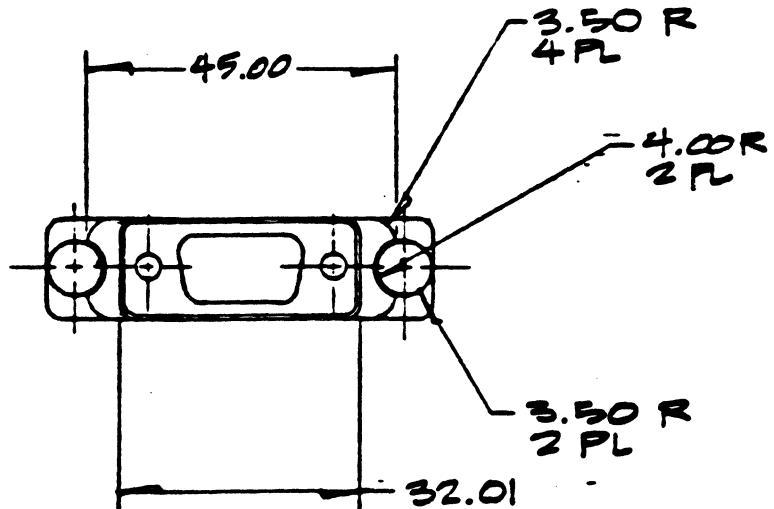


DB 9/DB 25
CONNECTOR

FIG. A3



DB 9



apple computer inc.

SIZE
A

DRAWING NUMBER
062-0190-A

SCALE: FULL

SHEET 10 OF 13





REV.	ZONE	ECO #	REVISION	APPD	DATE
A		J187	INITIAL RELEASE		

AppleBus Transformer Specification

DRAWING NUMBER
062-0215-B

SHEET
1 / 5

 METRIC DIMENSIONS ARE IN MILLIMETERS TOLERANCES X - _____ YZ - _____ ANGLES _____		 apple computer inc.	
MATERIAL _____ FINISH _____ DRFT _____ DRFT CA _____ FIN. APPL _____ NPL. APPL _____ RELEASE _____ WH. LNS _____		NOTICE OF PROPRIETARY PROPERTY THE INFORMATION CONTAINED HEREIN IS THE PROPRIETARY PROPERTY OF APPLE COMPUTER, INC. THE POSSESSOR AGREES TO THE FOLLOWING: (1) TO MAINTAIN THIS DOCUMENT IN CONFIDENCE (2) NOT TO REPRODUCE OR COPY IT (3) NOT TO REVEAL OR PUBLISH IT IN WHOLE OR PART	
DRAWN BY G. Crow		TITLE AppleBus Transformer	
SCALE _____		SIZE A	DRAWING NUMBER 062-0215-B
		SHEET 1 / 5	

1.0 Description

The AppleBus transformer is used in the AppleBus connection module to give isolation between the AppleBus cable and the units which are connected to the cable.

The transformer is a 1:1 turns ratio transformer with tight coupling between primary and secondary, and electrostatic shielding to give excellent common mode isolation.

The primary is wound as two windings of #32 AWG wire in series with one wound below the secondary and one above it. The secondary is a single continuous winding of #32 wire.

2.0 Environmental

The transformer shall operate properly and meet its specifications under the following environmental conditions:

Operating Temperature: 0 to 70 degrees C

Storage Temperature: -40 to 70 degrees C

Relative Humidity: 5 to 95%

Altitude: 0 to 4572 meters

In addition, the transformer must meet the Apple Computer shock and vibration requirements while mounted on a printed circuit board and tested to Apple specification number 062-0086.

3.0 Mechanical Strength and Workmanship

The transformer winding assembly, pins, mounting plate, core, and clamp shall be securely mounted and rigid with respect to each other.

The pins must be easily solderable; solderability must meet EIA specification RS-178B.

All components shall be free of undue mechanical stresses.

4.0 Identification Markings

The transformer shall be marked with the manufacturer's name or identification number, date of manufacture, country of origin, manufacturer's part number, and the Apple part number. The markings shall be clearly legible after typical assembly, wave-soldering, and cleaning processes, and after exposure to the above mentioned environmental extremes. The markings may be placed in any convenient and visible location on the transformer.



apple computer inc.

SIZE
A

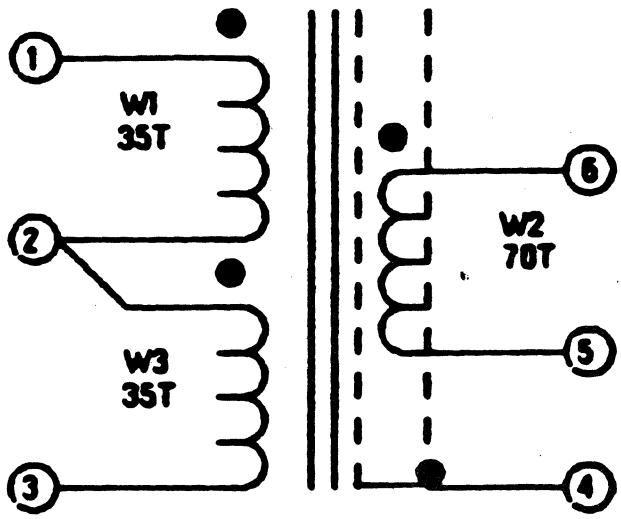
DRAWING NUMBER
062-0215-B

SCALE:

SHEET 2 OF 5

5.0 Electrical Specifications

Core Material: Siemens B65651-K000-R030 (or equivalent)
Bobbin: Siemens B65652-PC1,L (or equivalent)
Retaining Clip: Siemens B65653-T (or equivalent)
Magnetizing Inductance: 20 mH minimum
Leakage Inductance: 15 uH max
Capacitance: 5 pF max (primary to secondary with electrostatic shield and core guarded)
Turns Ratio: 1:1 accurate to the nearest 1/2 turn
Hi Pot: From W2 to core, shield and primary 1000 VDC for one minute with no significant leakage current.



All wire #32 AWG.

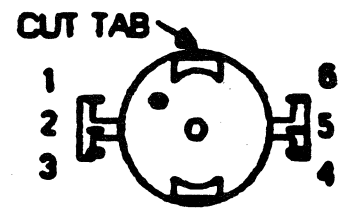
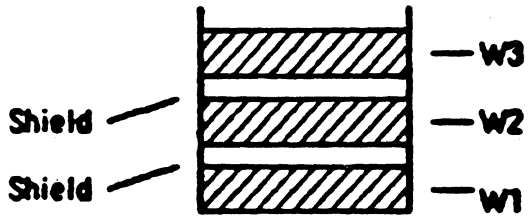
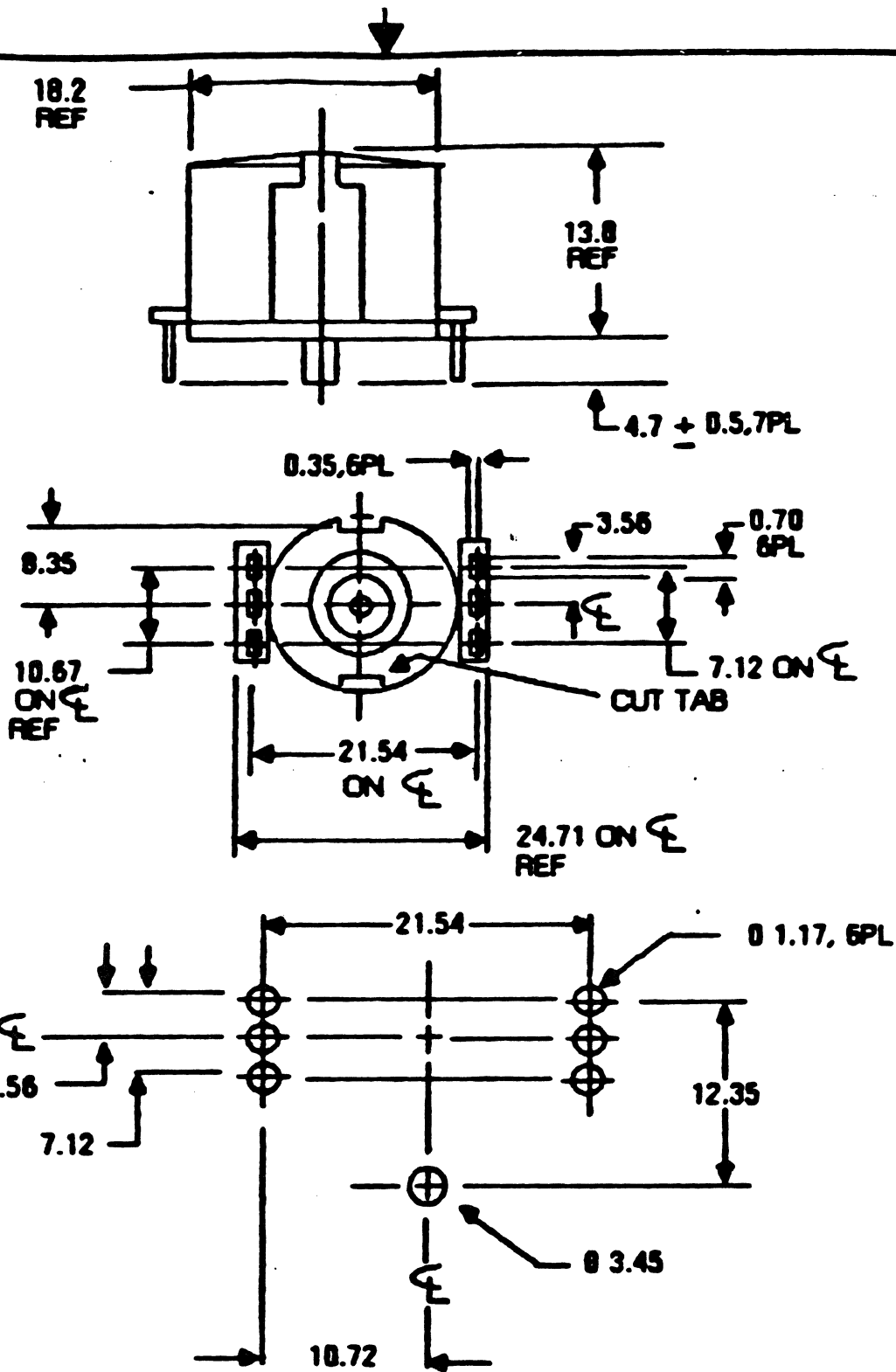


Figure 1.8 Schematic and Build Detail



P C Hole Pattern

Notes:

1. Transformer to fit easily into P.C. Hole Pattern
2. Unless noted, 0.XX +/- 0.20 mm

Figure 2.0 Mechanical Specification



apple computer inc.

SIZE
A

DRAWING NUMBER
062-0215-B

SCALE:

SHEET 5 OF 5

**AppleBus
Link Access Protocol
Specification
Version 1.0**

**Apple Computer, Inc.
May, 1984**

DISCLAIMER

This specification includes subject matter which may relate to patents of Apple Computer. No license under any such patents is granted by implication or otherwise as a result of publication of this specification. Applicable licenses may be obtained from Apple Computer.

This specification is furnished by Apple Computer for informational purposes only. Apple Computer does not warrant or represent that this specification or any products made in conformance with it work in the intended manner or to be compatible with other Apple products or network components. Nor does Apple Computer assume responsibility for any errors that the specification may contain, or have any liabilities or obligations for damages (including but not limited to special, indirect, or consequential damages) arising out of or in connection with the use of this specification in any way. Apple Computer may follow or deviate from the specification without notice at any time.

This specification or anything made from it is not represented or warranted to be free from infringements or patents of third persons.

PREFACE

This document contains the specification of the AppleBus Link Access Protocol, a network access protocol developed by Apple Computer. This document is intended as a design reference document. The document's structure is based on The Ethernet Data Link Layer and Physical Layer Specifications, Version 2.0, November, 1982. The AppleBus Link Access Protocol itself is significantly different from the Ethernet Data Link Protocol. An Overview of AppleBus and the The AppleBus Protocol Architecture, An Introduction are separate documents that include introductory descriptions of the AppleBus Link Access Protocol.

This specification contains four sections that describe the AppleBus Link Access Protocol from several aspects. Sections I, and II provide an overview of AppleBus and the functional model of the AppleBus architecture. Section III describes elements of the AppleBus Link Access Protocol, including the frame format, frame transmission, and frame reception. Section IV contains a procedural model of the Link Access Protocol as well as the interfaces to the Physical and Network Layers.

Readers looking for an overall understanding of the the AppleBus Link Access Protocol should read the first two sections. Implementors of the Link Access Protocol will find that sections III and IV contain the details of the specification.

Notation

Three different numerical representations will be used in this document:

%bb...b for Binary numbers, (b = 0 or 1)

\$hh for hexadecimal numbers, (h = 0, 1, ... , 9, A, B, ... , F)

numbers nn...n with no prefix for decimal numbers (n = 0, 1, ..., 9).

Nomenclature

The term *node* is used throughout this specification to refer to any computer, peripheral device, server, etc., that is attached to and communicates over the shared medium of AppleBus. Other commonly-used synonymous terms are *station* and *host*.

Implementation

This document is a specification of the function, form, and processes of the AppleBus Link Access Protocol. Although frequent reference is made to aspects of specific implementations of the protocol on various Apple computers, this is merely for the purpose of illustration. There is no implication that the mentioned hardware or software configurations are required by this protocol specification.

Table of Contents

Preface	iii
I. Introduction	1
I.1. Technical Specifications	2
I.2. Hardware	2
I.3. Protocols and Software	3
II. Functional Model of the AppleBus Link Access Protocol	6
II.1. Layering	6
II.2. Physical Layer	6
II.3. Data Link Layer	7
III. AppleBus Link Access Protocol	10
III.1. Dynamic Node Number Assignment	10
III.2. Frame Format	11
III.3. Frame Transmission	14
III.4. Frame Reception	17
IV. The Link Access Procedural Model	27
IV.1. Global Constants, Types and Variables	28
IV.2. Hardware Interface Declarations	29
IV.3. Interface Procedures and Functions	30

IV.4. InitializeLAP Procedure	31
IV.5. AcquireAddress Procedure	31
IV.6. TransmitPacket Function	32
IV.7. TransmitLinkMgmt Function	33
IV.8. TransmitFrame Procedure.	36
IV.9. ReceivePacket Procedure.	37
IV.10. ReceiveLinkMgmt Function	38
IV.11. ReceiveFrame Function	39
IV.12. Miscellaneous Procedures and Functions	42

Appendix 1. Serial Communications Controller (SCC) Implementation	43
---	----

Figures

Figure 1. AppleBus Protocol Architecture	3
Figure 2. FM-0 Encoding	9
Figure 3. AppleBus Architecture to Implementation Mapping	13
Figure 4. AppleBus Link Access Protocol Frame Format	16
Figure 5. ABLAP Timing Diagram	22

I. Introduction

AppleBus is a system designed to function in three major configurations: as a small work-area interconnect system, as a tributary to larger networks, and in its most elemental form, as a peripheral bus between an Apple computer and its dedicated peripheral devices.

As a stand-alone work-area network, AppleBus provides communication and resource sharing among up to 32 computers, servers, disks, printers, modems, and other peripherals. More importantly, AppleBus supports a wide variety of forthcoming services such as shared files, electronic mail, and communication with computers and resources on other networks. Further, AppleBus has been designed to allow its incorporation, through bridging devices, into larger networks.

When functioning as a peripheral bus, AppleBus in effect implements the concept of slots by providing an external serial bus for the connection of a variety of peripheral devices.

Applebus is designed to fit the needs of the individual user who wishes to extend his maximum number of directly-attached peripherals, and provide for the sharing requirements of a small cluster of computers. Careful attention has also been paid to businesses, where work-area networks feed large backbone networks and corporate communication implies a mixture of networks and technologies. The design goals of AppleBus can be summarized as: low cost, easy to install, easy to extend, and an open system architecture.

AppleBus access hardware is built into Macintosh and Lisa systems, and is essentially free for these systems. The shielded, twisted-pair cable used to form the bus, and its associated cables and connectors, bring the actual per-node connect cost to only about \$25.

Electrically, AppleBus has been designed with passive (transformer) coupling of nodes to a trunk cable, with simple, self-terminating miniature DIN connectors. No special installation is necessary, and computers, peripherals, or servers may be easily relocated or added. The actual connection of a device to AppleBus is simpler than hooking stereo speakers to a "hi-fi" system.

The AppleBus Protocols Package supports a range of configurations from simple dedicated device attachment through internetworking. With the same software, AppleBus may be extended through AppleBus bridges to other AppleBuses, and through gateways to communications services and other networks (local or long-haul).

Apple has developed a suite of protocols for AppleBus which provide functionality corresponding to the various layers of the International Standards Organization (ISO) Open Systems Interconnection (OSI) reference model (see Figure 1). Protocols at the ISO-OSI layers 1 through 5 (Physical, Data Link, Network, Transport, and Session) form the core of the AppleBus Protocol Architecture. The use of layering allows developers to design new functions and applications for AppleBus. Throughout the development of AppleBus, Apple will make all technical documentation completely public.

1.1 Technical Specifications

Physical Layer:

Data Rate: 230.4 Kilobits per second

Maximum Node Separation: 300 meters

Maximum Number of Nodes: 32

Medium: Shielded, twisted-pair cable, baseband signalling

Topology: Linear, non-branching bus

Data Link Layer:

Link Control Procedure: Fully distributed peer protocol, with statistical contention resolution (CSMA/CA)

Message Protocol: Variable size frames, "best effort" delivery.

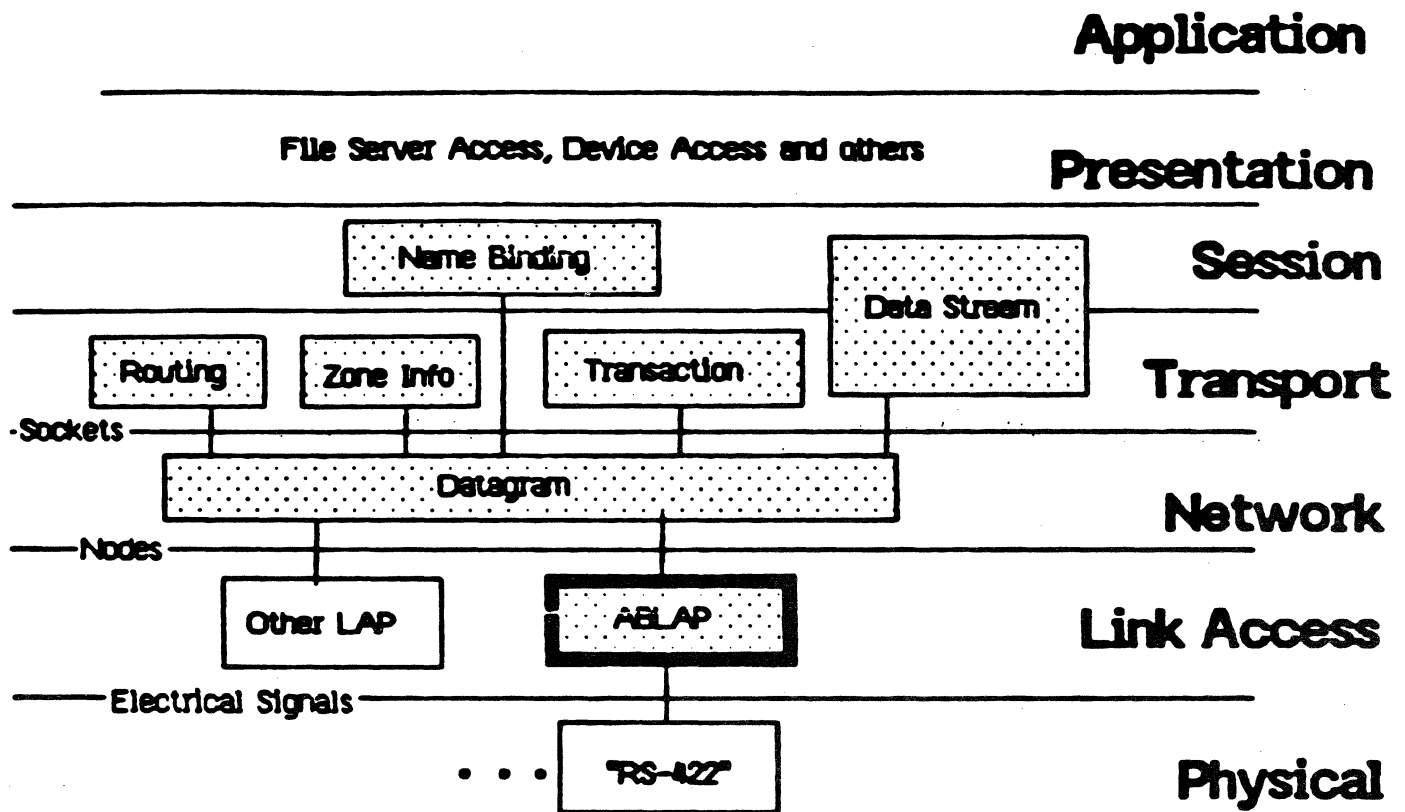


Figure 1. AppleBus Protocol Architecture

L2 Hardware

AppleBus at the Physical level has a bus topology consisting of a linear trunk cable with intervening connection modules to which nodes attach via a short drop cable. Electrically and mechanically, AppleBus is a multi-drop, balanced, transformer-isolated, serial communications system for up to 32 nodes. The raw data rate is 230.4 Kilobits per second over a distance of 300 meters.

Data packets are transmitted as frames using an HDLC/SDLC format; the frames bits are encoded using FM-0. FM-0 is a bit encoding technique that provides self-clocking. Balanced signalling is achieved using RS-422 driver and receiver ICs in each of the attached devices. The transformer provides ground isolation as well as protection from static discharge. Since a node is passively connected to the trunk cable via a drop cable, a device may fail without disturbing communications. Nodes may be added and removed from the bus with only minor disruption of service.

Since end-user installation is assumed, assembled cables will be sold in standard 2, 6, and 15 meter lengths with molded miniature DIN connectors. The trunk cable connects to an AppleBus connection module. The connection module is a small plastic case (3"x 2"x 1") containing a transformer, resistive and capacitive circuits for noise immunity, and two 3-pin miniature DIN connectors with switches to a 100 ohm terminating resistor. Attached to this plastic case is an 18-inch "drop cable" which terminates at the node with either a DB-9 or DB-25 connector. The trunk cable is shielded, twisted-pair cable (Belden 9272 or equivalent).

For detailed hardware information, refer to AppleBus Electrical/Mechanical Specification, March, 1984 (Apple Doc No. 062-0190-A).

L3. Protocols and Software

Underlying the use of AppleBus is a specific set of protocols, or communication "rules".

These protocols correspond to the ISO-OSI layers 1 through 5 (Physical, Data Link, Network, Transport, and Session).

While Apple recommends the use of these protocols, communication over AppleBus is not dependent on their exclusive use. The protocols are layered, functionally distinct entities, which allow easy access and addition of alternative protocols. This implies that a software developer could, for instance, leverage upon the physical and data link layers to build a different protocol architecture.

For further details refer to the document, The AppleBus Protocol Architecture, An Introduction, and individual documents on each of the AppleBus protocols. Protocol specifications will be made available as they are completed.

Briefly, the three layers of the architecture that are relevant to this specification include:

Physical Layer

The Physical layer encompasses the physical and electrical characteristics of AppleBus as described earlier in the Hardware section.

Link Access Protocol

The *AppleBus Link Access Protocol* (ABLAP) corresponds to the Data Link layer of the ISO-OSI reference model. This protocol, which must be common to all systems on the bus, provides "best effort" delivery of data packets between nodes. The ABLAP manages the encapsulation and decapsulation of data in an *ABLAP Frame* and then provides access to the bus for transmission and reception of frames. The format and interpretation of the data encapsulated by the frame is left to higher level protocols. Detailed description of ABLAP function are in section II of this document. In summary, the LAP manages:

1. Bus Access Control

All AppleBus nodes compete for use of the bus. It is the function of ABLAP to resolve this contention, and provide fair access for all nodes. AppleBus uses an access discipline that is termed *Carrier Sense, Multiple Access with Collision Avoidance (CSMA/CA)*. Carrier Sense means that a sending node first "senses" the line, and defers

to ongoing transmission. *Collision avoidance* means that the protocol attempts to minimize collisions. A *collision* occurs when two (or more) nodes transmit frames at the same time. In the AppleBus CSMA/CA technique, all transmitters wait until the line is idle for a minimum time plus an additional time as determined by a generation of a (pseudo-) random number whose range is adjusted based upon perceived bus traffic. ABLAP does not require hardware for the actual detection of collisions.

2. Node ID Assignment

AppleBus uses an 8-bit identification number (*node address* or *node ID*) for node addressing on the bus. AppleBus nodes have no permanent address or identification number configured into them. When a node is activated on an AppleBus, it makes a guess at its node number, either by extracting its number from some form of long-term (parameter/disk) memory, or by generating a random number. The node then sends a special ABLAP enquiry frame to this guessed address, and waits for an acknowledgement. If it receives an acknowledgement, it knows that the chosen node number is already in use, and repeats the process with a different guess until it succeeds (no acknowledgement received after repeated transmission of the special enquiry frame).

3. Node-to-Node Addressing Mechanism

ABLAP is ultimately responsible for the destination and source node addresses encoded in the header portion of a frame. In addition to the ability to direct packets to a specific node on the AppleBus, ABLAP allows the broadcasting of packets to all nodes on the bus using destination address SFF.

4. Frame Length and Integrity

Frame length may vary arbitrarily with a stipulated maximum of 600 data bytes. This data is assembled into frames with a three-byte ABLAP header and an appended 16-bit *Frame Check Sequence* (FCS). The FCS is calculated using the standard CRC-CCITT polynomial. Packets received with an invalid FCS are discarded by the receiving node.

This document provides a precise specification of the Data Link layer of the AppleBus architecture. It does not specify the higher level protocols required to complete a network architecture, nor does it detail the physical interface to the communication

medium. The higher level protocols are expected to provide error recovery, flow control, and end-to-end transport of user data, as well as support for application services.

The primary objective of this specification is to ensure compatibility among various AppleBus implementations. While different higher level protocols will be implemented on AppleBus, the enforcement of a unique Link Access Protocol will guarantee standard access to the network that is both fair and stable.

II. Functional Model of the AppleBus Link Access Protocol

The AppleBus Link Access Protocol performs the functions corresponding to those of the Data Link Layer of the ISO-OSI reference model. ABLAP also performs the additional function of dynamic node address assignment. The following functional model describes the general concepts of the protocol.

II.1. Layering

One of the major architectural features of AppleBus is the use of layering. Layering lends clarity to the design by defining and interrelating functionally distinct protocols. Each layer of the protocol architecture builds upon the services of lower layers and itself provides services to higher layers. In the specific case of the AppleBus Link Access Protocol, the Physical layer consists of the transmission medium and the associated electrical signalling facilities used by ABLAP to transfer data frames. ABLAP in turn provides the Network layer protocols (the *ABLAP client*) with a "best effort", error-free, node-to-node delivery of data packets.

II.2. Physical Layer

The Physical layer of AppleBus is a 230.4 Kilobit per second channel through a shielded, twisted pair cable. The channel is driven in conformance with EIA Standard RS-422 balanced voltage specifications. The Physical layer performs the functions of bit encoding/decoding, synchronization, bit transmission/reception, and carrier sense. The layered architecture of AppleBus allows for the Physical layer to be replaced by another medium as long as these functions are provided and the interface between the Physical and Data Link is maintained the same. An explanation of each of the functions performed by the Physical layer is summarized below. The details of the AppleBus Physical layer are in the AppleBus Mechanical/Electrical Specification referred to earlier.

Bit Encoding/Decoding

Bits are encoded using a self-clocking technique known as FM-0 (bi-phase space). In FM-0, each bit cell (nominally, 4.34 μ sec) contains a transition at its end, thus providing timing information. Zeros (0's) are encoded by adding an additional transition at mid-cell (see Figure 2).

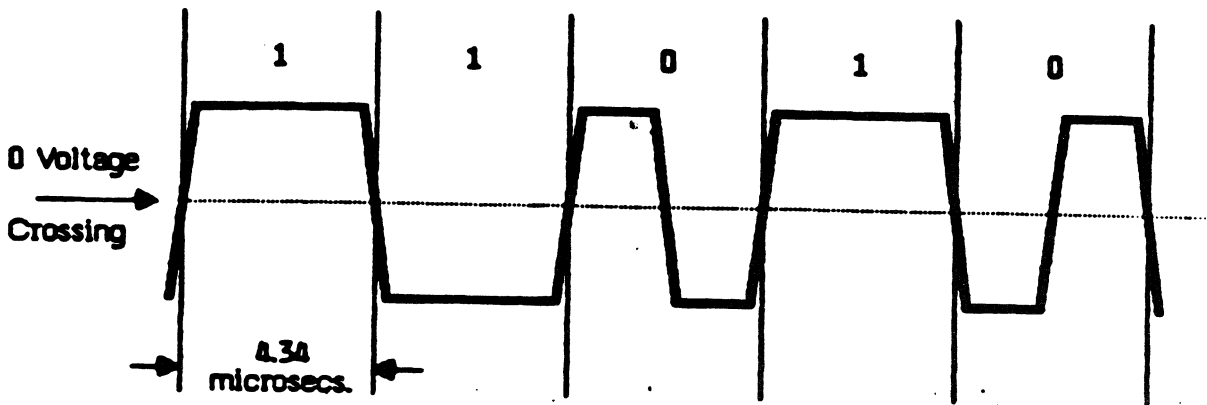


Figure 2 FM-0 Encoding

Synchronization

At the beginning of a frame ABLAP transmits a *synchronization pulse*. This is a transition on the bus, followed by an idle period greater than two bit-times. The synchronization pulse is obtained by momentarily enabling the line driver for at least one bit time, before disabling it. This causes a transition which will be taken as a clock by all receivers on the AppleBus. However, since it is followed by an idle period of sufficient length, a Missing Clock is detected by the receivers. The Missing Clock allows transmitters to synchronize their access to the line.

Signal Transmission and Reception

The use of the EIA RS-422 signalling standard for transmission and reception over AppleBus provides significantly higher data rates over longer distances than with the EIA RS-232C standard. AppleBus uses differential, balanced voltage signalling at 230.4 Kilobits per second over a maximum distance of 300 meters. The balanced

configuration provides better isolation from ground noise currents and is not susceptible to fluctuating voltage potentials between system grounds or common mode electromagnetic interference (EMI).

Carrier Sense

The Physical layer provides an indication to the Link Access Protocol when activity is sensed on the cable. For instance, on the Macintosh, carrier is sensed when the start of a valid frame is detected by the Zilog™ 8530 Serial Communications Controller (*Hunt Bit* equals zero).

II.3. Data Link Layer

The AppleBus Link Access Protocol is the key mechanism allowing AppleBus nodes to share the communication medium. It performs the functions of uniquely addressing each physical node on AppleBus, encapsulation and decapsulation of user data within an *ABLAP frame*, transparent transmission and reception of the frame through the physical medium, management of access to the physical medium, and detection of transmission errors.

Addressing

Each node on an AppleBus has a unique 8-bit address, which is also known as the *node identifier* (node ID). This address is used to identify the source and destination of each ABLAP frame. Unlike other networks that use fixed, universally unique addresses, AppleBus uses a dynamic node address assignment scheme. Thus, AppleBus node addresses are unique only within their local bus and may change over time.

The 8-bit AppleBus Destination address is used to "filter" frames at the data link layer. Frames whose destination address does not match the receiving node's address are not accepted by that node. The address 255 (SFF) has a special significance. Frames received with destination address equal to this value are accepted by all nodes. This permits the "broadcasting" of data packets to all nodes of an AppleBus. It is important to note that the node address zero (0) is not allowed and is treated as "unknown".

Data Encapsulation/Decapsulation

Data encapsulation/decapsulation is the general method used by the AppleBus Protocol Architecture for moving user information up or down through the protocol layers. A unit of user information is enclosed by a layer-specific header and/or trailer as it moves through each layer from a user application down to the data link layer. In the case of ABLAP, a preamble, an ABLAP header and trailer/postamble enclose the data passed from the next higher layer (network). The corresponding protocol layer at the receiving end examines and removes the layer-specific protocol information as user information moves up through the layers to the receiving application.

Data Transmission/Reception

AppleBus uses a bit-oriented data link protocol for data transmission and reception. Unlike byte-oriented protocols, a bit-oriented protocol permits the use of all bit patterns within the frame. The frame delimiter for ABLAP is called a FLAG. A flag is the distinguished bit sequence %01111110 (\$7E). Typically, flags are generated by (hardware) transmitters at the beginning and end of frames and used by (hardware) receivers to detect frame boundaries.

In order for the data link protocol to transmit all bit patterns within a frame the protocol must insure data transparency. This is provided by a technique known as "bit stuffing". ABLAP guarantees that data sent on the bus contains no sequences of more than five consecutive 1's. It does this by inserting a 0 after each string of five consecutive 1's detected in the user data stream. A receiving ABLAP performs the inverse operation, "stripping" a 0 which follows five consecutive 1's.

Access Management

AppleBus is a shared, communication medium and so requires that access to the medium be managed. This access management is performed by the ABLAP protocol. ABLAP uses a contention resolution method known as *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) for link access management. CSMA/CA uses the Carrier Sense signal provided by the Physical Layer to determine if the channel is busy and, if so, waits for a random period. A frame dialogue is used to implement a collision avoidance scheme. This scheme is discussed in detail in section IV.2.

Error Detection

Because communication channels are error prone, an important function of the data link layer is to assure correct reception of data. ABLAP uses a frame check sequence known as Cyclic Redundancy Check (CRC) to block check the data frame. ABLAP uses the standard CRC-CCITT polynomial to compute the FCS by performing a polynomial division on the data frame. Frames detected with an invalid FCS are discarded by the receiving node.

Implementation Note

It should be noted that FLAG generation and recognition, node address recognition, bit-stuffing (and stripping) and FCS generation/validation will typically be done by interface hardware. Figure 3 illustrates the relationship between the lowest two layers (Physical and Link Access) of the AppleBus Protocol Architecture and a typical implementation (the Macintosh) in hardware and software.

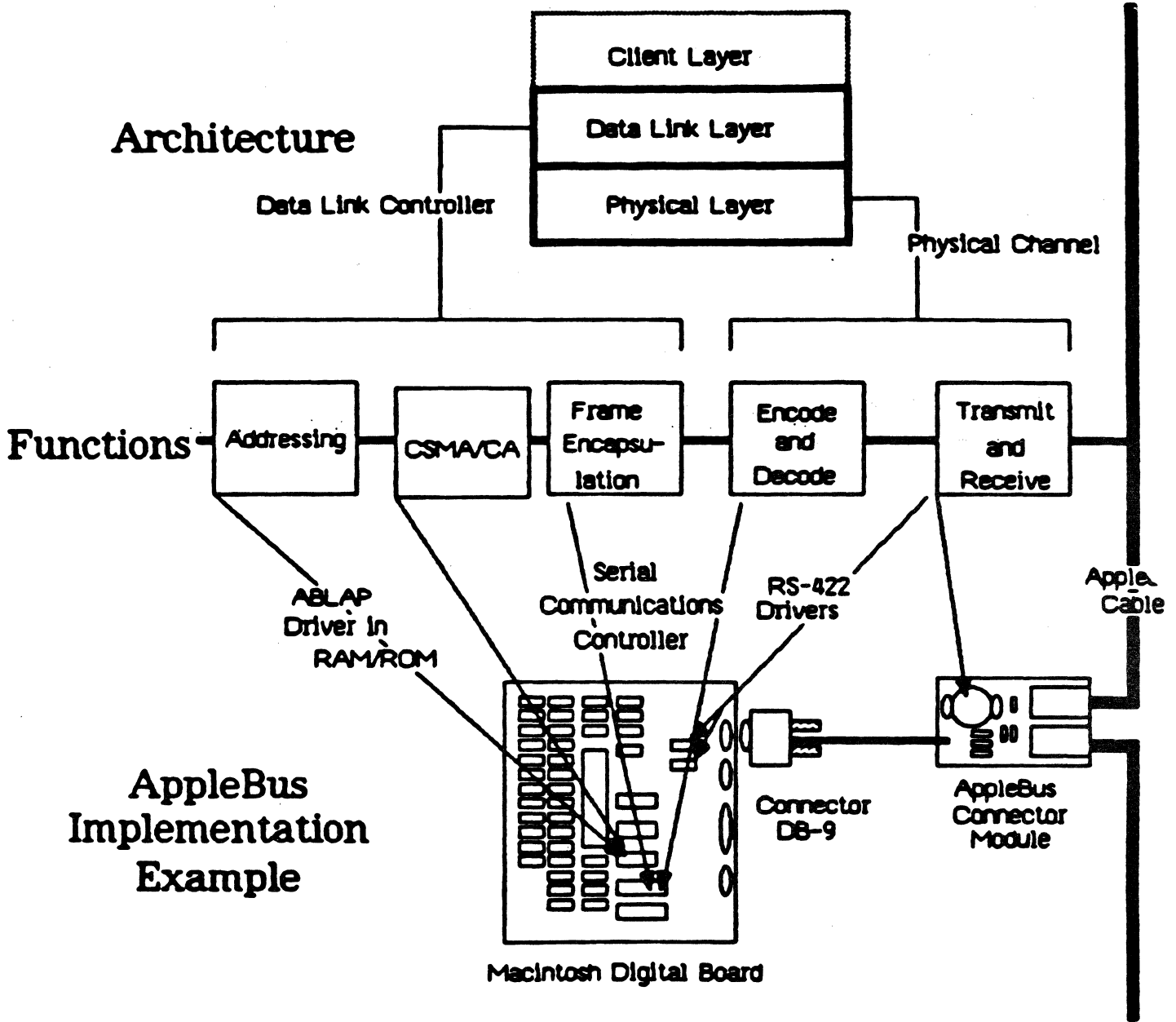


Figure 3 AppleBus Architecture to Implementation Mapping

III. AppleBus Link Access Protocol

The following sections describe the major features of the AppleBus Link Access Protocol. These include a dynamic node number assignment mechanism, the frame format, and frame transmission and reception processes.

III.1. Dynamic Node Number Assignment

AppleBus does not require that a node's address be recorded/configured in it permanently. The primary motivation for this feature is that when a node is moved between AppleBuses, it is possible that its old node number conflicts with one already in use on the new bus. Hence, it will have to acquire a new node number.

The procedure for determining a node number is as follows:

When a node first becomes active, it makes a guess at its node number. This guess may be provided by some sort of long-term memory (e.g., parameter memory) or may be generated each time by some pseudo-random mechanism (e.g., the low-order bits of some timer, etc.).

In any case, the new node must then verify that this guessed number is not already in use by some other node on that AppleBus. This is done by repeatedly sending out a special directed *ABLAP Enquiry control frame* to the guessed node address. If the guessed address is in use, then the corresponding node will, upon receiving the enquiry respond with an *ABLAP Acknowledge control frame*. The reception of this frame indicates that the guessed number is already in use and a new guess must be generated.

Repeated transmission of Enquiry frames is used to account for cases where a node using the guessed address might currently be busy, and thus miss an Enquiry frame.

Node addresses are divided into two classes: *server node addresses* and *user node addresses*. The range of addresses for user nodes is 1..127 (\$01..\$7F); the range for server nodes is 128..254 (\$80..\$FE), with \$00 reserved for "unknown" and \$FF reserved for broadcast.

The address space is split into two parts due to the fact that some nodes may for extended periods of time disable reception from AppleBus (for instance, if they are in the middle of a device-intensive operation, such as disk access or transferring a bitmap document to a laser-printing device). Such a node will not respond to another node's enquiry frames. This could lead to two nodes acquiring the same node address.

It is extremely important that no node acquire the same address as an already-functioning server node since this would disrupt service not only for the conflicting nodes but also for other users of the server. By excluding user (non-server) nodes from the address range used by servers the possibility of user nodes (switched on and off with greater frequency) conflicting with servers is eliminated.

Within the user node address range, verification can be done more expeditiously (fewer retransmissions of the Enquiry frame) to decrease the initialization time for such nodes. A more thorough and extended address assignment scheme is used by servers, i.e they take extra time during the address verification process to ensure that, once chosen, they will be unique on the AppleBus. This is not detrimental to a server's operation since such nodes are rarely switched on or off.

If, during normal operation after acquiring its address, a node detects the use of the same address by some other node (e.g., it receives a Clear-to-Send frame with source address the same as its own address), then the ABLAP driver should set a flag to indicate an address conflict error condition to its client(s). Recovery from this condition, if desired, is the responsibility of higher level protocols.

III.2 Frame Format

The basic unit of ABLAP transmission is a *frame* (see Figure 4). Prior to transmitting the frame, ABLAP sends out a synchronization pulse followed by a *frame preamble* consisting of two or more flag bytes (01111110). The frame is followed by a *frame postamble/trailer* consisting of a flag byte (01111110) and an abort sequence (seven or more 1 bits).

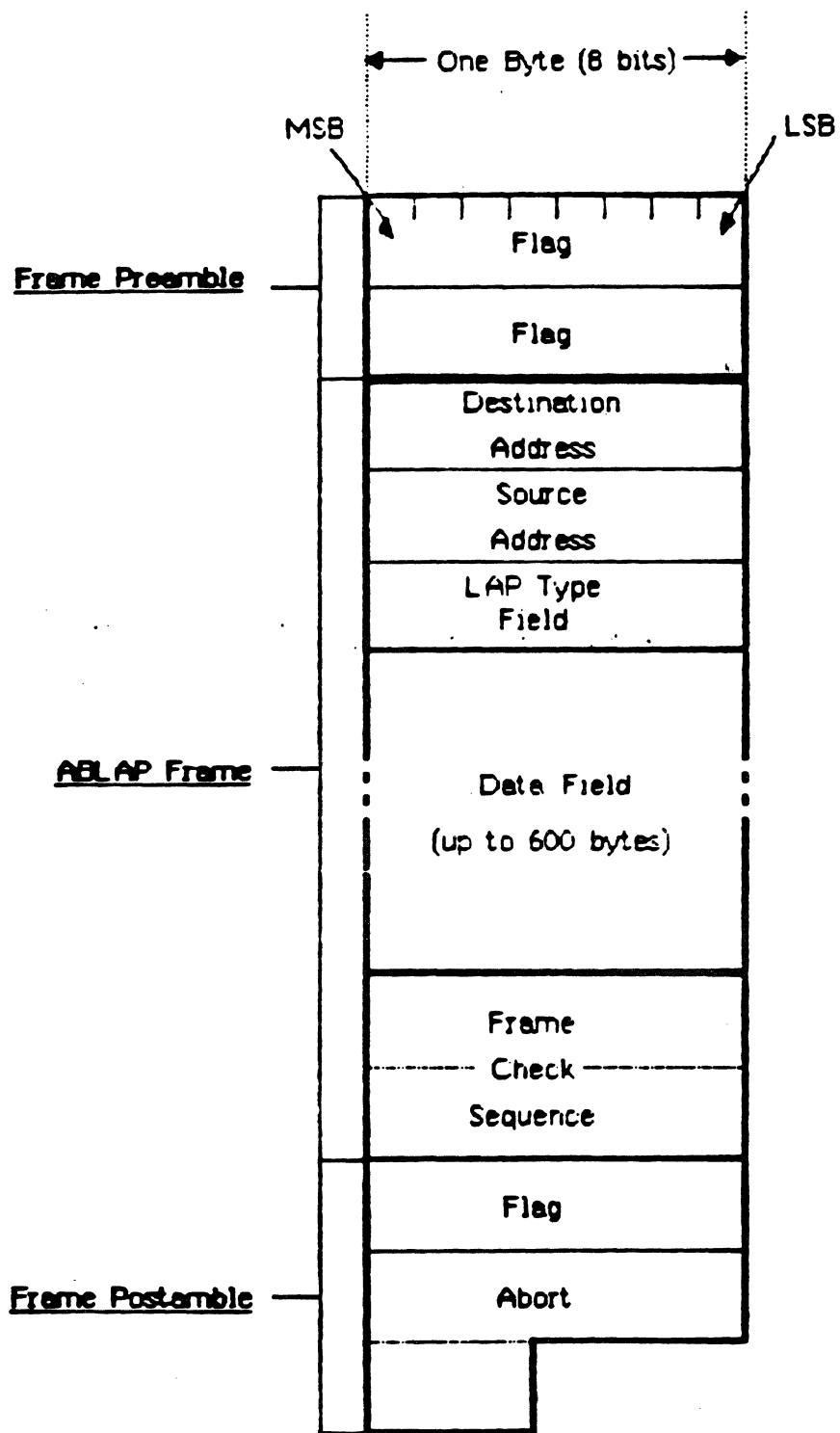


Figure 4 AppleBus Link Access Protocol Frame Format

The ABLAP frame itself consists of a three byte *ABLAP header* (the destination node address, the source node address, a one byte *LAP type field*) followed by a variable length data field (0 to 600 bytes), and a 16-bit frame check sequence.

Destination Node Address

The Destination node address (8 bits) specifies the address of the node for which the frame is intended.

Source Node Address

The Source node address (8 bits) specifies the address of the node sending the frame.

LAP Type Field

The *LAP Type Field* (8 bits) is used to specify the type of the frame. Values 128 to 255 (\$80 to \$FF) are used by ABLAP for its own control frames. ABLAP frames do not include a data field. The different ABLAP control frames are as follows:

- **lapENQ (LAP type = \$81):** This identifies the ABLAP Enquiry frame type used by the dynamic node address assignment mechanism;
- **lapACK (LAP type = \$82):** A node receiving a lapENQ frame must send a lapACK (ABLAP Acknowledge) frame in response;
- **lapRTS (LAP type = \$84):** This frame is used as part of the 3-way handshake to transmit data packets. It notifies the destination that the transmitter wishes to send a packet to it. The destination must respond with either a lapCTS;
- **lapCTS (LAP type = \$85):** This frame is sent by a destination node in response to a lapRTS. It indicates the willingness of the receiver to accept a data packet.

ABLAP control frames (LAP type in the range \$80 to \$FF) received with a type field other than those listed above are simply discarded.

LAP type field values 1 to 127 (\$01 to \$7F) are used in ABLAP frames carrying client data.

In such frames, the type field plays the role of an identifier of the client's protocol type. This allows the simultaneous use of ABLAP by several network level protocols (crucial to maintaining an open systems architecture). The ABLAP implementation in a receiving node uses the value of this type field to determine the client to whom the frame's data must be delivered. This client in turn uses the LAP type field value to decide how to interpret this data (format of the data, higher level protocol header, etc.).

As an example, the Datagram protocol of Figure 1 specified by Apple (called the Datagram Delivery Protocol) uses LAP type field values of 1 and 2. (See the already referenced document on the AppleBus Protocol Architecture).

Data Field

ABLAP transmits and receives data packets on behalf of its clients. The format and interpretation of packets is defined by higher level protocols. Client data is sent by the ABLAP by encapsulating it into an *ABLAP Data* frame (LAP type field = \$01 to \$7F). The data field contains a sequence of up to 600 octets. It should be noted that the length in bits of client data in a frame must be an integral multiple of 8.

Frame Check Sequence Field

The 16-bit frame check sequence (FCS) is computed as a function of the contents of the source address, destination address, LAP type, and data fields. The encoding of CRC-CCITT is defined in terms of the standard generating polynomial:

$$G(x) = x^{16} + x^{12} + x^5 + x^1$$

The CRC-CCITT frame check sequence value corresponding to a given frame is calculated based on the following polynomial division identity:

$$\frac{M(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)}$$

where:

$M(x)$ = binary polynomial (corresponding to the frame after complementing its first 16 bits);

$R(x)$ = remainder after dividing $M(x)$ by the generating polynomial (its coefficients are the bits of the FCS).

The implementation of the CRC for the FCS field, at the transmitter, computes the CRC starting with the first bit of the destination address following the opening flag and stopping at the end of data field. The FCS field at this point is an inversion, or ones-complement, of the transmitter's remainder. The result of a correctly received transmission is a constant: $\times 0001110100001111 (x^{15} \dots x^0)$.

In the SDLC implementation of CRC, a modified polynomial expression (modulo 2) of the transmitted data to be checked is divided by the generating polynomial, $x^{16} + x^{12} + x^5 + 1$. Integer quotient digits are ignored, and the transmitter sends the complement of the resulting remainder value as the FCS.

In addition to the division of the binary value of the data by the generating polynomial to generate the remainder for checking, the following manipulations occur:

1. The dividend is initially preset to all 1's. This adds the binary value of the preset bits to that of the data bits.
2. The transmitter's remainder is inverted bit-by-bit (FCS field) as it is sent to the receiver. The high-order bit of the FCS field is transmitted first (x_{15}, \dots, x_0).
3. The receiver includes the FCS field as part of its dividend. Continued computation raises the value of the dividend polynomial by the factor x^{16} . Since the dividend and remainder at the receiver is equal to that at the transmitter at the beginning of the FCS field, the remainder at the receiver at the end of the FCS field is a constant that is characteristic of the divisor.

If the receiver computation does not yield the constant, $\times 0001110100001111$, it is assumed that the frame was received in error. The entire frame is suspect and discarded.

Frame Size Limitations

Since the ABLAP header consists of 3 octets and the data field has 0 to 600 octets, the smallest valid frame (not including the FCS) is 3 octets long, while the largest is 603 octets long.

III.3. Frame Transmission

ABLAP transmits client data in ABLAP data frames using a special dialogue involving one or more ABLAP control frames followed by an ABLAP data frame. The exact form of the dialogue depends on the destination of the frame. On this basis ABLAP distinguishes two kinds of frames: *Directed* and *Broadcast*.

A directed packet is one whose destination address is a single node; a broadcast packet is intended to be received by all nodes.

The purpose of the dialogue mentioned above is to control the access to the shared bus in an orderly fashion that reduces the probability of a collision. This is based on a CSMA/CA technique.

The dialogues must be separated by a minimum Inter-Dialog Gap (IDG) of 400 μ sec.; the different frames of a single dialog must follow one another with a maximum Inter-Frame Gap (IFG) of 200 μ sec.

The frame transmission procedure is described separately for directed and for broadcast frames.

Consider the case of a directed data frame (see Figure 5(a)). The transmitting node uses the physical layer's ability to sense if the line is in use. If the line is busy the node waits until it becomes idle (the node is said to *defer*). Upon sensing an idle line, the

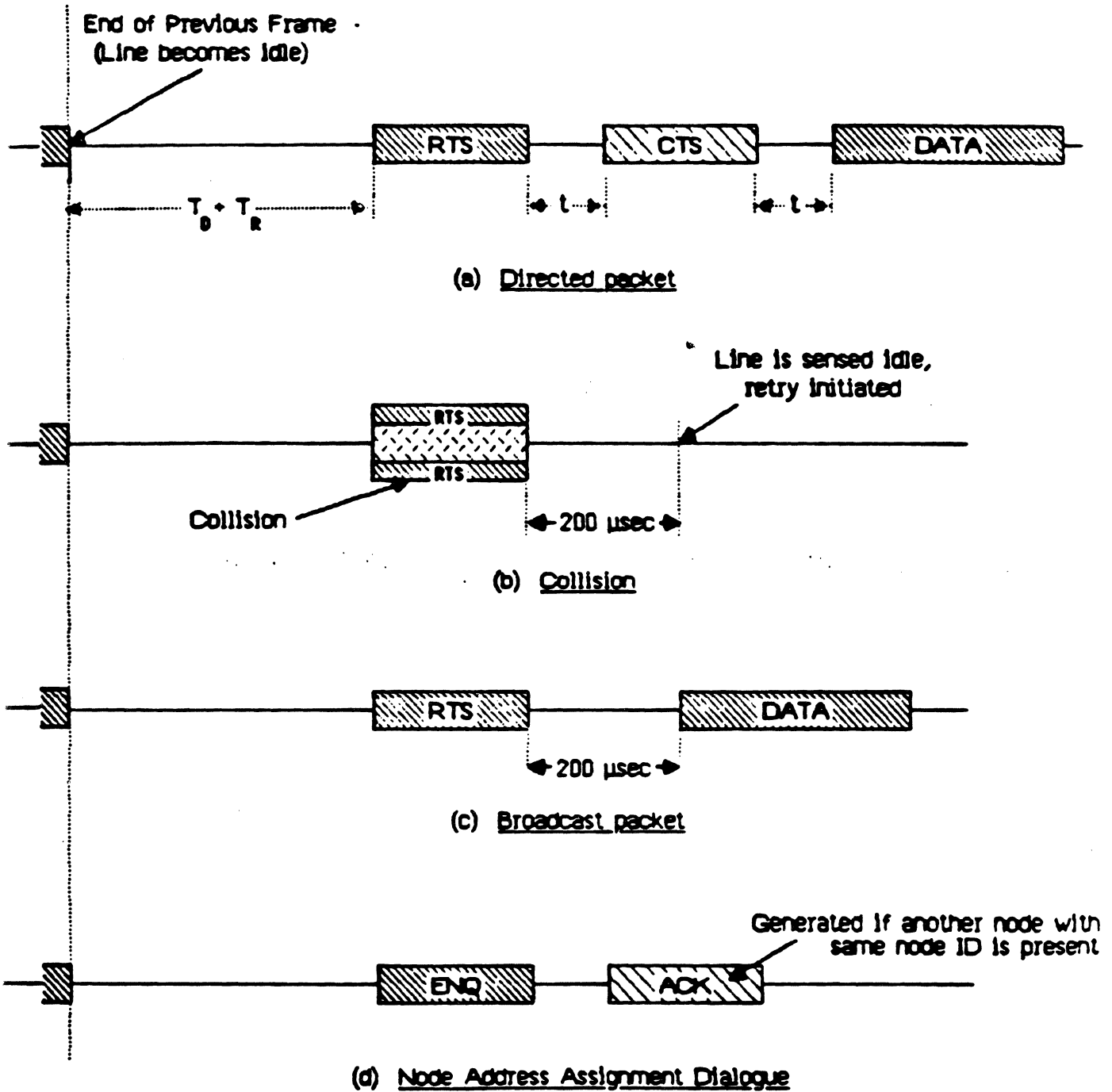
transmitter waits for a time equal to the minimum IDG plus a randomly generated amount. During this "wait", the transmitter continues to monitor the line. If the line remains idle throughout this wait period, then it sends an ABLAP lapRTS frame to the intended receiver of the data frame. The receiver must within the maximum IFG return a lapCTS frame to the transmitting node. Upon receiving this frame, the transmitter must within the maximum IFG send out the data frame.

The purpose of this algorithm is twofold: (1) to restrict the periods in which collisions are highly likely (this is during the lapRTS-lapCTS exchange), and (2) to spread out in time several transmitters waiting for the line to become idle. The lapRTS-lapCTS exchange, if successfully completed signifies that a collision did not occur, and that all intending transmitters have heard of the coming data frame transmission and are deferring/waiting.

If in fact a collision does occur during the lapRTS-lapCTS exchange, a lapCTS will not be received, and the sending node will then *back off* and retry. The sending node is said to presume a collision.

The range of the random wait time is adjusted if such a collision is detected. In fact, this adjustment or back off is done using a linear back off algorithm that dynamically modifies this range in response to recent traffic history. The idea is that if collisions have been presumed for recently sent packets, this signifies heavier loading and higher contention for the bus. Then the random wait should be generated over a larger range, thus spreading out (in time) the different contenders for the line.

Two factors are used for adjusting the range: (a) the number of times the node had to defer, and (b) the number of times it had to back off. This history is maintained in two 8-bit *history bytes*, one each for deferrals and for back offs. At each attempt to send a packet these bytes are shifted left one bit. The lowest bit of each byte is then set if the node had to defer or back off, respectively, on that attempt, else this bit is cleared. In effect the history bytes remember the deference and back off history for the last eight attempts.



- t -- Inter-Frame Gap (< 200 microsecs)
- T_D -- Inter-Dialogue Gap (> 400 microsecs)
- T_R -- randomly generated time interval

Figure 5 ABLAP Timing Diagrams

The history byte is used to adjust a *global backoff mask*. The mask takes values in the range \$00 to \$0F. When the first attempt is made to send a particular frame, and the node must defer, the mask is OR'ed with \$01, thus setting its lowest bit.

The number of set bits in each history byte provides a count of the number of times the node had to defer (back off) in the last eight attempts. This is used to adjust the mask as follows:

- If the number of times the node backed off during the last 8 attempts is greater than 2, the mask is extended by one bit (up to the maximum of 4 bits) and the back off history byte is then cleared.
- Else, if the number of times the node had to defer is less than 2, the mask is reduced by one bit (down to the minimum of 1 bit).
- Else, if neither of these apply, the mask is left as is.

Directed Transmissions

Directed Packets are sent via a 3-frame dialog as follows:

- L1) The transmitter senses the bus until the bus is idle for the minimum IDG time.
- L2) The transmitter waits an additional time determined by a random number.
(The actual procedure for generating this random value is discussed below).
- L3) The transmitter sends a lapRTS frame to the intended Destination node.
- L4) The receiver sends a lapCTS frame to the transmitter.
- L5) The transmitter, upon successful reception of the lapCTS, sends a Data frame
(in which it encapsulates the client's packet).

Note: all response frames must be sent within the maximum IFG time.

The extra wait during L2 tends to spread out transmitters which are contending for the line and, thus, minimize (or avoid) collisions. A transmitter assumes that a collision has

occurred when it does not receive a CTS frame in the required time (IFG). When this happens, the transmitter retries starting at L1. For each attempt, a new random number must be generated in step L2. If after 32 attempts the transmitter is unable to send the data frame, it reports failure to its client.

Broadcast Transmissions

Broadcast packets are distinguished by their Destination address being SFF. When the ABLAP TransmitPacket procedure detects this, it performs the following procedure:

- B1) The transmitter senses the bus until the bus is idle for the minimum IDG time.
- B2) The transmitter waits an additional time determined by a random number.
(The actual procedure for generating this random value is discussed below).
- B3) The transmitter sends a lapRTS frame with destination address SFF.
- B4) The transmitter senses the line for the maximum IFG time.
- B5) After sensing the line idle for IFG, it sends its DATA frame.

The purpose of the RTS in step B3 is to ensure that other transmitters are cognizant of the intent to transmit, and to force a collision if another transmitter starts at the same time. If the transmitter detects bus activity during B4, it makes up to 32 further attempts beginning with step B1. If it fails to transmit the data frame after 32 attempts, it reports failure to its client.

Broadcast packets are sent without collision except in the unlikely case of another transmitter attempting a broadcast at the same time.

III.4. Frame Reception

Frame reception is in many ways the inverse of the frame transmission process.

Receive Data Decapsulation

ABLAP recognizes the boundaries of an incoming frame by monitoring the Receive Character Available boolean variable provided by the Physical Layer. Each incoming octet is passed to the Receive Frame function until an End of Frame boolean variable is set by the Physical Layer. The received frame can generate several conditions that must be handled by ABLAP: overrun Error, bad frame Size, underrun Error, bad frame Type, and bad frame CRC.

Frame Size Error

If a frame (excluding the FCS), of greater than 603 bytes or smaller than 3 bytes is received then it is rejected and a badframe error status generated.

Overrun/Underrun Error

If ABLAP was not able to stay synchronized with the incoming data, causing receiver overruns or underruns, the frame will be rejected and an overrun or underrun error generated.

Frame Type

The frame's LAP type field is checked against each of the valid values for a ABLAP frame. Otherwise the frame is rejected and a bad frame type error is generated.

Frame Check Sequence

The CRC-CCITT frame check sequence is computed for the incoming frame. When the entire frame has been received, ABLAP tests the computed FCS for the frame. The CRCok Boolean variable is tested to determine whether the frame was received without transmission errors. If the CRC is in error the frame is rejected and a badframe CRC error is generated.

Address Recognition

If the destination address of the frame is not equal to that of the receiving node or a broadcast frame, then the rest of the frame is ignored.

IV.1. Global Constants, Types and Variables

The following global const's, type's and var's are used throughout the model.

```
const
  minFrameSize = 3;      { smallest (LAP header only) frame }
  maxFrameSize = 605;   { size of largest LAP frame including FCS }
  maxDataSize = 600;   { size of largest (encapsulated) ABLAP data field }
  bitTime = 4.34;      { bit time (usec) }
  byteTime = 39.0;     { worst case single byte time (usec) }
  minIDGTime = 400.0;  { minimum Inter-Dialog Gap in (usec) }
  IDGslotTime = 100.0; { slot time of transmit backoff algorithm (usec) }
  maxIFGTime = 200.0;  { maximum Inter-Frame Gap in (usec) }
  maxDefers = 32;      { " " defers for a single packet }
  maxCollsns = 32;     { " " collisions for a single packet }
  lapENQ = $81;        { LAP type field value of ENquiry frame }
  lapACK = $82;        { " " " ACKnowledgement frame }
  lapRTS = $84;        { " " " RequestToSend frame }
  lapCTS = $85;        { " " " ClearToSend frame }
  hdlcFLAG = $7E;     { value of an HDLC FLAG }

type
  { global result types from LAP functions }
  TransmitStatus = ( transmitOK, excessDefers, excessCollsns, dupAddress );
  ReceiveStatus = ( receiveOK, Receiving, nullReceive, frameError );
  FrameStatus = ( noFrame, lapDATAframe,
                 lapENQframe, lapACKframe, lapRTSframe, lapCTSframe,
                 badframeCRC, badframeSize, badframeType,
                 overrunError, underrunError );

  { Data Link types and structures }
  bit = 0..1;
  bitVector = packed array [0..7] of bit;
  octet = $00..$FF;
  anAddress = octet;
  aLAPtype = octet;
  aDataField = packed array [1..maxDataSize] of octet;

  { basic structure of an ABLAP frame, not including FLAGS, FCS }
  frameInterpretation = ( raw, structured );
  aFrame = packed record
    case frameInterpretation of
      raw:
        rawData : packed array [1..maxFrameSize] of octet;
      structured (
        dstAddr : anAddress;
        srcAddr : anAddress;
        lapType : aLAPtype;
        dataField : aDataField )
    end;
```

```

var
  MyAddress : octet;           { set during initializeLAP }
  Backoff : integer;         { current backoff range }
  fAdrValid,                 { MyAddress has been validated }
  fAdrInUse,                 { another node has same MyAddress }
  fCTSexpected : boolean;    { RTS has been sent, CTS is valid }
  deferCount, collsnCount : integer; { optional, for statistics only }
  deferHistory, collsnHistory : bitVector;
  outgoingLength, incomingLength : integer;
  outgoingPacket, incomingPacket : aFrame;

```

IV.2. Hardware Interface Declarations

The following declarations refer to hardware specific interfaces which are assumed to be available to the LAP procedures. The functions are typically bits and or bytes contained in the relevant hardware interface chip(s). Similarly, the procedures are expected to be represented in actual hardware by means of control bits within the hardware.

We will briefly describe the assumed attributes of each of these. Appendix 1 defines the correspondence between these definitions and the actual interface relevant to the hardware of the Lisa and/or Mac (which implements AppleBus via the Zilog™ Serial Communications Controller (SCC)).

CarrierSense indicates that the hardware is sensing a frame on the bus.

RcvDataAvail indicates that a data byte is available.

rxDATA is the next data byte available (as indicated by RcvDataAvail).

EndOfFrame indicates that a valid closing FLAG has been detected.

CRCok indicates that the received frame's FCS is correct (when EndOfFrame is true).

OverRun indicates that the code did not keep up with data reception.

MissingClock indicates that a missing clock has been detected.

setAddress sets the hardware to receive frames directed to MyAddress.

enableTxDrivers and disableTxDrivers control the operation of the RS-422 drivers.

enableTx and disableTx control the operation of the data transmitter.

txFLAG causes the transmission of a FLAG.

txDATA causes the transmission of a data byte; this is WRB.

txFCS causes the transmission of the Frame Check Sequence.

txONES causes 12+ ones (1's).

resetRx, enableRx and disableRx control the receiver.

resetMissingClock causes the MissingClock indication to be cleared.

{ hardware interface functions/procedures }

```

function CarrierSense : boolean;      external;
function RxDATAAvail : boolean;       external;
function rxDATA : octet;              external;

```

```

function EndOfFrame : boolean;          external;
function CRCok : boolean; external;
function OverRun : boolean;            external;
function MissingClock : boolean;       external;
procedure setAddress( addr : octet );  external;
procedure enableTxDrivers;              external;
procedure disableTxDrivers;            external;
procedure enableTx;                      external;
procedure bFLAG;                        external;
procedure bDATA( data : octet );        external;
procedure bFCS;                         external;
procedure bONES;                        external;
procedure disableTx;                    external;
procedure resetRx;                      external;
procedure enableRx;                     external;
procedure disableRx;                    external;
procedure resetMissingClock;            external;

```

IV.3. Interface Procedures and Functions

The ABLAP model's interface to the next higher layer (its client) is specified in terms of the following three calls:

Procedure InitializeLAP(hint : octet; server : boolean)

This procedure initializes the ABLAP; it is expected to be called exactly once. The hint parameter is a suggested starting value for the node's AppleBus physical link address; a value of 0 indicates that ABLAP generate a starting value. Upon return from the call, the station's actual address is available in the global variable MyAddress. If server is true then the internal procedure acquireAddress will spend extra time to determine if another node is using the selected node address.

Function transmitPacket(dstParam : anAddress; typeParam : aLAPtype; dataField : aDataField; dataLength : Integer) : TransmitStatus;

This is the call provided to transmit a packet across. The internal function TransmitLinkMgmt : TransmitStatus performs the transmission Link Access algorithms.

Function receivePacket(var dstParam : anAddress; var srcParam : anAddress; var typeParam : aLAPtype; var dataField : aDataField; var dataLength : Integer);

This is the entry provided to receive a packet. The internal function ReceiveLinkMgmt : FrameStatus; implements the reception Link Access algorithms.

IV.4. InitializeLAP Procedure

InitializeLAP is called to reset the global variables to known states; it calls acquireAddress to initialize MyAddress.

```
procedure InitializeLAP( hint : octet; server : boolean );
var   i : integer;
begin
  Backoff := 0;

  [ Initialize history data for Backoff calculations ]
  for i := 0 to 7 do
    begin
      deferHistory[i] := 0;
      collisionHistory[i] := 0
    end;

  deferCount := 0; collisionCount := 0; {optional}

  acquireAddress( hint, server )
end;
```

IV.5. AcquireAddress Procedure

The procedure acquireAddress provides the dynamic node assignment algorithm. A special frame (of type lapENQ) is created and sent. When no node responds after repeated attempts, the current value of MyAddress is assumed to be safe for use by this node; the state of fAdrValid reflects this fact. (If the global fAdrInUse ever becomes true after a call to acquireAddress, another node that is using the same MyAddress has been detected.)

```
procedure acquireAddress( hint : octet; server : boolean );
var   maxTrys, trys : integer; ENQframe : eFrame;
begin
  if hint > 0
  then
    MyAddress := hint
  else
    if server
    then
      MyAddress := Random( 127 ) * 128
    else
      MyAddress := Random( 127 ) * 1;
  setAddress( MyAddress );
  fAdrValid := false;
  if server
  then
    maxTrys := 1500
  else
    maxTrys := 50;
```

[the main loop of acquireAddress; repeatedly check for response to ENQ]

```

repeat
  trys := 0; fAdrInUse := false;
  while trys < maxTrys do
    if (transmitPacket( MyAddress, lapENQ, ENQframe.dataField, 0 )
      = transmitOK) or fAdrInUse
    then
      begin
        if server
          then
            MyAddress := Random( 127 ) * 128
          else
            MyAddress := Random( 127 ) * 1;
        setAddress( MyAddress );
        trys := 0
      end
    else
      if trys < maxTrys
        then
          trys := trys + 1;
        else
          fAdrValid := true
      until fAdrValid
end;

```

IV.6. TransmitPacket Function

The function transmitPacket is called by the ABLAP client to send a packet. After constructing (encapsulating) the caller's dataParam, it calls upon TransmitLinkMgmt to perform the actual link access.

```

function transmitPacket(
  dstParam : anAddress;
  typeParam : aLAPtype;
  dataParam : aDataField;
  dataLength : Integer ) : TransmitStatus;
begin
  if fAdrInUse
  then
    transmitPacket := dupAddress
  else
    begin
      { copy interface data into frame for TransmitLinkMgmt }
      with outgoingPkt do
        begin
          dstAddr := dstParam;
          srcAddr := MyAddress;
          lapType := typeParam;
          dataField := dataParam
        end;
    end;
  end;

```



```
    outgoingLength := dataLength + 3;
    transmitPacket := TransmitLinkMgmt
end
end;
```

IV.7. TransmitLinkMgmt Function

The function TransmitLinkMgmt implements the Carrier Sense, Multiple Access w/ Collision Avoidance algorithm; TransmitLinkMgmt is at the heart of the AppleBus Link Access Protocol.

The typical AppleBus hardware is not capable of performing collision detection.

LAP attempts to minimize collisions by requiring transmitters to wait a randomly generated amount of before sending their RTS frames after the bus has been sensed idle for a minimum time (the IDG). Any transmitter which detects that another transmission is in progress while it is in this random wait must defer.

In order to minimize delays under light loading, but still be able to minimize the probability of collisions under moderate to heavy loading, the random delay is picked in a range that is constantly adjusted based upon the recently observed history of a node's attempts to access the bus. Two history vectors (deferHistory and collsnHistory) are used to keep track of the last 8 access attempts at the bus. Each attempt adds a single bit of history data to these vectors by shifting left the current values and updating the vacated bit with the appropriate value. deferHistory[0] is set if a deference is required during an access; collsnHistory[0] is set if an attempts result in an assumed collision (i.e. back off).

The range of the current Backoff is adjusted upwards (to a maximum of 16) when the observed collisions exceeds 2 in the last 8 attempts; Backoff is adjusted downwards if the number of observed deferences is less than 2 in the last 8 attempts. When an adjustment is made, the corresponding history data is set to the "maximum" value so that further adjustments are inhibited until more history data has accumulated. The maximum value for deferHistory is all 1's; the maximum value for collsnHistory is all 0's.

Note that a local backoff (LclBackOff) is used during the retry attempts of a given frame.

This has the effect of spreading out attempts to a non-listening node for a longer time, thus increasing its chances of receiving our packet.

```

function TransmitLinkMgmt : TransmitStatus;
var
  LclBackOff, l : Integer;
  fBroadcast, fENQ : boolean;
  xmttimer : real;
  rcvdfame : FrameStatus;
  RTSframe : aFrame;
begin
  with RTSframe do
    begin
      dstAddr := outgoingPacketLdstAddr;
      srcAddr := MyAddress;
      lapType := lapRTS
    end;
    fBroadcast := outgoingPacketLdstAddr = $FF;
    fENQ := outgoingPacketLlapType = lapENQ;

    { adjust Backoff, based upon recent history }
    if bitCount( collsnHistory ) > 2
    then
      begin
        Backoff := min( max( Backoff + 2, 2 ), 16 );
        for l := 0 to 7 do
          collsnHistory[l] := 0
        end
      end
    else
      if bitCount( deferHistory ) < 2
      then
        begin
          Backoff := Backoff div 2;
          for l := 0 to 7 do
            deferHistory[l] := 1
          end;
        end;
      end;

    { shift history data }
    for l := 7 downto 1 do
      begin
        collsnHistory[l] := collsnHistory[l-1];
        deferHistory[l] := deferHistory[l-1]
      end;
    collsnHistory[0] := 0; deferHistory[0] := 0;

    { initialize main loop }
    deferTries := 0; collsnTries := 0;
    LclBackOff := BackOff; transmitDone := false;

    {begin main loop of TransmitLinkMgmt}
    repeat
  
```

```

( wait for minimum InterDialogGap time )
repeat
  ( wait for any packet in progress to pass )
  if CarrierSense
  then
    begin
      ( ensure minimum backoff if packet in progress )
      LclBackoff := max( LclBackoff, 2 );
      deferHistory[0] := 1;

      ( perform watchdog reset of Rx for "stuck" CarrierSense )
      xmtTimer := RealTime + 1.5 * maxFrameSize * byteTime;
      repeat
        until not CarrierSense or (RealTime > xmtTimer);
      if CarrierSense
      then
        resetRx (something's wrong, clear it)
      end;
    end;

  ( wait for minimum IDG after packet (or idle line) )
  xmtTimer := RealTime + minIDGtime;
  repeat
    until (RealTime > xmtTimer) or CarrierSense;
  until not CarrierSense;

( wait our additional backoff time, deferring to others )
xmtTimer := RealTime + Random( LclBackoff ) * IDGslotTime;
repeat
  until (RealTime > xmtTimer) or CarrierSense;

if CarrierSense or MissingClock
then
  (defer)
  begin
    incr( DeferCount ); (optional)
    LclBackoff := max( LclBackoff, 2 );
    deferHistory[0] := 1;
    if deferTries < maxDefers
    then
      deferTries := deferTries + 1
    else
      begin
        TransmitLinkMgmt := excessDefers;
        transmitDone := true
      end
    end
  end

else (not (CarrierSense or MissingClock))
begin
  (send our RTS/ENQ and await CTS/ACK)
  if FENQ
  then
    transmitFrame( outgoingPacket, 3 )
  else
    transmitFrame( RTSframe, 3 );
end

```

```

{ use common code to detect line state }
fCTSexpected := true;
rcvdfame := receiveFrame;
fCTSexpected := false;
If fBroadcast and (rcvdfame = noFrame)
then
begin
transmitFrame( outgoingPacket, outgoingLength );
TransmitLinkMgmt := transmitOK;
transmitdone := true
end
else
If rcvdfame = lapCTSframe
then
begin
transmitFrame( outgoingPacket, outgoingLength );
TransmitLinkMgmt := transmitOK;
transmitdone := true
end;

{ assume collision if we don't receive the expected CTS }
If not transmitdone
then
begin
(CTS not seen, initiate retry)
incr( CollsnCount );           {optional}
collsnHistory[0] := 1;        {update history data}
If collsnTries < maxCollsns
then
collsnTries := collsnTries + 1
else
begin
TransmitLinkMgmt := excessCollsns;
transmitdone := true
end
end

end {else, not CarrierSense}

until transmitdone
end;

```

IV.8. TransmitFrame Procedure

The procedure transmitFrame is responsible for putting data on to the bus. Notice that certain details, such as how a FLAG is forced and a packet terminated (which includes sending of the FCS) are not explicitly detailed here due to their extreme hardware dependence. Note: the 12 ones at the end of the frame are required to finish clocking of data by a receiver.

```

procedure transmitFrame( var frame : aFrame; framesize : integer );

var
  i : integer;
  bitTimer : real;
begin
  disableRx;

  { generate the Synchronizing Pulse }
  bitTimer := RealTime * 1.5 * bitTime;
  enableTxDrivers;
  while RealTime < bitTimer do
    begin end;
  disableTxDrivers;
  bitTimer := RealTime + 1.5 * bitTime;
  while RealTime < bitTimer do
    begin end;

  { start the actual frame transmission }
  enableTxDrivers;
  enableTx;
  txFLAG; txFLAG;           { output 2 opening FLAG's }
  for i := 1 to framesize do
    TxData( frame.rawData[i] );
  txFCS;                    { send the FrameCheckSequence }
  txFLAG;                   { the trailing FLAG }
  txONES;                   { send 12 ones for extra clocks }
  disableTxDrivers;

  { re-establish default listening mode }
  resetMissingClock;
  enableRx
end;

```

IV.9. ReceivePacket Procedure

The procedure receivePacket is the primary interface routine to higher levels. It is specified as if it is synchronously called by the user. Note that in many implementations, the lower-level ReceiveLinkMgmt function would be invoked by an interrupt routine.

```

procedure receivePacket(
  var dstParam : anAddress;
  var srcParam : anAddress;
  var typeParam : aLAPtype;
  var dataParam : aDataField;
  var dataLength : integer );
var status : ReceiveStatus;
begin
  repeat

```

```

status := ReceiveLinkMgmt;
If status = receiveOK
then
  begin
    with IncomingPacket do
      begin
        dstParam := dstAddr;
        srcParam := srcAddr;
        typeParam := lapType;
        dataParam := dataField
      end;
    dataLength := IncomingLength
  end
until status = receiveOK
end;

```

IV.10. ReceiveLinkMgmt Function

The function ReceiveLinkMgmt implements the receiver side of the Link Access Protocol; it would typically be called from an interrupt routine rather than receivePacket.

```

function ReceiveLinkMgmt : ReceiveStatus;
var
  status : ReceiveStatus;
  CTSframe, ACKframe : sFrame;
begin
  status := Receiving;
  while status = Receiving do
    case receiveFrame of
      badframeCRC, badframeSize, badframeType,
      underrunError, overrunError:
        status := frameError;
      lapENQframe:
        If fAdrValid
        then
          begin
            ACKframe.dstAddr := incomingPacket.srcAddr;
            ACKframe.srcAddr := MyAddress;
            ACKframe.lapType := lapACK;
            transmitFrame( ACKframe,3 );
            status := nullReceive
          end
        else
          begin
            fAdrInUse := true;
            status := nullReceive
          end;
      lapRTSframe:
        If fAdrValid
        then
          begin

```

```

CTSframe.dstAddr := IncomingPacket.srcAddr;
CTSframe.srcAddr := MyAddress;
CTSframe.lapType := lapCTS;
.transmitFrame( CTSframe, 3 )
end
else
begin
fAddrInUse := true;
status := nullReceive
end;
lapDATAframe:
if fAddrValid
then
status := receiveOK
else
begin
fAddrInUse := true;
status := nullReceive
end;
noFrame:
status := nullReceive
end (case receiveFrame);
ReceiveLinkMgmt := status
end;

```

IV.1.1. ReceiveFrame Function

The function receiveFrame is responsible for interacting with the hardware.

```

function receiveFrame : FrameStatus;
var rcvtimer : real;
begin
{ provide timeout for idle line }
rcvtimer := RealTime + maxIFGtime;
repeat
until CarrierSense or (RealTime > rcvtimer);
if not CarrierSense
then
begin
receiveFrame := noFrame;
exit( receiveFrame )
end;
{ line is not idle, check if frame is for us }
rcvtimer := RealTime + maxIFGtime;
repeat
until RxCharAvail or (RealTime > rcvtimer);
if RxCharAvail
then

```

```

begin                                     { receive frame }
error := false; framedone := false; incomingLength := 0;
repeat
  rcvtimer := RealTime + 1.5 * byteTime;
  repeat
    until RxCharAvail or (RealTime > rcvtimer);
  if RxCharAvail
  then
    begin
      if OverRun
      then
        begin
          receiveFrame := overrunError;
          error := true
        end
      else
        if incomingLength < maxFrameSize
        then
          begin
            incomingLength := incomingLength + 1;
            incomingPacket.rawData[incomingLength] := rxDATA
          end
        else
          begin
            receiveFrame := badframeSize;
            error := true
          end;
        if EndOfFrame
        then
          begin
            if CRCok
            then
              begin
                incomingLength := incomingLength - 2; {account for CRC}
                if incomingLength < minFrameSize
                then
                  begin
                    receiveFrame := badframeSize;
                    error := true
                  end
                else
                  framedone := true
                end
              else {bad CRC}
              begin
                receiveFrame := badframeCRC;
                error := true
              end
            end {if EndOfFrame}
          end {if RxCharAvail}
        else { RealTime > rcvtimer }
        begin
          receiveFrame := underrunError;
          error := true
        end
      end
    end
  end
end

```



```

        end
        until framedone or error;
    { check on validity of the frame }
    if framedone
        then
            if fAdrValid
                then

                    { If our address is valid, check on actual type }
                    if IncomingPacket.lapType < $80
                        then
                            receiveFrame := lapDATAframe;
                        else
                            case IncomingPacket.lapType of
                                lapENQ:
                                    receiveFrame := lapENQframe;
                                lapRTS:
                                    receiveFrame := lapRTSframe;
                                lapCTS:
                                    if fCTSok
                                        then
                                            receiveFrame := lapCTSframe
                                        else
                                            begin
                                                fAdrInUse := true;
                                                receiveFrame := badframeType
                                            end;
                                        otherwise
                                            receiveFrame := badframeType
                                    end {case IncomingPacket.lapType}
                            else
                                { we received something which we didn't expect }
                                begin
                                    fAdrInUse := true;
                                    receiveFrame := noFrame
                                end

                            end {if RxCharAvail}

                        else { no RxCharAvail }
                            receiveFrame := noFrame;

                    resetRx; resetMissingClock
                end;

```

IV.12. Miscellaneous Procedures and Functions

The following low-level routines are referenced by the above code.

```
procedure incr( var i : Integer );
begin
  i := i + 1
end;
```

```
function bitCount( bits : bitVector );
var i, sum : Integer;
begin
  sum := 0;
  for i := 0 to 7 do
    sum := sum + bits[i];
  bitCount := sum
end;
```

```
function min( val1, val2 : Integer ) : Integer;
begin
  if val1 < val2
  then
    min := val1
  else
    min := val2
end;
```

```
function max( val1, val2 : Integer ) : Integer;
begin
  if val1 > val2
  then
    max := val1
  else
    max := val2
end;
```

```
function Random( maxval : Integer ) : Integer;
begin
  ( note: this is implemented as any "good" random number generator
    which produces a result in the range [0,maxval-1] )
end;
```

Appendix 1. SCC Implementation

This appendix explains how the hardware interface routines declared in section IV.3 are actually implemented on the Mac, which uses the Zilog™ SCC as its primary interface for AppleBus. Note that all of the register and bit names below are those used by Zilog™ in its SCC documentation.

CarrierSense indicates that the hardware is sensing a frame on the bus; this corresponds to the complement of the SYNC/HUNT bit in RR0.

RcvDataAvail indicates that a data byte is available; this corresponds to the Rx Character Available bit in RR0.

DDATA is the next byte from the bus; it is RR8.

EndOfFrame indicates that a valid closing FLAG has been detected; this is the End of Frame bit in RR1.

CRCOk indicates that the received frame's FCS was correct (when EndOfFrame is true); this is the complement of the CRC/Framing Error bit in RR1.

OverRun indicates that the code did not keep up with data reception; this is the Rx Overrun Error bit in RR1.

MissingClock indicates that the hardware has detected a missing transition on the bus; this is the One Clock Missing bit in RR10.

setAddress is a procedure which sets the hardware to receive packets whose destination address matches MyAddress; this simply sets WR6 in the SCC.

enableTxDrivers and disableTxDrivers control the operation of the RS-422 drivers. On the Mac, this is controlled by the RTS bit in WR5.

enableTx and disableTx control the operation of the transmitter; this is done by means of the Tx Enable bit in WR5.

txFLAG causes the transmission of a FLAG. This happens automatically at frame opening when Tx Enable is set; however, the code must delay long enough to cause the extra FLAG. The trailing FLAG is generated automatically at frame end as part of the Tx Underrun processing.

txDATA causes the transmission of a data byte; this is WR8.

txFCS causes the transmission of the Frame Check Sequence. This is caused automatically by letting the Tx Underrun occur.

txONES causes 12+ ones (1's). This is done by disabling the SCC transmitter (setting

Tx Enable to 0), while leaving the 422 drivers on (RTS set to 1), and delaying.

resetRx, enableRx and disableRx control the receiver; this is done by means of the Rx Enable bit in WR3. resetRx should also flush of the receive FIFO.

resetMissingClock causes the MissingClock indication to be cleared; this is done by a Reset Missing Clock command via WR14.

AppleBus PROTOCOL ARCHITECTURE

by

Gursharan S. Sidhu

and

Richard F. Andrews

(with contributions by Alan B. Oppenheimer)

Chapters

- I. Introduction
- II. Datagram Delivery Protocol
- III. Routing Table Maintenance Protocol
- IV. Name Binding Protocol
- V. Zone Information Protocol
- VI. AppleBus Transaction Protocol
- VII. Data Stream Protocol

Date: May 1, 1984

I. INTRODUCTION

This is a general document covering the overall architecture and other global protocol issues for AppleBus. Simple but effective protocols are defined at the network, transport and session levels. Protocols for the presentation and higher levels will be discussed in another document.

1.1 Basic Goals:

In setting a clear goal for the architecture we must keep the following in mind:

(1) Open Systems Architecture:

We must not build a system that is a "straitjacket" that restricts developers of AppleBus systems from expressing their creativity. Nor should we restrict their ability to add to the architecture specific functions and features for their particular applications. With this in mind, AppleBus has a *simple, layered architecture* that allows protocols to be added at any level, and that specifies a small repertoire of protocols and access methods for only the lowest layers (network, transport and session). *No attempt is made to mandate the use of these protocols; they are simply recommended by Apple for efficiency and compatibility.*

(2) Simplicity and Versatility:

Compared to the many currently available local-area interconnect systems with speeds in the megabit-per-second plus range, AppleBus is a modest speed interconnect system. This leads to some important conclusions relevant to protocol design:

- to keep the station latency low, *the number of nodes on an AppleBus will in most cases be modest (on the order of twenty-five nodes),*
- it is important to keep packet headers as short as practical *without loss of functionality.*
- *interconnection of AppleBuses among themselves and with other networks is an important part of the protocol design.*

The architecture must be versatile enough to satisfy the needs of both a *peripheral bus and a network interconnect system.*

The protocols and access methods proposed here should be viewed as versatile primitives with which developers can incorporate AppleBus into an interconnected system of personal computers, peripherals, servers, and communication to other networks and mainframe resources.

(3) Support Third Party Development:

Apple is not proposing to build the entire network system by itself, or as a complete closed system. Apple wants to provide the correct framework to facilitate development by others and will provide the key building blocks and certain exemplary services (e.g. file servers) that can serve as the basic foundation for other services (e.g. mail, print spooling, etc.).

The architecture is discussed here using the terminology of the widely referenced ISO *Open Systems Interconnection* framework.

While layered architectures might to some appear complex, layering can in fact reduce complexity and enhance maintainability by partitioning the system into small, manageable chunks, each performing a well-defined function. But complexity and inefficiency can result from the implementation of each layer as a separate module with interfaces based on high overhead operating-system calls. It is important to realize that architecture diagrams should not be used as models for modular software implementation; they are pictorial representations of the logical interrelationship of the diverse protocol functions and services offered by the architecture.

Layered architectures do not imply "buffer-copying" to move data from layer to layer. The implementer can design the various interfaces to avoid copying.

1.2 Overall Requirements and a Global Structure:

The AppleBus Link Access Protocol (ABLAP) provides the basic underlying service of packet transmission between the nodes of an AppleBus. The main goals of the datalink protocol for a shared bus system like AppleBus are as follows:

1. provide access control;
2. provide a node addressing mechanism;
3. ensure packet integrity.

The ABLAP uses a CSMA/CA (*Carrier Sense Multiple Access with Collision Avoidance*) protocol for access control. Node addressing is done in the LAP header using one-byte source and destination node IDs. Packet integrity is ensured through the use of a two-byte checksum (CRC-16) at the end of a LAP frame.

The LAP's service must be augmented by the higher layers of the architecture with certain additional features:

(1) Sockets: Nodes on the system should be allowed to establish/create network-addressable logical entities known as sockets.

(2) Socket-to-Socket Transport: The basic transport functions of the system will provide a best-effort datagram transport service between any pair of sockets on the system.

(3) Reliable Transactions: The basic socket-to-socket transport of datagrams is inadequate in many respects. In particular, its level of reliability (loss of packets, out-of-sequence and duplicate delivery of packets, etc.) is often inadequate. A very useful added-value transport service allows the client of a socket to send a transaction request packet to the client of another socket and to receive a transaction response with high reliability, in the face of the usual problems of packet loss, etc.

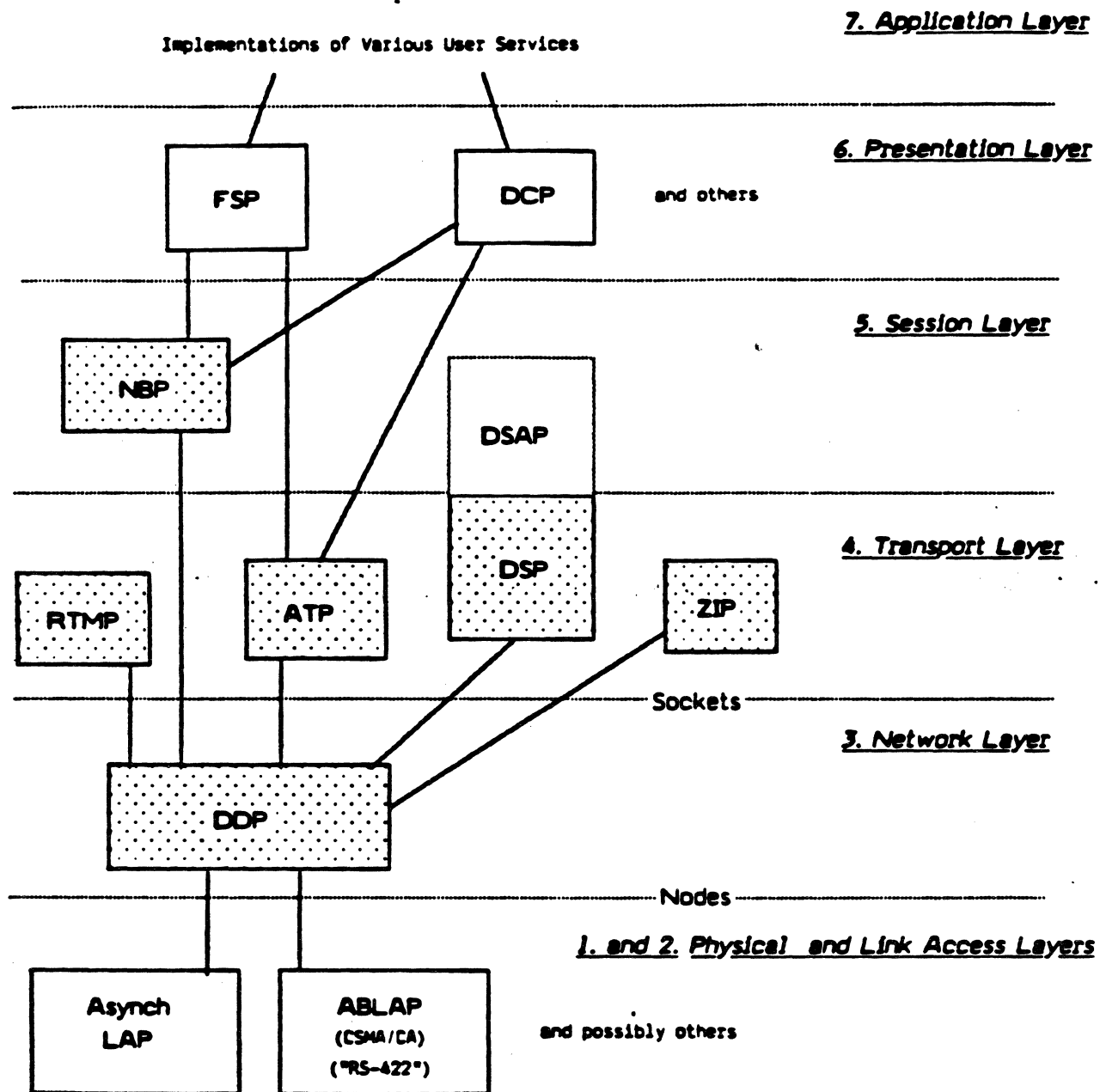
(4) Reliable Data Streams: Another class of socket clients will be exchanging sequences of data (e.g. screen contents, keyboard input, etc.) that they perceive as an arbitrary stream of bytes. Such clients need a reliable data stream service that takes care of problems such as data packetization, lost packets, out-of-sequence packets, and duplicates.

(5) Name Binding: The basic network-visible entities are the clients of sockets. These can be identified by their socket identification numbers (plus the node ID). This use of numbers is very efficient and appropriate for the network protocols but extremely inconvenient for the users of the network system. The latter would much rather refer to objects by their names (strings of characters). To make this possible we need a service that translates user-provided names into network addresses/identifiers such as socket identifiers, node numbers, etc.

(6) Special Function Protocols: This group includes protocols for controlling devices, file server access, etc. These are usually tailor-made for the specific service/function and will not be discussed here.

The AppleBus protocol architecture illustrated in figure Arch 1 was developed with these considerations in mind. This document covers the protocols indicated by gray shaded boxes referred to as the core of the AppleBus protocol architecture.

Although several of these core protocols have been implemented and tested, others are, as of this date, in a state of further refinement and development. In particular, the Data Stream and Zone Information protocols should be treated as proposals potentially subject to significant change.



Abbreviations

- | | |
|--|-------------------------------------|
| ABLAP -- Apple Bus Link Access Protocol | DSP -- Data Stream Protocol |
| ATP -- AppleBus Transaction Protocol | DSAP -- Data Stream Access Protocol |
| DCP -- Device Control Protocol | FSP -- File Server Protocol |
| DDP -- Datagram Delivery Protocol | NBP -- Name Binding Protocol |
| RTMP -- Routing Table Maintenance Protocol | ZIP -- Zone Information Protocol |

Figure Arch1: AppleBus Protocol Architecture

II. DATAGRAM DELIVERY PROTOCOL

II.1 Terminology

As noted in the introduction, the AppleBus Link Access Protocol (ABLAP) provides a node-to-node delivery of packets, with a best effort performance, on a single AppleBus. The Datagram Delivery Protocol (DDP) extends this mechanism to socket-to-socket delivery. DDP packets are referred to as *datagrams*.

DDP has been designed to provide this service over an AppleBus internet (see *Figure DDP1*). As illustrated there, an *AppleBus internet* consists of one or more AppleBus systems interconnected by *bridges*. Bridges are also referred to as *internet routing nodes* or *internet routers*. A bridge will often consist of a single node connected to two AppleBuses, or it might consist of two nodes each of which is connected to an AppleBus with the two nodes further connected to each other through a communication channel. As far as the protocol architecture is concerned, the channel between the two halves of such a bridge could be a leased or a dialup line or, in fact, it could be another network, e.g. a wide-area packet-switched or circuit-switched public network, or a higher speed broadband or baseband LAN used as a "backbone".

Bridges are packet forwarding agents. Packets can thus be sent between any two nodes of the internet by using a store-and-forward process through a series of internet routers. By combining this facility with DDP, packets can be sent between any two sockets on the internet.

Sockets, Socket Numbers:

Sockets are logical entities in the nodes of the network. *Datagrams*, packets that are delivered as single, independent entities, are exchanged between sockets. Sockets should be visualized as the sources and destinations of datagrams.

Sockets are identified by a *socket number*. Socket numbers are one byte (8 bits) long. Thus there can be at most 256 different socket numbers in a given node. Two of these values are reserved "0" to mean an unknown socket, and 255 for future use. The theoretical maximum number of sockets in a single node is thus 254.

Statically-Assigned and Well-Known Sockets

Sockets are owned by *socket clients*. These are typically processes (or functions in processes), implemented in software in the node. A socket client can send and receive datagrams only through sockets that it owns.

Sockets are classified into two groups: *statically* and *dynamically-assigned*.

Socket numbers decimal 1 through 127 are reserved for use by clients of *statically assigned sockets (SAS)*. Examples of such clients are AppleBus core protocols (e.g. ATP, NBP, RTMP, etc.) and low-level network built-in services such as echoers. The range of socket numbers decimal 1 through 64 is reserved by Apple for use in AppleBus core protocols and for future use. Numbers 65 through 127 are available for unrestricted experimental use. Use of these experimental SAS is not recommended for commercial products since there is no mechanism for eliminating conflicting usage by different developers. (see section II.6) Socket numbers decimal 128 through 254 are assigned dynamically by DDP upon request from clients in that node. Sockets of this type are said to be *dynamically assigned (DAS)*.

Socket Addresses, Network Numbers and InterNet Addressing:

No two sockets in the same node can have the same socket number. Thus, the socket number taken together with the ABLAP node ID provides an unambiguous identifier for any socket on a single AppleBus. This is called the socket's *AppleBus address*.

This can be extended to a socket address that is unique on an AppleBus internet. This is done by assigning a unique *network number* to each AppleBus in the internet. The *internet address* of a socket consists of the socket number, the node ID (of the node in which the socket is located) and the network number (of the network on which the node is located). The internet address of a socket uniquely identifies it on the internet. Thus the source and destination sockets of a datagram can be fully specified by their internet addresses.

Network numbers are two bytes (16 bits) long. Of these the number 0 (zero) is reserved to mean unknown, i.e. the local network to which the node is connected. The value 65,535 (all bits set to one) is reserved for future use. Packets whose destination

network number is zero are addressed to a node on the local network. This allows systems consisting of a single AppleBus to operate without the need for an explicit network number. Although we do not expect our clients to build such a large internet, the two byte network number theoretically allows proper operation on an internet with up to 65,534 AppleBuses.

DDP Protocol Type Field:

The AppleBus architecture allows the implementation of a large number (up to 255) of parallel protocols at the level above DDP. It is important to realize that socket numbers are not associated with a particular protocol type and should not be used to demultiplex among parallel protocols at the transport level. Instead, for this purpose a protocol type field is provided in the DDP header. This is one byte wide and is known as the *DDP protocol type field*.

Socket Listeners:

Socket clients provide code that is said to be the socket's *listener*. This is the code that "receives" datagrams addressed to that socket. The specific implementation of a socket listener is node dependent. For efficiency, the socket listener should be able to receive datagrams asynchronously through either an interrupt mechanism or an I/O request completion routine.

The code that implements the DDP in the node must contain a data structure called a *sockets table* to maintain an appropriate descriptor of each active socket's listener.

II.2 DDP Interface

As shown in figure DDP2, the DDP interface is the boundary at which the socket client can issue calls to and obtain responses from the DDP implementation module in the node.

The DDP Implementation Module supports four calls:

(i) Open a Statically-Assigned Socket:

The caller specifies the socket number and the socket listener for that socket. The call returns with a result code:

- success: socket activated

- error: various cases: socket already active, not a statically-assigned socket (outside the range), socket table full.

(ii) Open a Dynamically Assigned Socket:

This is similar to the preceding except that the caller does not specify the socket number. The call returns a result code and, if this signals success, then the activated socket's number. The result code takes the following possible values:

- success: socket activated
- error: various cases: socket table full,
all dynamic sockets busy

(iii) Close a Socket:

This request specifies the number of the socket to be deactivated. If the socket is currently active it is removed from the sockets table. The result code has the following possible values:

- success: socket deactivated
- error: no such socket.

(iv) Send a Datagram:

The request specifies number of the source socket, the internet address of the destination socket and the DDP protocol type field value. Also the length and location of the data part of the datagram are provided in the request. Since DDP includes an optional software checksum in internet datagrams, the requester must specify whether or not this checksum is to be generated.

The result code has the following possible values:

- success: datagram sent
- error: sending socket not active or invalid,
datagram data too long.

In addition to these four calls, a socket listener must provide a mechanism for the reception of datagrams.

(v) Datagram Reception by the Socket Listener:

We do not specify this function since it is dependent on the implementation in the node. Some mechanism is needed to deliver datagrams within the node to the destination socket's listener. The DDP package should attempt this only if the destination socket is currently active. The DDP must discard datagrams addressed to inactive sockets. Furthermore, internet datagrams received with an invalid DDP checksum must be discarded.

II.3 DDP Internal Algorithm:

DDP is a simple, best-effort protocol for delivery of datagrams. As such there is no mechanism for recovery from packet loss or error situations.

Within the DDP implementation module, the primary function is to form the DDP header on the basis of the destination address, and then to pass the packet on to the appropriate LAP. Similarly, for packets received from the LAP, DDP must examine the datagram's destination address in the DDP header and route the datagram accordingly. Details of this operation depend on the nature of the node, namely whether it is a bridge or not. These are discussed in more detail in section II.5.

II.4 DDP Packet Format:

A datagram consists of the DDP header followed immediately by the data.

There is a 10-bit datagram length field in the DDP header. (See figures DDP3 and DDP4.) The value in this field is the length in bytes of the datagram starting with the first byte of the DDP header and including all bytes up to the last byte of the data part of the datagram. *Upon receiving a datagram the receiving node's DDP implementation must reject all datagrams whose indicated length is not equal to the actual received length.* The maximum length of the data component of a datagram is limited to 586 bytes. *Longer datagrams must be rejected.*

In addition, the DDP header contains the source and destination socket addresses and the DDP protocol type. Each of these addresses could be specified as a four byte internet address. However, for packets whose source and destination sockets are on the same AppleBus, the network number fields are unnecessary; likewise, for such datagrams the source and destination node IDs are repeated in the LAP header and are thus redundant in

the DDP header. For these considerations, DDP uses two types of header: a short form and an extended form.

The short form version of a DDP datagram is as shown in figure DDP3. The datagram header is five bytes long. The first two bytes of the header contain the datagram length, with the more significant byte first. The upper 6 bits of this byte are not significant and should be set to zero. The datagram length field is followed by a one byte destination socket number, a one byte source socket number, and a one byte DDP protocol type field. Such short header datagrams are sent if the source and destination sockets are on the same AppleBus.

The extended form datagram is shown in figure DDP4. The extended form DDP header is thirteen bytes long. It contains the full internet addresses of the source and destination sockets, as well as the datagram length and DDP type fields. For such packets, there is a 6-bit hop count field in the most significant bits of the first byte of the DDP header. Datagrams exchanged between sockets on different AppleBuses of an internet must use this form of header. In addition, the extended header includes a two byte (sixteen bit) DDP checksum field. If the sending client so desires, the source node's DDP implementation calculates and inserts a software-generated checksum into this field; otherwise, the sending node sets this field to 0. The datagram's destination node recomputes this checksum and rejects the datagram if the received and computed values do not agree. (see section II.7)

The DDP checksum has been provided to allow the detection of errors caused by faulty operation (e.g. memory and data bus errors) within bridges on the internet. Implementors of DDP should treat it as an optional feature.

On packets received from the LAP, DDP uses the value of the LAP type field to determine if the packet has a short or an extended DDP header. The LAP type field values for the two cases are 1 for short form, and 2 for the extended form.

Hop Counts:

For datagrams that are exchanged between sockets on two different AppleBuses in an internet, a provision is made to limit the maximum number of internet routers the datagram can visit. This is done by including, in such internet datagrams, a *hop count field*.

The source node of the datagram sets this field to zero before sending the datagram. Each internet router increments this field by one. *A bridge receiving a datagram with a hop count value of 15 should not forward it to another bridge, but should instead discard it.* This provision is made to filter out of the internet packets that might be circulating in closed routes. Such closed routes (loops) are a transient situation that can occur for short periods of time while the routing tables are being updated by the Routing Table Maintenance Protocol (RTMP). Nodes other than bridges ignore this field.

The upper two bits of the hop count currently are not used by DDP, but are reserved for future use (such as the extension of the maximum value of the hop count beyond the currently allowed value of 15).

II.5 The DDP Routing Algorithm:

A datagram is conveyed from its source to its destination socket over the internet through the bridges. The DDP implementation in the source node examines the destination network number of the datagram and determines whether the destination is on the local network or not. If it is, then the short form DDP header of Figure DDP3 is adequate and the LAP layer is called to send the packet to its destination node. If, however, the destination is not on the local network, the DDP builds the extended header of Figure DDP4 and calls the LAP to send the packet to a bridge (if there is more than one bridge, any one will do) on the local network. The bridges examine the destination network number of the datagram, and use *routing tables* to forward the datagram (through the LAPs of appropriate intervening local networks to which the bridge is connected) to subsequent bridges to get it to a bridge connected to the destination network. There the datagram is sent to its destination node through the local network's LAP.

Routing tables are maintained by bridges by using the Routing Table Maintenance Protocol (RTMP) discussed elsewhere. The routing tables indicate, for each network number in the internet, the node ID (on the appropriate local network) of the next bridge on the proper route.

Simple nodes (i.e. those that are not bridges) do not need to maintain these tables. Such nodes only need two pieces of information: the network number of the local network (THIS-NET), and the node ID of any bridge (A-BRIDGE) on the local network. This can

be done by implementing a simple subset of RTMP (called the RTMP Stub) in each such node. For nodes on systems consisting of a single AppleBus, the values of THIS-NET and A-BRIDGE are zero ("unknown").

Here is the internal routing algorithm for the DDP implementation module for a simple node (i.e. not an internet router). The sending client issues the send datagram call, specifying therein the destination socket's internet address. Then the algorithm is as follows:

If (Destination-Network-number = 0) OR (Destination-network-number = THIS-NET) then

begin

build the short-form DDP header;

call LAP to send it to the destination node

end

else

begin

build the extended-form DDP header;

call LAP to send the packet to A-BRIDGE

end

For packets received by simple nodes from the LAP, the routing function is simply one of delivering the packet to the destination socket in the node. It is advisable for such nodes to verify that the destination node ID and destination network number in an extended DDP header in fact matches the stations node ID and network numbers.

In internet routers, the algorithm is somewhat more complex. Details are provided in the discussion of the Routing Table Maintenance Protocol.

II.6 Recommendations on the use of Sockets in Conjunction with Name Binding

Implementations of AppleBus in commercial products must not rely on Statically Assigned Sockets. Apple has chosen this approach in order to provide a flexible approach for developers without the need for a central administering body.

Developers should instead use the name binding technique to allow work stations to discover their server/service access point addresses. Thus developers would identify their server/service by a unique name. Workstations would then use NBP to bind an address to this name. [For details of this process see the document on the name binding protocol (NBP)]. Once the client process has determined the proper destination socket number, it can then proceed and transmit packets to that socket.

A potential disadvantage of this approach is that developers must implement NBP in their servers. This is not significant for larger, complex servers, but assumes importance for smaller memory-bound cases. In fact NBP has been so designed that a subset is all that is needed for such servers. This subset simply responds to Name Lookup packets received over the network. This subset does not need to implement general names table management, etc., since this table contains just a single name in it.

II.7 DDP Checksum Computation

The 16-bit DDP checksum is computed as follows:

1. CkSum := 0 ;
2. For each datagram byte starting with the byte immediately following the CkSum repeat the following algorithm:
 - a. CkSum := CkSum + byte; (unsigned addition)
 - b. Rotate CkSum left one bit, rotating the most significant bit into the least significant bit.
3. If at the end, CkSum = 0, then CkSum := hex FFFF (all ones).

Reception of a datagram with CkSum := 0 implies that it is unchecksummed.

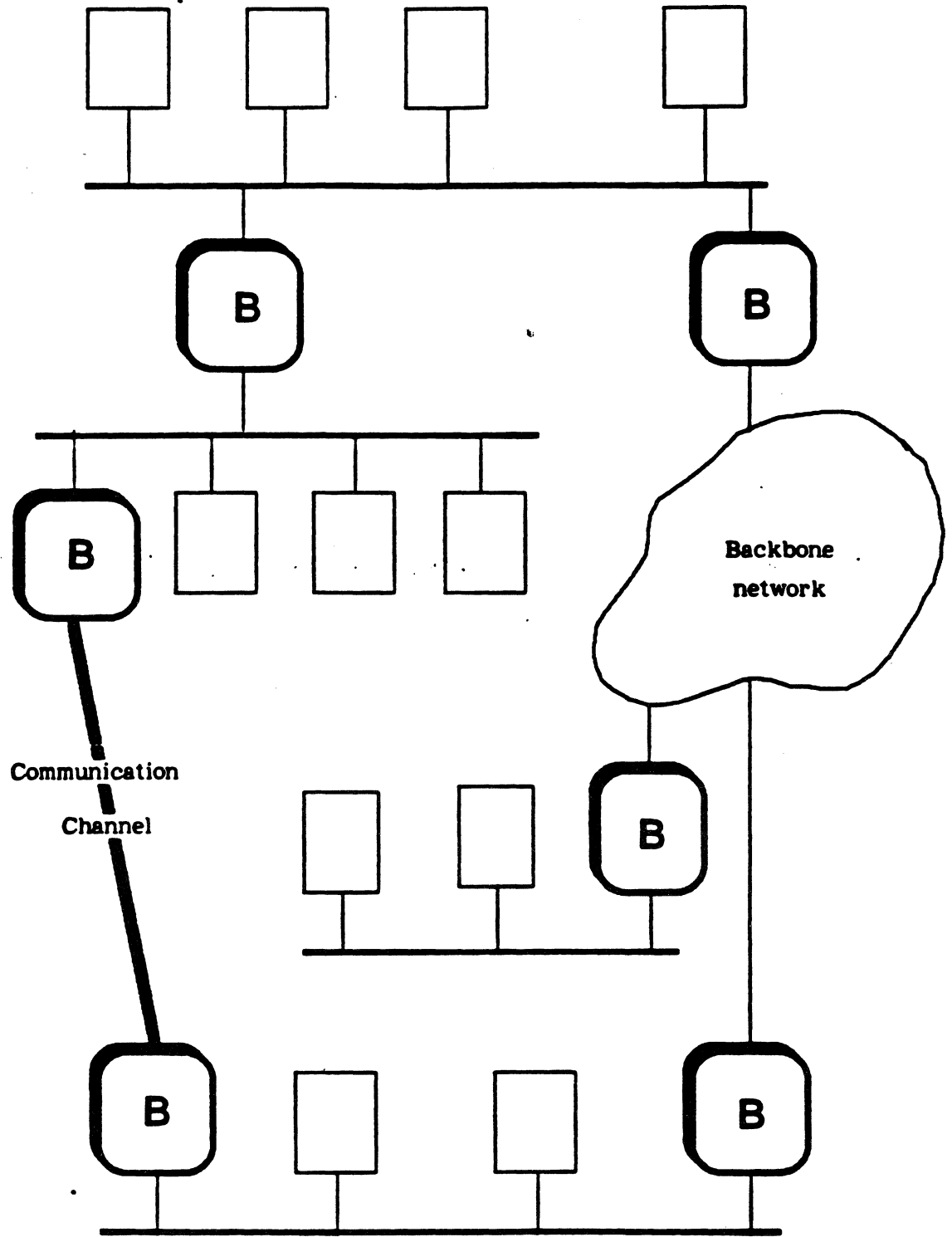


Figure DDP1: AppleBus internet and bridges/internet-routers (B)

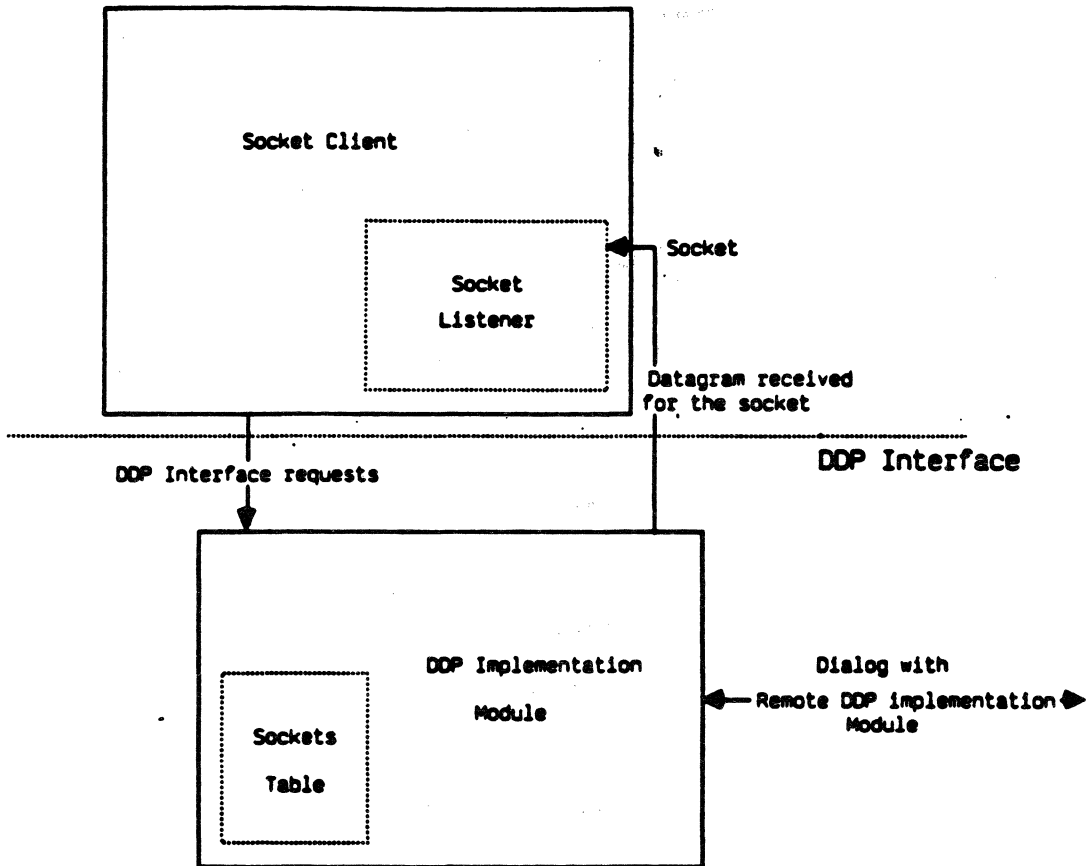


Figure DDP2: Socket Terminology

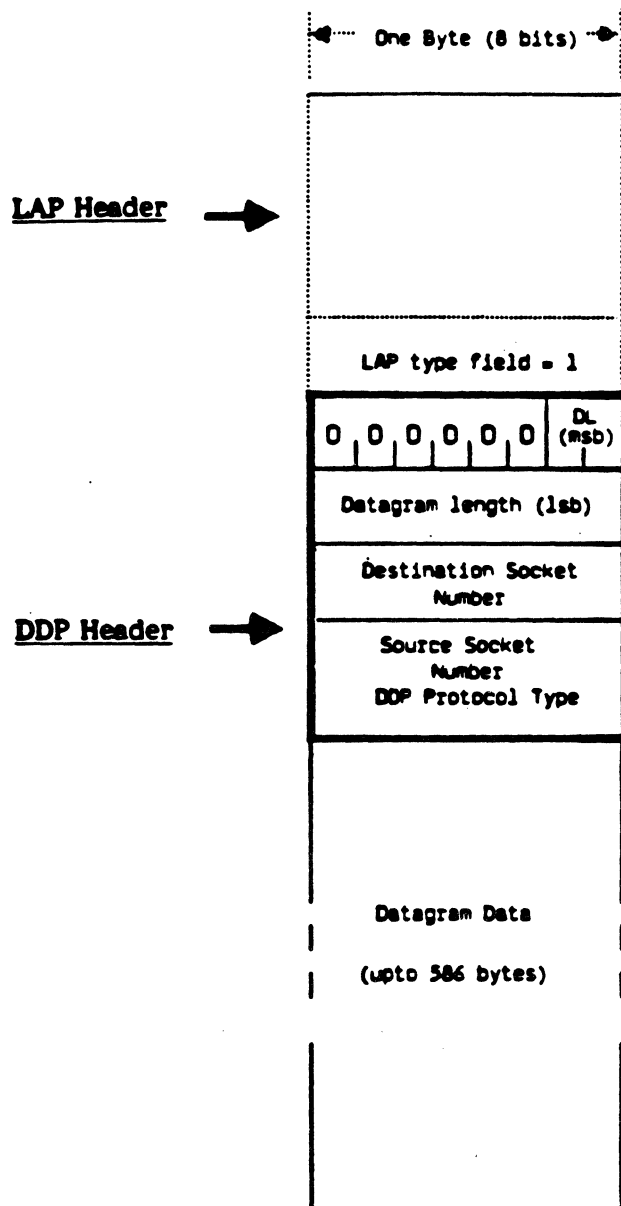


Figure DDP3: DDP Packet Format (Short header)

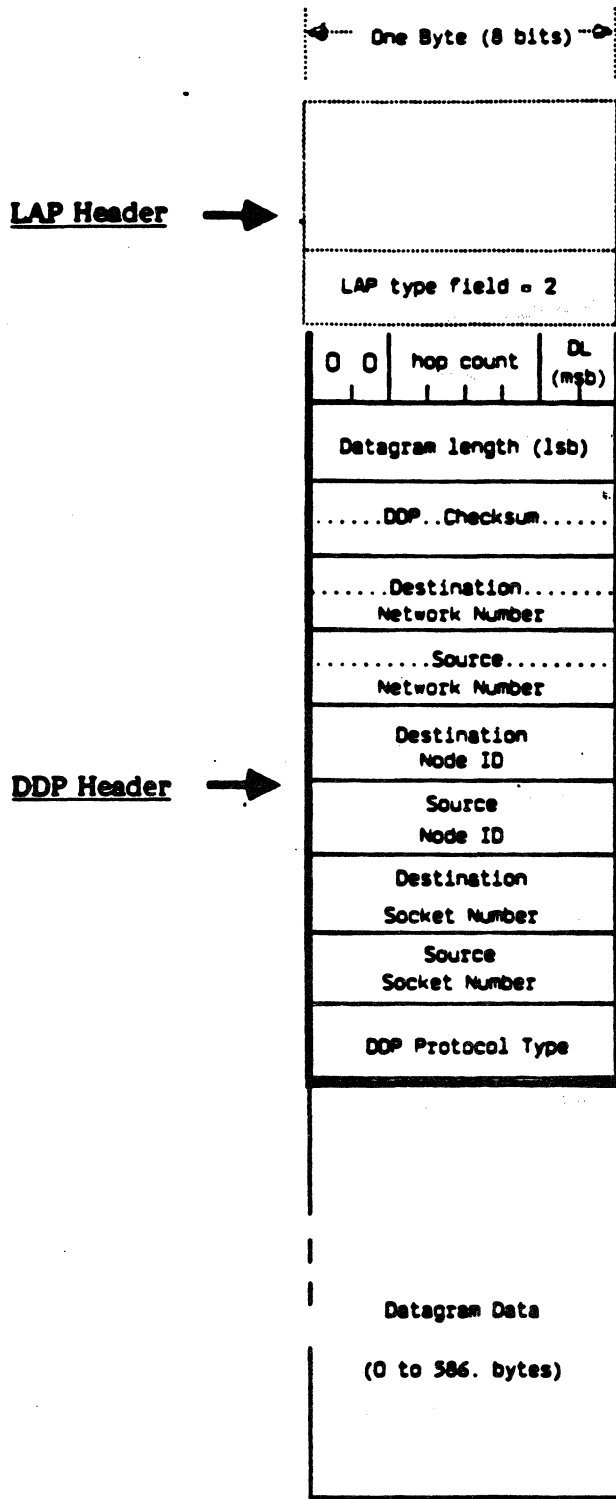


Figure DDP4: DDP Packet Format (extended header)

III. ROUTING TABLE MAINTENANCE PROTOCOL

The Datagram Delivery Protocol introduced the concept of *bridges* or *internet routers* (IR). It mentioned the mechanism by which datagrams are forwarded/routed, through intervening IRs, from any source socket to any destination socket on the internet. Mention was made of *routing tables*, maintained in the IRs, that were central to this routing mechanism.

This section describes the content, establishment and maintenance of these routing tables by means of the *Routing Table Maintenance Protocol* (RTMP). Furthermore, it is through this protocol that any AppleBus node can "discover" the number of the local network to which it is directly attached and the host ID of an internet router on its AppleBus.

III.1 Bridges — Some Terminology :

In general terms, a bridge is a device that can be connected to more than one network for the purpose of serving as a datagram forwarding agent between those networks. Bridges are thus the key components in extending the datagram delivery mechanism to an internet setting.

It is useful to distinguish the principal manners in which bridges may be used to build an internet of AppleBus systems. *Figure RTMP1* illustrates the three major situations.

(a) Local Bridges :

Configuration A shows a bridge used to interconnect several AppleBus systems that are in close proximity (such a bridge might be said to be a *local bridge*). Such local IRs are connected directly to each of the AppleBuses they serve to bridge. They are useful in allowing the construction, within the same building, of an AppleBus internet with more than 32 nodes.

(b) Half Bridges :

Configuration B illustrates the use of two bridges interconnected by a long distance communication link. Each bridge is directly connected to an AppleBus. The combination of the two bridges and the intervening link serves in effect as a bridging unit between the AppleBus systems. Each bridge in this unit will be referred to as a *half bridge*. The intervening link of

course can be made up of several devices, such as modems, and other networks, such as public data networks, etc. The primary use of half bridges is to interconnect remote AppleBus systems. An important characteristic of such bridges is that the throughput is in general lower than in the previous case, due to the lower speed of the communication link. Furthermore, usually these communication links display poorer error characteristics than the local networks of the internet.

(c) Backbone Bridges:

Although these might be placed in one of the previous categories, they present an important set of properties that make it appropriate to single them out. These bridges are used to interconnect several AppleBus systems through a backbone network. Each bridge could be a local bridge connected on one side to an AppleBus, and on the other to the backbone network. This is illustrated as configuration C. Another manner of connecting a backbone bridge to the backbone network might be through a long-distance communication link.

RTMP addresses these three important cases. A particular bridge might be a combination of two or more of the above as is indicated in configuration D. Thus the model of a bridge is general enough to allow for any of these four configurations.

III.2 Bridges — A Model:

We model a bridge as a device with several hardware ports, referred to as *bridge ports*. (see *figure RTMP2*) A bridge port can be connected in three ways: (a) directly to a local AppleBus (local bridge case), (b) to one end of a communication link the other end of which is connected to another bridge (half bridge case), (c) either directly or via a communication link to a backbone network. In our model a bridge can have any number of ports, which are numbered starting with the number 1.

Each bridge port has associated with it a *port descriptor*. This consists of three fields: (a) the *port number*, (b) the *port node ID*, ie the bridge's node ID corresponding to that port, and (c) the *port network number*, ie the network number of the AppleBus to which the port is connected. The values of these three fields are obvious for a port that is directly connected to

a local AppleBus. For a port connected to one end of a communication link (half bridge case), the port node ID and port network number are both 0 ("unknown"). For a port connected to a backbone network, the port network number is 0 ("unknown"), while the port node ID is the appropriate node ID of the bridge on the backbone network. In this latter case, provision must be made for this field to be of any size (possibly variable length) depending on the nature of the backbone net.

It is important to understand that the AppleBus node ID of a local bridge is different for each of its ports. Put another way, for each AppleBus to which it is directly connected, the bridge acquires a different AppleBus node ID.

Returning our attention to figure RTMP2, the bridge internals can be modelled as a suitable *link access process* for each port, a *router*, the *routing table*, and the *routing table maintenance process* implemented on a statically-assigned socket known as the *RTMP socket* (socket number = 1). The router accepts incoming datagrams from the LAPs and then reroutes them out through the appropriate port depending on their destination network number. This routing decision is made by the router by consulting the routing table. The routing table maintenance process receives RTMP packets, from other bridges, through the RTMP socket and uses these to maintain/update the routing table.

III.3 Routing Tables:

All bridges maintain complete routing tables that allow them to determine how to forward a datagram on the basis of its destination network number. The RTMP protocol allows bridges to periodically exchange their routing tables and thus respond to changes in the connectivity of the internet. Examples of changes are the installation or switching on of a new bridge, or its going down.

A routing table that has stabilized to all changes will consist of one entry for each reachable network in the internet. Each entry provides the number of the port through which packets for that network must be forwarded by the bridge, the node ID of the next IR/bridge, and a measure of the "distance" to the destination network. The entry in the routing table corresponds to the shortest path (known to that bridge) for the corresponding destination

network. The distance corresponding to a network to which the bridge is directly connected is always zero.

This distance to a network is measured in terms of "hops", each hop representing one IR/bridge that the packet will encounter in its path from the current bridge to the destination network. This simple measure of distance is adequate for an RTMP that responds to the connectivity of the network and adapts to changes in this aspect.

[Other modified measures could reflect the speeds/capacities of the intervening links and thus try to find a minimum time path. Yet another enhancement might use current traffic conditions on a particular path to modify its contribution to distance. This makes the routing tables adapt to traffic patterns and conditions. We have, for reasons of simplicity, chosen to use the hop-count measure. The basic algorithm remains unchanged if more complex measures are used.]

A bridge receiving the routing table of another bridge compares it with its own table and updates the latter to record the shortest path for each destination network.

Each table entry has associated with it an entry state value. This is a variable which takes on one of three values: *good*, *suspect*, *bad*. The significance of the entry state will become clear when the table maintenance mechanism is discussed in more detail.

Figure RTMP3 illustrates a typical routing table for a bridge with three ports in an internet consisting of seven networks. The corresponding port descriptors are shown in the figure as well.

III.4 Routing Table Maintenance — Conceptual description:

Bridges do not have any knowledge of the topology or connectivity of the internet. Thus, the system must provide a mechanism that allows them to construct their routing tables and to maintain them in the face of changes in the internet (bridges coming up or going down). There are two parts to this process: *initialization* of the routing table, and its *maintenance*.

When a bridge is switched on, it initializes its table by examining the port descriptor of each of its ports. Any port with a non-zero port network number signals that the bridge is directly connected to that network. Thus an entry is created in the table for that network number, with a distance of zero and with that port's number in the appropriate part of the entry. This initial table is called the *routing seed* of the bridge.

Every bridge must periodically *broadcast* one or more RTMP packets *through each of its ports*, addressed to the RTMP socket. Thus every bridge's routing table maintenance process will periodically receive RTMP packets from every bridge on its directly connected networks, backbones or communication links. The RTMP packets (see *figure RTMP4*) carry the port node ID and port network number of the bridge port through which the RTMP packet was sent, and the <network number, distance> pairs (such pairs are called *routing tuples*) of the entries in the sending bridge's routing table. With these RTMP packets the routing table maintenance process adds to or modifies its own routing table.

The basic idea is that if an RTMP packet (received by the bridge) contains a routing tuple for a network not in the bridge's table, then an entry is added for that network number with a distance one larger than the tuple's distance. In essence, the RTMP packet indicates that a route exists to that network through the RTMP packet's sender.

Likewise, if an RTMP packet indicates a shorter path to a particular network than the one in the bridge's routing table (ie if the tuple distance t plus one is less than the table entry's distance), then the corresponding entry must be modified to indicate the RTMP packet's sender as the next IR for that network with the new distance.

Clearly, this process allows for the growth and adaptation of routing tables to the addition of bridges/routes.

"Aging" Routing Table Entries :

However, if bridges die or are switched off, the corresponding changes will not be discovered through the foregoing process. To respond to such changes, the entries in the routing tables must be "aged" and, in the absence of confirmation via new RTMP packets, be declared "suspect" and later "bad". Bad entries are purged from the routing tables.

Each entry in the routing table (corresponding to a network to which the bridge is not directly connected) was obtained from an RTMP packet received by the bridge from the next IR for that network. The RTMP protocol considers such an entry to be valid for a limited amount of time only (*entry validity time*). Before starting the validity timer, the bridge goes through its routing table and changes the state of every "Good" entry to "Suspect". An entry must be revalidated from a new RTMP packet before the timer expires.

Let us suppose for instance that the next IR, for a particular entry in bridge B's routing table, dies. Then, that IR will not be sending RTMP packets, ie the entry's validity timer will expire and bridge B will not have received confirmation of the entry. At that time the entry's state is changed from "Suspect" to "Bad". Now any other RTMP packet received with path information to the network of the entry can be used to replace the entry with the new values from that packet. If no new route is discovered, then the bad entry will be deleted when the validity timer goes off a second time.

More precise understanding of this process is available from the algorithms provided later in this chapter.

III.5 RTMP Packet Format:

Each bridge's RTMP process must periodically send out its routing table information through each bridge port. This is broadcast as one or more datagrams addressed to the RTMP socket. *Figure RTMP4* illustrates the RTMP packet. Clearly, the datagram need use only the short form of the DDP header. The DDP type field is set to 1 to indicate that it is an RTMP packet. The DDP data part of the packet consists of three parts:

Sender's Network Number:

The first two bytes of the RTMP packet's DDP data is the port network number from the port descriptor (of the port through which the packet is sent by the bridge). This field allows the receiver of the RTMP packet to determine the number of the network through which the packet was received. This is thus the number of the network to which the corresponding port of the receiver is attached.

Sender's Node ID:

This field follows the sender's network number field. It contains the port node ID from the port descriptor (of the port through which the packet is sent by the bridge). To allow for the case of ports connected to backbone networks other than AppleBuses, this field must be of variable size. The first byte of the field contains the length (in bits) of the sender node's ID. This is followed by the ID itself. If the length of the ID in bits is not an exact multiple of 8 then we prefix it with enough zeros to make a complete number of bytes. The bytes of this modified ID are then packed into the ID field of the packet, most significant byte first. It is from this field that the receiver of the RTMP packet determines the ID of the bridge sending the packet.

Routing Tuples :

The last part of the RTMP packet consists of the routing tuples from the sending bridge's routing table. They are <network number, distance> pairs with two bytes for the network number and one byte for the distance.

For internets with a large number of AppleBuses, the entire routing table may not fit in a single datagram. In that case, the tuples are distributed over as many RTMP packets as are necessary.

III.6 Table Initialization and Maintenance Algorithms :

We discuss these in two parts: initialization first, and maintenance later.

Routing Table Initialization :

A bridge upon being switched on performs the following algorithm :

For each port of the bridge

**If the port network number $\neq 0$ then create an entry for that network number
with distance zero and the entry port number = the port.**

This algorithm creates an entry for each network to which the bridge is directly connected.

Routing Table Maintenance :

The bridge is assumed to have two timers running continuously. These are the *validity timer* and the *send-RTMP timer*. The events to which the bridge's RTMP process responds are :

RTMP packet is received, the validity timer expires, the send-RTMP timer expires. The corresponding algorithms are given below:

(i) RTMP packet is received through port P :

The following algorithm is executed:

If port P 's port network number is zero then

port network number := RTMP packet's sender network number;

For each routing tuple in the RTMP packet do

If there is an entry in the routing table corresponding to the tuple's network number then Update-the-Entry else Create-New-Entry;

Update-the-Entry and Create-New-Entry are as follows:

Update-the-Entry:

If (Entry-State = Bad) then

If (tuple distance < 15) then Replace-Entry

else

If Entry's distance \geq (tuple distance + 1) then Replace-Entry

else

If Entry's next IR = RTMP packet's sender node ID then

Begin

Entry distance := tuple distance + 1;

If entry distance < 16 then

Entry state := Good

else

Entry state := Bad

End;

Create-New-Entry:

Entry's network number := tuple's network number;

Replace-Entry;

Replace-Entry:

Entry's distance := tuple's distance + 1;

Entry's next IR := RTMP packet sender's ID;

Entry's port number := P;

Entry's state := Good;

(ii) Validity Timer Expires:

In this case the algorithm is as follows:

For each entry in the routing table do

Case Entry State of

Good: If entry distance $\neq 0$ then Entry state := Suspect;

Suspect: Entry state := Bad;

Bad: Delete the entry

End;

(iii) Send-RTMP Packet Timer Expires:

Now the following algorithm is executed:

If routing table is not empty then

Begin

Copy the network number and distance pairs of each Good or Suspect entry of the routing table into the routing tuple fields of the RTMP packet:

For each bridge port do

Begin

Packet's sender network number := port network number;

Packet's sender node ID := port node ID;

Call DDP to broadcast the packet through the port's LAP to the statically-assigned RTMP socket;

End

End;

III.7 How networks acquire their numbers:

One of the major problems is how to administer and assign numbers to the networks in an internet. The basic idea we are proposing is that the network numbers are set into the port descriptors of the bridge ports, and are then dynamically propagated out through RTMP to the other nodes of each network.

Not all bridges on a particular network must have the network number set into their corresponding port descriptors. The precise requirement is that at least one bridge (called the *seed bridge*) on a network should have the network number built into its port descriptor. The other bridges could have a port network number value of 0. These will acquire the correct network number by receiving RTMP packets sent out by the seed bridge.

An absolute requirement is that the bridges on a particular network should not have (in their port descriptors) conflicting port network numbers for that network. (The value zero does not cause a conflict).

III.8 What part of RTMP do ordinary nodes have to implement?

Nodes that are not bridges do not need to maintain routing tables. As noted in the chapter on DDP, these nodes only need to know the number of the network to which they are connected and the node ID of any bridge on that network. We referred to these quantities as THIS-NET and A-BRIDGE.

When such a node first comes up the values of both of these variables are zero ("unknown"). These nodes can obtain the correct values dynamically by listening for RTMP packets being sent out by the bridges on the network. For this purpose these ordinary nodes implement a very trivial RTMP process. This process sits on the RTMP socket in that node, and upon receiving an RTMP packet, copies the packet's sender network number and sender node ID fields into THIS-NET and A-BRIDGE respectively. This is done every time an RTMP packet is received. Thus although THIS-NET will stabilize to a constant value, A-BRIDGE will constantly be changing (if there is more than one bridge on the network).

An optional extension can be made that ages the values of THIS-NET and A-BRIDGE just as in the case of a bridge's routing table entries. This might be important in the case of a node on a network that has only one bridge. Now, if that bridge goes down or is switched off, the node must discover that event through the aging mechanism. When starting the node's validity timer, its THIS-NET and A-BRIDGE values, if good, are declared to be suspect. Now if the timer goes off and these values have not been confirmed from an RTMP packet then they become bad: when the timer goes off and the values are still bad then they are both reset to zero ("unknown") and marked as bad. When updating these values from an RTMP packet the values are marked as good.

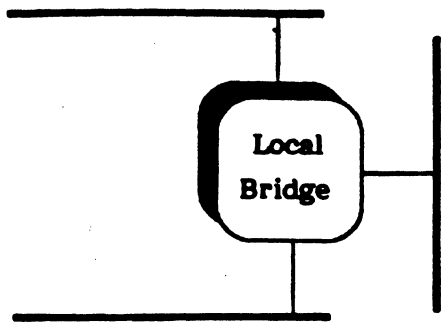
III.9 Values for the Validity and Send-RTMP Timers :

These values have a significant effect on the dynamics of the propagation of routing table adaptation in response to changes in the internet's connectivity. The exact values of these parameters will be determined by tuning actual internets and will be published at that time. For the present we propose to use a value of 5 seconds for the Send-RTMP timer and about 10 seconds for the Validity timer. Since these values do not affect ordinary nodes in any fundamental way, the impact of changes will impact bridges only.

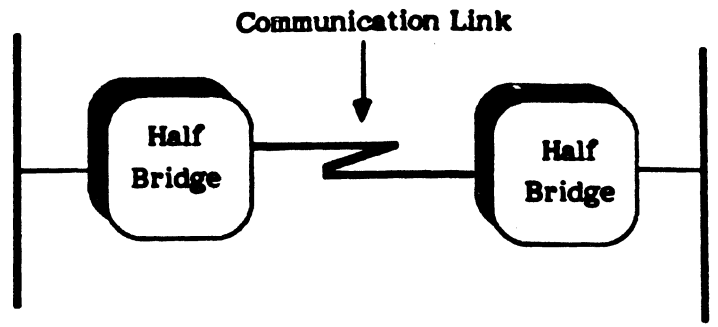
III.10 Bridges — the Routing Algorithm :

Having established the appropriate terminology we can now spell out the details of the routing algorithm used by a bridge to forward internet datagrams. This discussion applies only to the forwarding of packets received by the bridge through one of its ports, ie it does not hold for packets generated within the bridge.

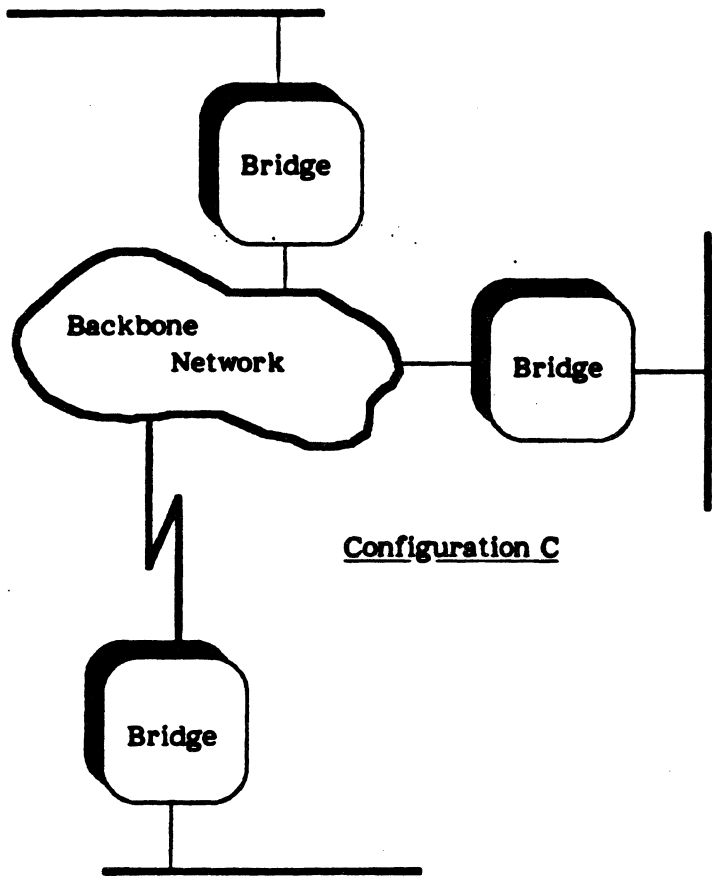
Furthermore we assume that when a packet is received through one of the bridge ports, it is tagged with the number of the port and placed in a queue. The router takes packets off this queue and executes the algorithm in the flow chart of *figure RTMP5*.



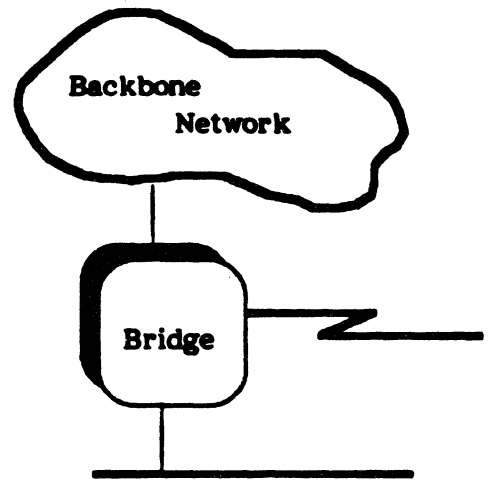
Configuration A



Configuration B



Configuration C



Configuration D

Figure RTMP1. Bridge configurations

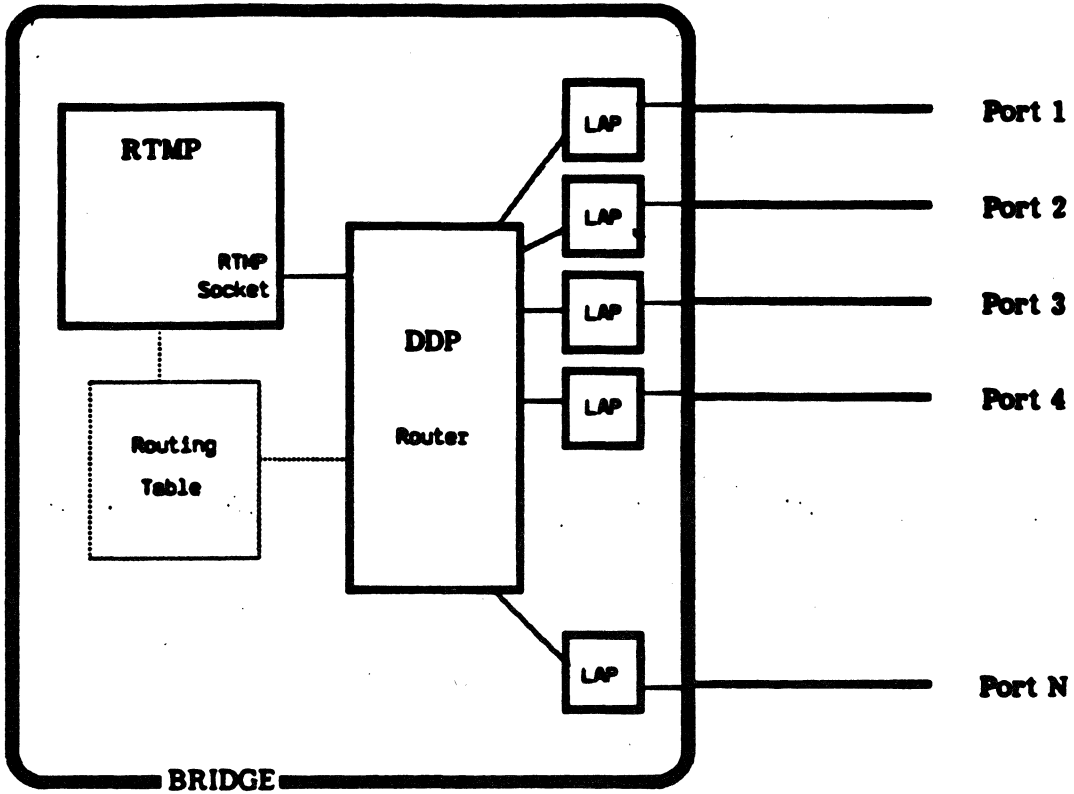
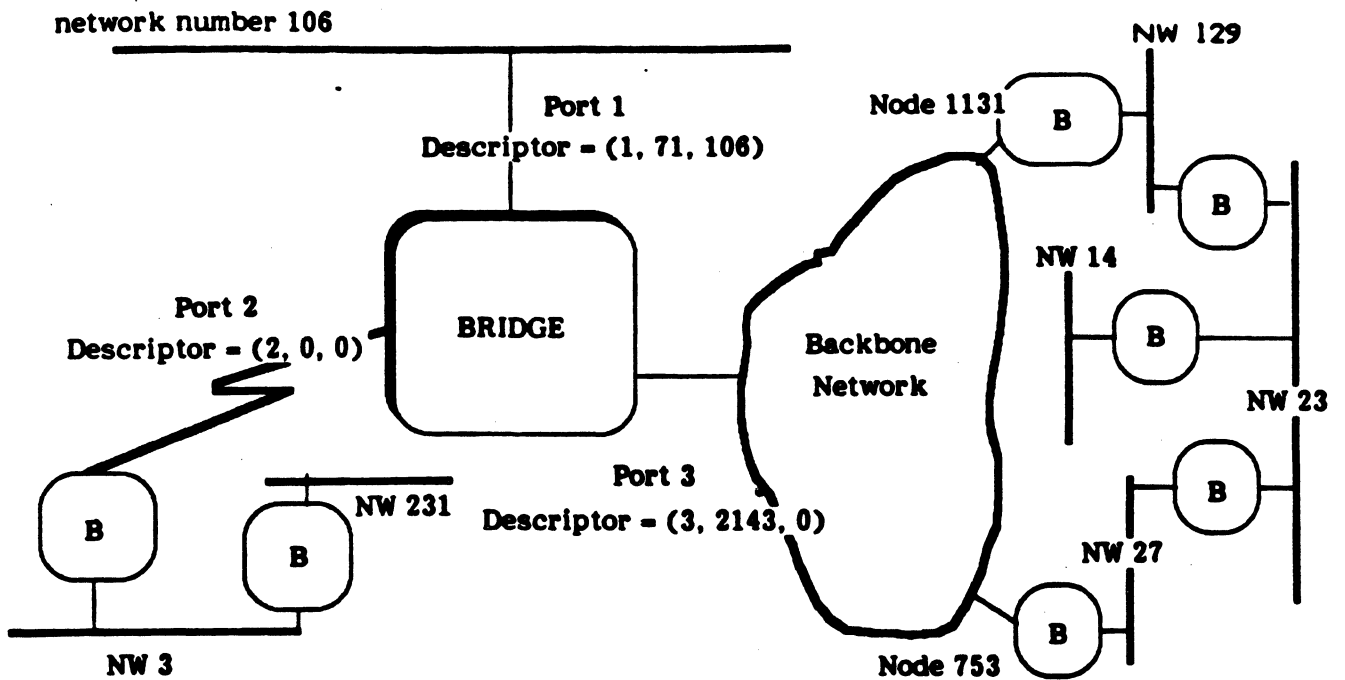


Figure RTMP2. Bridge model



Network Number	Distance	Port	Next Bridge/IR	Entry State
3	1	2	0	Good
14	3	3	1131	Good
23	2	3	753	Good
27	1	3	753	Good
106	0	1	—	Good
129	1	3	1131	Good
231	2	2	0	Good

Figure RTMP3. Example of a Routing Table

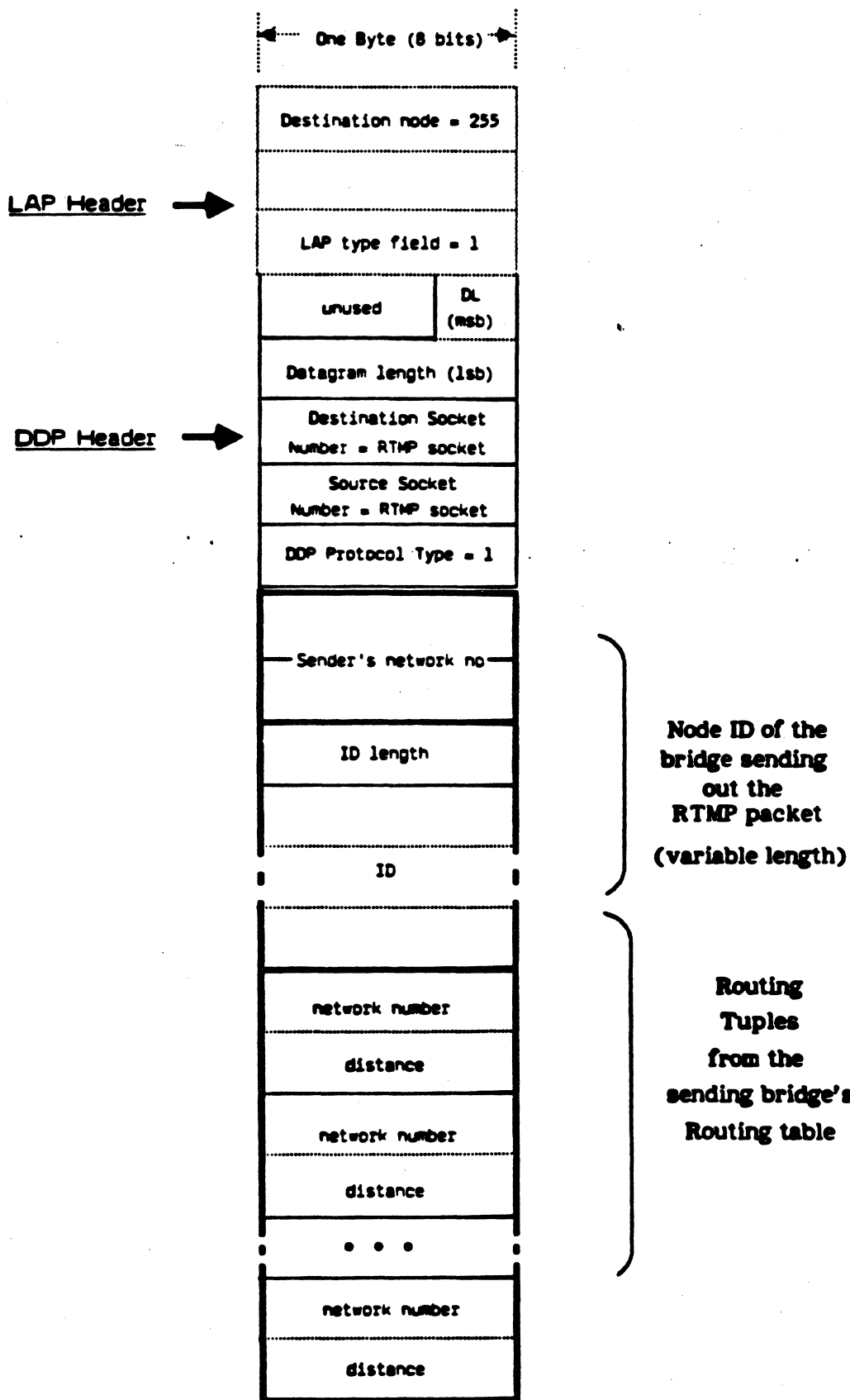


Figure RTMP4: RTMP Packet Format

IV. NAME BINDING PROTOCOL

IV.1 Terminology

AppleBus protocols rely on numerical identifiers, such as node IDs, socket numbers and network numbers, to provide the basic addressing capability essential for communication over a packet-switched network. These addresses are efficient and easily handled by the system's protocol software and hardware. However, human users of a network do not find numbers the most convenient form of identification. Numbers are hard for them to memorize and are easily confused and misused. Names are more easily used by the human user. Thus, if they are allowed to refer to objects by their names, the name must be converted into a network address for use by the other protocols. The *Name Binding Protocol* (NBP) performs this conversion.

Applebus uses dynamic node id assignment, which does not allow building in (or configuring in) addresses into software for accessing network resources. The name binding approach provides the preferable solution towards this end. (see section II.6)

Network-Visible Entities:

We start by defining the concept of a network-visible entity. In informal terms this is any entity that can be accessed over the network. More specifically, the socket clients on an internet are its *network-visible entities* (NVE).

In this context we should make it clear that the nodes of the network are not network-visible entities. Rather, any services in the nodes available for access over the network are network-visible entities. Consider for instance a print server on a network. The server is not the network-visible entity. The print service will typically be a socket client on what might be called the server's request listening socket. The latter is the server's network-visible entity.

The same distinction applies to the human users of the network. They are not

network-visible themselves. But a user may have an electronic mailbox on a mail server. This mailbox is network-visible and will have a network address. This distinction is quite valid, since the network does not provide any protocols for conversing with the individual user, but rather to certain applications/services through which the user may be accessed.

Entity Names:

Network-visible entities may assign names to themselves, though not all NVEs have names. Such a name will be called simply an *entity name* (EN). Entity names are character strings. A particular entity could in fact possess several names (*aliases*).

In addition to a name, an entity could also have certain attributes. For instance, a print server's request receiver might have associated with it a list of attributes of the printer, such as its type (daisy wheel, dot matrix, laser), the type and size of paper, etc. So the mapping of the EN into its socket address might be complemented with some form of entity attributes "look-up" service.

In defining the permissible structure of an entity name, we have decided to code the attributes into the name as a separate field, known as the *entity type*.

In addition to attributes, it might prove useful to have some logically-defined location information about the entity. For instance, a given file server might belong to a particular department or building. The users of the network should be able to select a file server on the basis of its being in the vicinity or in their department, etc. Thus, the concept of a zone has been defined by us and a *zone field* included in the entity name. A precise definition of the concept of a zone is provided later.

Thus, an *entity name* is a character string consisting of three fields: *object*, *type*, and *zone* concatenated together in this order with colon and @-sign separators. For example, Sidhu:Mailbox@Bandley3. Each field is an alphanumeric string of at most 32 characters.

Certain *meta-characters* can be used in place of explicit strings. For the object and type fields, an '=' can be substituted, signifying 'all possible values' (wildcard). For the zone

field, a '*' may be substituted, meaning the default value (the zone in which this node resides). If a network name does not contain any meta-characters, it is said to be *fully specified*. As an example, `=:Mailbox@*` refers to all mailboxes in the same zone as the information requester. Likewise, `:-@*` would mean all named entities of all types in the requester's zone. Again, `Sidhu:-@*` refers to all entities named Sidhu in the requester's zone regardless of their type.

Name Binding

Before a named entity can be accessed over an AppleBus or an AppleBus internet, the address of that entity must be obtained. This is done by a process known as *name binding*.

Name Binding may be visualized in several entirely equivalent ways: as a mapping of an entity name into its internet address, as a lookup of the address in a large data base, etc. Both characterizations are used in our discussion of NBP. [Note: we will refer to the entity's internet address. This includes the case of a single AppleBus, where the network number field will always be equal to zero ("unknown").]

Name binding can be done at various times. One strategy is to configure the address of the named entity into the system trying to access that entity. This so-called *static binding* is not appropriate for systems such as AppleBus where the node ID can change every time a node comes up.

It is useful in this context to remember that although entities can move on a network, their names seldom change. For this reason, it is best to configure names into systems, and then use services such as NBP to bind *dynamically*. This may be done when the user/accessor's node is first brought up (known as *early binding*) or just before the access to the named entity is performed (known as *late binding*). Early binding runs the risk of using incorrect (out-of-date) information when the resource is actually accessed, possibly long after the user's node was brought up. However since the binding process might be a slow one, late binding might impinge unfavorably upon the response obtained by the user when accessing the named entity. Late binding is the method most

appropriate when the entity is expected to "move" on the internet.

IV.2 AppleBus Name Binding

AppleBus Names Directory and Names Tables

Name binding is done on AppleBus by using NBP to look up the entity's address in AppleBus' *Names Directory* (ND). This is a distributed data base of entity name to entity address mappings. The data base does not require different portions to be replicated. It can be distributed among all nodes containing named NVEs. NBP places no restriction on the use of name servers, nor are such servers necessary.

The ND is the union of individual *names tables* (NT) in the nodes of the internet. Each node maintains in its names table the name-to-address mappings of *all* entities located in *that* node. As a consequence of this, name lookup performed on names corresponding to entities which reside on a node that is down or unreachable will fail.

Aliases and Enumerators

NBP allows an NVE to have more than one name. Each of these *aliases* must be included in NT as an independent entity.

To simplify and speed up the ability to distinguish between multiple names associated with a given socket, an *enumerator* value is associated with each NT entry. This is an 8-bit integer, totally invisible to the clients of NBP. Each NBP implementation can develop its own scheme for generating enumerator values to be included in its NT, subject to the condition that no two entries corresponding to the same socket have the same enumerator value.

Names Information Socket

Each node implements an *NBP process* on a statically-assigned socket (socket number = 2) known as the *names information socket* (NIS). This process is responsible for

maintaining the node's names table, and for accepting and servicing name lookup requests from within the node and from the network (through the NIS).

Name Binding Service

The name binding service consists of the following:

(a) *Name Registration:*

Any entity can enter its name and socket number into the ND (actually into its node's NT) to make itself "visible by name". This is done by using the *name registration* call to the node's NBP process.

This process must first verify that the name is not already in use. This is done by performing a name lookup in the node's zone. If the name is already in use, the registration attempt is aborted. Otherwise, the name and the corresponding socket number are inserted into the node's names table. This enters the corresponding name-to-address mapping in the ND.

When a node comes up, its names table is empty. Each network visible entity must re-register its name(s) when restarted.

(b) *Name Deletion:*

A named entity should delete its name-to-address mapping from the ND when it wishes to make itself "invisible". Reasons for doing so range from the obvious one of the entity wishing to terminate operation to sophisticated entity-specific control requirements.

The entity issues a *name deletion* call to the node's NBP process. The latter simply deletes the corresponding name-to-address mapping from the node's NT.

(c) *Name Lookup:*

Before accessing a named entity, the user (or a surrogate application) must perform a binding of the entity's name to its internet address. This is done by issuing a *name*

lookup call to the user node's NBP process. The latter then uses the NBP to perform a search through the ND for the named entity. If it is found, then the corresponding address is returned to the caller. Otherwise an "entity not found" error condition is returned.

It is possible to find more than one entity matching the name specified in the call. This is especially true when the name includes the '=' wildcard. The interface to the user must have provisions for handling this case.

Another feature of NBP is that it does not allow abbreviated names; for instance, it does not permit reference to *Sidhu:Mailbox*. The complete reference *Sidhu:Mailbox@Bandle3* or *Sidhu:Mailbox@** is required by NBP. Provisions may be made in the user interface to permit abbreviations; the interface must then "flesh out" the name before passing it on to NBP.

(d) *Name Confirmation:*

Name lookup performs a zone-wide ND search. More specific confirmation is needed in certain situations. For instance, if early binding was performed, the binding must be confirmed when the named entity is actually accessed. For this purpose, NBP has a *name confirmation* call in which the caller provides the full name and address of the entity. This call in effect performs a name search in the entity's node to confirm that the mapping is still valid.

Although a new name lookup can lead to the same result, the confirmation is much cheaper in terms of total network traffic generated. It is the recommended and preferable call to use when confirming mappings obtained through early binding.

Now we examine the binding protocol first for the case of a single AppleBus and later for the internet/multi-zone case.

IV.3 NBP on a Single AppleBus

Name searching/look-up is quite simple on a system consisting of a single AppleBus. We single out this case, since it is expected to be the most common instance of an AppleBus system, and because it serves to introduce the basic search technique.

NBP relies heavily on the ability to broadcast packets on an AppleBus. The lookup proceeds as follows.

The user issues a name registration or lookup request to the NBP process in its node. This process first examines its own names table to determine if the name is available there. If so, in the case of a registration attempt, the call is aborted with a "name already taken" result. For the case of a name lookup, the information in the names table is a partial response (there may be entities in other nodes that match the specified name).

The NBP process now prepares an NBP lookup packet (*LkUp packet*) and then calls DDP to broadcast it over the AppleBus for delivery to the Names Information Socket. Only nodes that have an NBP process will have this socket activated. In these nodes the LkUp packet is delivered to the NBP process which searches its names table for a potential match. If no match is found then the packet is ignored, else a LkUp reply packet is returned to the address from which the LkUp packet was received. This reply packet contains the matching name-to-address mappings found in the replying node's names table.

The receipt of one or more replies allows the requesting NBP process to compile a list of name-to-address mappings for the original user. If the lookup was performed in response to a name registration call, then the call must be aborted as the name is already taken.

Since broadcasts are not very reliable, the requesting NBP process sends the LkUp packet several times before returning the compiled mappings, if any, to the requesting user. If no replies are received, then it is concluded that there is currently no entity using the specified name. For a name lookup call, the user is informed of a "no such entity" outcome. For a name registration call, the requested name-to-address mapping is

entered into the node's names table.

Sending the LkUp packet several times implies that the same name-to-address mapping could be received by the requesting node several times in LkUp-Reply packets. These duplicates must be filtered out of the list of mappings. The obvious way is to see if the mapping is already in the list, i.e. compare the name strings and the address fields with each entry in the compiled list. This method is inefficient. Comparison of the four byte address fields is insufficient because of the possibility of aliases. The enumerator value together with the address resolves this problem and accelerates the duplicate filtering.

Name confirmation is similar in nature, except that the caller provides the name-to-address mapping to be confirmed. The LkUp packet is not broadcast but is sent directly to the NIS at the specified address. This can be repeated several times to protect against lost packets or the target node being temporarily busy.

IV.4 NBP on an Internet

The use of broadcast packets to perform name searching is inconvenient in internets. DDP does not allow a destination address corresponding to a broadcast to all nodes in the internet. DDP can broadcast datagrams to all nodes of any single specified network in the internet (these are said to be *directed broadcasts*). If NBP sent a directed broadcast to every network in the internet, the traffic generated would be considerable.

On AppleBus internets we provide for the formation of zones. NBP name searches are restricted to the nodes in a zone.

Zones

The AppleBus internet can be split up into *zones*. A zone consists of a subset of the AppleBuses in the internet.

The individual AppleBuses in a particular zone need not be in any way contiguous or

"neighbors". A zone is an arbitrary subset of the buses in the internet. A particular AppleBus belongs to exactly one zone. The union of all zones is exactly the internet.

The concept of zones is provided to assist the establishment of departmental or other user-understandable groupings of the entities of the internet. Zones are intelligible only to the NBP (and to the related Zone Information Protocol discussed elsewhere).

A zone is identified by a string of at most 32 characters.

Name Lookup on an Internet

Bridges participate in the name lookup protocol of an internet.

The NBP process in the requesting node prepares an NBP "broadcast request" packet (called a *BrRq packet*) and sends it to the NIS of A-BRIDGE. The NBP process in the bridge, in cooperation with the NBP processes in the other bridges of the internet, arranges to convert the BrRq packet into one LkUp packet for each AppleBus in the target zone of the lookup request. Each of these LkUp packets is then sent to the NIS socket as a directed broadcast on the corresponding AppleBus. The replies are returned to the original requester as before.

The important point is that ordinary nodes (other than bridges) do not have to know anything about zones. The process of generating a zone-wide broadcast is done by the collection of bridges. The latter must of course jointly have a complete mapping of zone names into the list of corresponding AppleBuses. The establishment and maintenance of these lists is the purpose of the Zone Information Protocol discussed elsewhere.

IV.5 NBP Interface

Four calls provide to the user all the functionality of name binding. They are as follows :

(1) REGISTER-NAME:

This call is used by an NBP client to register an entity name and its associated socket address. Except for a default zone specification, meta-characters are not allowed in the name. If a '*' is not used, the zone field must be the same as the default.

Call Parameters:

entity name,
socket number

Returned Parameters:

outcome code: success
failure (name conflict, invalid name or socket)

(2) REMOVE-NAME:

This call removes an entity name from the node's names table. Meta-characters are not allowed in the name.

Call Parameters:

entity name

Returned Parameters:

outcome code: success
failure (name not found)

List of fully specified entity names and their corresponding AppleBus addresses

(3) LOOKUP-NAME:

This call performs the mapping between entity name and address. Meta-characters are allowed in the name to make the search as general as needed. It is possible for more than one AppleBus address to match the call's entity name and be returned. The corresponding fully-specified name is returned to the caller as well.

Call Parameters:

entity name

maxMatches

Returned Parameters:

outcome code: success

failure (name not found)

List of fully specified entity names and their corresponding
AppleBus addresses

The parameter maxMatches is a positive integer that specifies the maximum number of matching name-to-address mappings needed. This is useful if wildcards are used by the caller in the entity name parameter.

(4) CONFIRM-NAME:

This call confirms a caller-supplied mapping between entity name and address. Meta-characters are not allowed in the name.

Call Parameters:

entity name.

socket address

Returned Parameters:

outcome code: success (mapping still valid)

failure (mapping invalid)

IV.6 NBP Packet Formats

All NBP packets use a DDP type field value of 2.

There are three different types of NBP packets: BrRq, LkUp and LkUp-Reply. The format is as shown in Figure NBP1. The various fields of the NBP part of these packets are as follows:

Control

The most significant four bits of the first byte are used to indicate the type of NBP packet. The values are 1 for BrRq, 2 for LkUp and 3 for LkUp-Reply.

Tuple Count

The packets contain name-address pairs called *NBP tuples* (see figure NBP2). The least significant four bits of the first NBP byte contain a count of the number of tuples in that packet.

BrRq and LkUp packets carry exactly one tuple (with the name being looked up or confirmed). The tuple count field for these packets is always equal to 1.

NBP ID

In order to allow for multiple pending lookup requests from a given node, an 8-bit ID is generated by the NBP process issuing the BrRq or LkUp packets. The LkUp-Reply packets must contain the same NBP ID as the LkUp or BrRq packet to which they correspond.

Requester's Address

BrRq and LkUp packets include the 4-byte address field of the NBP tuple, which provides the complete internet address of the requester. This field allows the responder to properly address the LkUp-Reply datagram.

NBP Tuple

The NBP tuple consists of an entity's name and its 4-byte internet address and a one-byte enumerator field. The address field appears first in the tuple. The fifth byte in a tuple is the enumerator field, followed by the entity name. This consists of three string fields: one each for the object, type and zone names, in that order. Each of these strings consists of a leading 1-byte string length followed by up to 32 string bytes. The string length represents the number of bytes/characters in the string. The three strings are concatenated without any intervening padding.

Enumerator Field

This field is included to handle the situation where more than one name has been registered on the same socket. It should be noted that NBP specifically permits this use of aliases (or alternately, the use of a single socket by more than one NVE).

In this case, each alias is given a unique enumerator value, kept in the NT along with the name-address mapping. The enumerator field is not significant in a LkUp or BrRq packet, and is ignored by the recipient of these packets. (Recall that the address part of the tuple, in this case, is the requester's address).

In a LkUp-Reply packet, the correct enumerator value must be included in each tuple. This value is used for duplicate filtering.

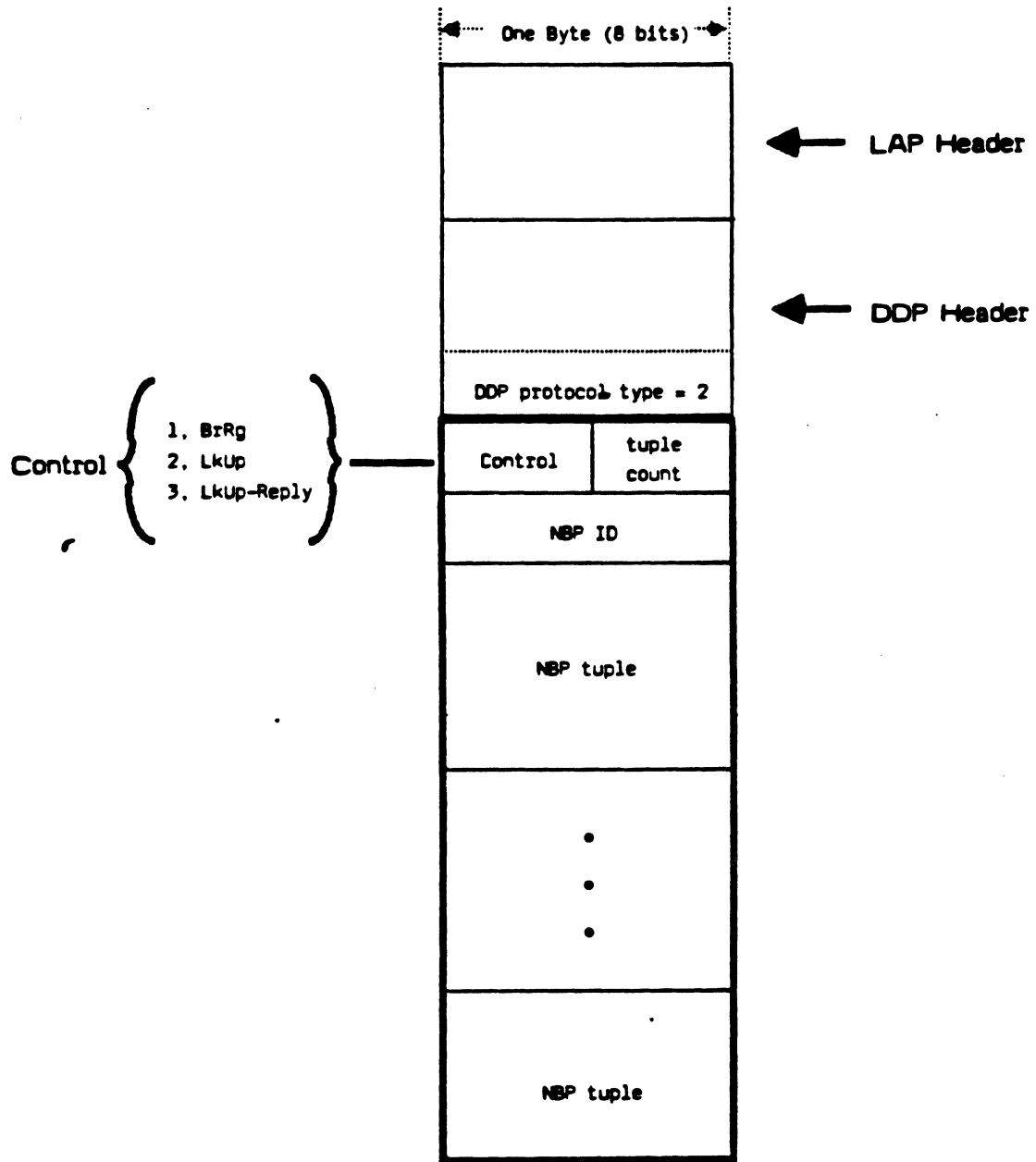


Figure NBP1: NBP Packet Format (LkUp-Reply packets)

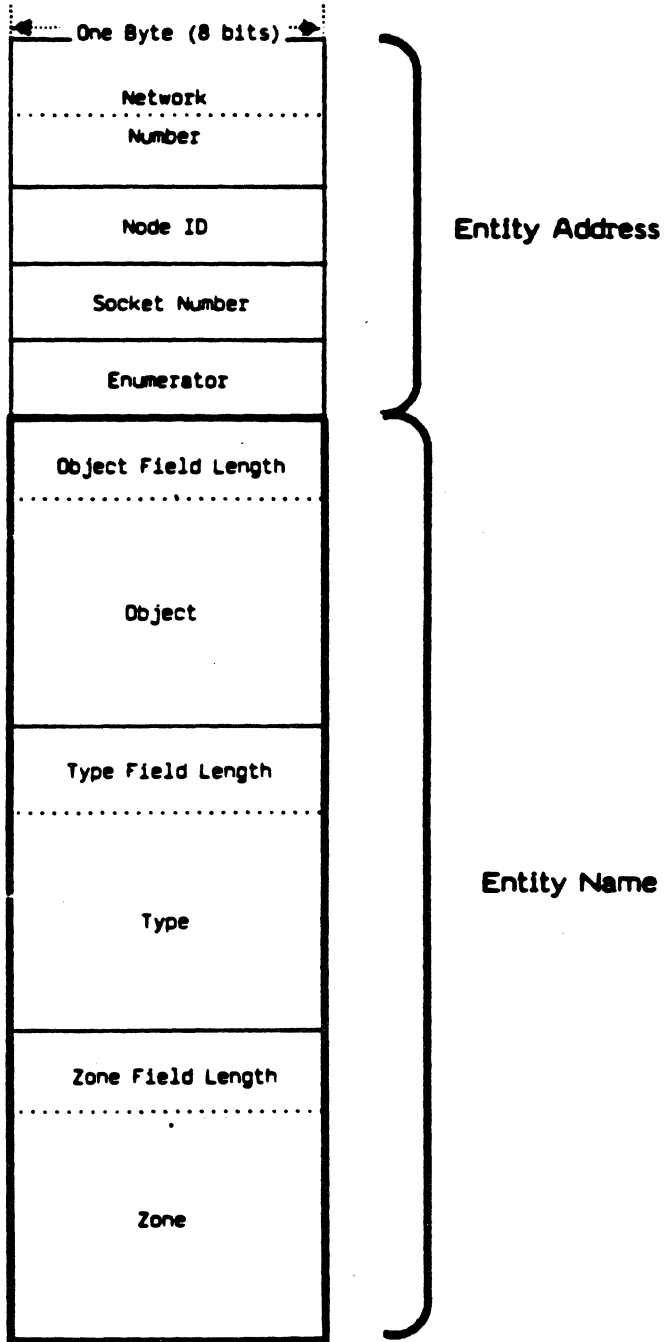


Figure NBP2: NBP Tuple

VI. APPLEBUS TRANSACTION PROTOCOL

The AppleBus Transaction Protocol (ATP) is a transport protocol that adequately fulfills the transport needs of the vast majority of peripheral devices, and satisfies the transaction needs for more general networking on AppleBus. Our goal has been to strike a compromise between features needed for networking and for peripheral device control. Particular attention has been paid to simplicity and ease of implementation, so that nodes with tight memory space restrictions will be able to support a sufficient subset of ATP.

VI.1 Introduction

The fundamental purpose of reliable transport protocols is to provide loss-free delivery of client packets from source socket to destination socket. Various features can be added to this basic service, to obtain the service characteristics appropriate for its specific needs. Transport characteristics can be added by clients or built into standard value-added transport services. ATP is such a value-added service.

Transactions - Terminology:

Frequently, a socket client must request the client of another socket to perform a particular higher level function and to report the outcome to the requester. This interaction, consisting of a *request* and a *response*, is termed a *transaction*.

The basic structure of a transaction, in the context of a network, is illustrated in Figure ATP1. A transaction has a *requesting end* and a *responding end*. The requesting end initiates the transaction by sending a *transaction request*, TReq, from the requesting end's socket to the responding end's socket. The responding end executes the request and returns a *transaction response*, TResp, reporting the transaction's outcome.

At-Least-Once Transactions:

This basic process must be performed in the face of various situations inherent in the loosely-coupled nature of networks, for example:

- (i) TReq is lost in the network;
- (ii) TResp is lost or delayed in transit;
- (iii) the responding end "dies" or becomes unreachable from the requesting end.

In the first place, there could be several transaction requests outstanding, and the requesting end must be able to distinguish between the respective responses received by it over the network. This can be done by using a transaction ID (TID), sent with the request. The response must contain the same ID as the corresponding request. The TID, in a sense, "binds" together the request and the response portions of a transaction.

In every one of the above three error situations, the requesting end will not receive the transaction's response and must conclude that the transaction was not completed. A recovery procedure must then be activated *at the requesting end*. This consists of a timeout and an automatic retry mechanism. If the timer expires in the requesting end and the response has not been received, the requester retransmits the TReq. This process is repeated until a response is received or until a retry count maximum is reached. If the retry count hits its maximum value then it is concluded that the responding end is either "dead" or unreachable, and the transaction requester (the ATP client at the requesting end) is so notified.

This mechanism does its best to ensure that the transaction request is executed at least once. Such a mechanism is adequate if the request is essentially *idempotent*, e.g. repeated execution of the request is the same as executing it once. [An example of an idempotent transaction is asking a destination station to identify itself].

Exactly-Once Transactions:

Nevertheless, with this recovery mechanism, lost or delayed responses could cause the transaction to be executed more than once. If the request is not idempotent, serious damage could result. For such requests it is desirable to have a transaction protocol

which ensures transaction execution once and exactly once.

Whether the at-least-once or the exactly-once level of service is appropriate can only be determined by the transaction requester.

The basic technique for implementing an exactly-once transaction protocol is as follows. The responding end maintains a transactions list of all recently received transactions. Upon receiving a TReq it searches through this list to determine if the request has already been received (duplicate transaction-request filtering). A new request is inserted into the list before being executed (and then a response is sent out). Upon receiving a duplicate request for which a response has already been sent, the responder retransmits the response.

Upon receiving a TResp the requesting end should return a transaction release (TRel) packet to release the request from the responding end's transactions list. If this TRel gets lost then the request would stay in the list "forever". To eliminate this situation, the responding end "timestamps" a request before inserting it in its list. The list is checked periodically and requests that have been in it for too long are eliminated.

VI.2 AppleBus Transaction Protocol — Multi-Packet Responses

ATP uses the basic idea that a transaction is a request issued by a client in a requesting node to a client in a responding node. The client in the responding node is expected to service the request and generate a response. It is assumed that the two clients have some method of unambiguously identifying the data/operation sought in the request (eg. read a disk block, block number, etc.).

This is a very simple model, in principle sufficient for all interactions. The difficulty is that the underlying network places a size restriction on the packets that may be exchanged. Thus, for instance, the transaction response might not fit in a single packet. For this reason we look upon the transaction request and response as "messages" (not packets). Although ATP restricts its transaction requests to single packets, it allows the transaction response message to be made up of several packets, which of course bear

a sequential relationship to one another. When the requesting node receives all the response packets (i.e. the complete response message), the transaction is considered complete and the response is delivered as a single logical entity to the ATP client (the transaction requester).

ATP supports both 'at-least-once' and 'exactly-once' modes as client-elected options.

If the 'at-least-once' case is chosen then, ATP handles timeouts and retransmission of requests but does not attempt to prevent duplicate requests from being passed to the transaction responder. In this case, if request actions are not idempotent, it is up to the responding client to filter duplicates.

The maximum size (number of packets) of a transaction response message is limited to 8 packets.

Transaction IDs:

Transaction IDs are generated by the requesting end and sent along with the TReq packet. An important design issue is the size (16 bits for ATP) of these IDs. This is a function of the rapidity with which transactions are generated and of the *maximum packet lifetime* (MPL) for the complete network system. The longer the MPL the larger the TID must be. Similarly, if transactions are generated rapidly, then the IDs must again be larger. The basic problem is that the TID being of finite size wraps around and there is the danger that for a particular value an old packet stored in some internet router may arrive later on and be accepted as a valid packet.

For a single AppleBus, the time taken for exchanging a TReq and a TResp is bounded (by the ABLAP) to be greater than about 2.5 msec. Thus there can be no more than 400 transactions per second. From this point of view a single byte TID would allow half a second or so for wraparound of the TID.

However, with network interconnection through store-and-forward internet routers.

the impact of MPL (of the order of 30 seconds) makes a one-byte TID inadequate. A 16-bit ID would increase the wrap around time to be approximately two minutes. This obviates all concerns about old retransmitted transaction requests and responses "sneaking-in" due to wraparound.

The ATP BitMap/Sequence Number:

Every ATP packet includes in its header a *bitmap/sequence number*. This field is 8 bits wide. ATP handles lost or out-of-sequence response packets by using this bitmap. The significance of this field depends on the type of ATP packet (TReq, TResp or TRel).

In TReq packets this field is known as the *transaction bitmap*. The basic idea behind the use of this field is that the requesting end reserves enough buffers for the expected transaction response, and then sends out the TReq packet indicating to the responder the number of buffers reserved. This is done by setting a bit in the TReq packet's bitmap, for each reserved buffer. The responding end can then examine the TReq packet's bitmap and determine the number of packets the requester is expecting to receive in the transaction response message.

In TResp packets this field is known as the *ATP sequence number*. The value of this field in the TResp packet is an integer (in the range 0 to 7), indicating the sequential position of that packet in the transaction response message. The requesting end can use this value to put the received response packet in the appropriate response buffer (even if the response packet is received "out-of-sequence") for delivery to the transaction requester (ATP client). Furthermore, the requesting end clears the corresponding bit in its transaction bitmap.

The actual transaction response message may turn out to be smaller than was expected by the requester. Thus a provision is made in the response packet's header to signal "end-of-message" in the last response packet when it is sent out by the responder. Upon receiving a response packet with the end-of-message indication, the requesting end must clear all bits in the transaction bitmap corresponding to higher sequential positions. If the requesting end's retry timeout expires and the complete transaction response has

not been received as yet (indicated by one or more bits still set to one in the requester's transaction bitmap), then a TReq is sent out again with the current value of the transaction bitmap and the same TID. Thus only the missing transaction response packets are requested again.

This mechanism is illustrated in Figure ATP3 where a requesting end issues a TReq indicating that it has reserved six buffers for the response. For instance, the request might be for six blocks of information from a disk device. The TReq packet would have in its ATP data part the pertinent information: what file, which six blocks, etc. ATP builds the request packet and sets the least significant six bits in the bitmap. When the responder receives this request packet, it examines the bitmap and thus determines the range of the host's request to be serviced. The six blocks are fetched from disk, passed to the ATP layer in the device node and sent back to the host, each in a separate packet with its sequence number indicating the position in the response.

The example illustrated in the figure assumes that the third response packet is lost in the network. Thus the retry timeout will expire in the requesting end, which then retransmits the original request (transparently to the ATP requesting client) but with a bitmap reflecting only the missing third response packet.

Notice that single packet request-response transactions are simply the degenerate case in which the transaction request has only one bit set in its bitmap. If two nodes wish to communicate in this manner, very little extra packet overhead is added by the protocol.

Responders with Limited Buffer Space:

A potential difficulty with exactly-once transactions is that a responder might not have enough buffer space to hold the entire transaction response message until the end of the transaction (i.e. receipt of a TRel).

ATP provides a mechanism for such responders to reuse their buffers, through a confirmation of response packet delivery. This is done by piggy-backing, in a response packet, a request to Send Transaction Status (STS). The requesting end, upon receiving such a response packet, just immediately send out a TReq with the current bitmap (i.e. indicating which response packets have been received correctly). The responding end

can then use this bitmap to free buffers holding already delivered response packets.

Figure ATP4 illustrates this with an example in which the responding end, with two buffers, services a request for a seven packet response.

TReq packets sent in response to an STS do not consume the retry count, but do reset the Retry TimeOut.

VI.3 ATP Details

Details of the ATP protocol are provided in three parts: the packet format, the ATP interface, and the mechanism (state/action model of the ATP protocol package).

ATP Packet Format

The format of an ATP packet is illustrated in figure ATP5. It consists of an 8-byte ATP header plus up to 578 ATP data bytes.

The first byte of the ATP header is used for *command/control information (CCI)*. The two most significant bits of the CCI are the packet's *function code*. These two bits are encoded as follows:

01	—	TReq packet
10	—	TResp packet
11	—	TRel packet.

The *EOM bit* is set in a TResp packet to signal that it is the last packet in the transaction response message. The *XO bit* must be set in all TReq packets that pertain to the exactly-once mode of operation of the protocol. The STS bit is set in TResp packets to force the requester to retransmit TReq immediately. The remaining three bits of CCI must always be 0.

The 8 bits immediately following the command/control field contain the *ATP bitmap/sequence number*, with the most significant four bits in the first byte of the ATP header. The packets comprising the transaction response message are assigned sequence

numbers 0 through 7. The sequence number (encoded as an integer) is sent in the ATP bitmap/sequence number field of the corresponding response packet.

In the case of a transaction request packet, a bit of the bitmap is set to one for each expected response packet. The least significant bit corresponds to the response packet with sequence number 0, up through the most significant bit which corresponds to sequence number 7.

The third and fourth bytes of the ATP header contain the 16-bit *transaction ID*. TIDs are generated in the ATP requesting end, and are incremented from transaction to transaction as unsigned 16 bit integers (the value zero is permitted).

The last four bytes of the ATP header are not examined by ATP, but contain user data. As such, they should, strictly speaking, not be considered part of the ATP header. However, they can be used by the ATP clients to build a simple header for a higher level protocol. The reason they have been separated out is to allow an implementation of ATP that handles the complete ATP response message's data in an assembled, contiguous, form, without interposed higher level headers. ATP-Client interfaces must build appropriate mechanisms for exchanging these four user bytes independently of the data.

ATP Interface

The interface to ATP is made up of the five calls described below. It is convenient to visualize the *ATP package* (an implementation of ATP) as consisting of two parts: the *ATP Requesting end* and the *ATP Responding end*. The calls are discussed in the context of each end.

It is not appropriate in this specification of the protocol to detail the interface, as several aspects are implementation dependent. The description below is in general terms, adequate for establishing the characteristics of the ATP service to the next higher layer.

Remember that we neither assume nor require the availability of a multiprocessing environment in the network node. Our descriptions of the various interface calls have

been written in a generic form indicating parameters passed by the caller to the ATP implementation, and results and outcome codes returned by the latter. The result codes and their interpretation depend on the specifics of the implementation of a call. If the call is issued synchronously, the caller is blocked until the calls operation has been completed or aborted. then the returned parameters become available when the caller is unblocked. In the case of asynchronous calls, a call completion mechanism is activated when the operation completes or aborts. Then the returned parameters become available through the completion routine mechanism.

We envision at least two kinds of interfaces: a packet-by-packet passing of response buffers to and from ATP, and a response message (i.e. in a contiguous buffer) interface. These are analogous to the familiar packet stream and byte stream interfaces available for data stream protocols. Details of both types of ATP interfaces for Apple systems will be made available in application notes.

1. ATP Requesting End Calls:

Only one call is defined for use by the transaction requester. This call is processed by the ATP requesting end.

(1.1) SEND-ATP-REQUEST:

The transaction requester (ATP client) issues this call to send a transaction request. It must supply several parameters with the call. These include the address of the destination socket, the ATP data part of the request packet, buffer space for the expected response packets, and whether the exactly-once mode of service is required or not. In addition the caller can specify the duration of the retry timeout to be used and the maximum number of retries.

Call Parameters:

Transaction mode (exactly-once or at-least-once)

Transaction Responder's address (network number, node ID
and socket number)

Call Parameters (cont.):

ATP request packet's data part and its length

Expected number of response packets

Buffer space for the transaction response message

Retry timeout in seconds

Maximum number of retries

Returned Parameters:

Outcome code (success, failure)

Number of response packets received

The following situations return an outcome code of failure:

- the caller requested exactly-once service, but the responding end does not support it;
- ATP has exhausted all retries and a complete response has not been received.

An outcome code of success is returned whenever a complete response message has been received. A complete response is said to have been received if either of the following occurs: (i) all response packets originally requested have been received, or (ii) all response packets with sequence number 0 to some integer n have been received and packet n had the EOM bit set.

In either case, the actual number of response packets received is always returned to the requesting client. A count of zero should indicate that the other end did not respond at all. In the case of a nonzero count, the client can examine the response buffer to determine which portions of the response message were actually received and to detect missing pieces for higher level recovery, if so desired.

2. ATP Responding End Calls:

There are four calls that a transaction responder (ATP client) can make to the ATP responding end.

(2.1) OPEN-ATP-RESPONDING-SOCKET:

An ATP client uses this call to instruct ATP to open a socket (well-known or dynamically assigned) for the purpose of receiving transaction requests. If well-known, the client passes the socket number to ATP; otherwise the dynamically assigned socket number is returned to the client.

When opening this socket, the client is in effect opening a "transaction listening socket". The call allows the socket to be setup so that requests are accepted from a specified network address (provided in the call). This address can include a zero in the network number, node ID, or socket number fields to indicate that any value is acceptable for that field.

Call Parameters:

Transaction listening socket number (if well-known)
Admissible transaction requester address (network number,
node ID, and socket number)

Returned Parameters:

Local socket number (if dynamically assigned)
outcome code (success, failure)

Note that this call does not set up any buffers for the reception of transaction requests. That is done by issuing the Receive-ATP-Request call.

(2.2) CLOSE-ATP-RESPONDING-SOCKET:

This call is used to close a socket opened with an OPEN-ATP-RESPONDING-SOCKET call.

Call Parameters:

Transaction listening socket number

Returned Parameters:

outcome code (success, failure)

(2.3) RECEIVE-ATP-REQUEST:

The transaction responder issues this call to set up the mechanism for actual reception of a transaction request through an already-opened transaction responding socket.

Call Parameters:

Local socket number on which to listen

Buffer for receiving the request

Returned Parameters:

the received request's ATP data

Transaction ID

Transaction Requester's Address (network number, node ID and socket number)

Bitmap

XO indication

(2.4) SEND-ATP-RESPONSE

When a transaction responder has finished servicing the request, it issues a SEND-ATP-RESPONSE call to send out one or more response packets. ATP will send out each response buffer with the correct transaction ID and a sequence number indicating the position of the particular response packet in the response message. There are many different ways of implementing the call; our description is generic.

Call Parameters:

Local socket number (the responding socket)

Transaction ID

Destination Address (Network number, node ID and socket number)

Transaction response message/packets (ATP data part)

descriptors to determine the sequence numbers of the response packets

Call Parameters (cont.)

EOM and STS control

Returned Parameters:

outcome code: (success, failure)

ATP State Model

Management of STS will be discussed in a future version of this document. The following description provides a sufficiently precise statement of the protocol internals for the purpose of implementing the protocol. It is not a formal specification, but an aid for protocol implementers.

The description is presented in terms of the actions to be taken in response to all possible events. [This description is subject to modification as we gain more experience through actual implementation of the protocol: it is the current state of our understanding of the protocol and will not be widely different in subsequent updates.] Certain special terms are employed in the description, and we start with a discussion of some of these.

First of all, the ATP requesting end must maintain all information necessary for retransmitting an ATP request and for receiving its responses. This is referred to by us as the *Transaction Control Block* (TCB). More specifically, this would contain all the information provided by the transaction requester in the SEND-ATP-REQUEST call, plus the TID, the request's bitmap and a response packets received counter. With each request and thus with each TCB we associate a retry timer, used to retransmit the request packet in order to recover from loss of request or response packet situations.

In the second place, the ATP responding end must maintain a *Request Control Block* (RqCB) for each RECEIVE-ATP-REQUEST call issued by a client in that node. This call would contain the information provided by that call including all pertinent data as to the buffers and delivery-to-client mechanism (implementation dependent).

A third data structure, the *Response Control Block* (RspCB) is needed only in nodes

implementing the exactly-once mode of operation. It holds the information needed to filter duplicate requests and to retransmit response packets in response to these duplicates. We associate a release timer with each RspCB. This timer is used to release the RspCB in the event that the release packet sent by the requesting end is lost.

1. ATP Requesting End:

SEND-ATP-Request call issued by a Transaction Requester in the Node:

- a. Validate the following call parameters:
 - Number of response packets should be at most 8;
 - the ATP Request's Data should be at most *MaxATPData* bytes long.

If either parameter is invalid, then reject the call.

- b. Create a TCB:
 - insert the call parameters in it
 - clear the response packets counter.
- c. Generate a TID:
 - this is the last used TID plus one modulo 2^{16} :

Save the TID in the TCB.
- d. Generate the bitmap for the request packet and save it in the TCB.
- e. Prepare the ATP header:
 - insert the TID and the bitmap;
 - set the function code bits to 01;
 - {only for systems implementing the exactly-once mode of operation} if the caller requested exactly once mode the set the XO bit.
- f. Call DDP to send the ATP request packet.
- g. Start the request's retry timer.

Retry timer expires:

- a. If Retry Count = 0 then:
 - set outcome code to 'failure';
 - notify transaction requester (the client in the node) of the outcome;
 - destroy the TCB.

- b. If Retry Count \neq 0 then:
- decrement the retry count;
 - change the bitmap in the ATP request's header to the current value in the TCB;
 - call DDP to retransmit the request packet;
 - start the retry timer.

ATP Response Packet received from the network (i.e. from DDP):

- a. Use the packet's TID and source address to search for the TCB;
- b. If a matching TCB is not found then ignore the packet and exit.
- c. If a matching TCB is found then check the packet's sequence number against the TCB's bitmap to determine if this response packet is expected. The packet is expected if the bit corresponding to the response packet's sequence number is set in the TCB's bitmap. If the packet is not expected then ignore it.
- d. If the response is expected then:
 - clear the corresponding bit in the TCB bitmap;
 - set up response packet's ATP data for delivery to the transaction requester;
 - increment the response packets counter in the TCB.
- e. If the packet's EOM bit is set then clear all higher bits in the TCB bitmap.
- f. If the TCB bitmap = 0 then: {a complete response has been received}
 - cancel the retry counter;
 - set the outcome code to 'success';
 - {only if exactly-once mode is implemented} if the transaction is of exactly-once mode (determined by examining the TCB) then call DDP to send a TRel packet to the responding end;
 - notify the transaction requester;
 - destroy the TCB.

2. ATP Responding End:

OPEN-ATP-Responding-Socket call issued by a Transaction Responder in the Node:

- a. If caller specifies a well-known socket then call DDP to open that socket else call DDP to open a dynamically assigned socket.
- b. If DDP returns with error then set outcome code to the error.
- c. If DDP returns without error then:
 - set outcome code to 'success';
 - save socket number and the admissible transaction requester address in an ATP responding sockets table.
- d. Return to caller.

CLOSE-ATP-Responding-Socket call issued by a Transaction Responder in the Node:

- a. Call DDP to close the socket.
- b. Release all RqCBs, and, for systems supporting exactly-once mode of operation, release all RspCBs (and cancel all release timers), if any, associated with the socket.
- c. Delete the socket from the ATP responding sockets table.

RECEIVE-ATP-REQUEST Call issued by the Transaction Responder in the node:

- a. If the specified local socket is not active then return to caller with error.
- b. Create a RqCB and attach it to the socket.
- c. Save the call's parameters in the RqCB.

SEND-ATP-Response call issued by transaction responder in the node:

- a. If the local socket is invalid OR response data lengths are invalid then return to caller with error.
- b. {only if exactly-once mode is implemented} Search for a RspCB matching the call's local socket number, TID, and transaction requester address (destination of the ATP Response). If found then save the response attached to the RspCB (for potential retransmission in response to duplicate requests received subsequently), and restart release timer.
- c. Send the response packets, setting the ATP header of each with function code 10, with the caller supplied TID, with the correct sequence number for the packet's sequential position in the response message, and with the EOM flag set in the last response packet.

Release timer expires: {only if exactly-once mode of operation is implemented}

- a. Destroy the RspCB and release all associated data structures.

ATP Request Packet (TReq) received from DDP:

- a. {only if exactly-once mode is implemented} If the packet has its XO bit set and a matching RspCB exists then:
 - retransmit all response packets;
 - restart the release timer;
 - exit.
- b. If a RqCB does not exist for the local socket or if the packet's source address does not match the admissible requester address in the RqCB then ignore the packet and exit.
- c. {only if exactly-once mode is implemented} If packet's XO bit is set then create a RspCB and start its release timer.
- d. Notify the client about the arrival of the request and destroy the corresponding RqCB.

ATP Release Packet (TRel) received from DDP: {only if exactly-once mode is implemented}

- a. Search for a RspCB that matches the packet's TID and source address. If not found then ignore the release packet.
- b. If a matching RspCB is found then:
 - destroy the RspCB and all release all associated data structures.
 - cancel the RspCB's release timer.

VI.4 Some Optional ATP Interface Calls :

In certain instances the clients of ATP might use certain contextual information to enhance their use of ATP through some additional interface calls. Here are two examples.

Release-RspCB:

In the first case two clients of ATP are communicating with each other, using the exactly once mode, and have decided to have at most one outstanding transaction at a time. Client A calls ATP to send a TReq packet to client B. B sends back the response. A upon receiving the response sends out a second request (but no release packet). The second request packet upon being received by B signals that the response to the previous request has been received by A. Now B could simply call its ATP responding end and ask it to release the previous transaction's response control block. This needs a Release-RspCB call to the ATP responding end. The parameters of this call are fairly obvious in this instance.

Release-RqCB:

Another instance arises when a client A wishes to send data to client B. A must first inform B of this intention, and thus allow B to request the data. To this end, A can send a TReq to B to signal "I want to write N bytes of data to you, please ask me for it on my socket number S". Instead of sending a TResp to this packet, B could just send a TReq to A's socket S asking for the data. The reception by A, on socket S, of B's request implies that A's original request has been received by B. A could call its ATP requesting end and ask it to eliminate the previous transaction's RqCB. This needs a Release-RqCB call to the ATP requesting end.

Details of these calls are implementation-dependent and do not affect the ATP protocol *per se*. We mention these as interesting variants that might be implemented by certain clients who wish to reduce the traffic generated on the network.

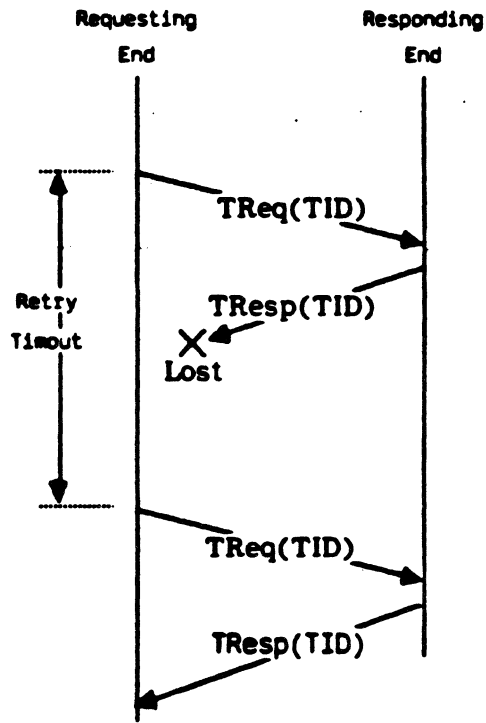
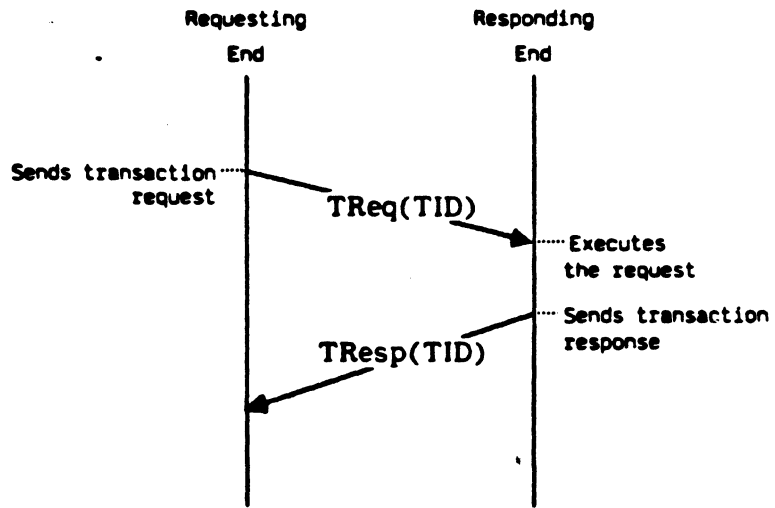


Figure ATP1: Transaction Terminology

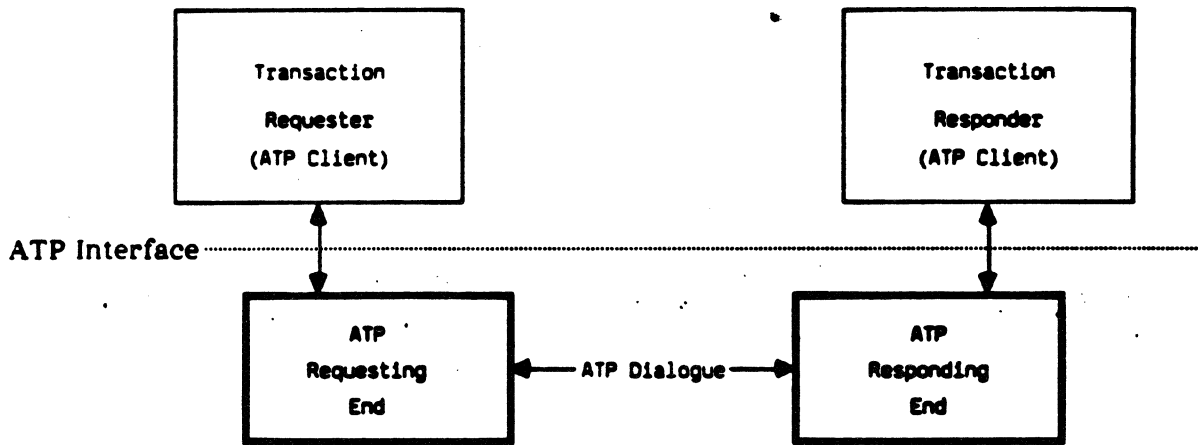


Figure ATP2: ATP Terminology

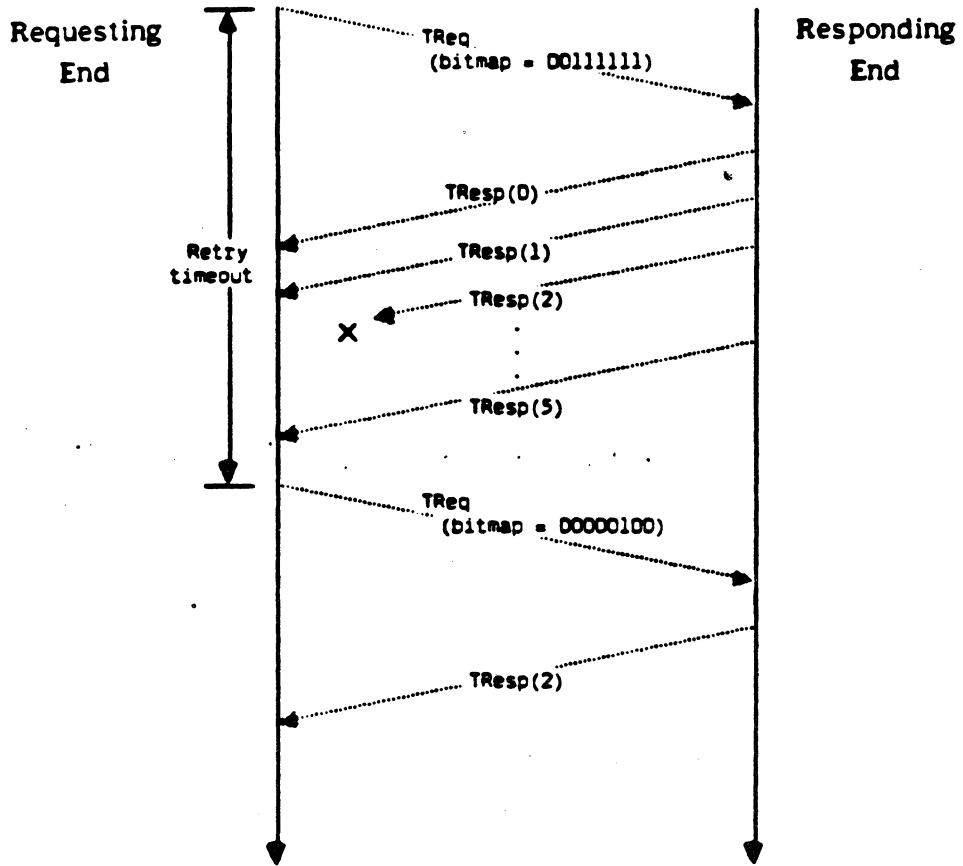


Figure ATP3 : Multiple-Packet Response

Requesting
End

Responding
End
(has only two
buffers)

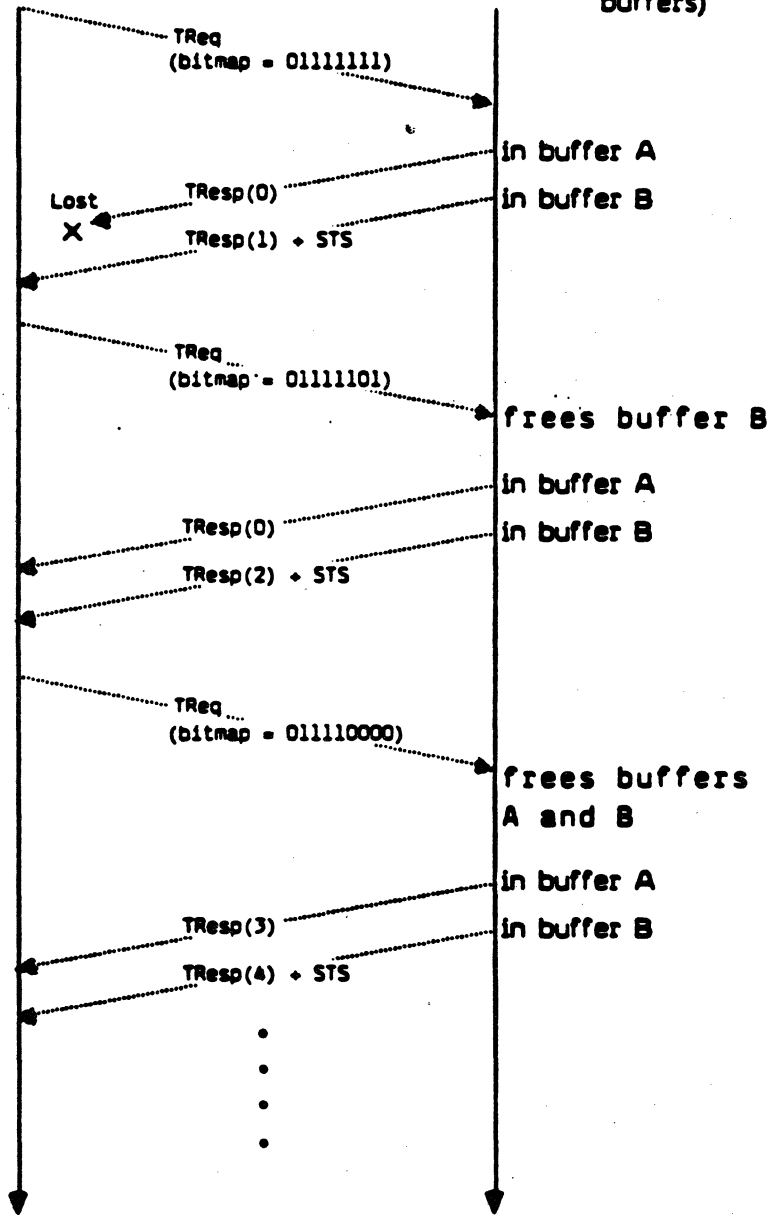


Figure ATP4 : Use of STS

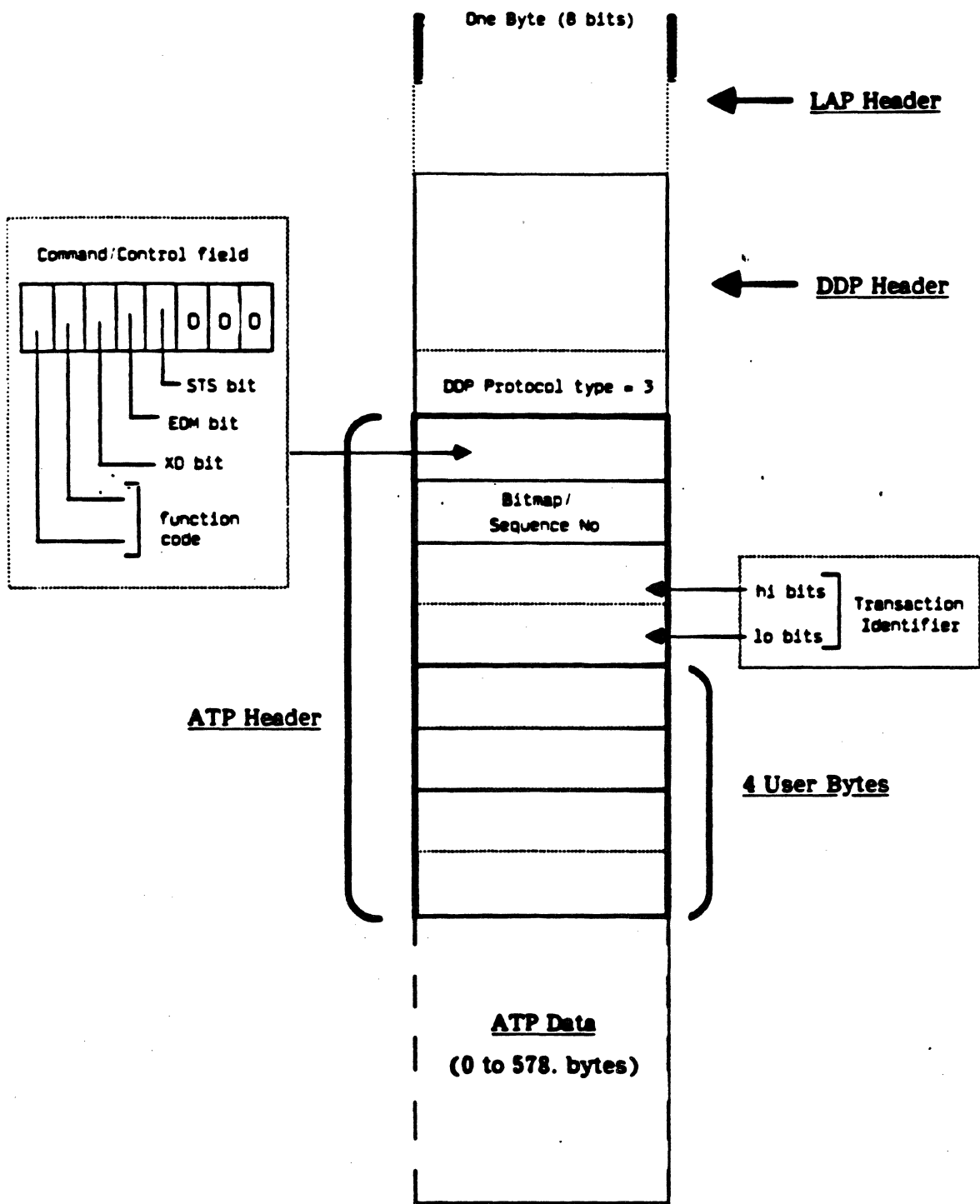


Figure ATP5. ATP Packet Format

VII. DATA STREAM PROTOCOL

The other important transport protocol is for the transfer of sequences of packets or more generally sequences of data bytes. We group this into the single term *data stream*.

This is an area in which an enormous amount of work has been done in the networking community under the general terminology of connection-oriented protocols, virtual circuits, liaisons, etc. A wide consensus has been arrived on the basic structure of such systems. Certain characteristics that stand out are the need for a symmetric protocol allowing full duplex data streams between a given pair of sockets, management of flow control by using credits sent by the intended receiver to the sender, sequencing by the use of sequence numbers in each datastream packet, connection establishment through handshakes that coordinate the initial state of the two connection ends (and hence the connection).

VII.1 Terminology:

We start by establishing the significance of certain terms and concepts. In particular, the ideas of a connection, of connection ends and of connection IDs and their significance play a central role in a data stream protocol.

Connections and Connection Ends:

A *connection* is an association between a pair of sockets that allows the reliable, full-duplex flow of packets between them without loss or duplicate packet delivery. Packets are delivered in the same order that they were inserted into the connection. Furthermore, there is a flow control mechanism built into the protocol that regulates the sending of packets according to the availability of buffers at the receiving end. Other terms used to describe connections are *virtual circuits*, *liaisons*, etc.

Connections can be set up at any time by either or both of the communicating parties, and are torn down when they are no longer required or if either connection end "dies". In essence setting up a connection involves establishing descriptors of various state and control variables at each end and bringing these into a synchronous state. A generic term

that covers both the communicating socket and the descriptor associated with it is *connection end*.

Opening and Closing Connections:

Connections are associations (between two sockets) that are set up and torn down. This can be done in many ways that are, strictly speaking, not a necessary part of the data stream protocol but more of a session layer issue (we discuss this at the end of this chapter).

A connection end can be in one of two states: *unestablished* and *established*. If both ends of a connection are established then the connection is said to established or open.

Data can flow on a connection only if it is established.

We will place the restriction that there can be only one connection between a given pair of sockets.

Connection IDs:

It might appear from the foregoing that a connection could be identified by the pair of socket addresses (e.g. node IDs and socket numbers of the connection ends). This works well as long as we are dealing with a single network with no alternate routes and without intermediate store-and-forward processing. Otherwise, since connections might be set up and torn down between a given pair of sockets, packets from a previous invocation could show up in a later invocation and possibly be accepted. This is not a common situation but is not as rare as might seem at first glance. For this reason, each connection end when it is set up generates a connection ID. The two connection IDs taken together with the socket address pair uniquely identify the connection. The size of a connection ID is a function of the rapidity with which connections are expected to be setup and broken down (ie how quickly will the ID wrap around) and the maximum packet lifetime. For our situation an 8-bit connection ID seems ample.

Sequence Numbers, Acknowledgement Numbers and Windows:

All packets sent on the connection carry a *sequence number*. This is used to ensure their in-sequence delivery, and duplicate packet filtering. Lost packets can be retransmitted using a timeout mechanism. Packets received correctly can be

acknowledged by sending back an *acknowledgement number*, which is the sequence number of the next packet the end expects to receive over the connection. The size of these numbers is related to how rapidly packets will be sent out (again the issue of wrap around) and the expected maximum packet lifetime on the network.

Flow control is done on the basis of credits or a *reception window* supplied by the receiving end of a flow. This is a value that indicates how many packets beyond the one indicated by the acknowledgment number are welcome. This need not be equal to the number of available buffers, though that would be the simplest strategy. The acknowledgment number plus the reception window value (minus one) is the maximum sequence number authorized for use by the sending end.

VII.2 DSP Interface:

The basic DSP calls are SEND-on-DSP, RECEIVE-from-DSP. [Connection opening and closing calls are discussed in section VII.6]. It is assumed that when a connection is opened DSAP (part of the *Data Stream Access Protocol*) returns to the connection client a *Connection RefNum* which is used to identify it in the SEND and RECEIVE calls. Buffers for the connection are provided by the client when the connection is opened.

(1) SEND-on-DSP:

The caller provides the Connection RefNum, and DSP data length and pointer. A function flag is provided to allow the caller to signal a logical end at the end of that packet (thus the data stream can be interpreted by the receiver as having messages that end with these logical end marks) and for the purpose of sending connection interrupts to the other end.

(2) RECEIVE-from-DSP:

This is a means for polling the DSP module (or setting up a completion routine address) to see if any packets have been received on the connection. DSP returns the buffer address and DSP data length, as well as the logical end or connection interrupt indication. RECEIVE-from-DSP also serves the other important function of supplying additional receive buffers to DSP (or returning buffers in which received packets have been delivered to the client by DSP).

VII.3 DSP Packet Format:

Figure DSP1 illustrates the DSP packet format. The header consists of one byte source and destination connection IDs ("source" refers to the source of the packet), a one byte sequence number of the packet, a one byte acknowledge number, a four bit window field, and a four bit control field. The MS bit of the control field is set to indicate a DSP control packet, ie one that does not carry any client data. The Request ACK bit can be set to force the destination to send a packet for the purpose of providing the latest values of its acknowledgement number and window value.

A packet that has its DSP control bit set must not carry data and cannot have the logical or interrupt bits set. Such a packet does not consume the sequence number; the sending end sends the next packet with the same sequence number.

If the DSP control bit is not set the packet contains client information, usually data in the DSP data part of the packet, though it could be just an interrupt or logical end indication.

VII.4 DSP Dialogue and Error Recovery Mechanisms:

The protocol dialogue in the normal situation is quite straightforward. Once a connection has been opened (see section VII.5), either end can send data packets to the other. Receiving ends of such packets can send out ACKs at any time though the sending end can force the return of ACKs by setting the Request ACK bit. Control packets can be sent at any time to elicit such information as well.

The sequence number at the sending end is increased every time a non-Control packet is sent out for the first time (as opposed to retransmissions). From the window parameter sent by the other end, the sending end obtains a maximum permissible value of the sequence number (equal to the received acknowledge number plus the received window value minus one); client requests to send a packet when this value has been reached are rejected.

Lost Packet and Out-of-Sequence Recovery:

Each end has a RCV.NXT value which is the sequence number of the next packet it expects to receive from the other end. When a packet is received its sequence number is compared with this value. A mismatch causes the received packet to be discarded. A

packet received with the correct sequence number is accepted and if it is a non-Control packet then the RCV.NXT value is incremented. Also, the receiving end maintains a value of its receive window RCV.WND. This grows and shrinks depending on the availability of receive buffers.

Recovery from lost or out-of-sequence situations is resolved by using a *retransmit timer* at each end. Every time this timer expires, all packets sent by that end but not yet acknowledged are retransmitted. The reception of a packet with acknowledge number A_n causes all packets with sequence numbers through $A_n - 1$ to be taken off the send queue.

Half-Open Connections:

Another situation is that of one of the ends "dying" or becoming unreachable from the other. The connection is said to be half-open. In such a situation, the sending end would keep on retransmitting the packets in its send queue and would needlessly consume bandwidth. Even in the absence of traffic, resources are tied up at this end in the form of connection descriptors, timers, etc. This situation is handled by using a connection timer, started when the connection is opened. If the timer expires and no activity has been seen, then a probe (a control packet) is sent to the other end. Failure to receive a response is taken to mean that the other end is "dead" or unreachable and the connection is torn down.

VII.5 Data Stream Access Protocol:

(to be described in a later version)

VII.6 Implementation Notes:

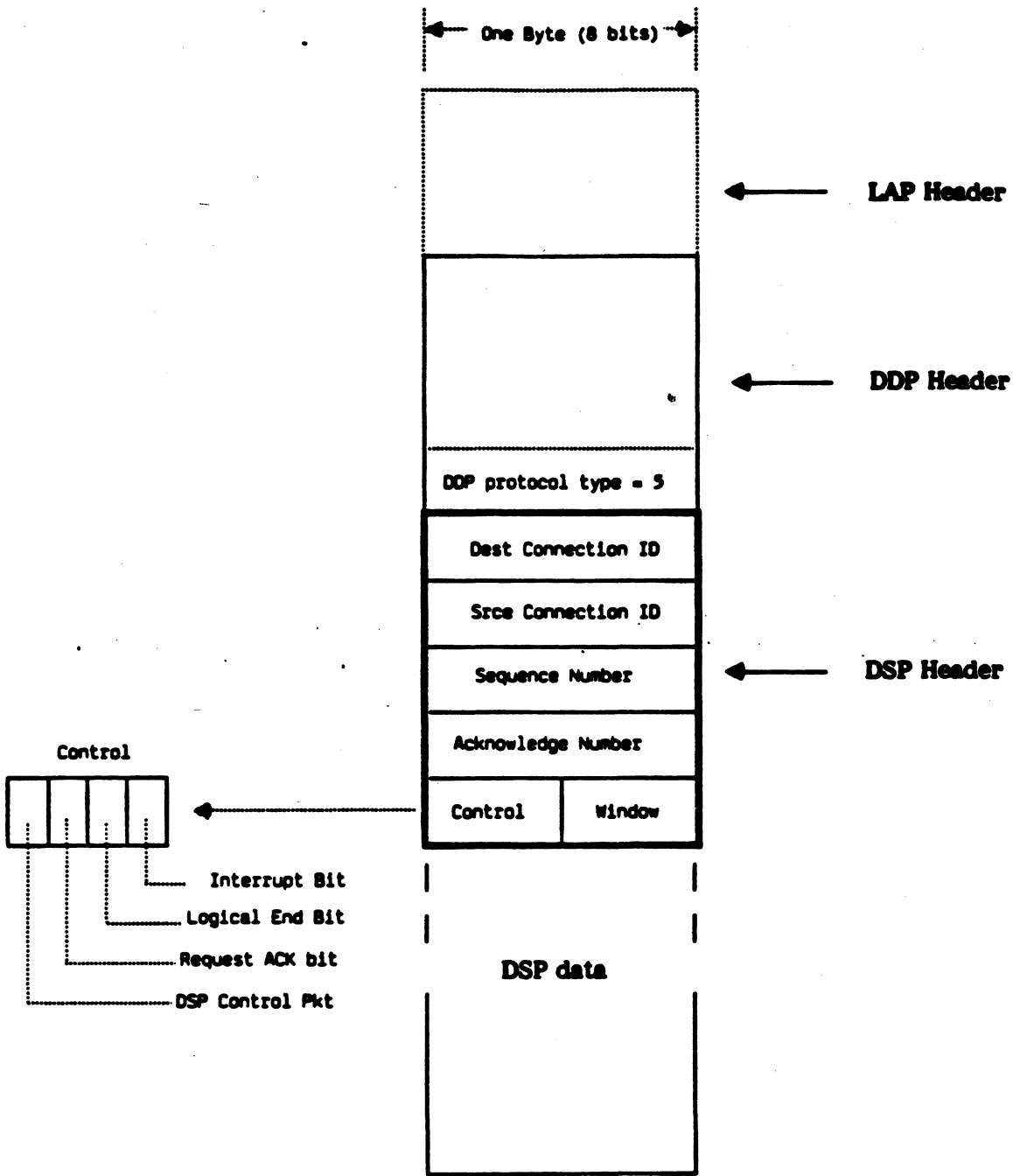


Figure DSP1. DSP Packet Format

AppleBus Protocols Architecture

- **Key Considerations/Goals
and Overall Functions**
- **LAP Type Field**
- **Datagram Service**
- **Internets**
- **Named Entities**
- **Reliable Transport**

**Gursharan Sidhu
July 27, 1984**

Key Considerations in Design of AppleBus Protocol Architecture

1. Open System

Protocols can be accessed and added at any level

2. Versatility

Not designed for a specific purpose. Rather,
rich enough to be essentially general purpose

3. Simplicity

Protocols have been made as simple as possible to:

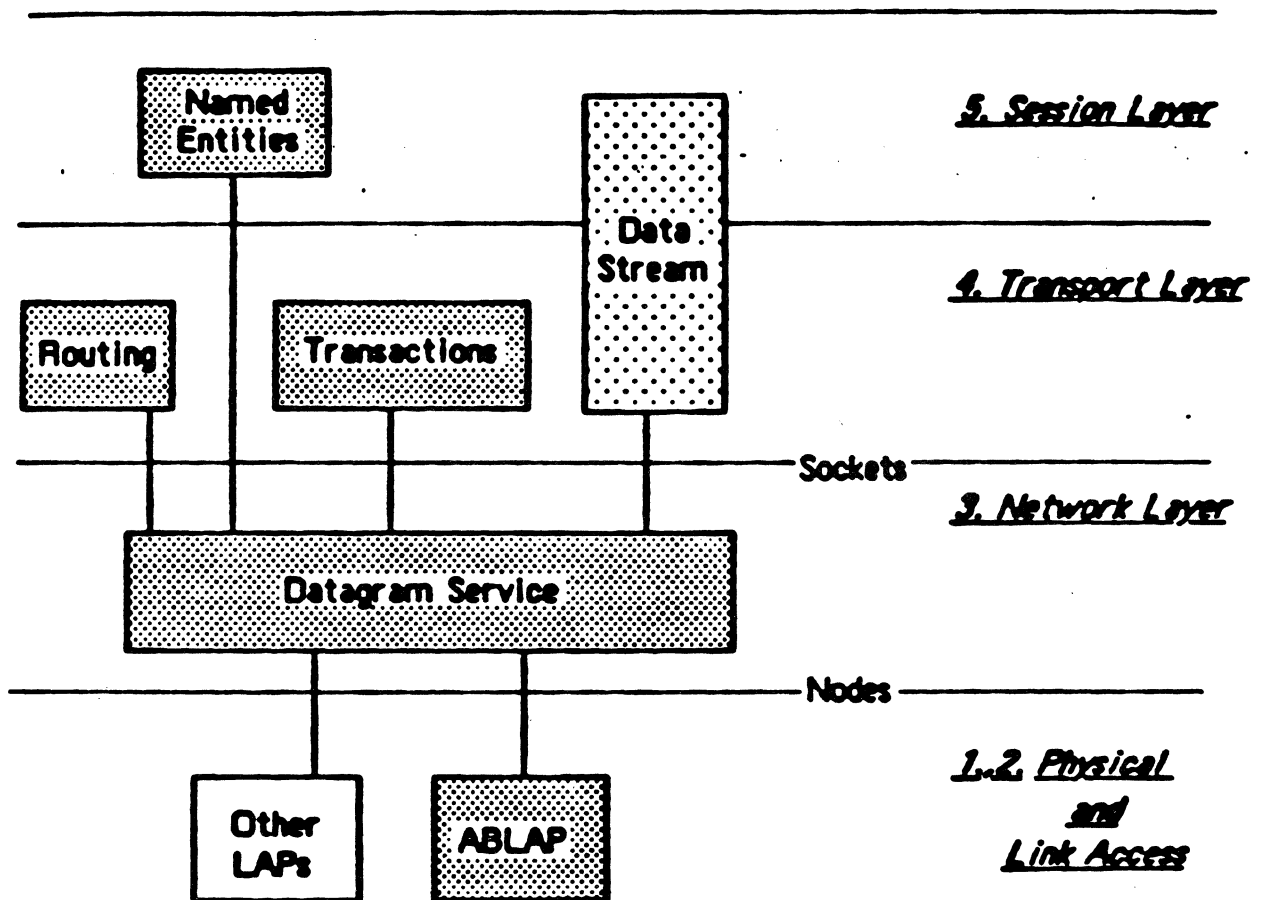
- reduce code size and complexity
- enhance performance

4. Support Third Party Development

Provide only the framework and basic
building blocks and services

AppleBus Core Protocols

ISO-OSI
Reference Model



AppleBus Protocols Architecture

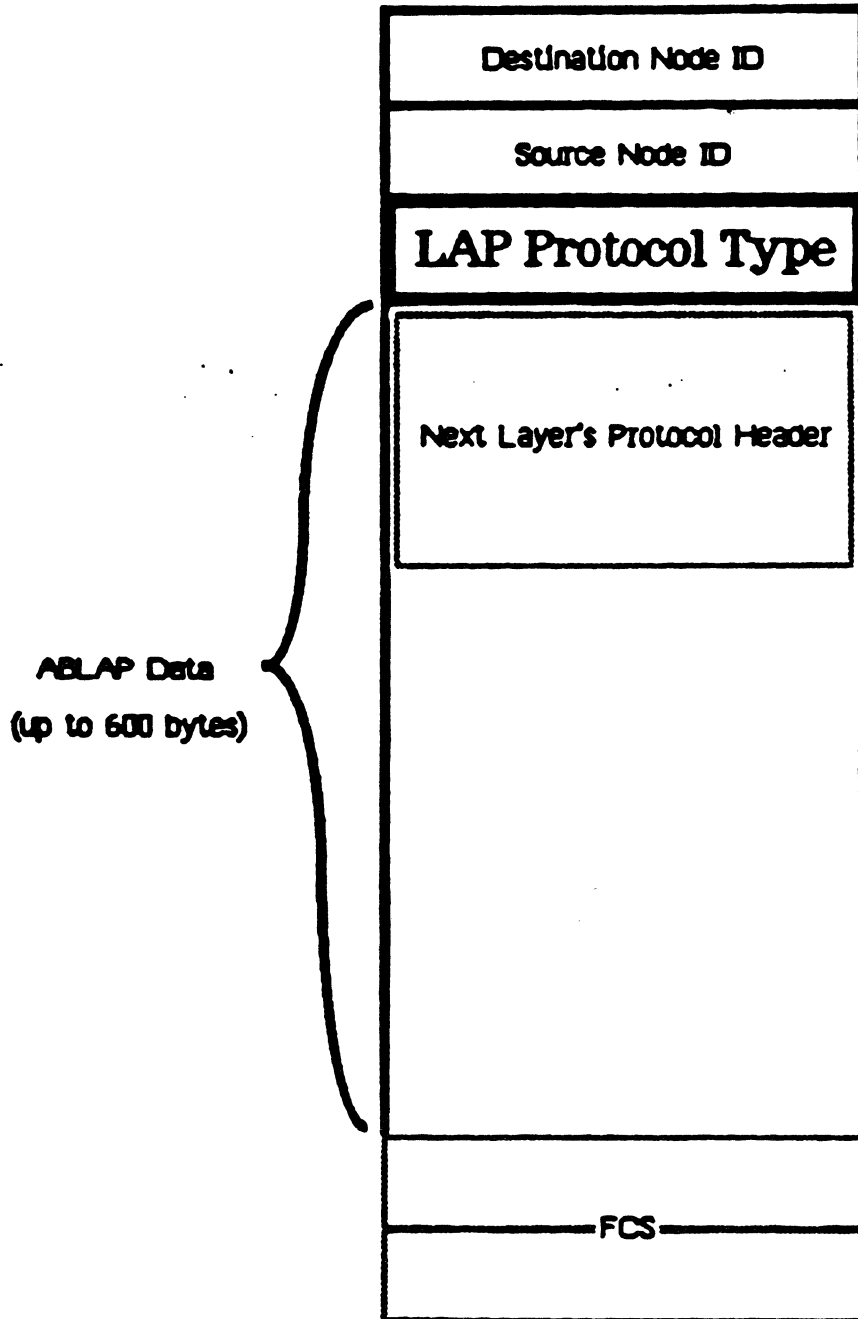
- **Key Considerations/Goals
and Overall Functions**



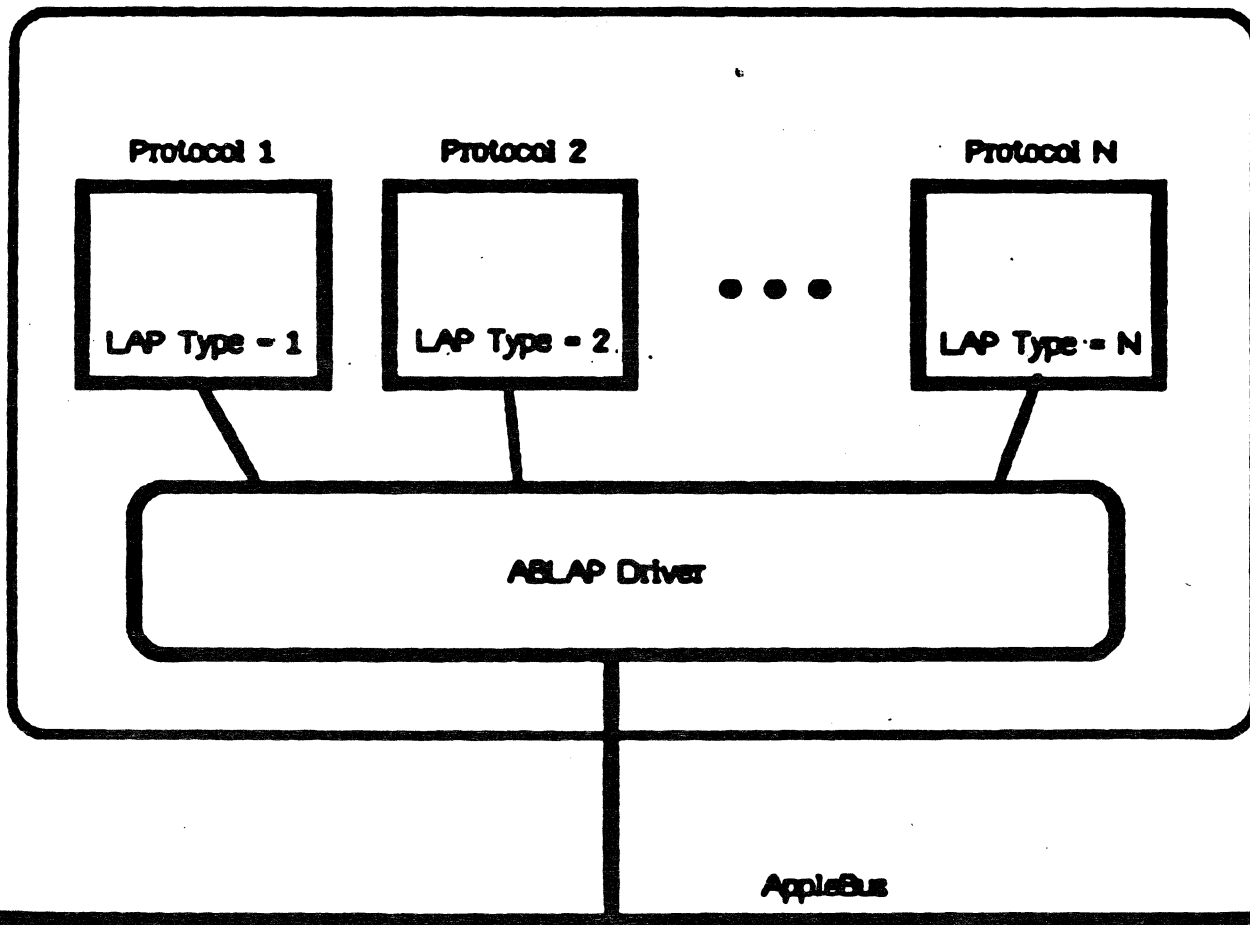
- **LAP Type Field**
- **Datagram Service**
- **Internets**
- **Named Entities**
- **Reliable Transport**

LAP Protocol Type

ABLAP Frame



Use of LAP Type Field



up to 254 different protocols can be located
immediately above the LAP

AppleBus Protocols Architecture

- **Key Considerations/Goals
and Overall Functions**

- **LAP Type Field**



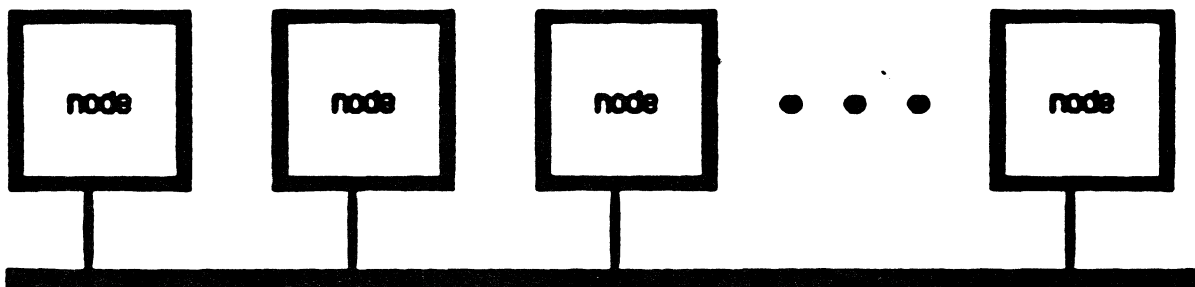
- **Datagram Service**

- **Internets**

- **Named Entities**

- **Reliable Transport**

AppleBus Node IDs

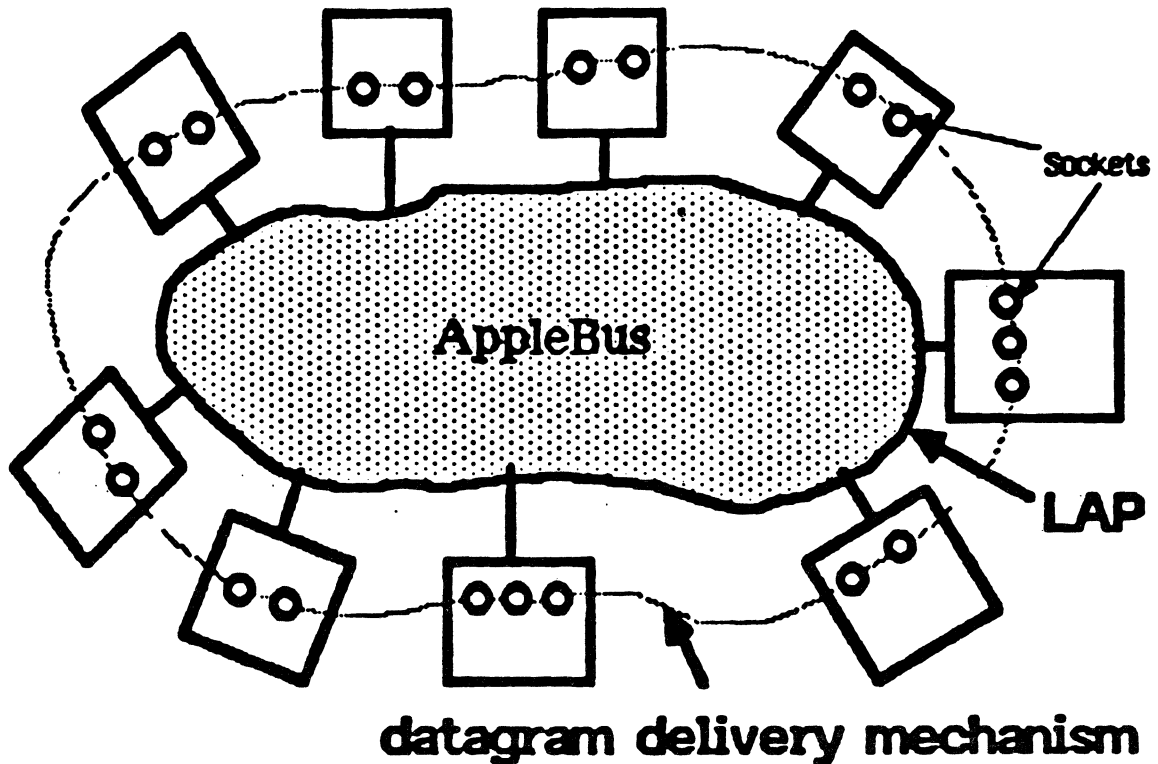


ABLAP uses the 8-bit Node Identifiers to provide a:

- node-to-node
- best effort •

packet delivery service on a single AppleBus

Sockets and Datagrams

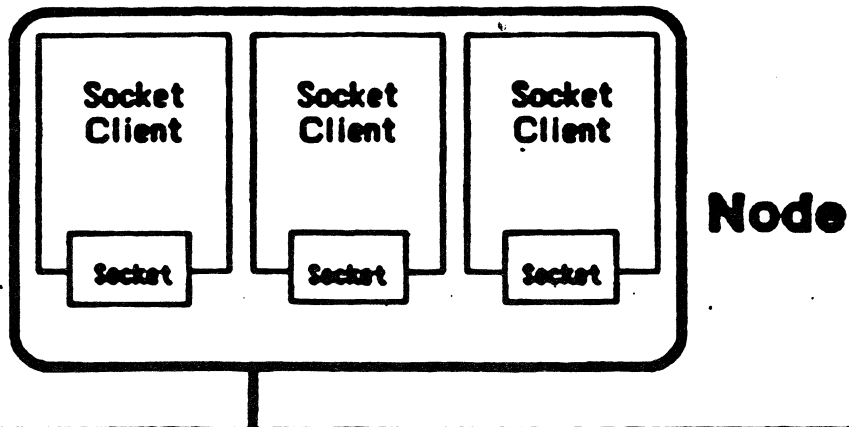


Sockets:

- logical addressable entities in network nodes
- identified by an 8-bit socket number
- Socket numbers are unique within a given node
- At most 254 sockets in a given node
- Datagrams are packets that are exchanged between sockets using the datagram delivery mechanism

Socket Clients

Sockets are "owned" by socket clients



Socket Client: a process (or "function"
in a process) (software)

Two types of sockets:

Statically assigned :

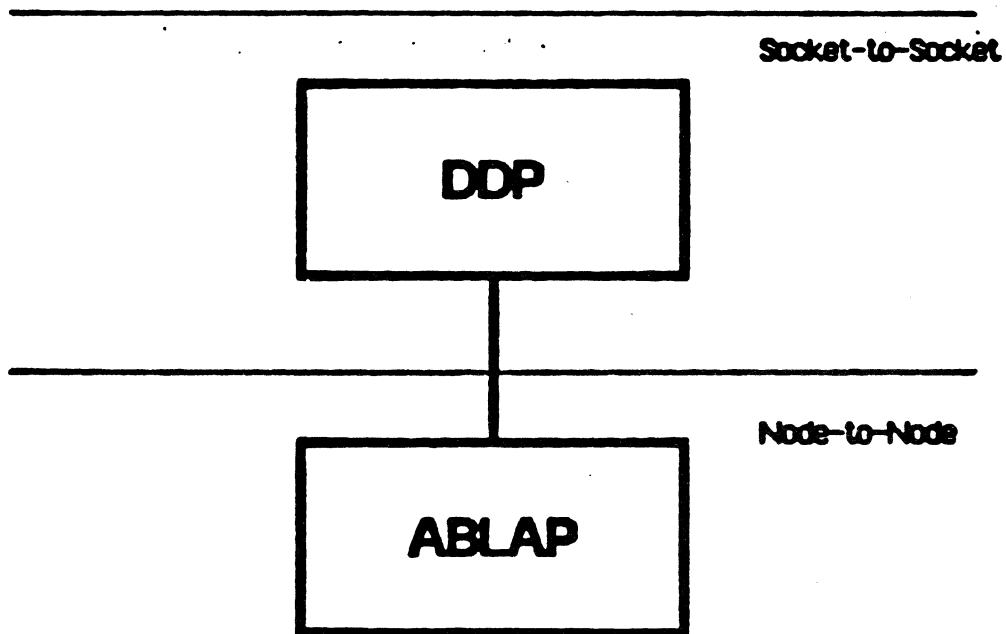
- 1 - 64. -- reserved
- 65. - 127. -- experimental use

Dynamically assigned : 128. - 254.

DDP

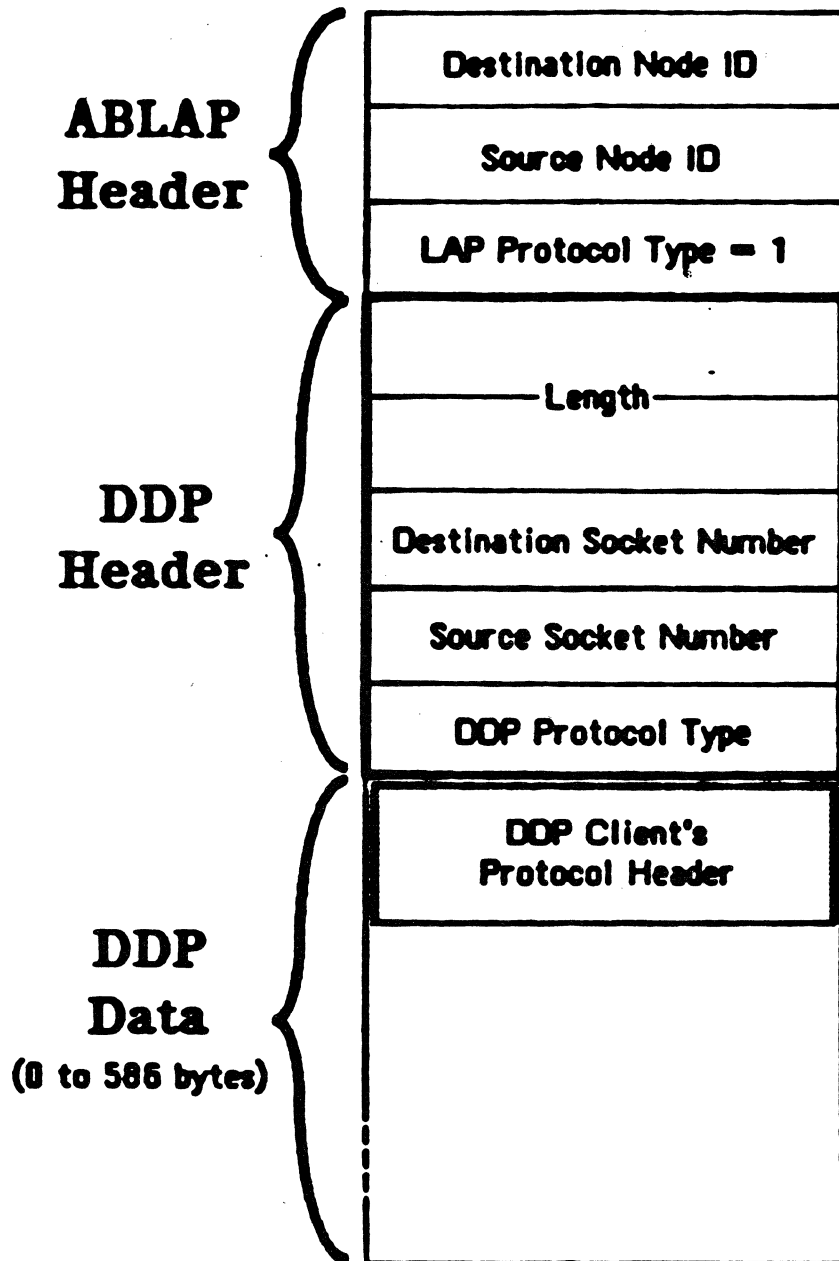
Datagram Delivery Protocol

the protocol that implements the socket-to-socket
delivery of packets (datagrams)



- Simple, "best effort", no retry mechanism
for packet loss

DDP Packet (single AppleBus)



AppleBus Protocols Architecture

- **Key Considerations/Goals
and Overall Functions**

- **LAP Type Field**

- **Datagram Service**

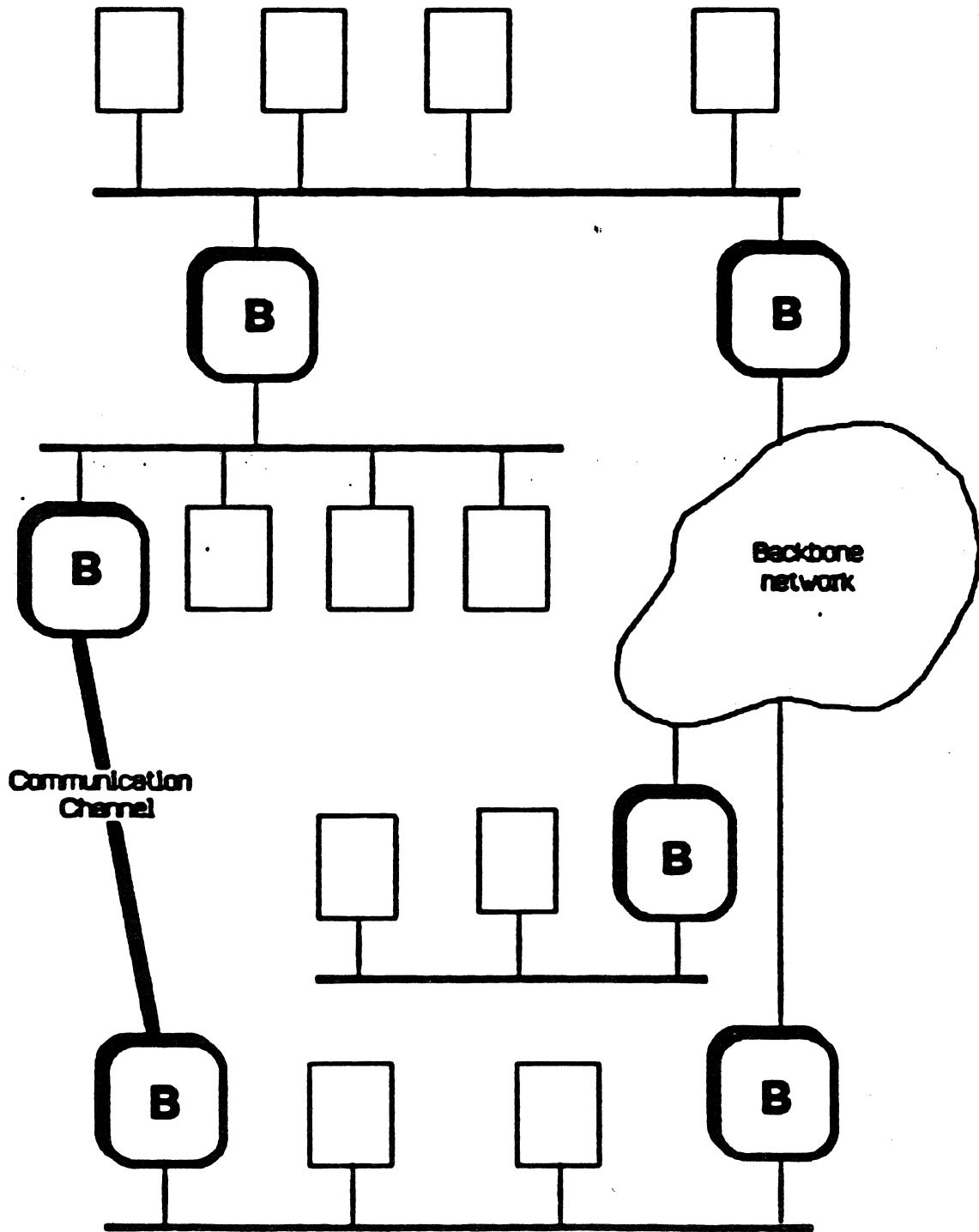


- **Internets**

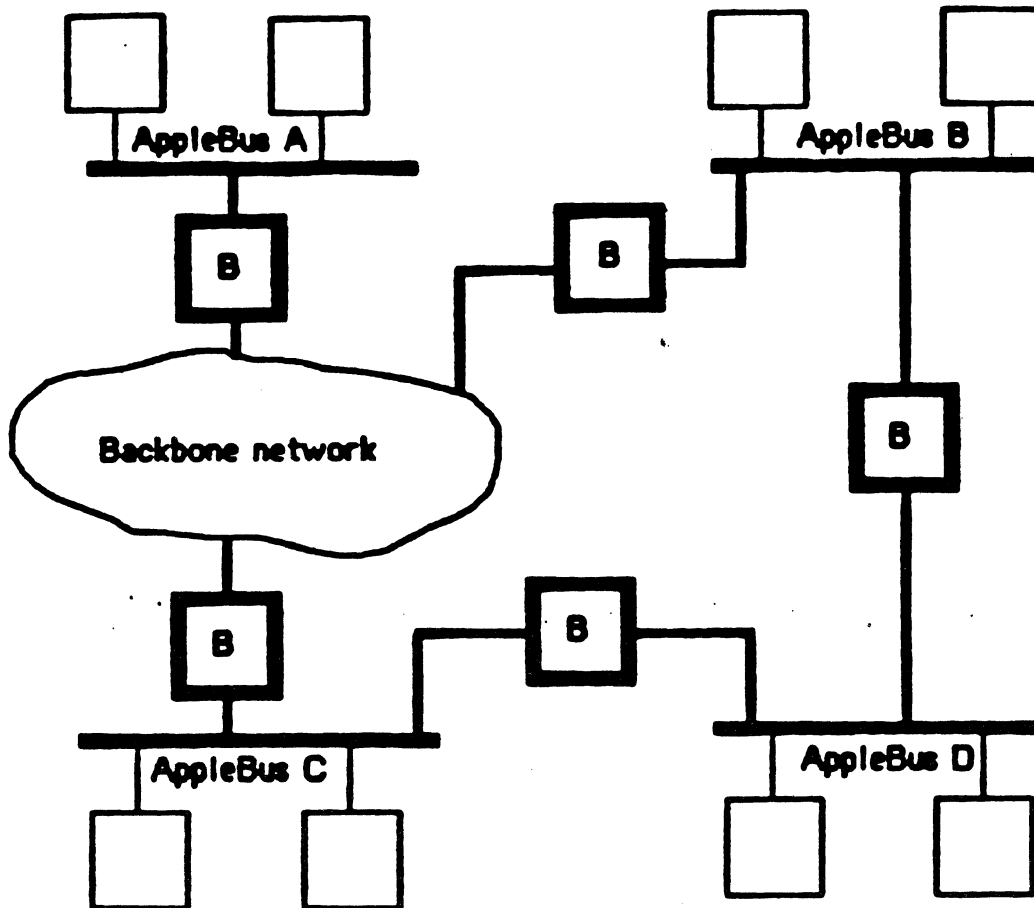
- **Named Entities**

- **Reliable Transport**

Internets and Bridges



Internet Node Addressing

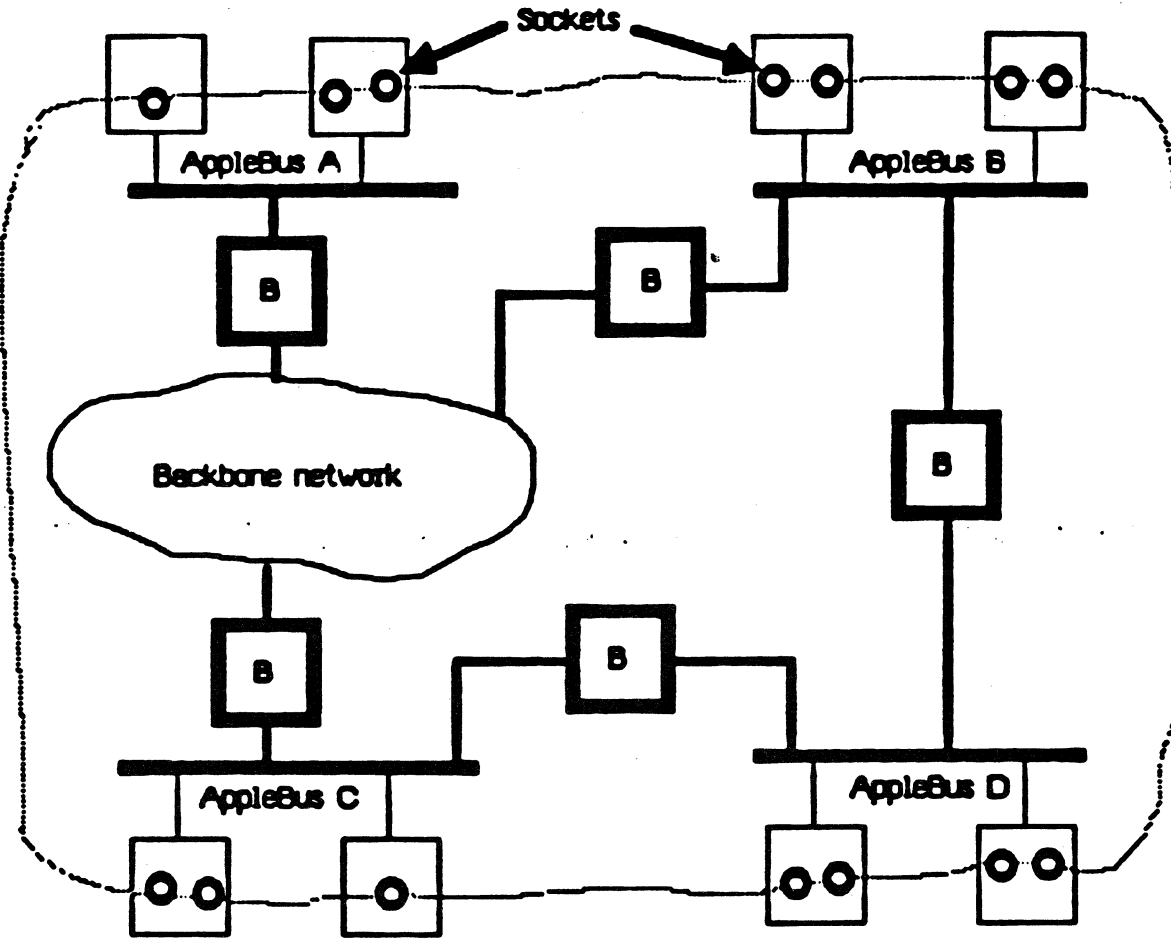


- Unique 16-bit network number for each network in the internet
- Unique way of addressing a node on the internet

Internet Node ID =

Network # : AppleBus Node ID

Internet Socket Addressing

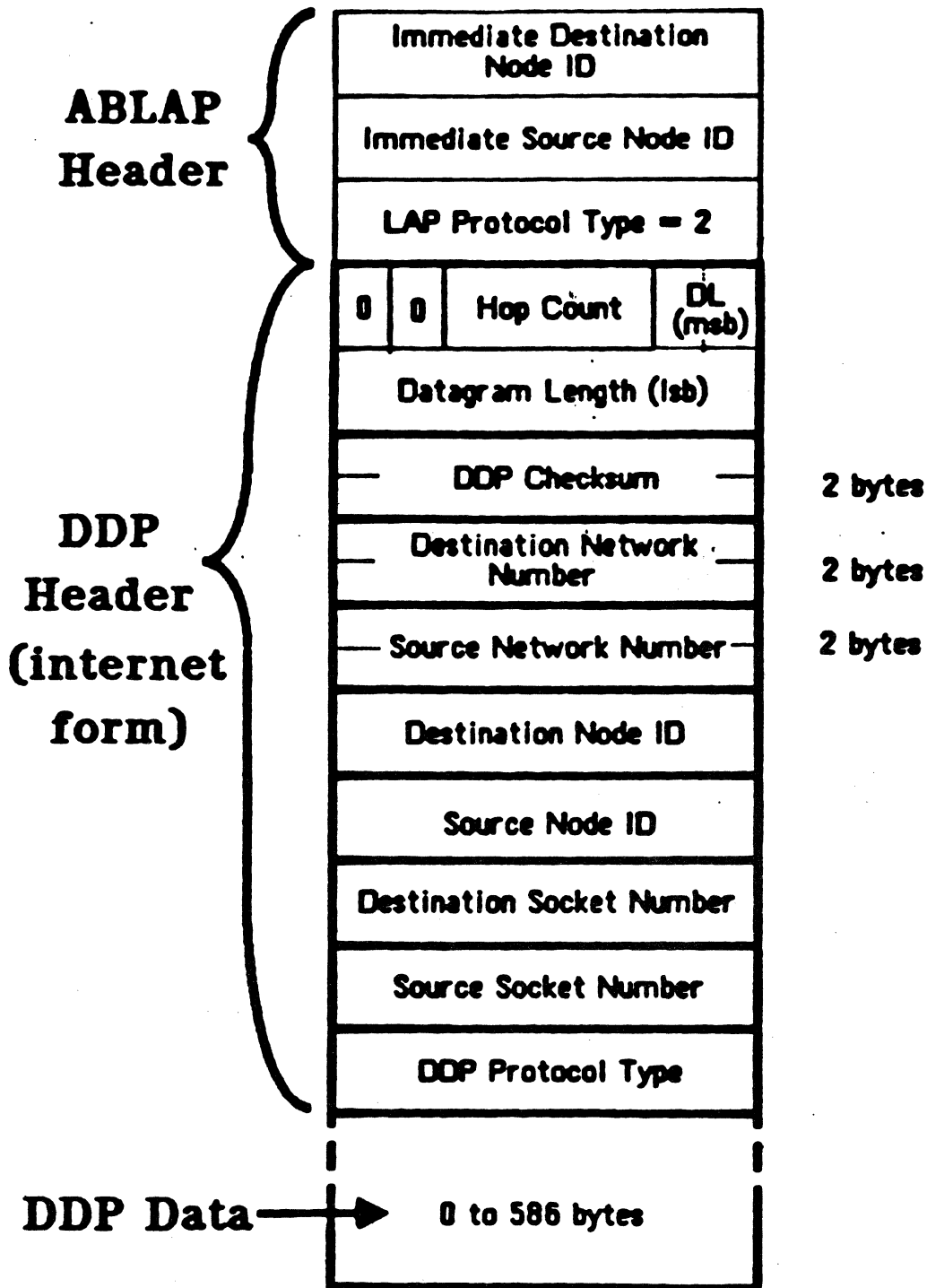


- Unique socket address on the internet

Internet Socket Address =

Network # : AppleBus Node ID : Socket #

Internet Datagrams



AppleBus Protocols Architecture

- **Key Considerations/Goals
and Overall Functions**

- **LAP Type Field**

- **Datagram Service**

- **Internets**



- **Named Entities**

- **Reliable Transport**

Addresses vs Names

Protocols use Addresses

People prefer Names

Addresses are Numbers

Names are Character Strings

Named Entities

Network-Visible Entities ==> Socket Clients

Names of Network-Visible Entities

Entity Name ==>

< Object > : < Type > @ < Zone >

each part of the name is a
string of up to 32 characters

Accessing Named Entities

Two Step Procedure:

(1) Name Binding:

NAME —————> SOCKET ADDRESS

(2) Access the Entity:

Use this address to access the entity
through its socket

Wildcards in Names

Object and Type Fields

= means "any"

Zone Field

* means "this zone"

Names Directory

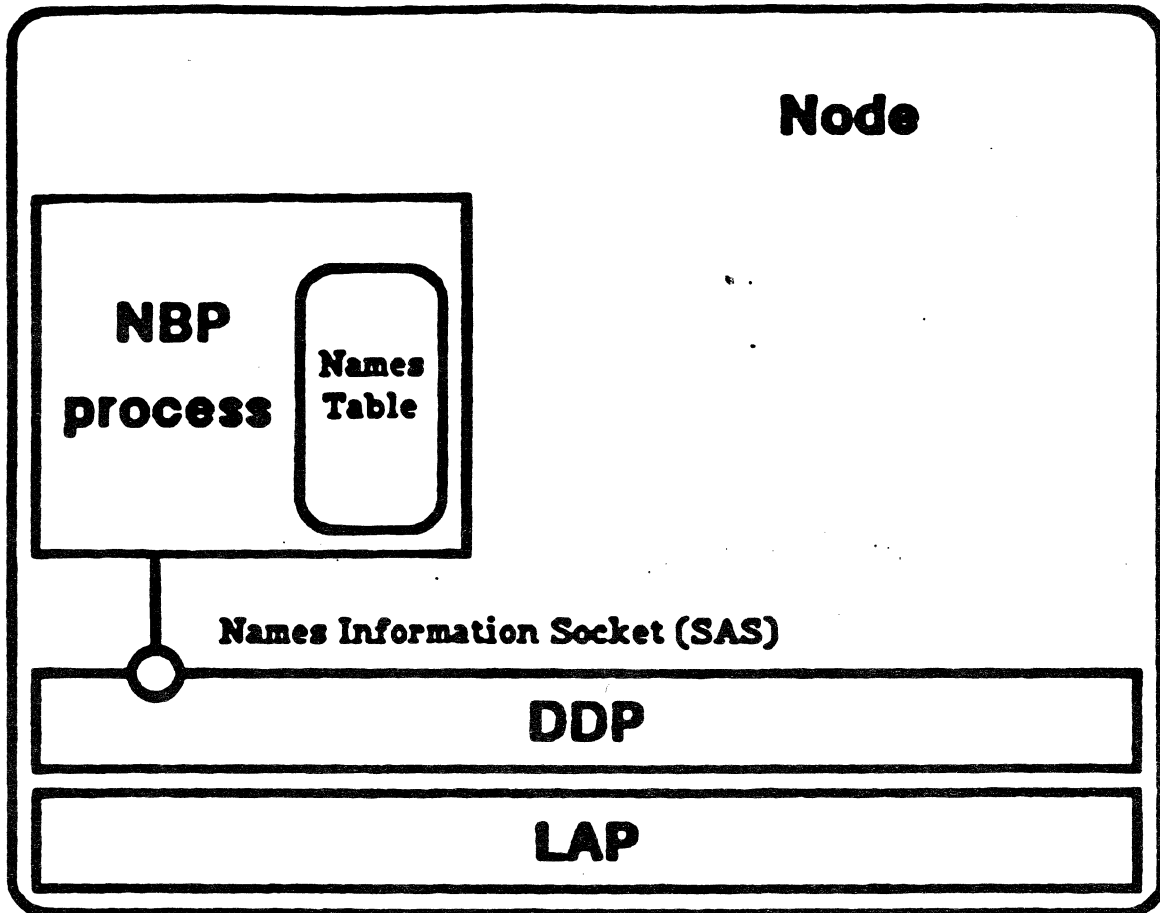
ND consists of the Names Tables in the nodes of the internet

Names Table contains

< Entity Name, Socket Address >

pairs for all named socket clients
in that node

Name Binding Protocol



(c) 1984 Apple Computer Inc.

Name Lookup using NBP

(single ABus case)

1. Requester broadcasts an NBP LookUp datagram
(with name to be looked up) to the
Names Information Socket
2. This is received by all nodes that have
implemented an NBP process
3. Each of these NBP processes searches its Names Table:
 - if a match is not found it ignores the request
 - if a match is found, it sends a datagram back to the lookup requester with the complete name(s) and socket address(es) of the matching entity(ies)
4. The requester repeats this process several times

AppleBus Protocols Architecture

- **Key Considerations/Goals
and Overall Functions**

- **LAP Type Field**

- **Datagram Service**

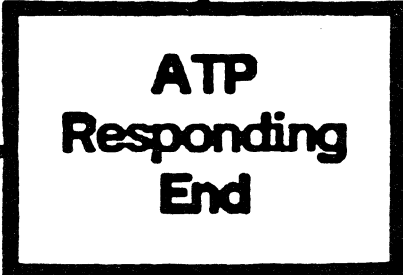
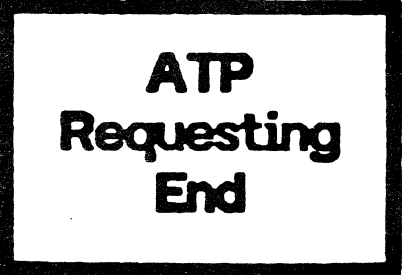
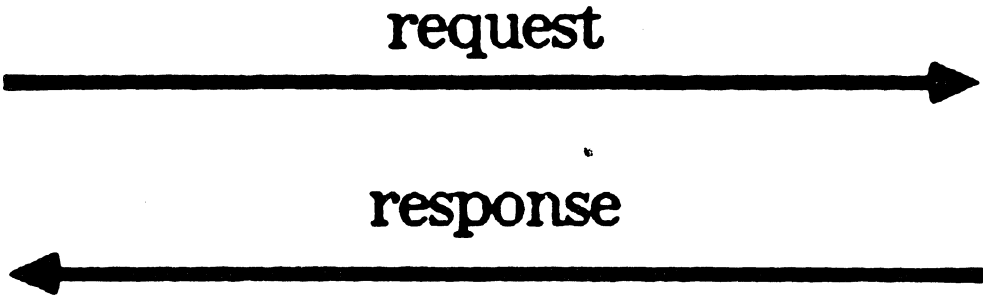
- **Internets**

- **Named Entities**

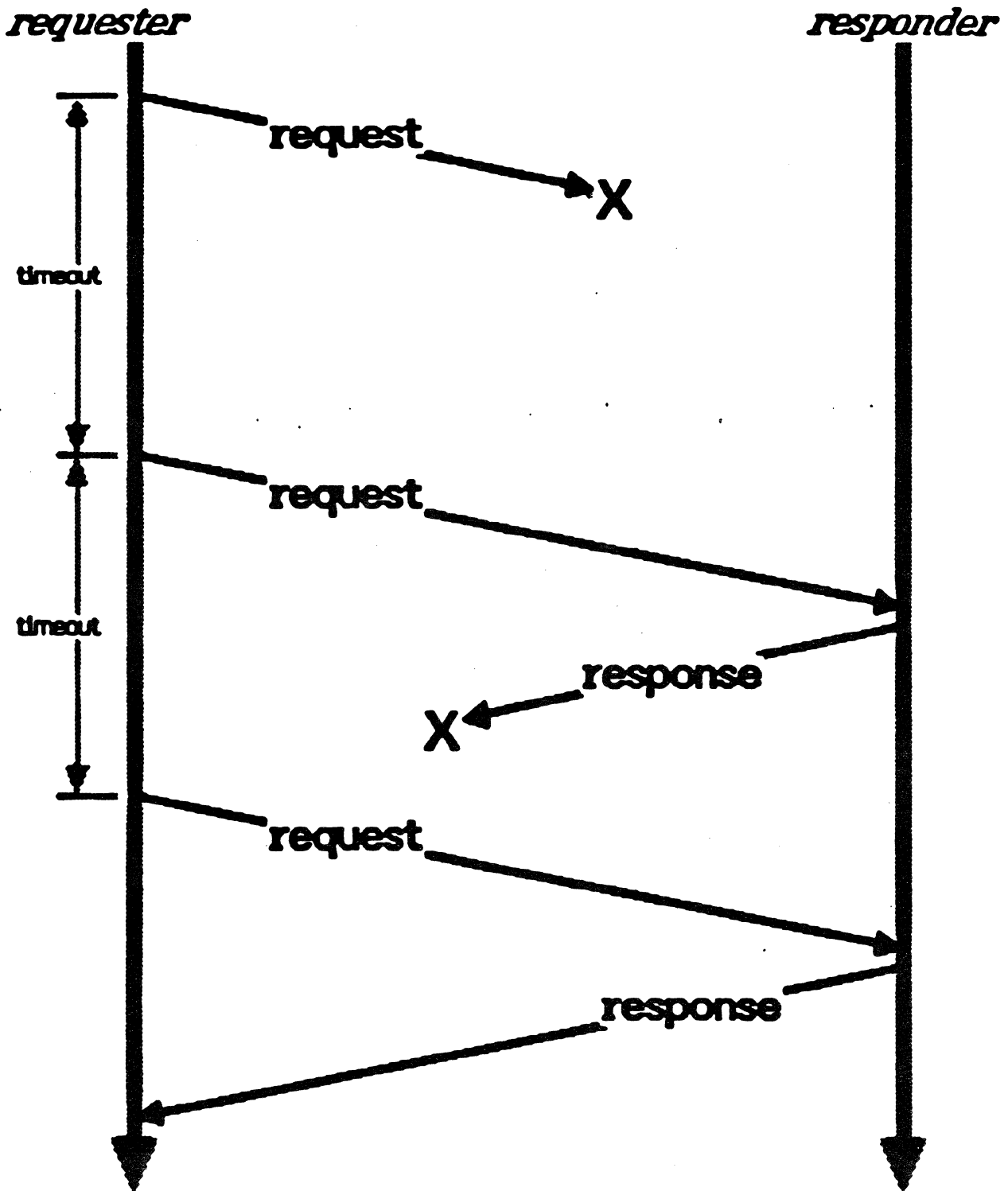


- **Reliable Transport**

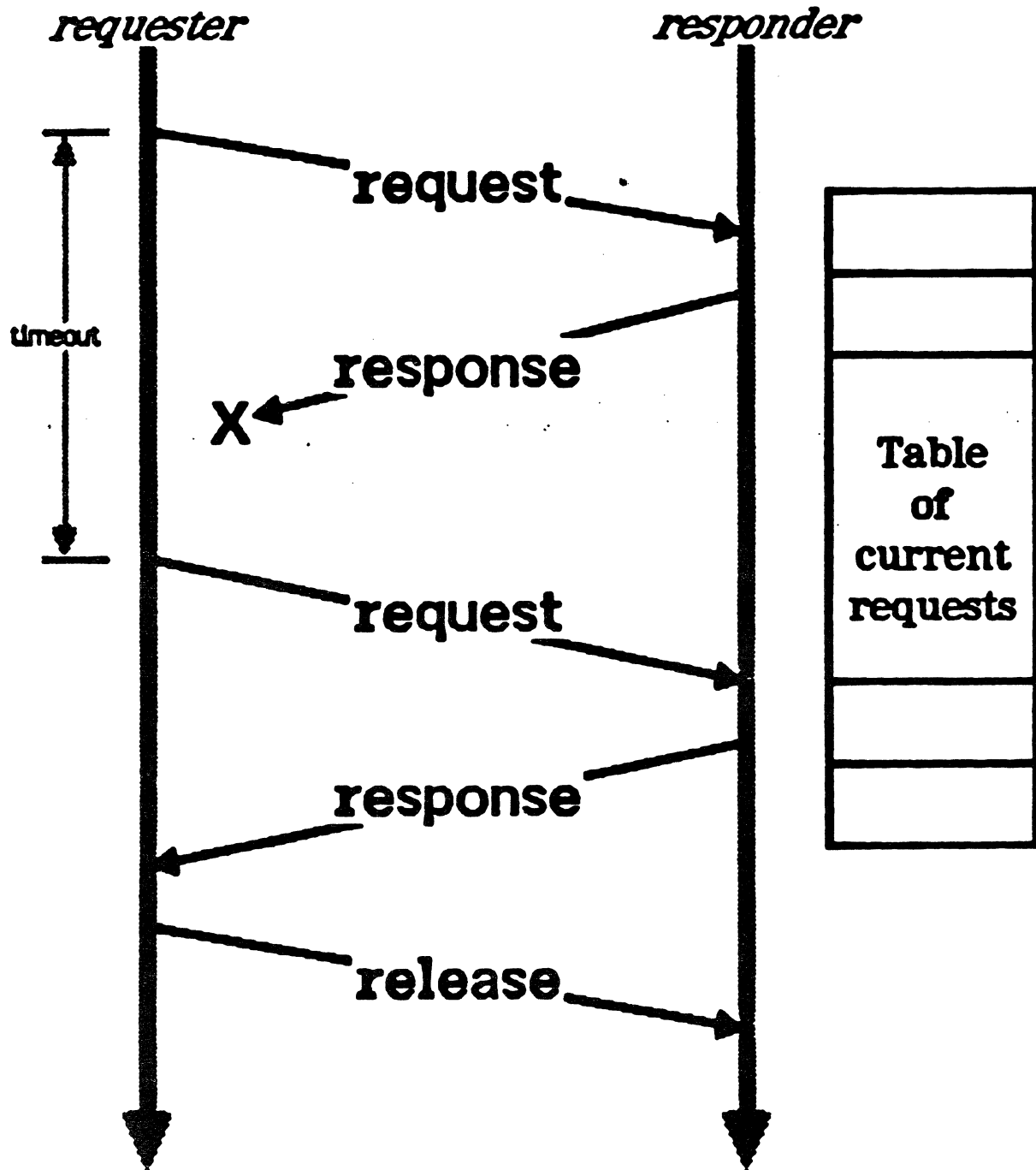
ATP Terminology



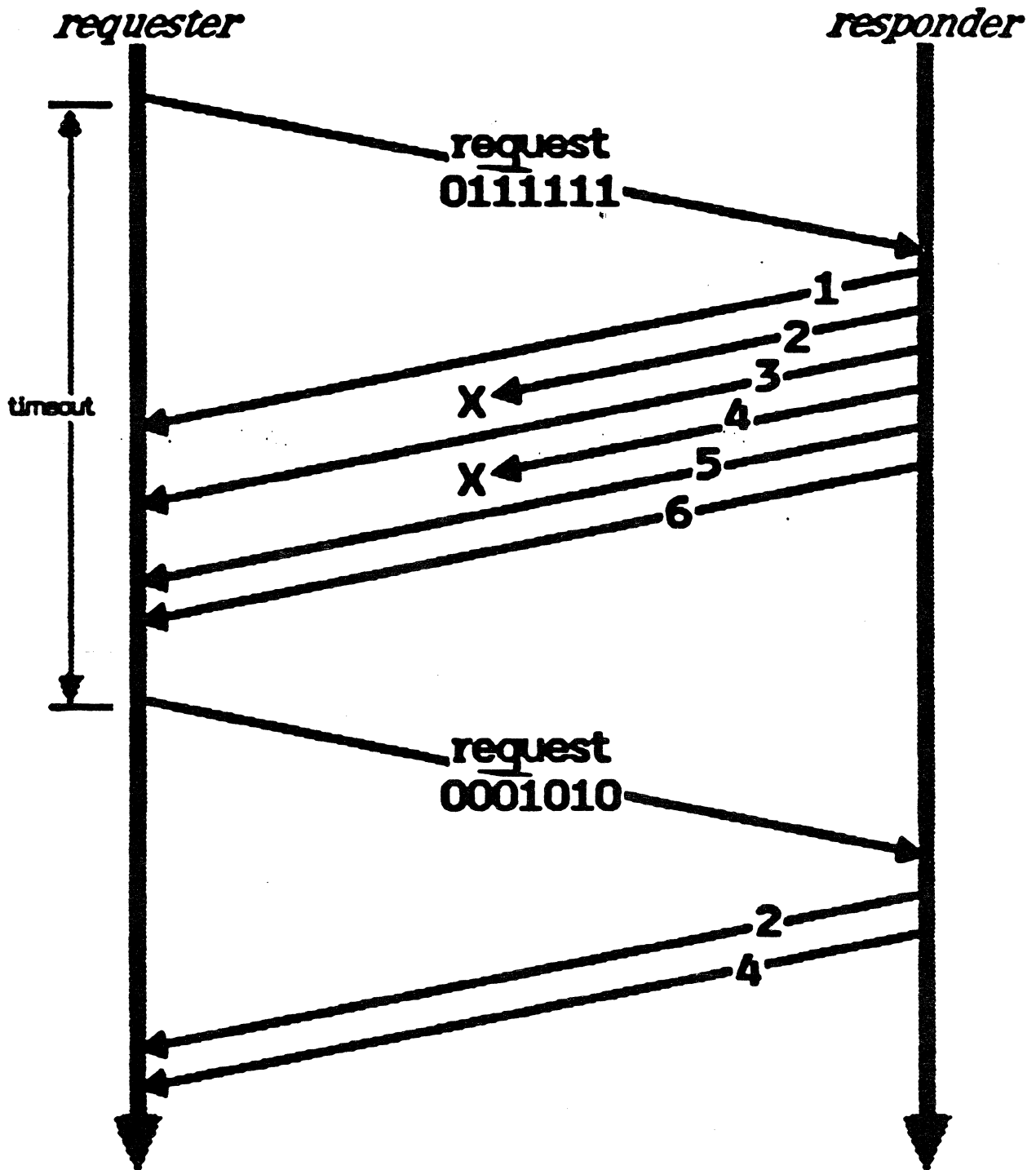
ATP Error Recovery



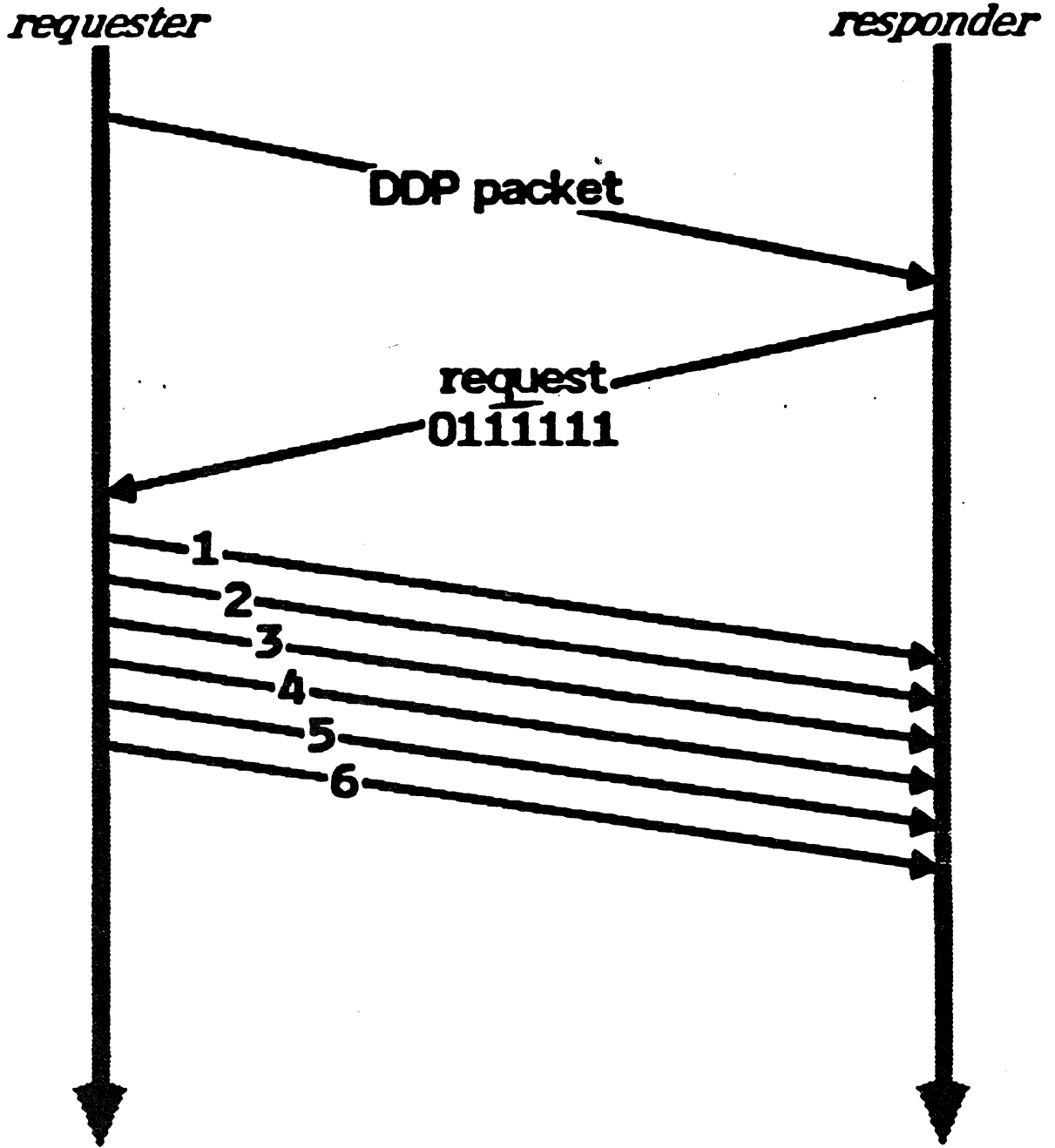
ATP Exactly-Once Service



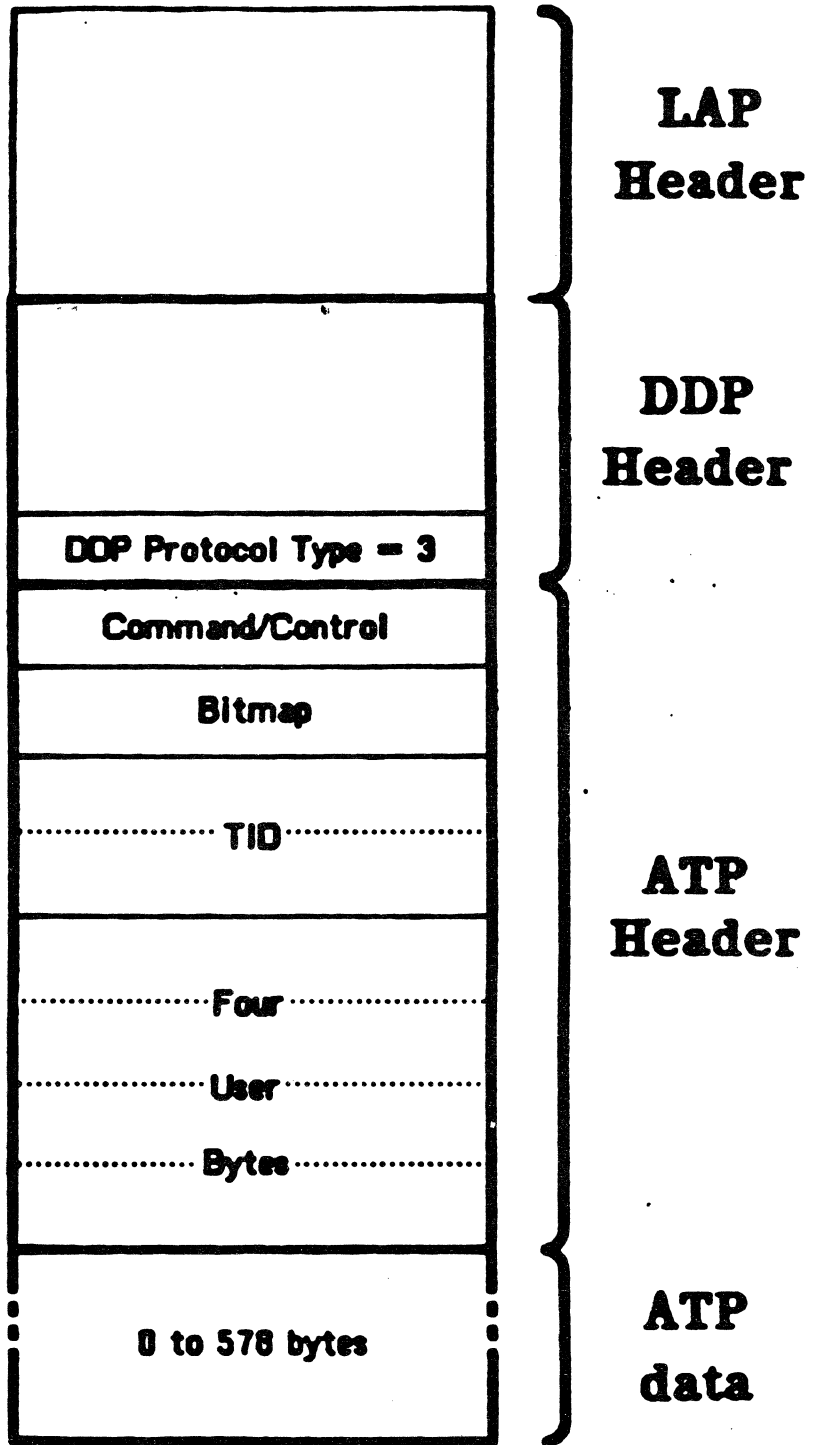
ATP Multi-Packet Responses



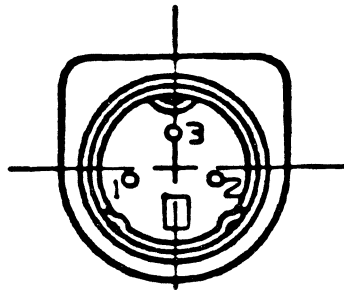
ATP Write Example



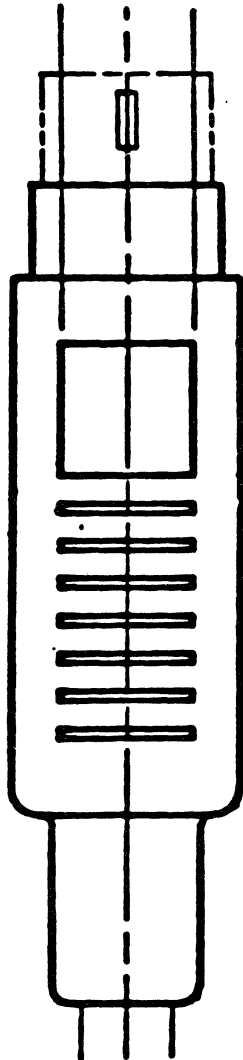
ATP Packet Format



AppleBus



WIRE CONNECTION TABLE		
DIN PIN #	FUNCTION	DIN PIN #
1	APPLEBUS-PLUS	1
2	APPLEBUS-MINUS	2
3	UNUSED	3
SHELL	SHIELD	SHELL



AppleBus Cable Connector

3-Pin Male
Miniature Din

Color: Apple Snow Beige

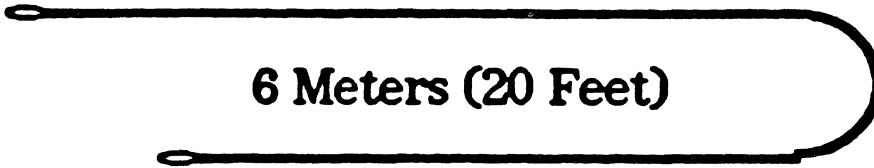
(c) 1984 Apple Computer Inc.

AppleBus

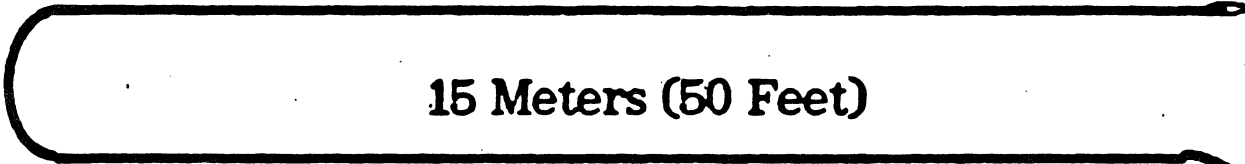
AppleBus Cables



2 Meters (6 Feet)



6 Meters (20 Feet)



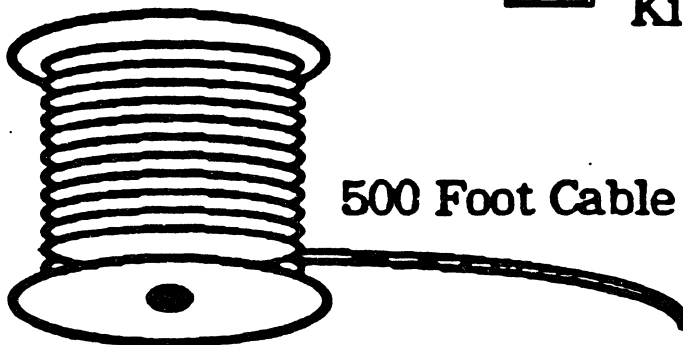
15 Meters (50 Feet)



Bulk Cables (PVC and Teflon)



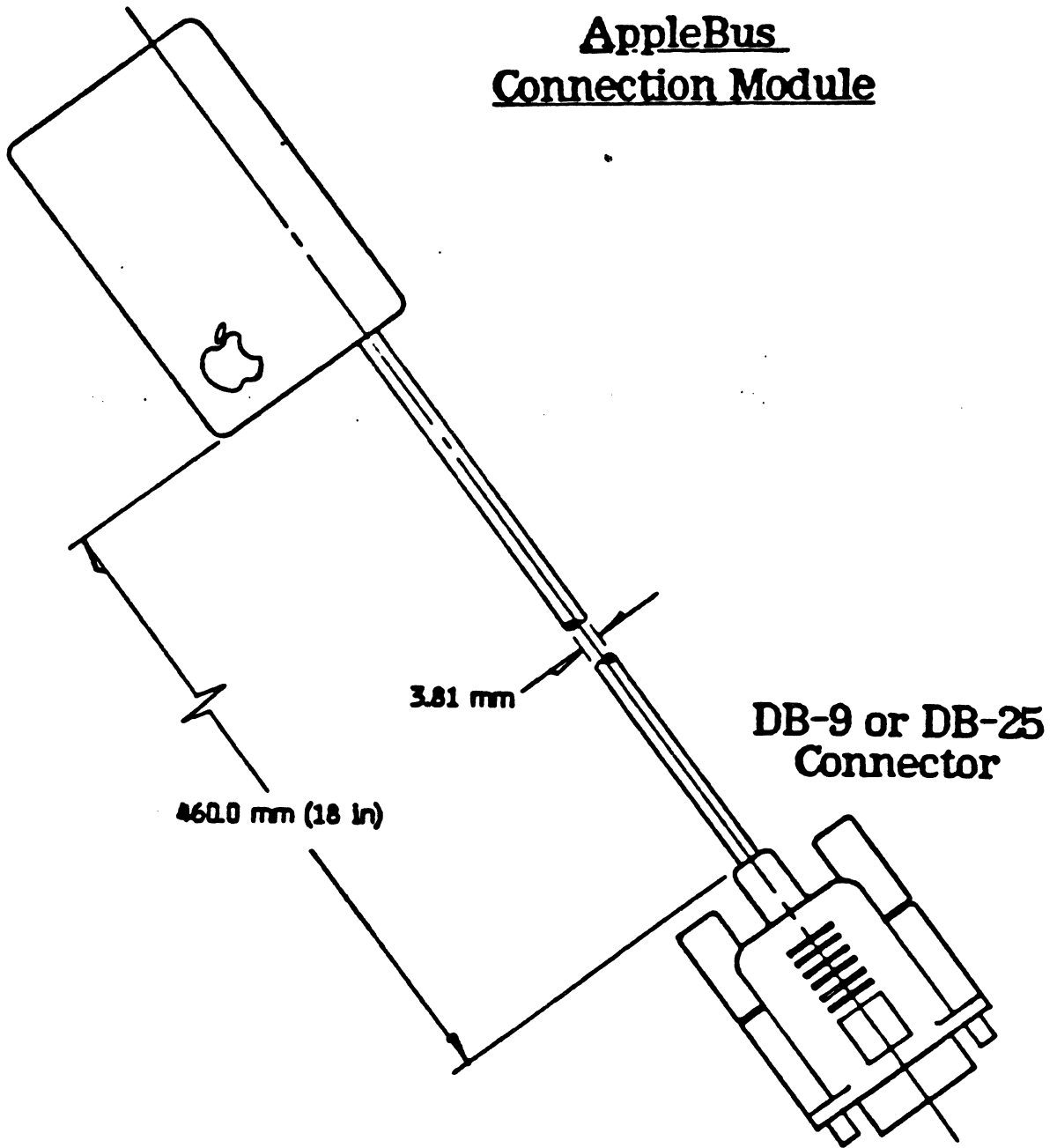
Connector
Kits (6 units)



500 Foot Cable Spool

AppleBus

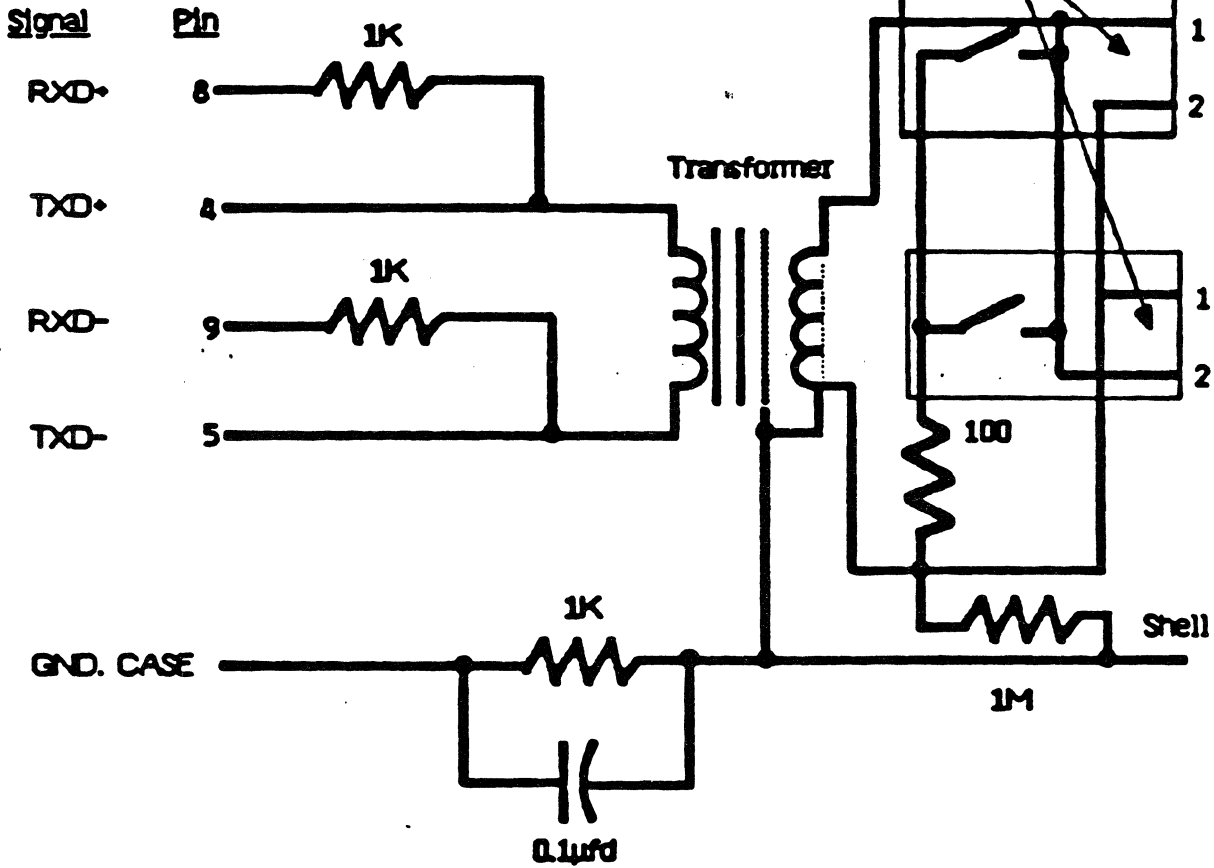
AppleBus Connection Module



(c) 1984 Apple Computer Inc.

AppleBus

DB-9 Connector

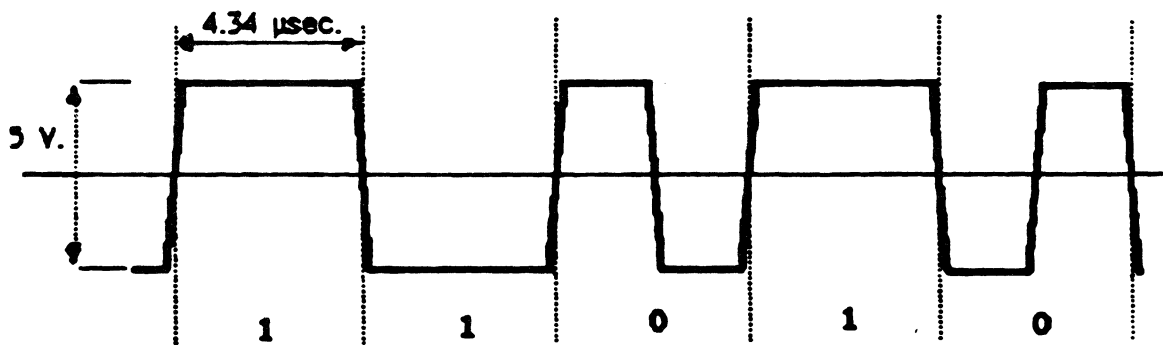
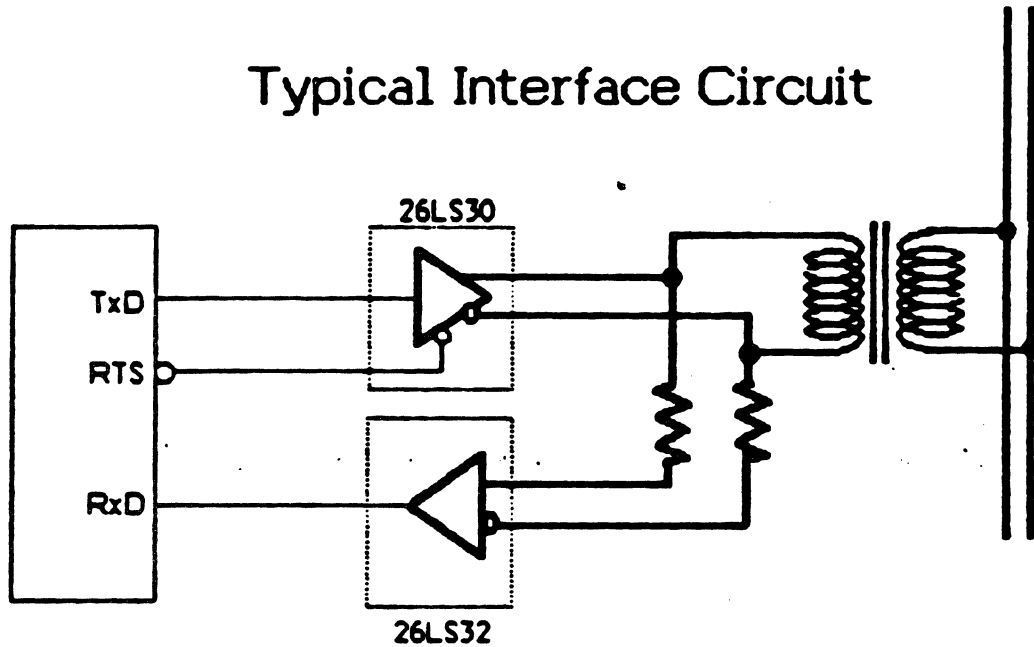


AppleBus Connection Module Circuit Diagram

AppleBus

Hardware Overview

Typical Interface Circuit



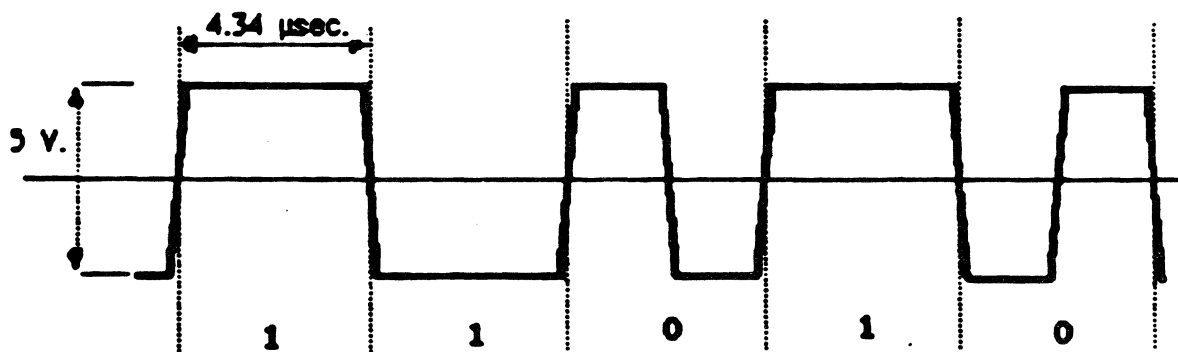
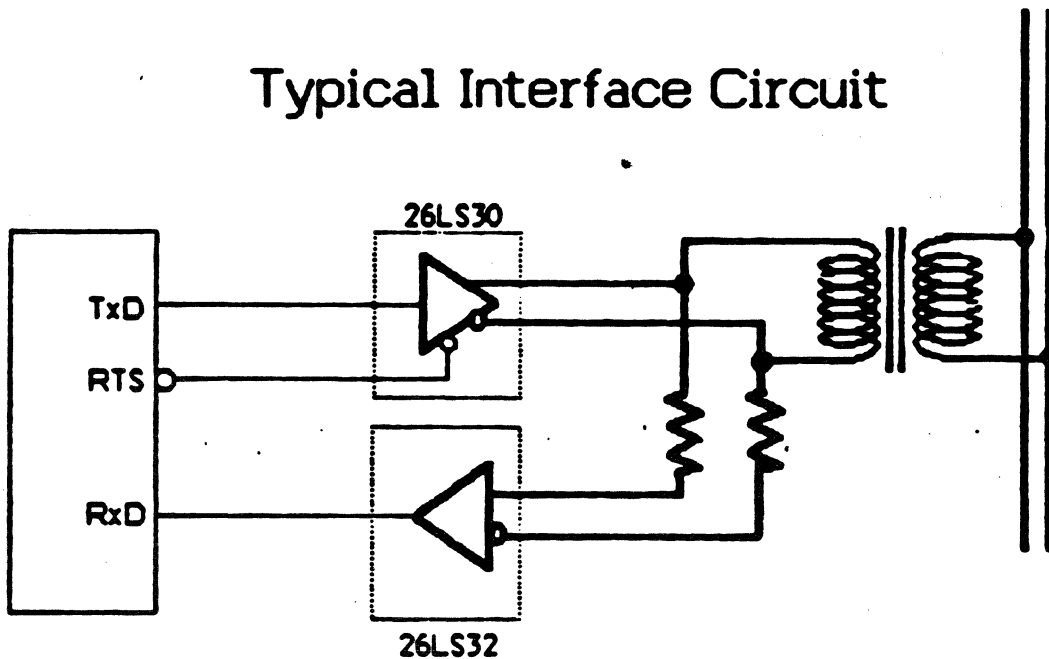
FM-0 Encoding, RS-422 Signalling

(c) 1984 Apple Computer Inc.

AppleBus

Hardware Overview

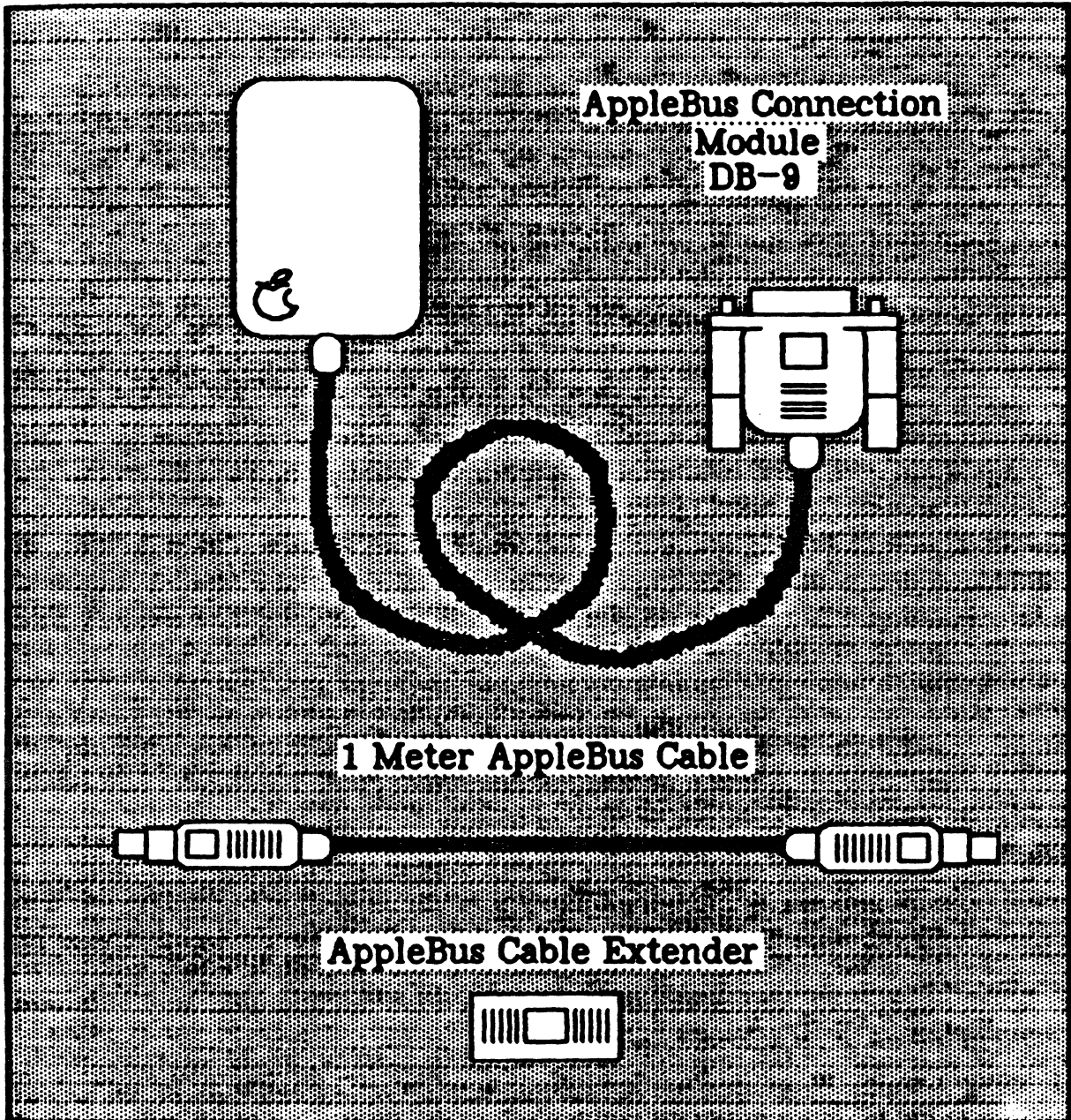
Typical Interface Circuit



FM-0 Encoding, RS-422 Signalling

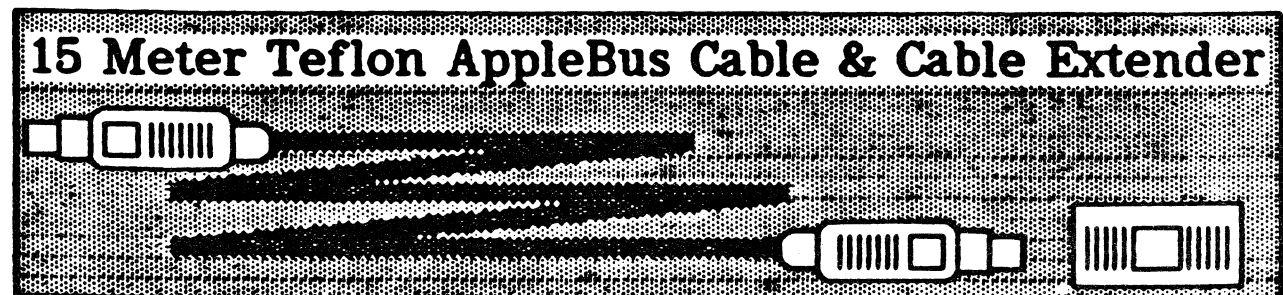
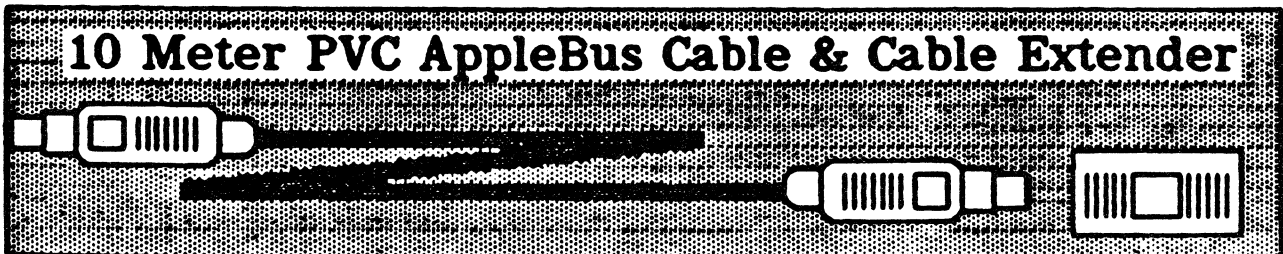
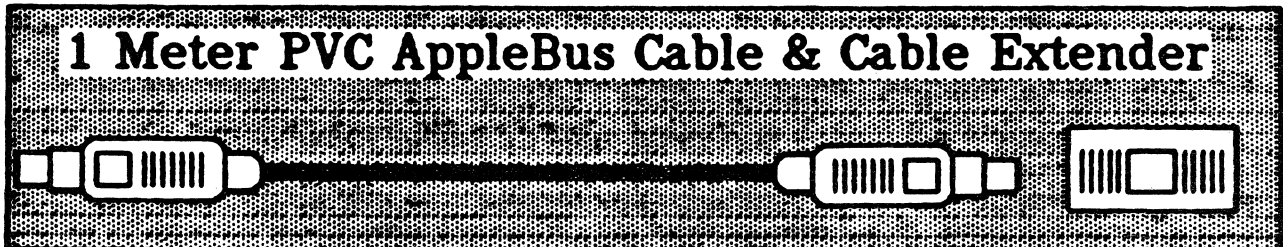
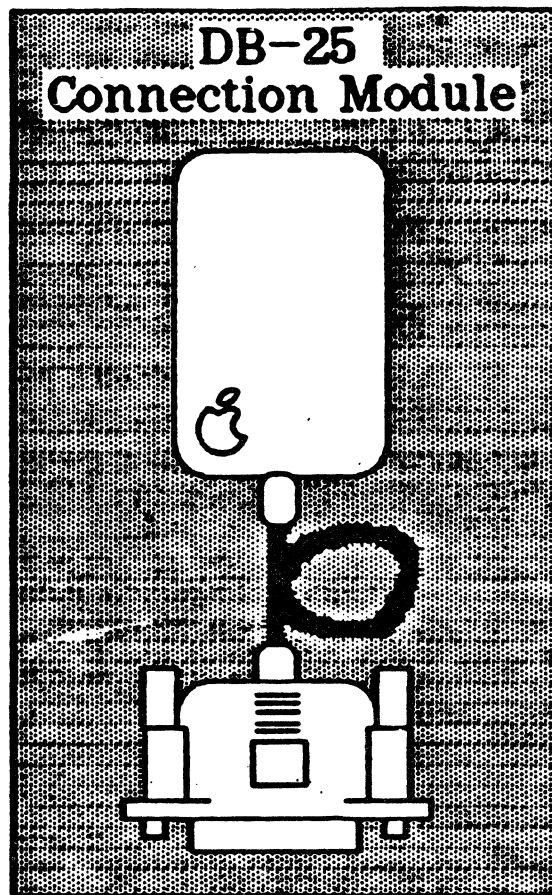
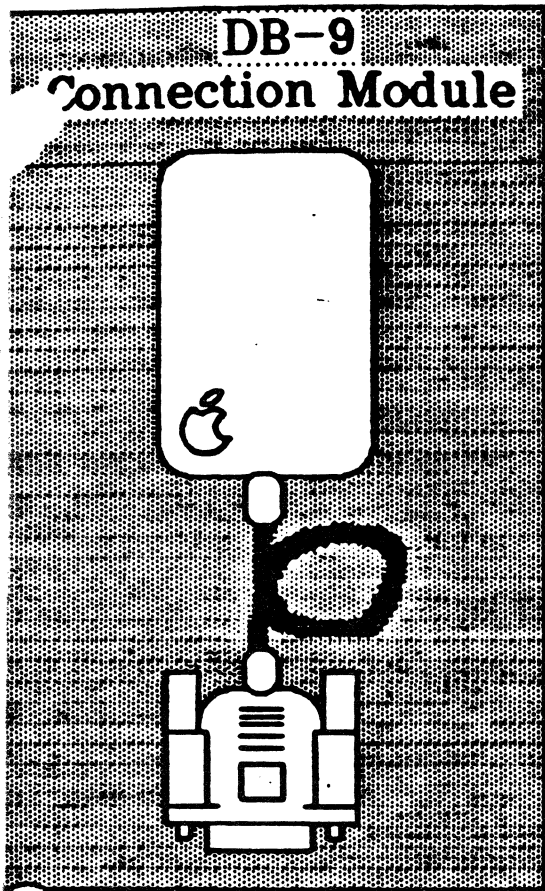
(c) 1988 Apple Computer Inc.

AppleBus Server Kit



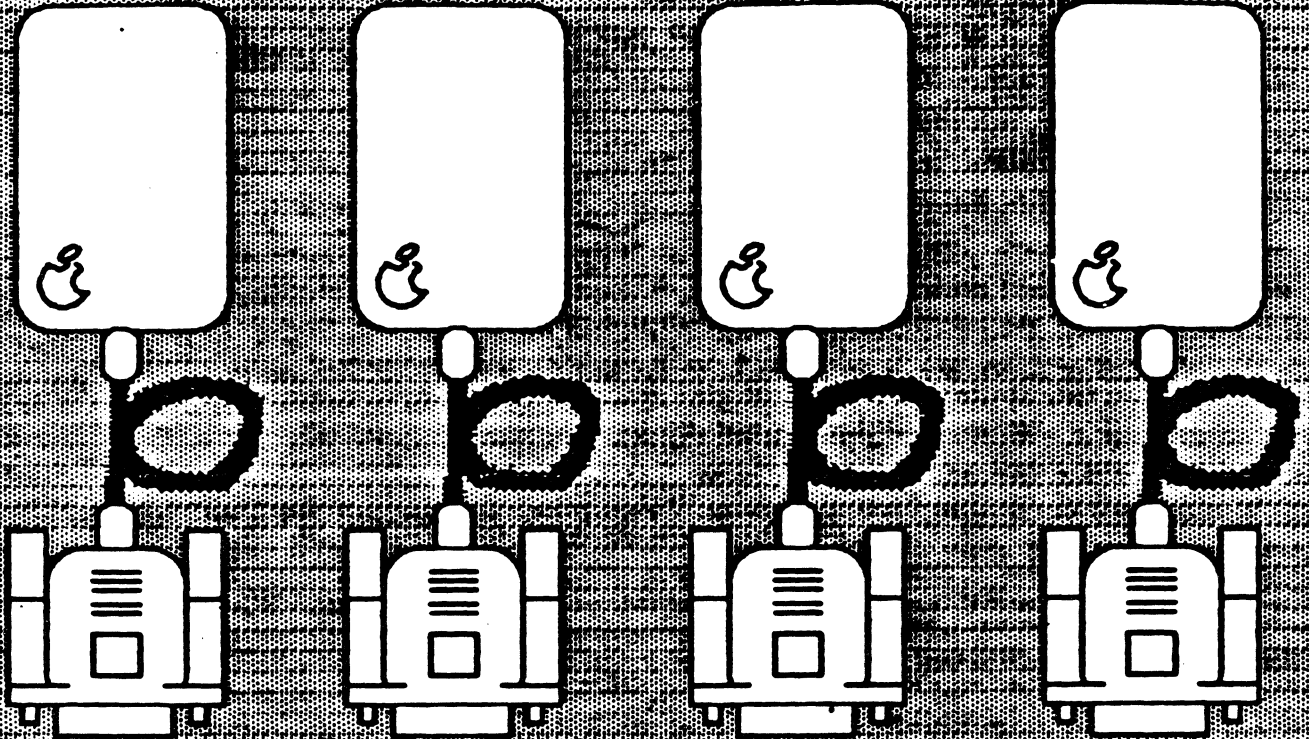
**AppleBus Server Kit packaged with
all Apple Server Products**

AppleBus Components

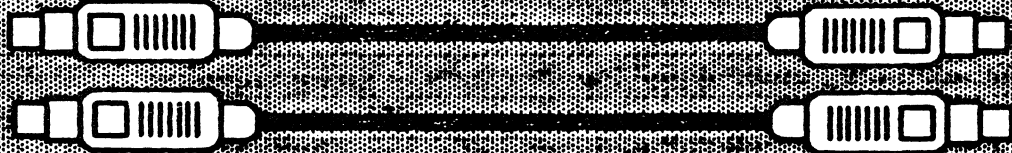


AppleBus Starter Kit

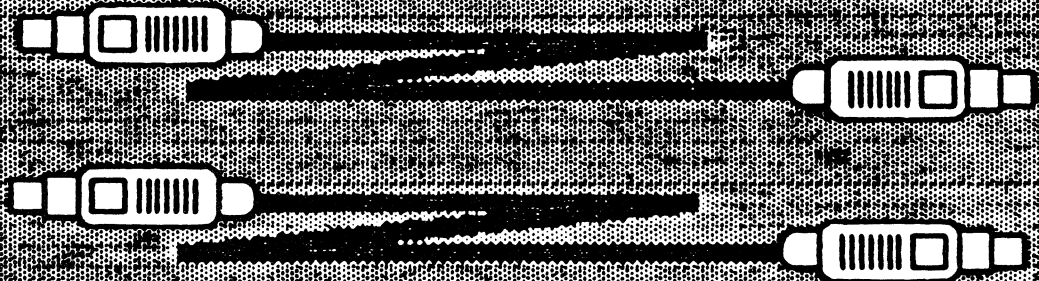
Four (4) AppleBus Connection
Modules (DB-9)



Two (2) 1 Meter AppleBus Cables



Two (2) 10 Meter AppleBus Cables

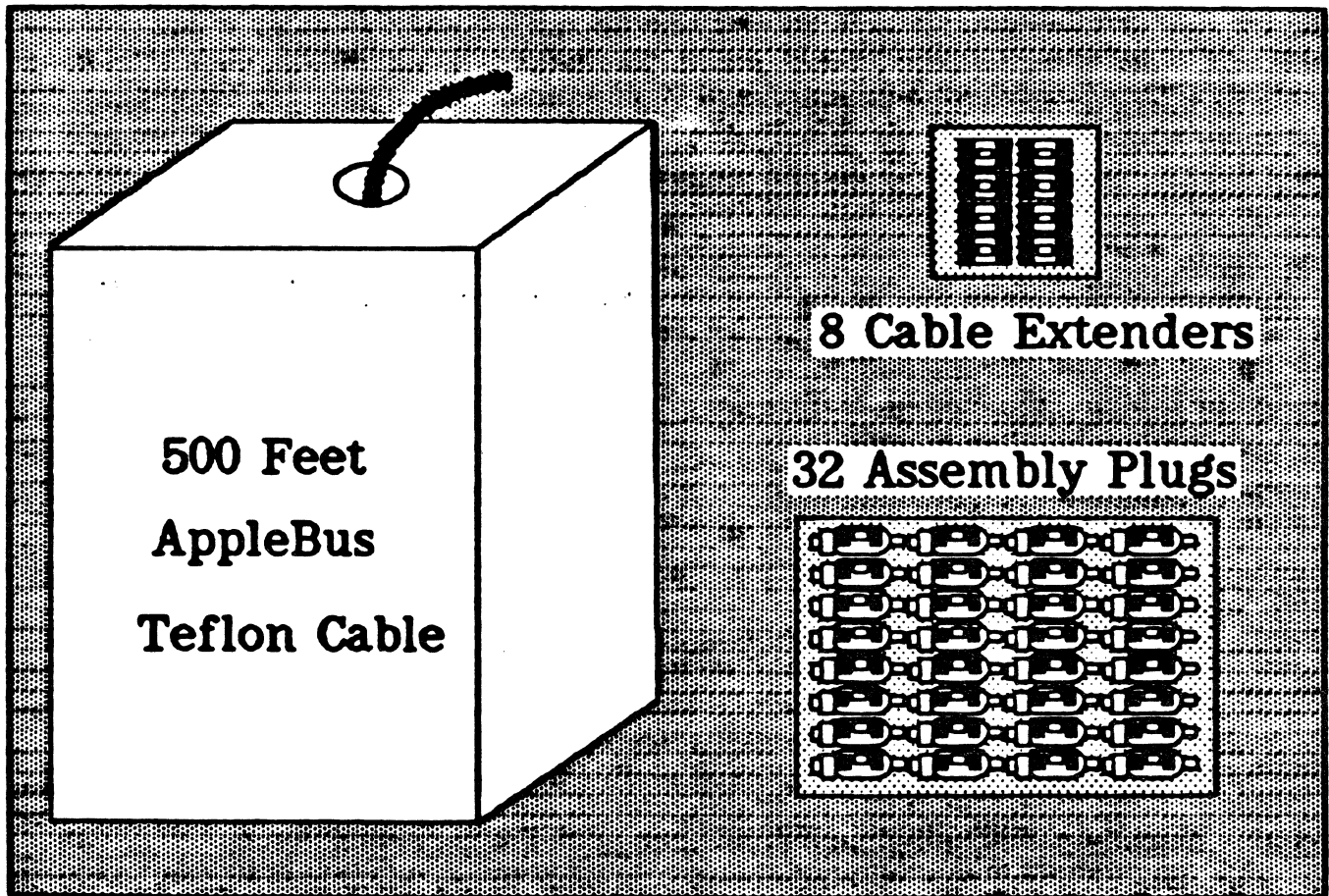


AppleBus Cable Extender

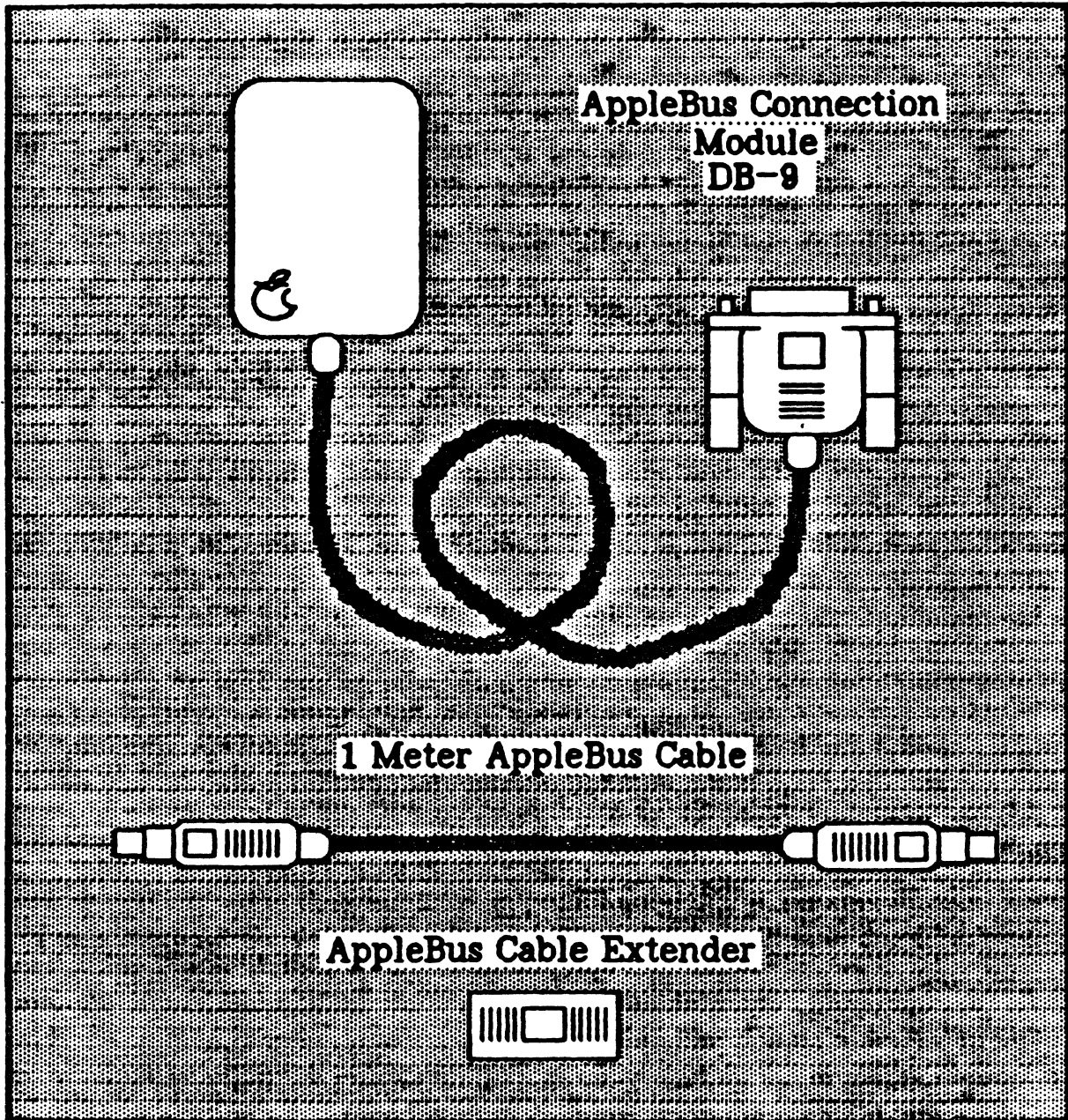


AppleBus Installation Guide

AppleBus Assembly Kit

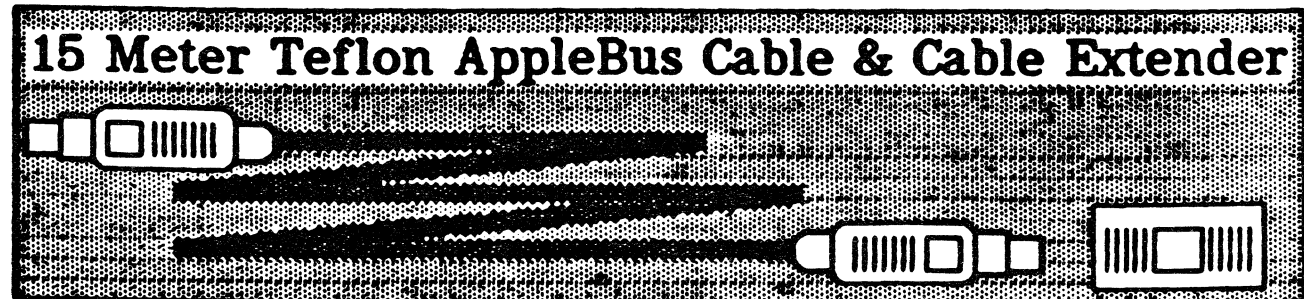
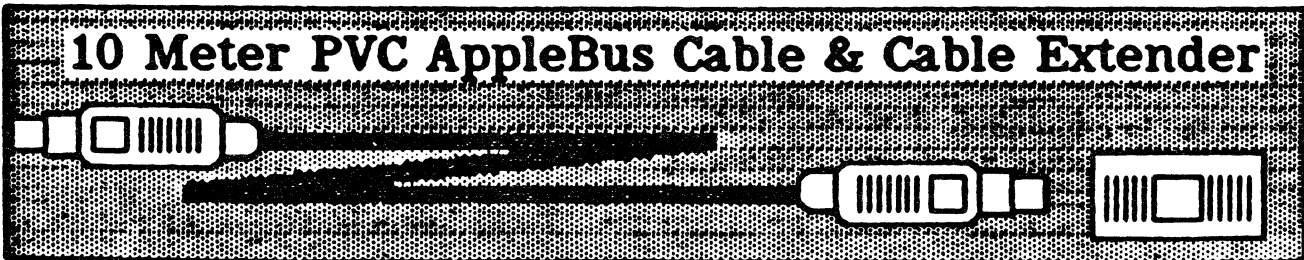
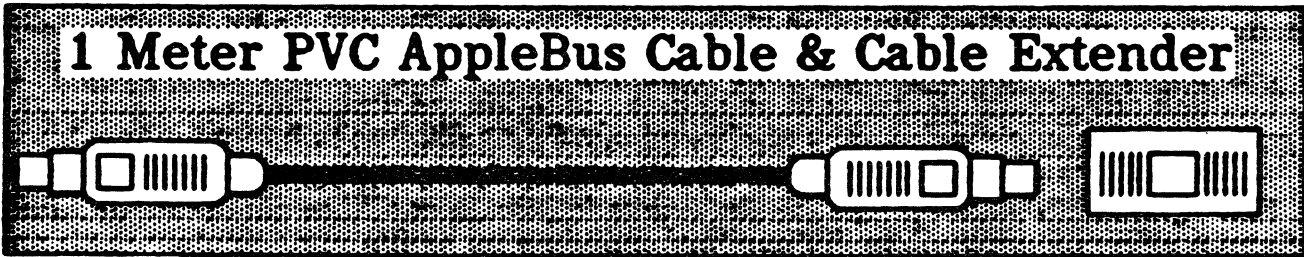
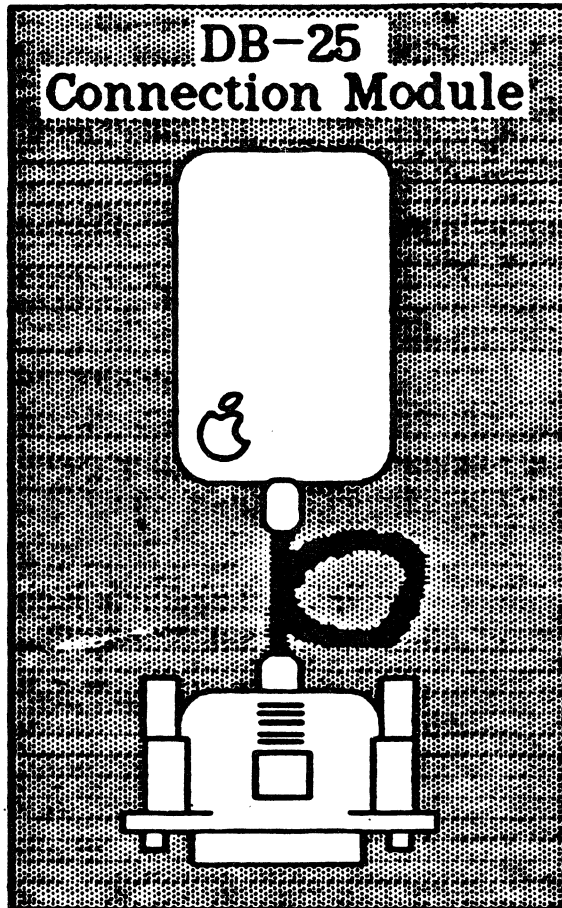
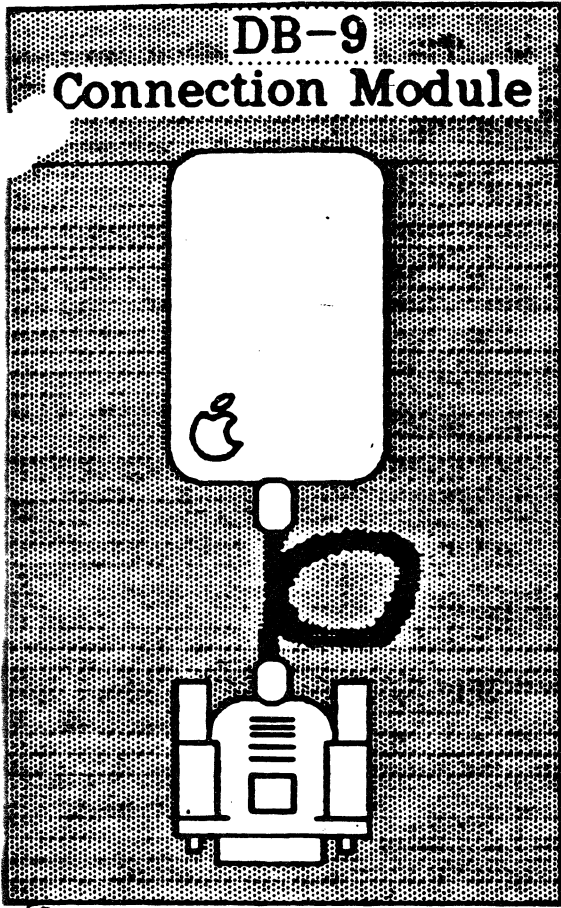


AppleBus Server Kit



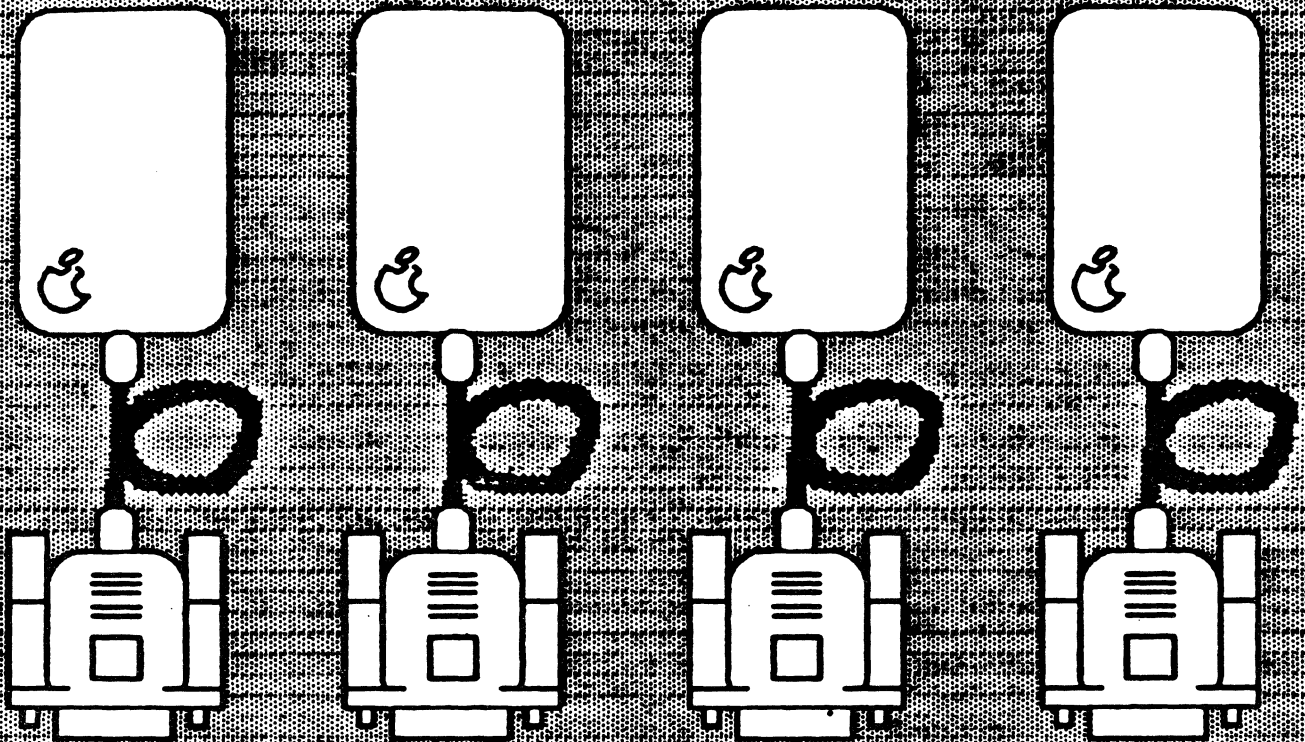
**AppleBus Server Kit packaged with
all Apple Server Products**

AppleBus Components

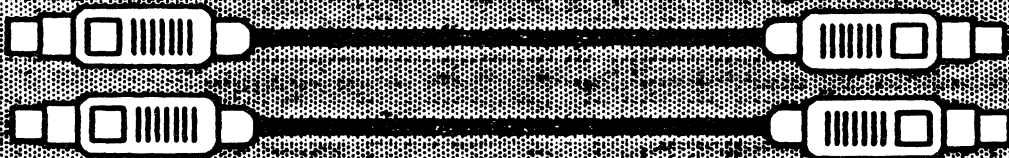


AppleBus Starter Kit

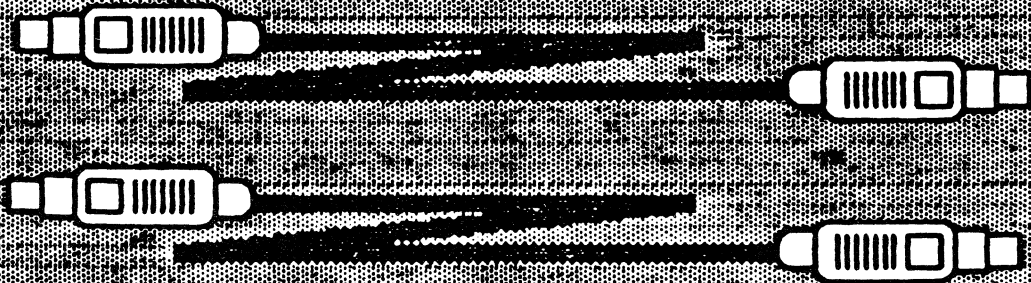
Four (4) AppleBus Connection
Modules (DB-9)



Two (2) 1 Meter AppleBus Cables



Two (2) 10 Meter AppleBus Cables

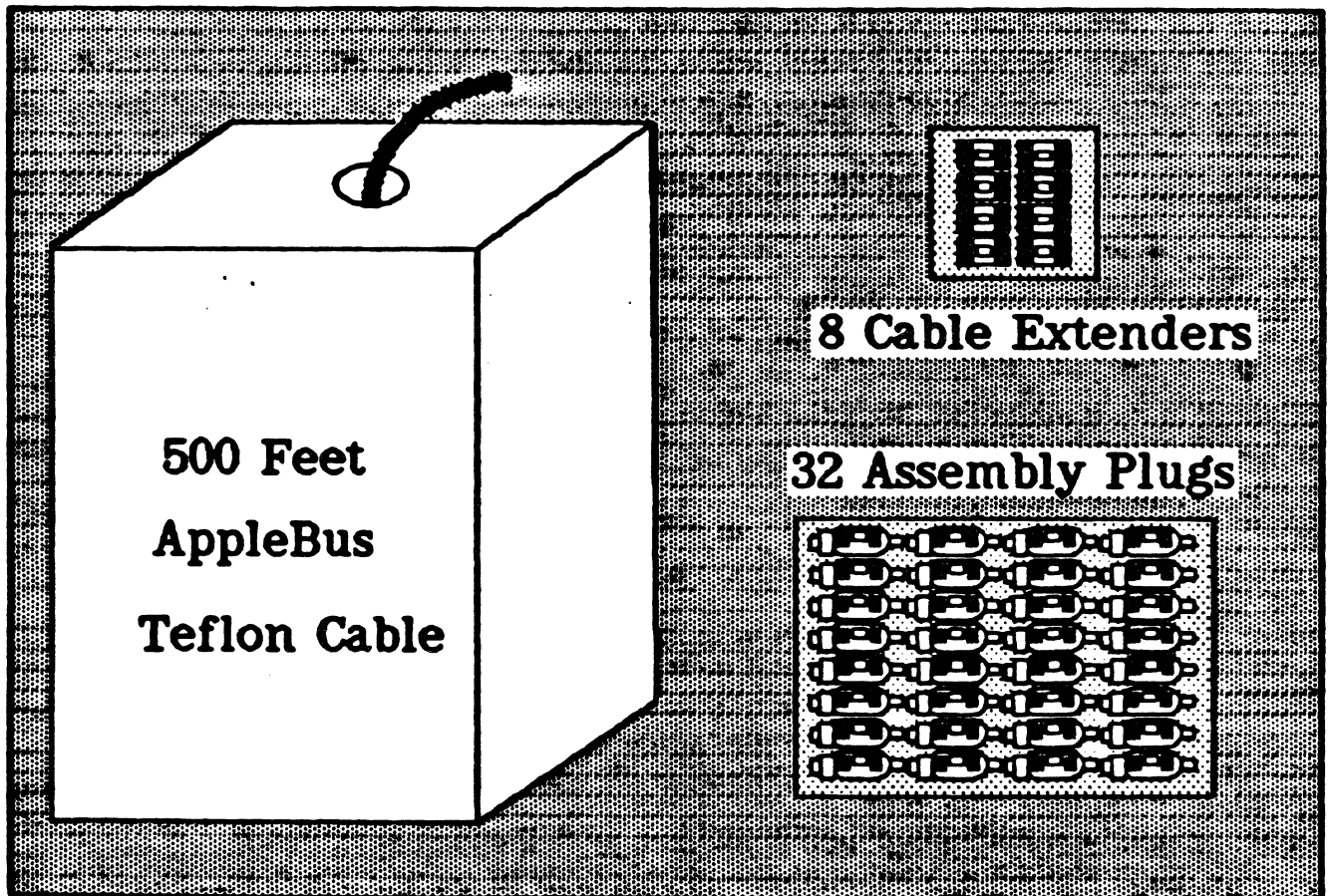


AppleBus Cable Extender



AppleBus Installation Guide

AppleBus Assembly Kit



AppleBus

Link Access Layer

ABLAP

CSMA/CA

Carrier Sense, Multiple Access
w/ Collision Avoidance

Dynamic Node Address Assignment

8-bit node addresses
Unique only on single bus link

HDLC/SDLC frame format

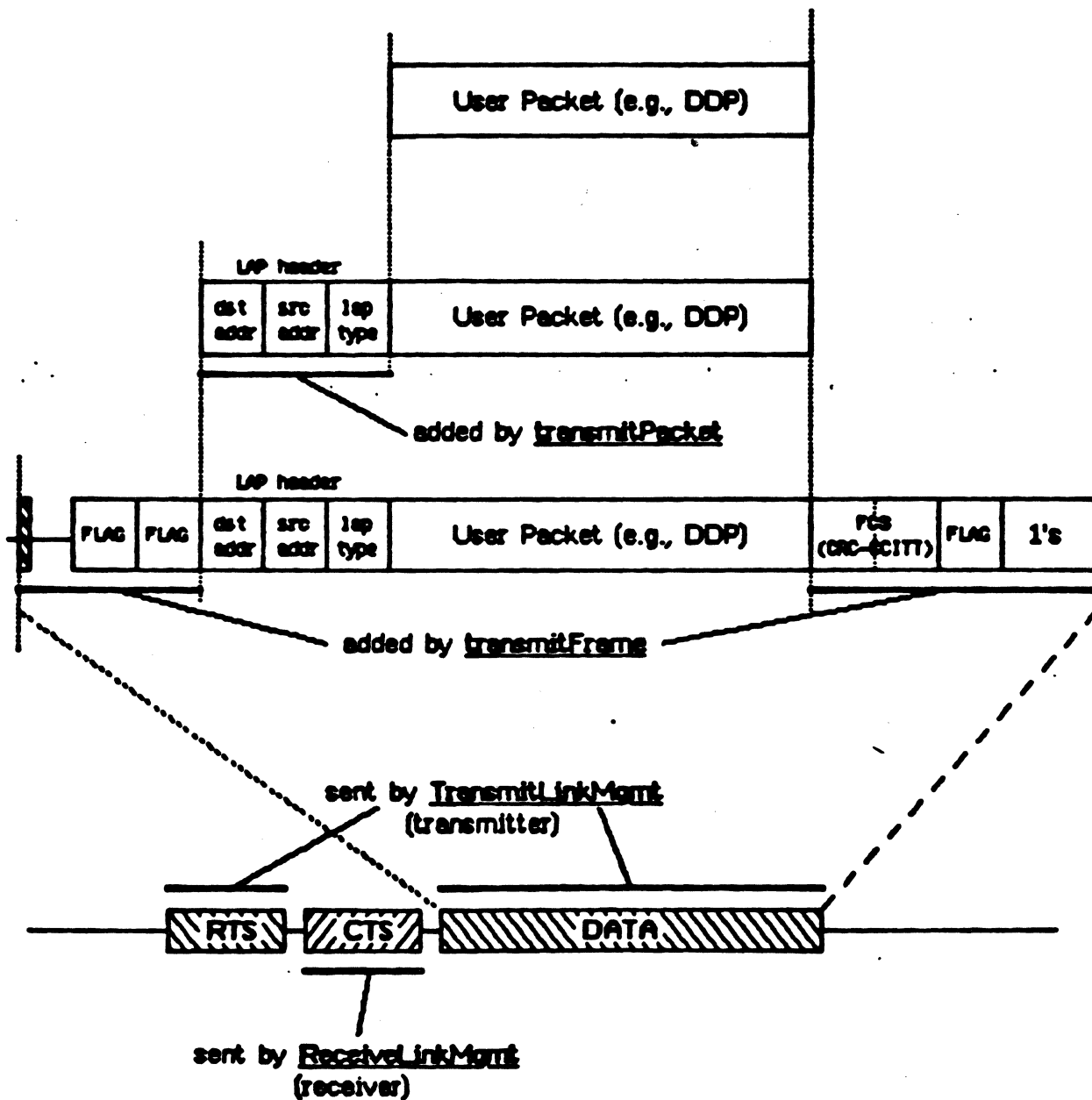
FLAG framing,
8-bit address,
bit-stuffing

CRC-SDLC Frame Check Sequence

(using CRC-CCITT polynomial - $x^{16} + x^{12} + x^5 + 1$,
w/ initialization of 1's)

AppleBus

Frame Format



AppleBus

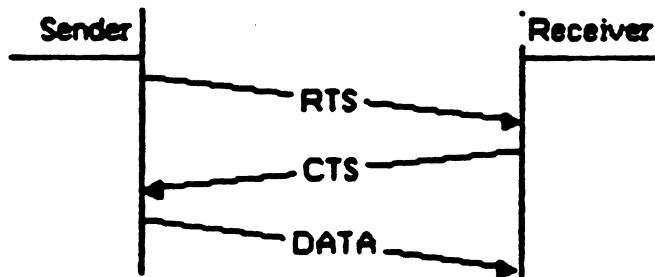
LAP Dialogs

Problems to be solved:

- station may not be listening
- contention (collisions)
- no true collision (carrier) sense

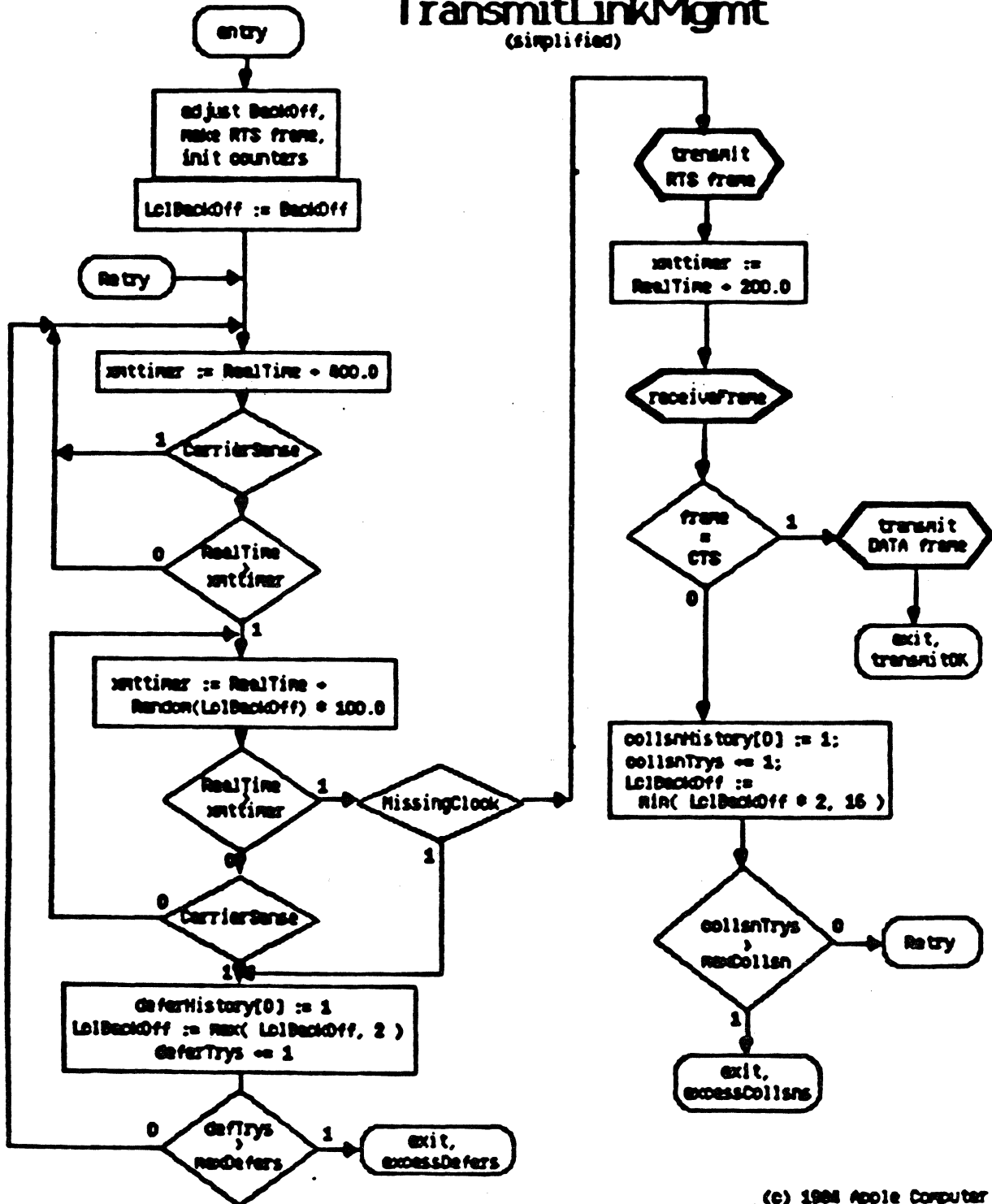
Solution:

3-way handshake



AppleBus

TransmitLinkMgmt (simplified)



(c) 1988 Apple Computer Inc.

**Performance Characteristics
of
AppleBus**

**as observed by the
Übung
simulation system**

James Nichols

March, 1984

Apple Computer, Inc.

Major LAN control schemes

Polling

- + guaranteed response time
- + simple work for slave nodes
- fixed overhead
- complex work for master

Tokens

- + guaranteed response time
- + high channel efficiency
- complex; difficult to implement

Random access schemes

- + easy to implement
- inherently unstable @ heavy load

Apple Computer, Inc.

Major Access Schemes

Pure random access:

ALOHA (c. 1970)

Listen before transmitting:

APPLEBUS (1984)

Listen during transmission:

ETHERNET (c. 1976)

APPLENET (1980-1983)

Apple Computer, Inc.

ALOHA

- ★ University of Hawaii packet radio network (9600 bps).
- ★ VHF transceiver connects nodes to network (split channel).
- ★ Node sends at *any time*. Positive ACK required. Retransmit after random interval.
- ★ low capacity: $1/2e = 1766$ bps.
- ★ slotted variant improves to $1/e$

Apple Computer, Inc.

Ethernet

- ✓ CSMA/CD: Carrier Sense, Multiple Access with Collision Detection
- ✓ Transmitting node monitors line and transmits when idle (1-persistent).
- ✓ Transmission terminated when collision is detected. Retransmit after binary - exponential delay.

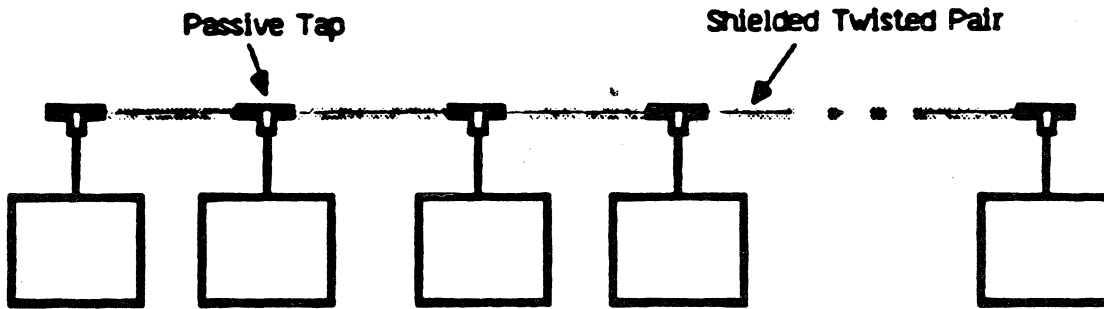
Performance

- ✓ Inherently unstable.
- ✓ Inefficient if propagation delay is large relative to packet tx time.

Apple Computer, Inc.

AppleBus

Physical Layer



Data Rate: 230.4 kilobits/second

Physical Data: maximum 32 nodes
maximum 1000 feet

Electrical: Balanced Signalling
Standard RS-422 driver
and Receiver ICs
Transformer Isolation
Passive Drops
FMO modulation

Apple Computer Inc.

AppleBus

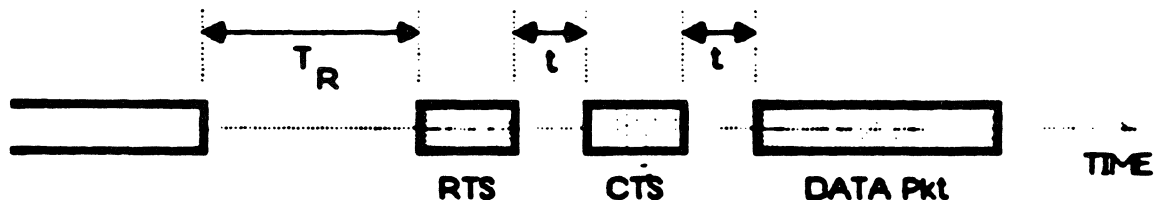
Link Access Layer

ABLAP

Addressing: 8 bit Node IDs,
Dynamically Assigned
Broadcast packets allowed

Framing: SDLC (using bit stuffing)

Access Control: CSMA/CA



T_R random (> 400 usec)

t less than 200 usec

Übung Simulator

TestBed

**Up to 32 nodes, Macs, Apple //e,
Lisas: one master, 31 slaves.**

Operation

Master controls testbed:

- * configure nodes from config file**
- * monitor performance and gather statistics.**

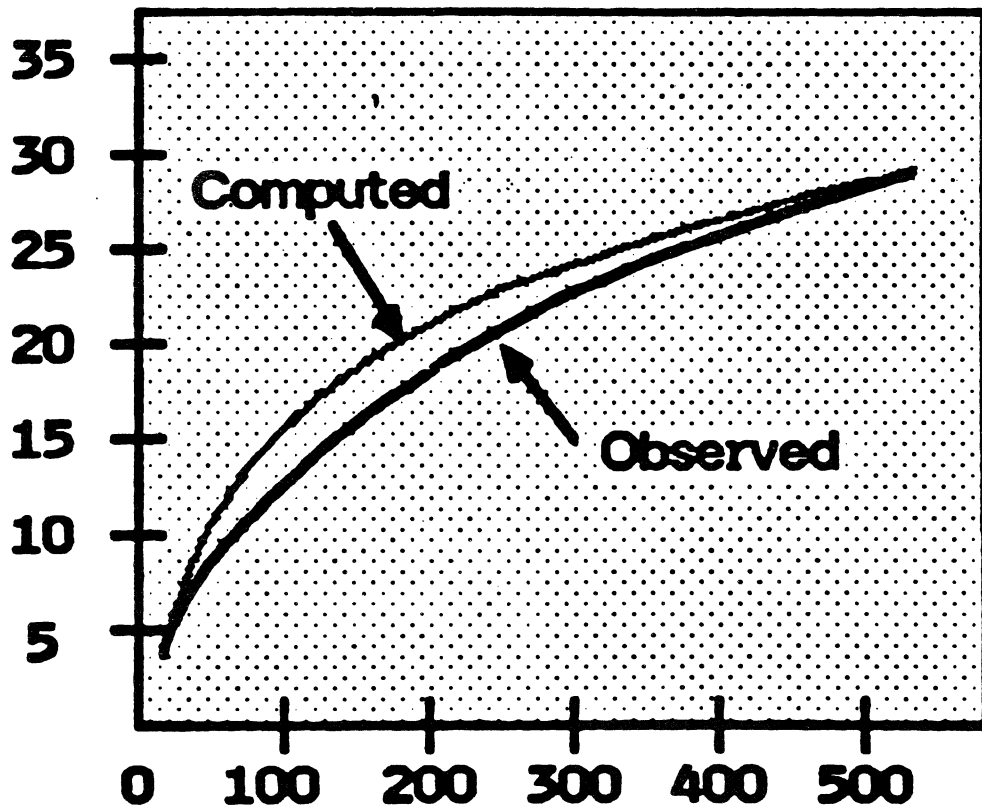
Statistics monitored

- * transmit queue delay**
- * throughput**
- * errors: CRC, fragmented packets, retries, etc.**

Apple Computer, Inc.

AppleBus Performance Analysis

KBytes/sec



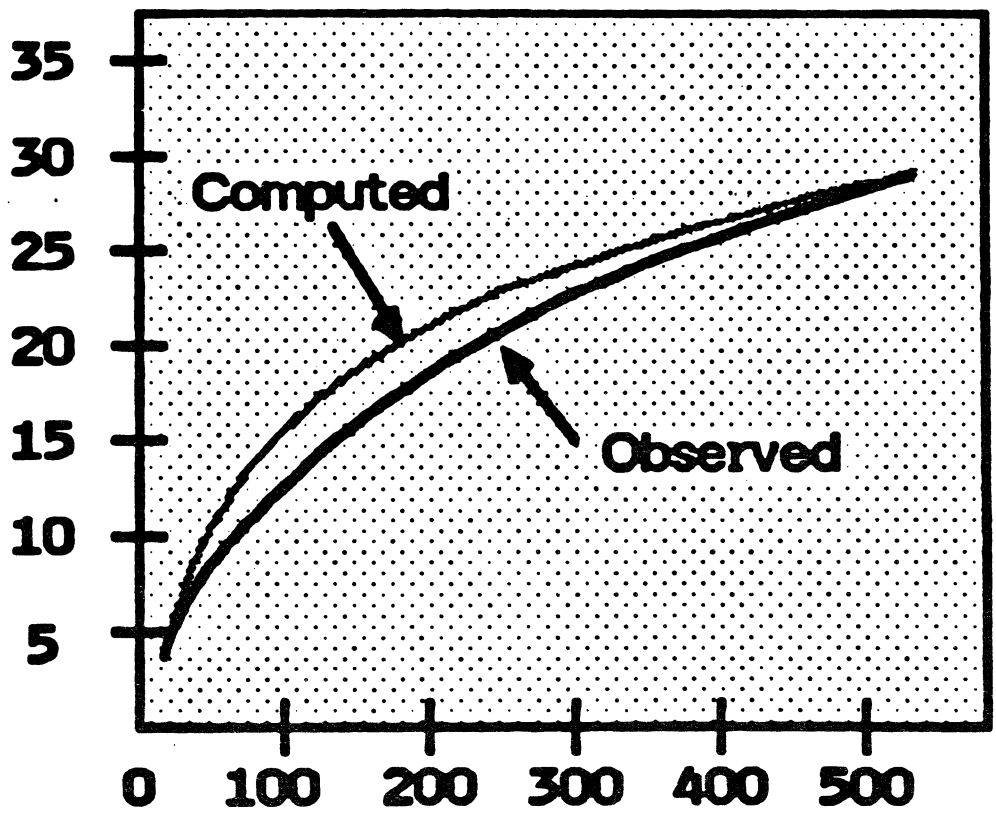
Packet Size, bytes

12 Nodes

Apple Computer, Inc.

AppleBus Performance Analysis

KBytes/sec



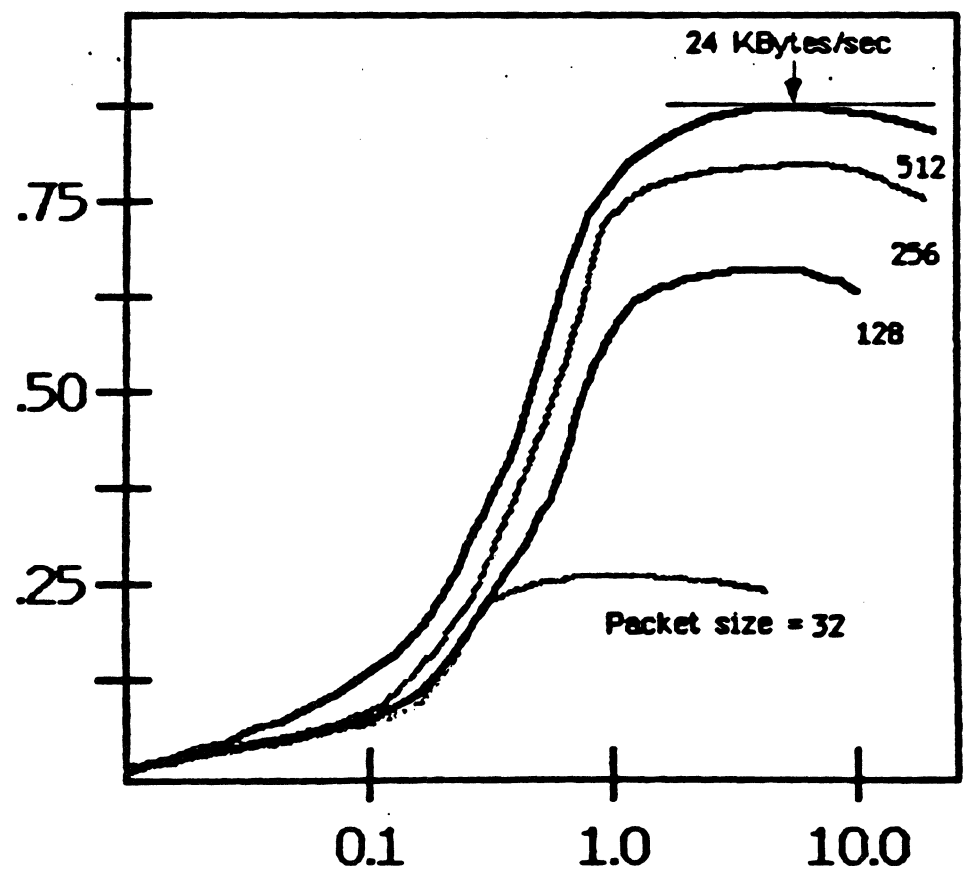
Packet Size, bytes

12 Nodes

Apple Computer, Inc.

AppleBus

Throughput Y vs Offered Load X



14 Nodes

Apple Computer, Inc.

Major LAN control schemes

Polling

- + guaranteed response time
- + simple work for slave nodes
- fixed overhead
- complex work for master

Tokens

- + guaranteed response time
- + high channel efficiency
- complex; difficult to implement

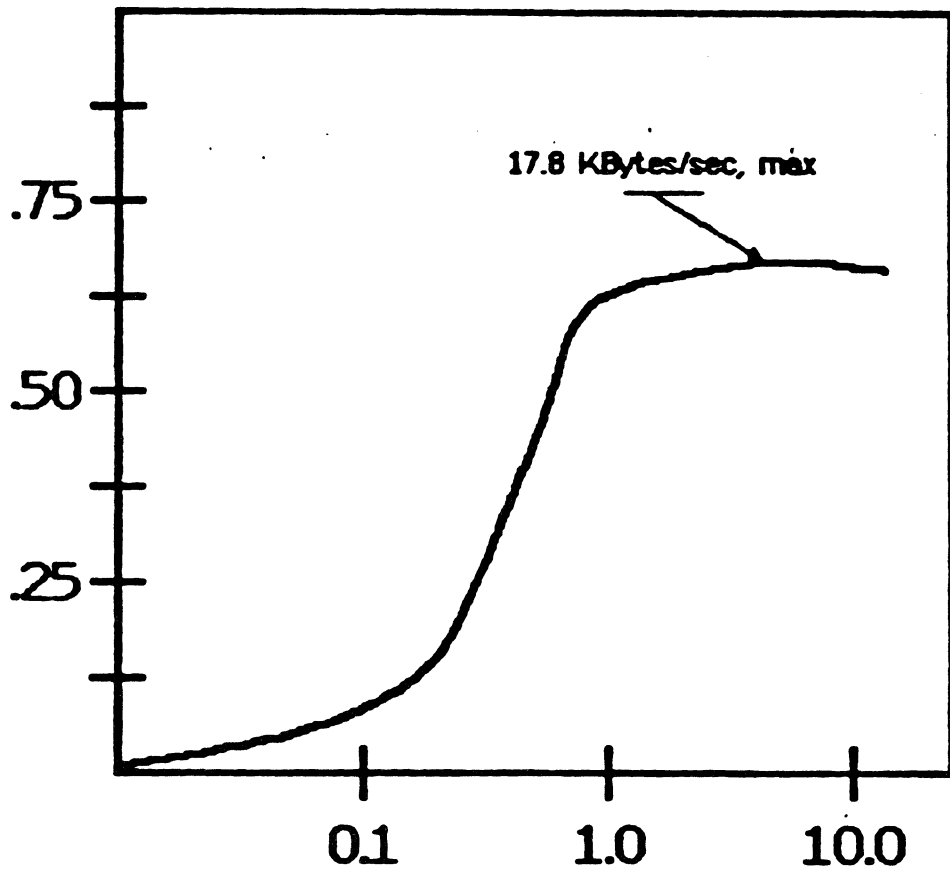
Random access schemes

- + easy to implement
- inherently unstable @ heavy load

Apple Computer, Inc.

AppleBus

Throughput Y vs Offered Load X
Server emulation, 50 ms. service time



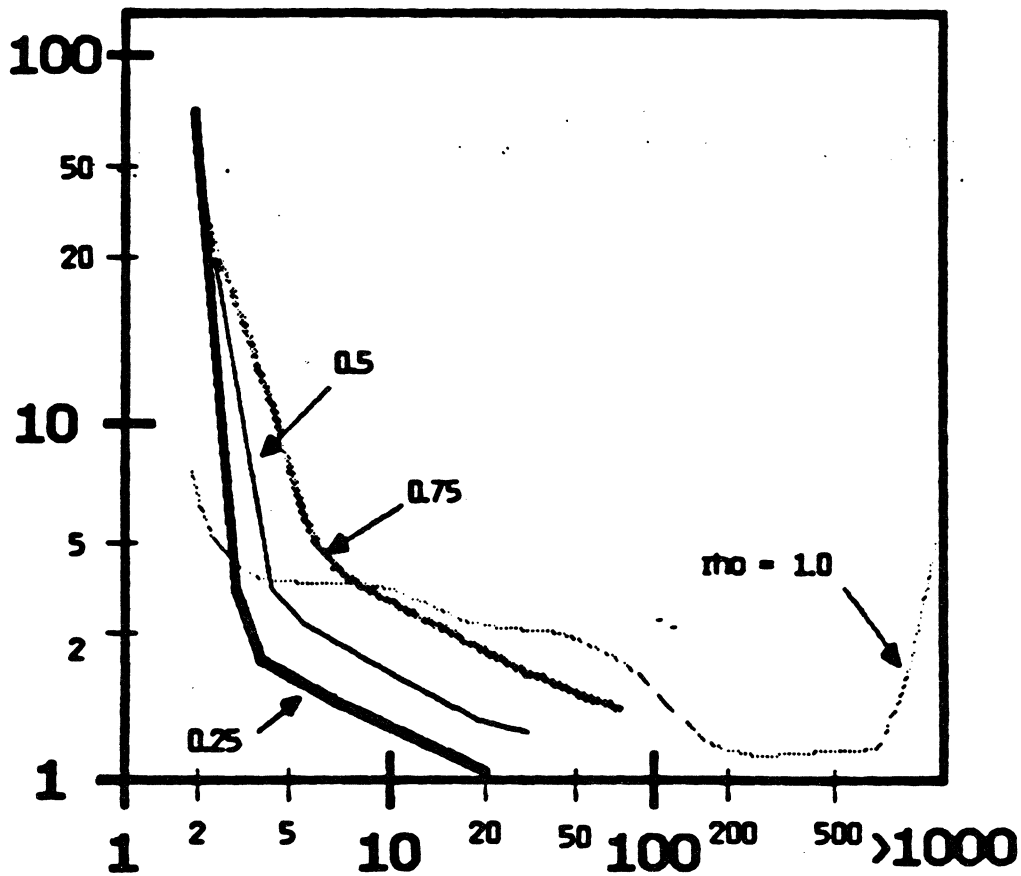
Packet size = 512, 14 Nodes

Apple Computer, Inc.

AppleBus

LAP delay distribution

$p[\text{Delay} \times \text{packet times}]$



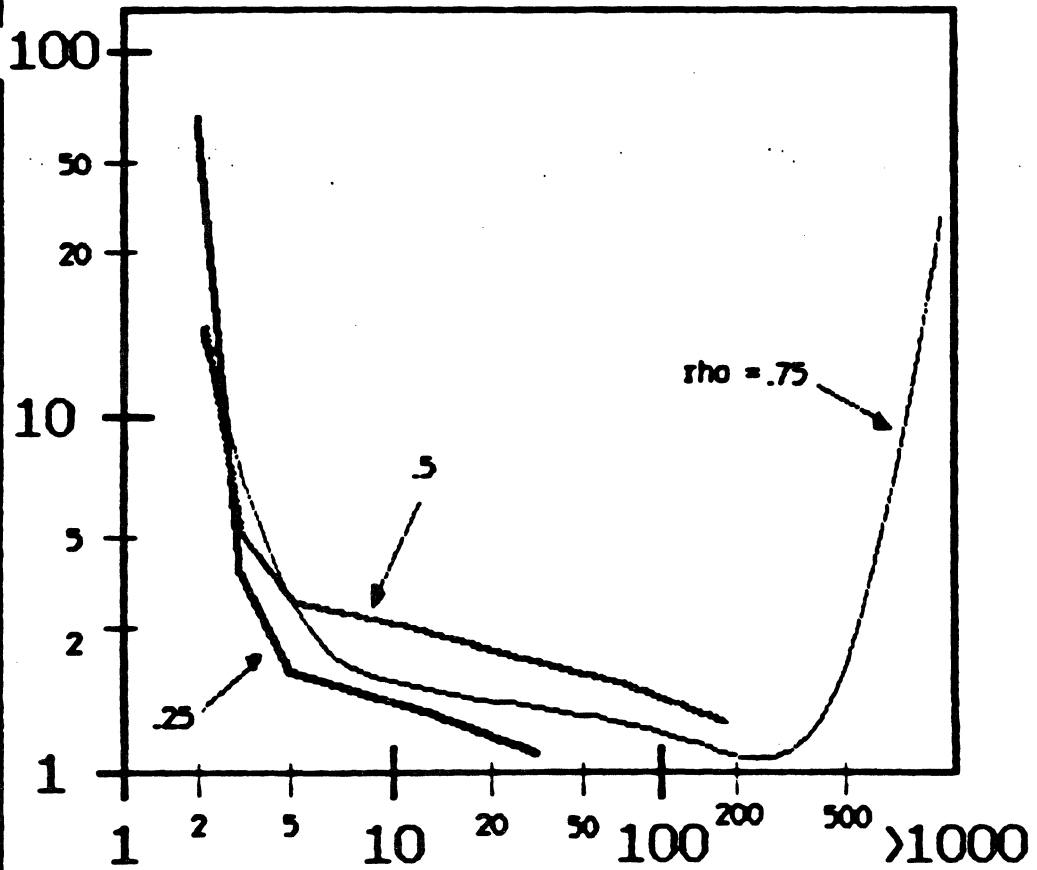
Echo mode, 512 bytes packets, 14 nodes

Apple Computer, Inc.

AppleBus

LAP delay distribution

$p[\text{delay} \times \text{packet times}]$



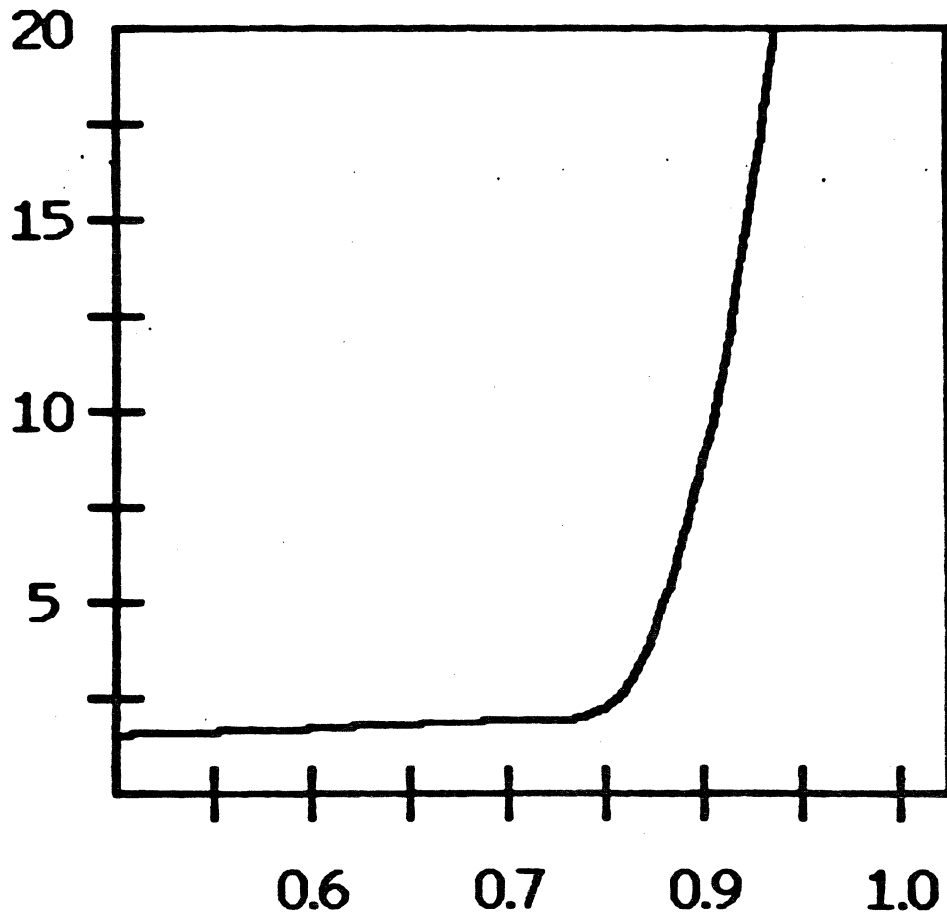
Non-echo mode, 13 Nodes, 512 byte packets

Apple Computer, Inc.

AppleBus

Delay Y vs. Throughput X

Delay in packet transmit time



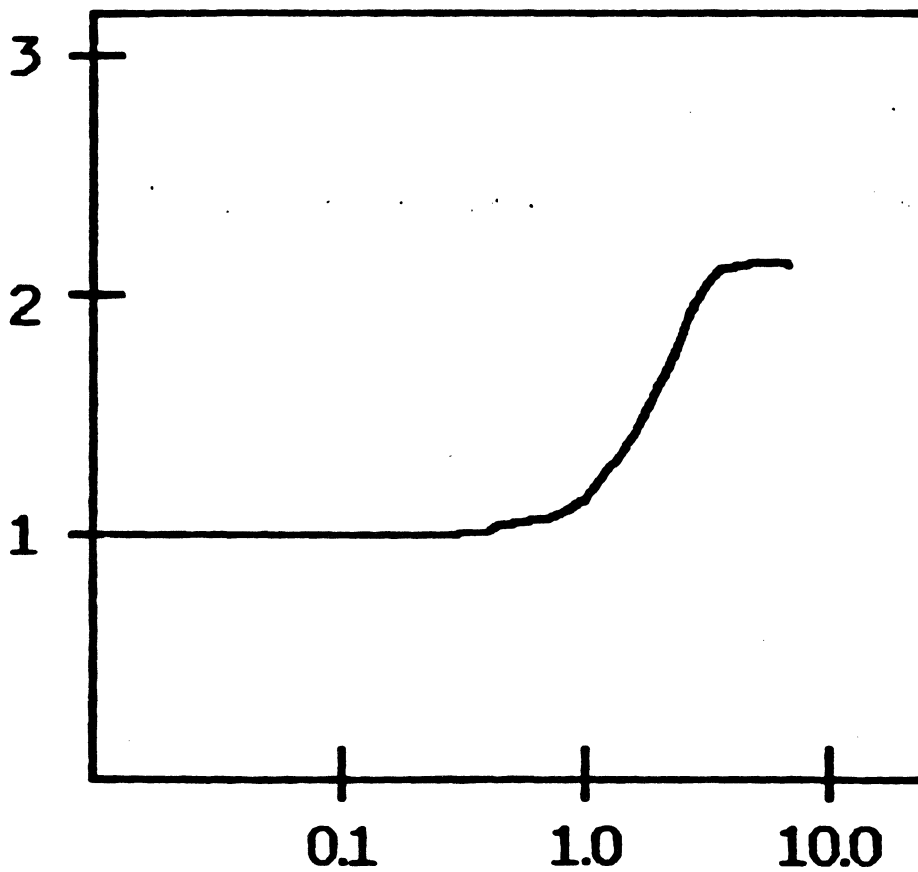
Packet size = 512, 14 nodes

Apple Computer, Inc.

AppleBus

Capture Effect

MinMax Delta Y vs Offered Load X

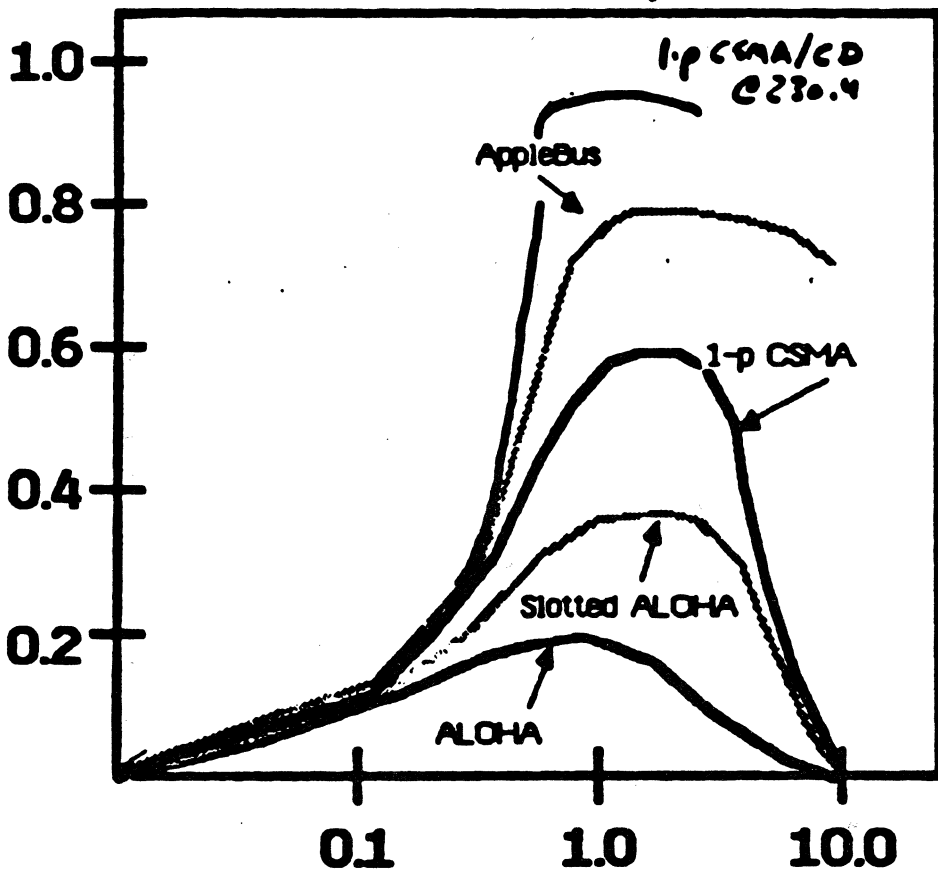


Packet size = 512, 14 Nodes

Apple Computer, Inc.

AppleBus

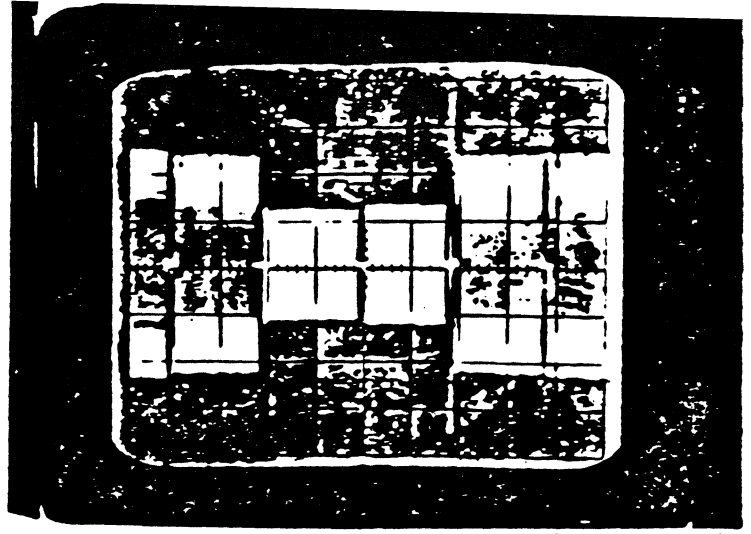
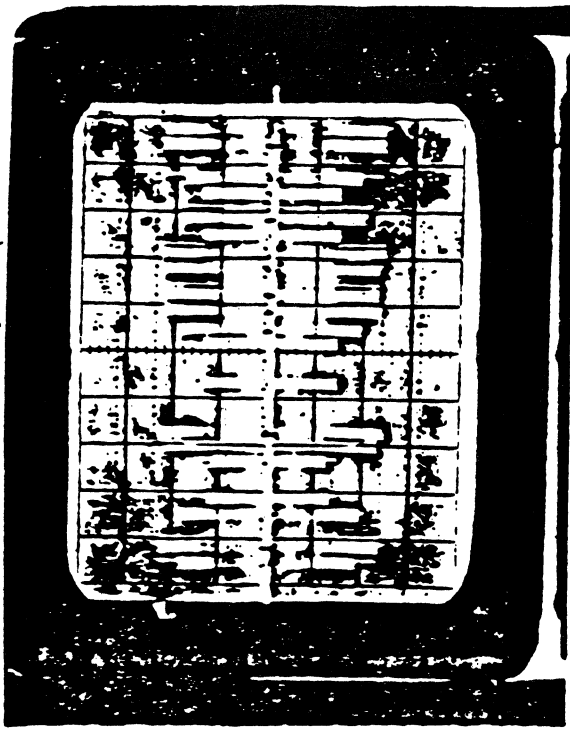
Throughput Y vs Offered Traffic X
for selected access schemes



$\tau = 0.01$, 256 byte packets

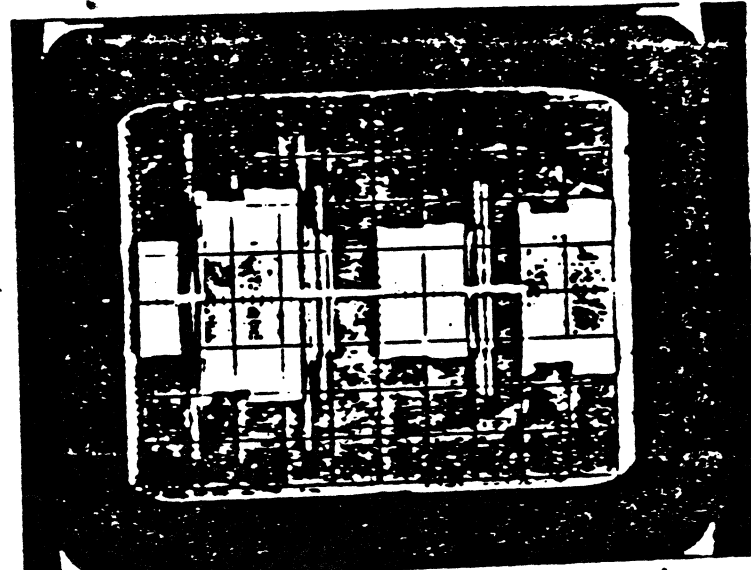
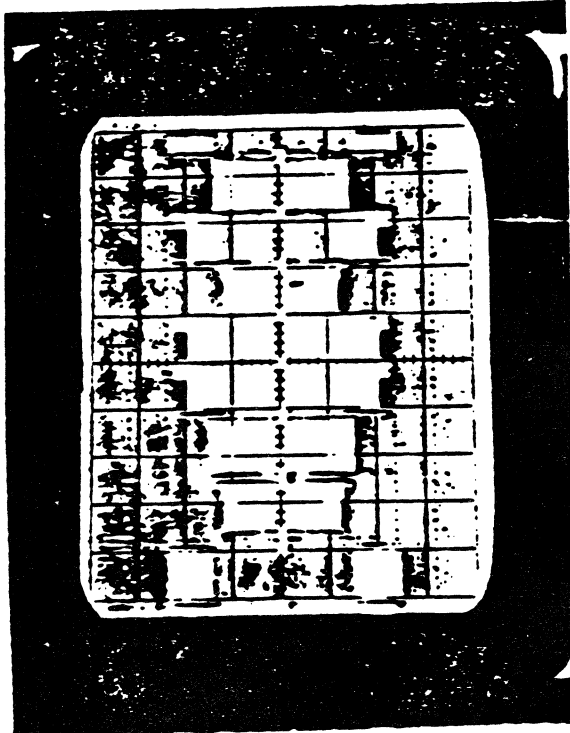
Apple Computer, Inc.

50/10ms/div P=25 PS=32, 14 nodes

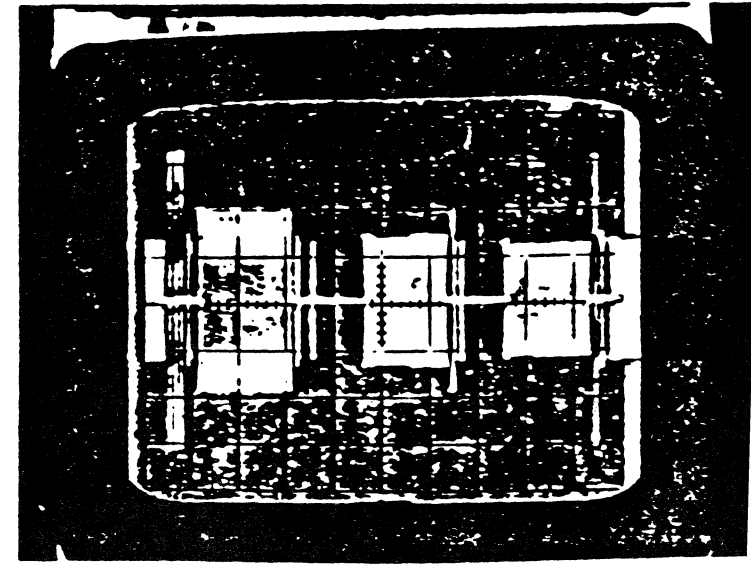


echo=2Ntd, P=500, PS=512, 10ms/10V .5V/div

P=100, 12 nodes S1&Bwe



No Echo P=500, PS=512, 10ms, .5V/DIV



No Echo, P=100, PS=512 10ms, .5V/div

Observations

from *Übung*

- ✓ Strong 'capture' effect biases service towards faster nodes.
- ✓ Backoff algorithm breaks down when offered load is greater than ≈ 0.80 .
- ✓ Very reliable when offered load is less than 80% channel capacity.
- ? Randomness of backoff algorithm affecting performance?
- ? Link-level NAK increases channel utilization with heavy loads?

Apple Computer, Inc.

Observations

from *Übung*

- ✓ Overruns reported on \approx 6-8% of packets received (independent of load).
- ✓ Underruns reported on \approx 2% of packets transmitted, at offered load $>$ 0.90.
- ✓ Runt packets (length $<$ 8 bytes, usually = 4 bytes) are received under heavy ($>$ 1.0) load, but infrequently.
- ✓ Runt packets are received more frequently (0.03%) with smaller (32 byte) packets.

Runt format: SrcAddr, DstAddr, SrcAddr, \$84

Apple Computer, Inc.

AppleBus

MacCollege

- Technical assistance to accelerate the completion of Macintosh programs
- Restricted to experienced Macintosh developers
- 3-day sessions beginning in June, \$500/session
- Conducted in a dedicated facility complete with 15 Macintosh/Lisa development systems

(c) 1988 Apple Computer Inc.

AppleBus

Co-Marketing Programs

Macintosh and Lisa Development
Team List

Own-A-Mac

Macintosh Goodie Box

Co-Promotion

(c) 1988 Apple Computer Inc.

AppleBus

Macintosh Registered Developer Program

- Technical assistance for serious, commercial development of Macintosh and Lisa products
- Telephone and consultation technical support
- 24 hour turnaround goal
- \$500 for 6 month period
- Enrollment limited to Apple Certified Developers

(c) 1984 Apple Computer Inc.

AppleBus

Apple Certified Developer Program

Workshops and Seminars

Conferences

Product Discounts

Newsletter and Mailings

Technical Information

(c) 1984 Apple Computer Inc.

AppleTalk Peek

Version 2.0

Richard F. Andrews and Gursharan S. Sidhu
Network Systems Development
© 1984, 1985 - Apple Computer Inc.

The **AppleTalk Peek** program is a network tool used to monitor packet traffic on a single AppleTalk network. Peek runs on a Macintosh (with AppleTalk connected via the Printer port), and can record all packets seen on the bus. In addition, it can detect certain errors, measure packet arrival times, and display packet data in hexadecimal and ASCII format.

Peek has enough queue space to hold a large number of packets. This queue is used in a circular fashion, so that Peek can continue to monitor packets even after the queue has been filled. Older packets are discarded to make room for newer ones.

The Window

When Peek is started, the program's window is drawn. It contains the control buttons, menus, and information display areas described below.

START

STOP

Peek is always in one of two states: recording or displaying packets. When the program is first started, it is in the record state. The **START** and **STOP** buttons are used to initiate and terminate a recording session, during which Peek listens on the bus and records traffic.

When the **START** button is pressed, packet recording is enabled. The button becomes gray to indicate that Peek is recording (see Figure 1). Peek's internal buffers are cleared, and packets from a previous session

are lost. The **STOP** button halts recording and causes packets to be displayed, if any were recorded during the session (see Figure 2).

Pkts in Q
0

When the **STOP** button is pressed, the **Pkts in Q** box shows the number of packets in Peek's queue. This box is not dynamically updated during a recording session, since queue wraparound makes this determination difficult. The word "Sampling" appears here while recording, as an indication that Peek is monitoring the bus.

The size of the queue is determined by the amount of free memory, so that Peek running on a 512K Macintosh will be able to record more packets than a 128K Macintosh. Part of the queue memory is devoted to "bookkeeping" information.

Pkts Rcvd
0

The total number of packets seen by Peek since the start of the recording session appears in this box. This count is updated dynamically, and hence provides a rudimentary visual indication of bus traffic. Since Peek's queue can wrap around and "forget" old packets, this count may be greater than the number of packets stored in the queue at that time.

During a particularly long recording session, this count may itself wrap around (when it reaches 32,767) and become invalid.

CRC errors:	0
Overruns:	0
Time Outs:	0

This box displays the tally of errors during a recording session. Peek can detect three types of errors:

CRC errors are noted when the 16-bit Frame Check Sequence (FCS) at the

end of the ALAP frame does not match the calculated FCS. This indicates a possible error in transmission (due to noise on the bus, collisions, etc.).

Overrun errors occur when the receiver reads bytes too slowly from the Macintosh's Serial Communications Controller (SCC), and this chip's three-byte FIFO buffer overflows. Note that if the node running Peek detects an overrun error, it does not necessarily mean that this will be the case for other nodes. This error is sometimes detected as a by-product of collisions on the bus.

Timeout errors are flagged when a byte is expected on the bus (the end of the packet has not been seen, yet a byte does not appear on the bus within about 400 microseconds of the previous byte). Every byte received thus far will be stored and displayed as "the packet", even though the true packet end was not detected. This usually indicates a problem in the packet sender's hardware, but it may also be a by-product of collisions on the bus.

The error fields are updated "on the fly" as packets are recorded, and are a measure of the total number of errors seen by Peek. Therefore, in a long recording session, it is possible, due to queue wraparound, for a packet to be received in error and not appear in the packet display, although the error is noted in the box.

Go-Away box

When you wish to terminate the Peek program, click in the small box in the upper left-hand corner of the window.

Display box

This box is used to view packets which were saved during the recording session. Each packet is preceded by a banner line (see Figure 2) in boldface which includes, from left to right:

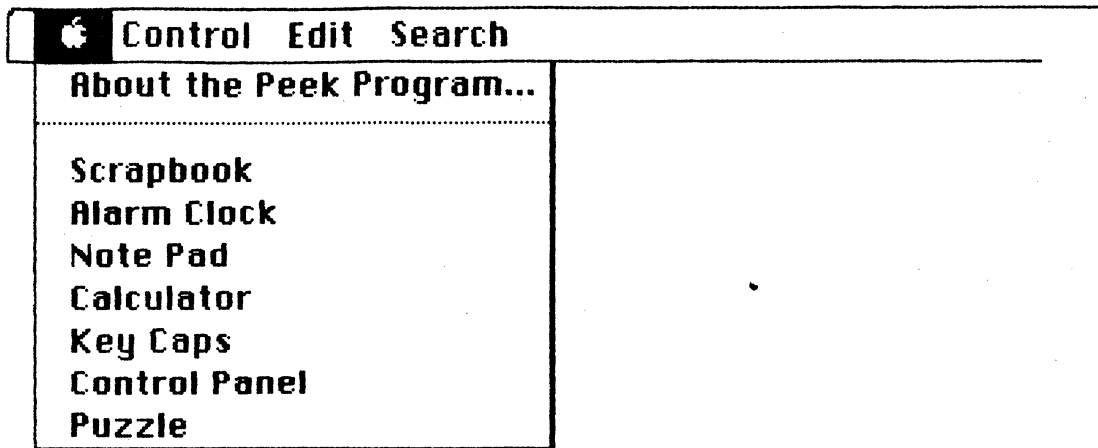
- a set of square brackets which may contain blanks or one of the following characters: **B** if the packet was a broadcast, **C**, **O**, or **T** if a CRC, overrun, or timeout error, respectively, was detected for that packet;
- the source **S** and destination **D** node IDs of the packet, in decimal format;

- the packet's arrival time T in milliseconds (measured relative to the first packet stored in the queue);
- the time since the previous packet's arrival (delta time or ΔT);
- the packet's sequence number in parentheses (in the order in which they were received, starting with zero for the oldest packet in the queue);
- the calculated length L of the packet (number of bytes in decimal, not including the FCS).

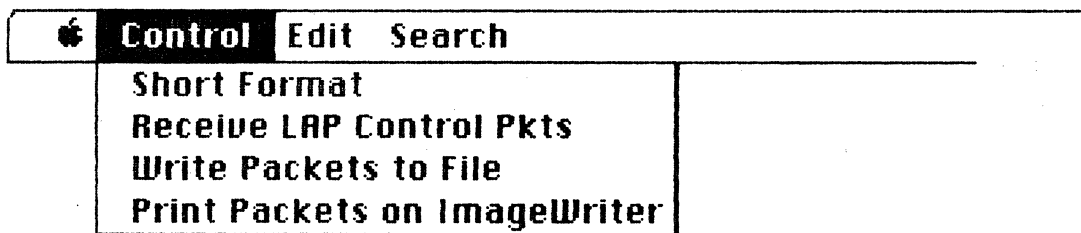
Following this banner line are two displays of the packet's contents, in hexadecimal on the left and the corresponding ASCII (if printable) on the right. A period is substituted for any unprintable character. Note that the FCS is not shown.

On the right side of the display box is a scroll bar which is enabled whenever there is more to be displayed than will fit in the box. The user can scroll through the display of packets by using the scroll bar's up and down arrows, or by dragging the thumb. Clicking in the gray area above or below the thumb shifts the display backwards or forwards one complete packet at a time. This is useful for scrolling past large packets.

Menus



The Apple menu, as usual, is used to invoke a variety of desk accessories. Choosing **About the Peek Program** will cause Peek to display some descriptive information, including the version number and the size of the queue in bytes.



The second menu is the **Control** menu, as seen above. When **Short Format** is selected, a check mark appears next to the menu item and packets are displayed in a more compact form. Only the banner line and the first line (up to the first 16 bytes) of each packet will be shown. The display can be scrolled as before. Choosing the menu item again will change the display back to long format. This item is inactivated during a recording session.

When **Receive LAP Control Pkts** is selected, all LAP control packets seen on the bus will be recorded. These are defined as any packets whose LAP type field is \$80 through \$FF hex (most significant bit set). Such packets will be recorded only when this option is selected. If their reception is

enabled in the middle of a recording session, any LAP control packets already seen on the bus will not have been saved. Changing this option while viewing the display of a previous session will have no effect on the display, but will affect the next recording session.


Selecting **Write Packets to File** will cause Peek to save away a copy of the display into a file on disk. If there are disks in the internal and external drives, Peek will attempt to write to whichever one has more free space. The name of the file will be "Peek Buffer 0", unless a file by that name already exists; in which case Peek will try "Peek Buffer 1" through "Peek Buffer 9". If those ten files already exist, Peek will give up and display an error message.

If there is not enough room on the disk to save all the packets, Peek will write as many as will fit, and notify the user that a "Write Error: -34" occurred. This means that the disk is full; you may wish to put an empty disk in the drive and try again. The file containing saved packets will be of type "TEXT" and creator "EDIT".

As an alternative, you may choose **Print Packets on Imagewriter**. Connect an Imagewriter to the modem port (not the printer port) before selecting this item, and Peek will print the entire display on the printer. (This could be a time-consuming process if there are many packets to print). Note that the **Short Format** option (described below) has no effect on packets written to a file or to the printer - long format is always used.

⌘	Control	Edit	Search
		Cut	⌘H
		Copy	⌘C
		Paste	⌘U

The **Edit** menu is used only in conjunction with the **Find Pattern** feature, described below. It allows you to use the standard text edit commands to create a string for which to search.

 Control Edit Search	Find Pattern ⌘F Find Same ⌘S
	<hr/> Find Overrun Find CRC Error Find Timeout
	<hr/> Search is Not Case Sensitive

The **Search** menu is used to look for a particular hexadecimal or ASCII string within the recorded packets. Selecting **Find Pattern** (or the equivalent command key-F combination) will cause the Find window to appear, as in Figure 3. Select **Hexadecimal** or **Ascii**, and type in the string for which to search. The standard text editing features available in the **Edit** menu may be used. Hex strings must be a sequence of bytes, each specified as a dollar sign (\$) followed by a two-digit hex number. In either format, a wild-card may be specified by the command key-equals sign combination. This will appear in the Find window as "Ω", and will match one or more characters of any value.

When the string has been entered, hit Return or the **Find Next** button. Peek will begin searching from the first packet appearing in the display box. The display will be scrolled down to the packet containing the string, if found, and the string will be highlighted. Otherwise, Peek will inform you that the string was not found. Selecting **Find Same** (or the equivalent command key-S combination) will cause Peek to look for the next occurrence of the same string, starting from the current packet.

Find Overrun, **Find CRC Error**, and **Find Timeout** work in a similar fashion. Selecting one causes Peek to search, starting from the first packet in the display box, for a packet exhibiting the particular error. If found, the display is scrolled to bring that packet to the top of the box (unless it is too close to the last packet to scroll up to the top). Since the error counts are cumulative from the time the **START** button was pressed, packets with errors may not always appear in the queue (if they were discarded to make room for newer packets).

The search can be made case-sensitive or not case-sensitive by selecting

the item Search is Not Case Sensitive. The menu item will display the current state of this option.

Notes

1. The newer versions of Macsbug (1/1/85 or later, with symbols) tend to slow Peek enough that it will frequently overrun. Use older debuggers on your Peek disk or none at all.
2. A node running Peek is in listen-only mode on an AppleTalk network. Such a node does not participate in the ALAP protocol and does not even consume a node ID. In fact, it is "invisible" to other nodes.
3. Peek does not use any of the standard AppleTalk drivers (e.g. the Macintosh Protocol Package), but assumes direct control of the Macintosh's AppleTalk port. However, the port is reset when Peek terminates, so it is possible to then run other AppleTalk software without powering down the Macintosh and powering it up again. (Note that this is not true for versions of Peek older than V2.0).
4. Peek will not run on a Macintosh XL under MacWorks.
5. If a packet that is longer than 4095 bytes is received by Peek, its subsequent behavior becomes unpredictable. If Peek terminates abnormally during a recording session, there is a strong probability that a node on the network has sent a packet of illegal size (e.g. a node is stuck in its transmit loop).

Acknowledgements

This program borrows ideas from a former Lisa WorkShop application developed by Jim Nichols and Steve Butterfield; in particular, the circular use of a buffer. AppleTalk Peek is a completely new program designed to exploit the Macintosh user interface. Thanks to Mark Neubieser and Paul Williams for implementing new features.

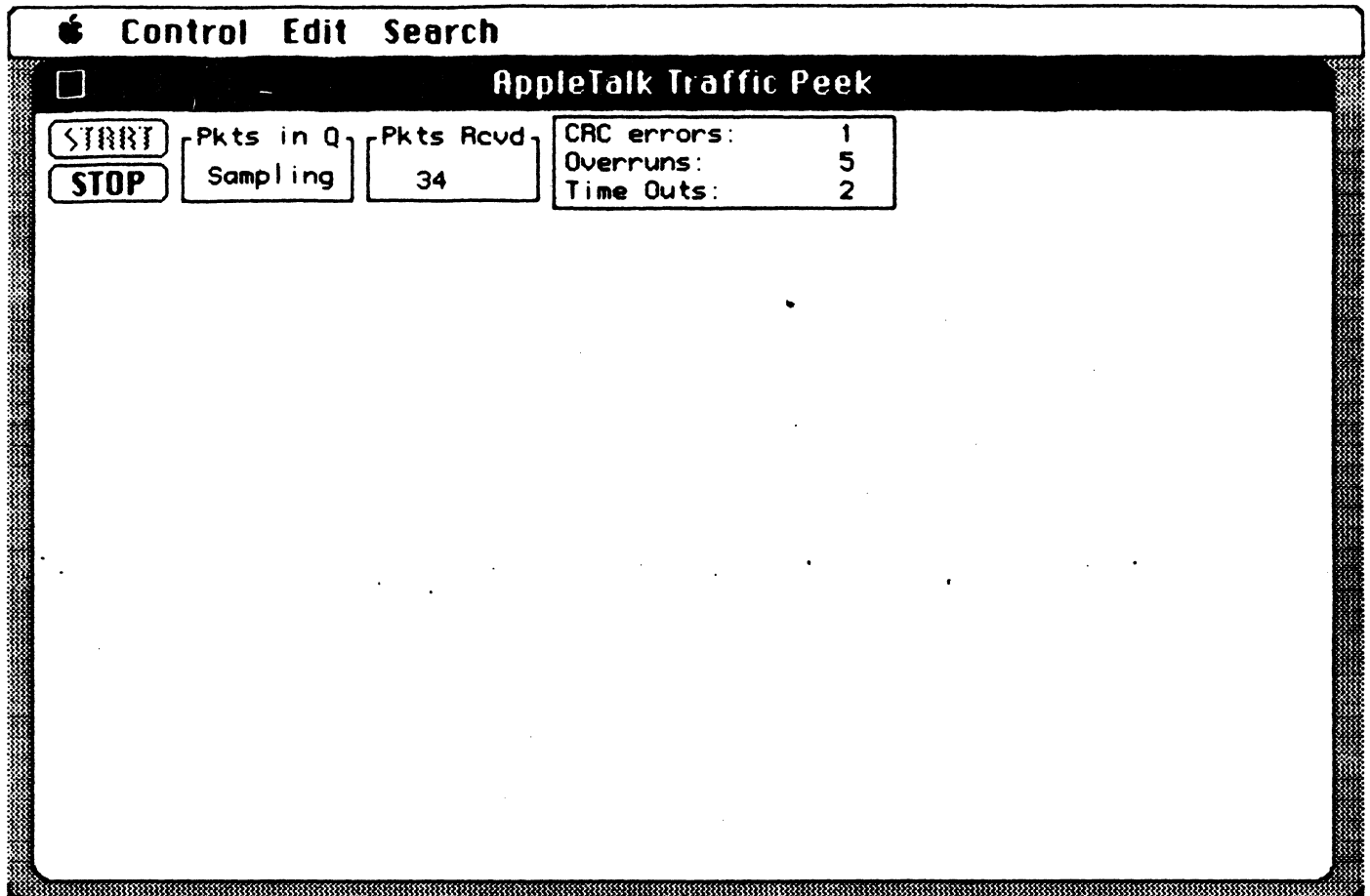


Figure 1: Peek Window (Recording State)

Control Edit Search

AppleTalk Traffic Peek

START	Pkts in Q	Pkts Rcvd	CRC errors:	3
STOP	2526	2597	Overruns:	3
			Time Outs:	0

```

20 61 FF 73 64 66 6A 39 38 20 61 39 64 20 20 61
73 70 99 3D 61 30 66 2D 3D 3B 6C 61 73 64 20 61
73 27 14 66 3B 6F 61 64 70 72 30 33 33 6C 34 2C
20 20 27 61 73 66 70 0F 6F 61 5B 20 6F 61 30 64
20 73
IB 1 S: 10 D: 255 T: 25885 AT: 32 (1550) L: 3
FF 0A 84
IB 1 S: 10 D: 255 T: 25885 AT: 0 (1551) L: 130
FF 0A 01 00 7F 00 00 03 3B 6C 64 73 66 6E 65 6B
7A 78 63 69 6A 20 61 06 73 64 6A 66 20 69 64 6B
20 61 64 6B 6A 66 11 6E 20 61 6B 6A 6B 6E 6D 69
39 38 38 39 34 35 1B 35 6A 20 61 65 39 75 76 20
20 61 FF 73 64 66 6A 39 38 20 61 39 64 20 20 61
73 70 99 3D 61 30 66 2D 3D 3B 6C 61 73 64 20 61
73 27 14 66 3B 6F 61 64 70 72 30 33 33 6C 34 2C
20 20 27 61 73 66 70 0F 6F 61 5B 20 6F 61 30 64
a.sdfj98 a9d a
sp.=a0f-=;lasd a
s'.f;oadpr033l4,
'asfp.oal oa0d
s
(1550) L: 3
.....;ldsfnek
zxcij a.sdjf idk
adkjf.n akjknmi
988945.5j ae9uv
a.sdfj98 a9d a
sp.=a0f-=;lasd a
s'.f;oadpr033l4,
'asfp.oal oa0d

```

Figure 3: Peek Window (Display State)

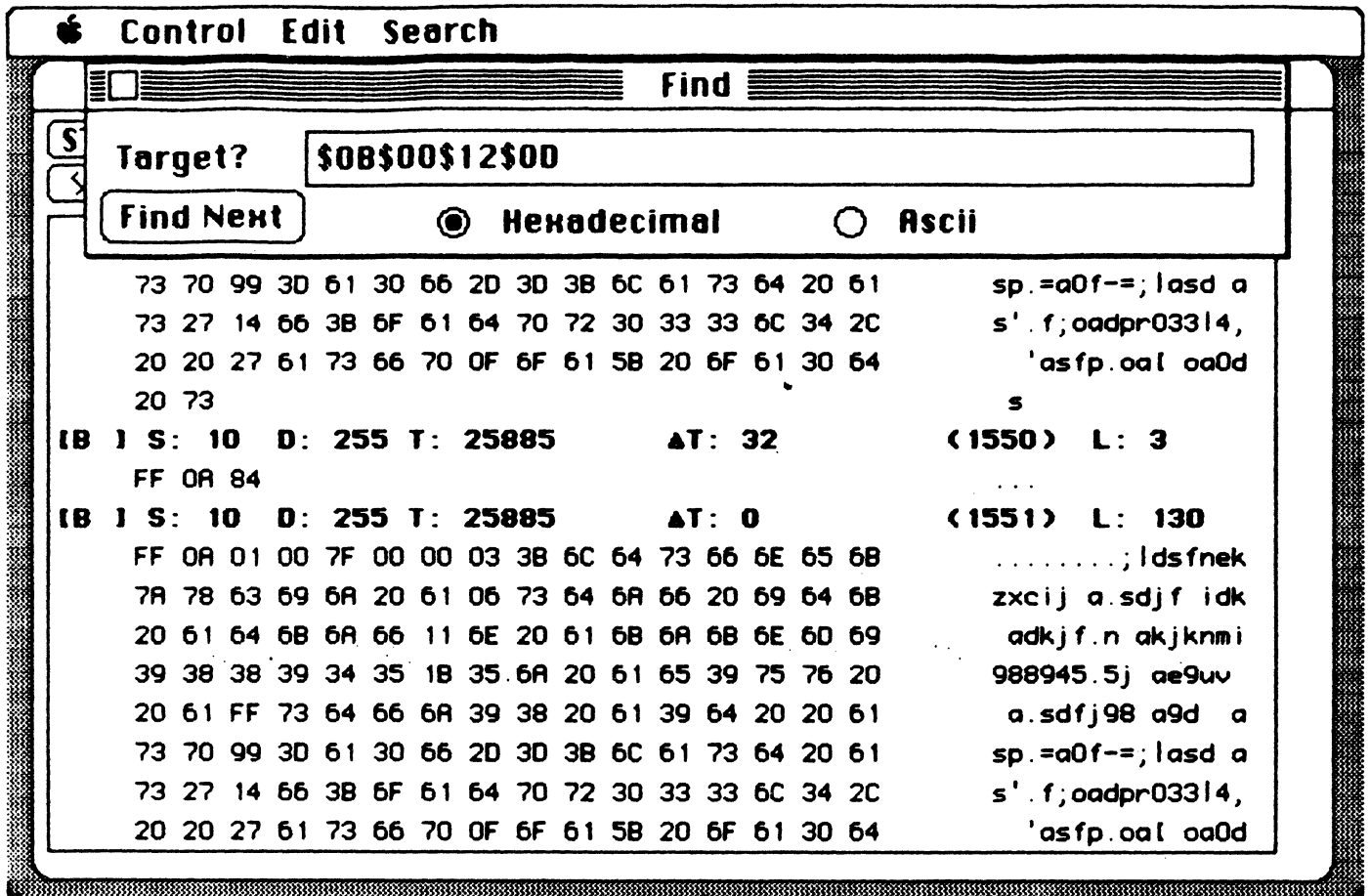


Figure 3: Find Window

AppleTalk Poke

Version 3.1

Gene Tyacke

Network Systems Development

© 1984,1985 - Apple Computer Inc.

AppleTalk Poke is a Macintosh application designed for use by AppleTalk developers. It allows the user to edit/create packets and to send them out on AppleTalk. Developers are expected to use Poke to test their protocol software/hardware implementations for AppleTalk products. Poke uses the Macintosh Protocol Package (MPP) for AppleTalk access. (Details of MPP are discussed elsewhere)

This document describes the features and use of Poke. It is not intended to instruct the user on the capabilities, features, or specifications of MPP or of the various AppleTalk protocols, nor does it discuss the normal use of the Macintosh's standard editing abilities. (For information on AppleTalk protocols, see the corresponding specification/description document.)

Startup

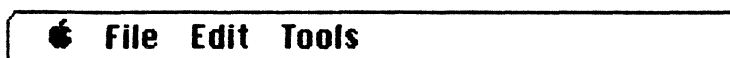
After starting Poke, the MPP driver is loaded in (if it isn't currently in memory) and the main window is brought up (figure 1). This window displays the Poke station's AppleTalk node ID and packet information. At this stage, the packet information indicates that no packets have been loaded into Poke (packet names are all set to empty). Next to each packet name, there is a pair of buttons labeled **EDIT** and **SEND**. Initially, all **SEND** buttons are dimmed (inactive) because no packets have been loaded. The main window includes an area for displaying any appropriate error or status messages.

The program operates in two different states. When started up, it is in

the send state with the main window displayed. When any EDIT button is pressed, it goes into an edit state and the packet editing window is displayed (figure 2). In this state, the selected packet can be edited. Clicking the edit window's OK button will return you back to the main window and the send state.

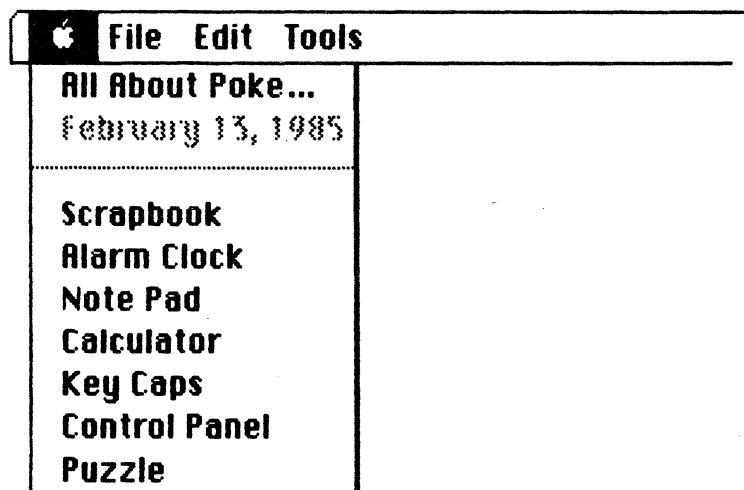
Menus and Commands:

Poke's menu bar contains four menus. These are:



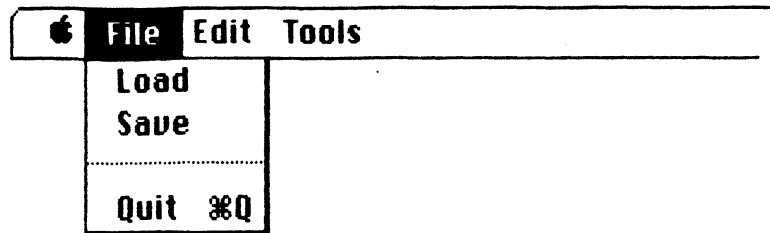
Each menu and its associated commands is displayed and described below:

🍏
Menu:



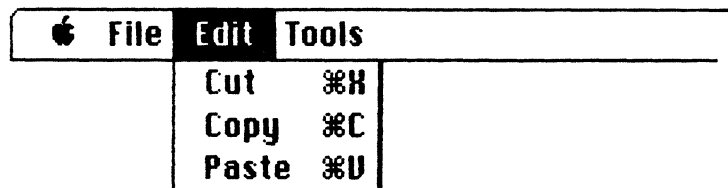
The "Apple" menu allows you to run an available desk accessory or to examine Poke's version information ("All About Poke..."). Selecting the "All About Poke..." command brings up an information window. Clicking the mouse or pressing a key causes this window to disappear and Poke returns to its original state.

**File
Menu:**



The **File** menu allows the user to load from (or save to) a file of 10 prepared or canned packets. The **Load** and **Save** operations follow the standard conventions for file loading and saving. **Note:** Older versions of **Poke** utilize a different file format. You cannot load in packets created by those versions.

**Edit
Menu:**



The **Edit** menu is used only while editing a packet. Please note the keyboard's optional command keys that can be used to invoke this menu's commands.

Tools

Menu:

🍏	File	Edit	Tools
Clone Packet			
Show Packet Length ⌘L			
Helpful Hints ⌘H			
Set Repeat Factor ⌘S			
Abort Send If Error Occurs			

Calculate CheckSum			

"Clone Packet" can only be selected from the main window. When selected, a dialog box appears which asks you for the name of the packet you wish to copy (the source). It also asks for the name of the packet which it is to be copied to (the destination). The names are searched in a top to bottom fashion starting at the top left corner of the main window (figure 1). The first packet whose name matches the one you entered will be chosen. If both source and destination names are found, then the source packet will be copied verbatim to the destination. Otherwise, an error message is displayed.

The "Show Packet Length" command can only be used while editing a packet. It returns the number of bytes in the packet's data field. This count does not include the packet's header, so the actual packet size will be larger. (See the AppleTalk protocol documentation for information on the size of the different headers.) If an error is detected while computing the length, an alert box will be displayed indicating the exact location of the error. Note: If you have entered more data into a packet than is allowed by the corresponding protocol (LAP,DDP,ATP) then Poke will truncate the data (at the end) to the maximum allowed value.

The "Helpful Hints" command allows you to obtain a quick summary of editing instructions. Clicking the mouse or pressing any key will return you to the currently active Poke window.

Packets can be transmitted repeatedly at user specified intervals. The number of times a packet is transmitted and the time interval between transmissions are set by the user by selecting the "Set Repeat Factor" command. This command will allow you to change transmission

information used by the **SEND** command (discussed later).

The delay time interval between transmissions is given in ticks (1 tick = 1/60 of a second). If you enter a number of transmissions value equal to zero, then Poke will keep sending packets out in a closed loop (i.e, indefinitely). When Poke is in such a loop, you can stop the **SEND** operation by either clicking the mouse button or by pressing a key. If you wish to send packets out at the fastest rate possible, enter a zero for the time interval. If this is done, packet statistics will not be displayed in the messages box.

Note: The user specified time interval is achieved only approximately. Network loading and ALAP overhead plus packet transmit time add to this interval.

The "Abort Send If Error Occurs" command is used in conjunction with the **SEND** operation. If selected, a checkmark will appear on the left side of the command informing the user that this feature is active. Now, if an error occurs while sending a packet, the **SEND** operation will abort. To deactivate this feature, select the command again and the checkmark will be removed. This command is especially useful when large numbers of packets are being sent out.

The last command, "Calculate Checksum", may be used in the edit window to replace the existing DDP checksum field with an updated checksum. This command is only valid with packets utilizing the DDP long format (LAP Type field \$2).

Preparing a Packet

When you press the edit button for a particular packet in the main window, the edit window of figure 2 will appear and you will be shown the information of that packet. This window is divided into two main sections: the header and the data, with 18 editing fields. Only one editing field is active at a time. This is indicated by highlighting that field's rectangular box. There are several circular buttons, check boxes and command buttons (OK, CANCEL and CLEAR) used in preparing the packet. The standard Macintosh editing features apply to most of these controls. Some, however, need further clarification. These are:

- o Pressing the **TAB** key causes Poke to verify the information in the current field before activating the next field. The same is true if you press the **RETURN** key (except within the packet's data field). If an error is detected while verifying a field, a beep will sound and Poke will return you back to the error's location. (Possible errors are described at the end of this section.)
- o Clicking the mouse on a different editing field will verify the information in the currently active field. If there are no errors, Poke moves to the field clicked on.
- o You may type data beyond that visible in the field. Leading blanks are automatically removed in the packet header fields.

Entering the Packet's Name

The packet's name is used only to visually distinguish the various packets from others in the main window. It may contain any sequence of printable characters, but it is suggested that you limit the number of characters to 16.

Entering Information in the Header Fields

Information in the packet header fields can be entered in any one of three ways:

- Decimal** : Type in the digits (e.g. 128). This is the default entry type.
- Hexadecimal** : All hexadecimal (hex) numbers are preceded by a dollar sign (e.g., \$80 = 128).
- Binary**: Binary numbers are preceded by a percent sign (e.g., %1111 = \$0F = 15).

Leading zeros are ignored. When a field has been verified, the number entered is automatically converted to hex format.

Possible Error Conditions:

- o Value in field is out of range. (see AppleTalk Protocol documents for the permissible ranges of the various fields)
- o Unknown character in field. Valid digits for decimal format are [0..9] (where this represents a range from zero to nine); valid digits for hex format is [0..9, a..f, A..F], and valid digits for binary numbers are [0,1].

Entering Packet Data Information

The following format must be followed when entering information into the packet data:

Data bytes can be entered into the packet in two ways: by typing in the ASCII character corresponding to the byte's value or by entering the byte's value in its hex form.

To enter the hex form, type a "\$" followed by the two digit hex number (e.g. \$84,\$01). Note that "\$1" is invalid, you must enter "\$01". Bytes whose value corresponds in the ASCII code to a graphic character can be entered by just typing in that character. Example: to enter a byte with the value "\$62", type "b"; for "\$42" type "B"; for "\$31" type "1". Other examples can be found in figures 2 and 3. Note: Since the dollar sign (\$) is a special character, you can only enter it in its hex form "\$24".

Poke will detect errors from the end of the data back to its beginning.

Editing Buttons

Various buttons in the edit window control the information that constitutes the packet. Each set of buttons is described below:

Packet Type: LAP DDP ATP

The Packet Type buttons are used to choose the header type as described in the protocols document. After clicking on a button, only the fields appropriate for that protocol type will be shown. The default is ATP. Only one button may be selected at a time.

Req Rsp Rel

These three buttons are only used for an ATP packet. They are used to format an ATP request, response or release packet. The default is Req. As above, only one button may be chosen at a time.

HO EOM STS

Each of these check boxes represents the corresponding bit in the ATP control field. If checked, the corresponding ATP control field bit will be set; otherwise the bit is cleared.

Packet Data Display
 Hex ASCII

The Packet Data Display buttons allow the user to select the type of display for the packet's data: hex strings or mixed ASCII and hex. [Note: This operation may take up to 10 seconds for large packets.] If an error occurs during the format conversion, an error message is displayed and the conversion will abort. You may enter data in either format at any time. The above buttons are used only when the display is updated or when you wish to convert data to the format immediately.

OK

CANCEL

CLEAR

The **OK** button should be pressed when you are through editing the packet. All fields are verified for correctness and the packet length is displayed before returning to the main window. You will also have the option, at this time, to calculate a checksum for the packet. If any errors are detected, you will be returned to the edit window.

The **CANCEL** button terminates the editing session without saving any changes to the packet. The packet is returned to the original form that it had prior to this editing attempt. Poke returns you to the send window.

The **CLEAR** button clears all editing fields and inserts the default information into them.

Sending Packets

To send packets, Poke must be in the send state (i.e., displaying the main window). Any one of the ten packets may be sent by clicking on its active **SEND** button. The number of times the packet will be sent and the delay between each of these transmissions is shown at the top right corner of the main window in the short form:

Rpt Factor = nx : d ticks

where: **n** = number of transmissions

d = time interval between transmissions (in ticks)

If a **SEND** button is inactive, you must first edit the packet. The result of the **SEND** operation is displayed in the message area at the bottom of the main window.

Possible Error Conditions:

- o No error; packet was sent to destination node (or broadcast)
- o -95; Packet was unable to be sent because either the destination node did not respond or the line was sensed "in use" 32 times.

Startup Notes:

When Poke starts up, the MPP driver is opened and initialized. If the open call fails and you are returned a -35 error, you will be forced to exit the program. Most likely the cause of this error will be that the MPP driver is not installed in the System resource file. In addition, if the system heap is fragmented such that the MPP driver cannot get enough memory to load, the same error will be returned.

If the serial port configuration byte (SPConfig) is not set correctly, you will get a -98 error when Poke starts. See the AppleTalk Manager manual for additional information on location and contents of this byte.

Caveats:

- o Editing of the packet's data field will slow down appreciably as its size increases. Whenever possible, display it under the ASCII mode to minimize the number of screen characters.
- o While in the ASCII display, all characters in the printable ASCII range (\$20-\$7E) and RETURN (\$0D) will be displayed in their ASCII form, even if they were entered as hex strings.
- o The packet data field is limited to 55 lines. Even short packets (e.g., entering more than 55 carriage returns in the packet's data field in ASCII mode) can go out of the scrolling range.
- o Numbers cannot be entered into the packet's data field in decimal or binary format.
- o In no case can the size of the packet be greater than 603 bytes, including ALAP header.
- o If an error occurs while verifying or converting the packet's data field, the information at the error location may change, as Poke tries to back out of the error gracefully.
- o If you have chosen DDP or ATP packet types from the edit window, DDP long format will always be displayed, even if the ALAP type of \$01 (short format) was entered.
- o If you enter more than 600 bytes of packet data, the checksum calculation may not work correctly until you have exited and reentered the editing window. (This will truncate off all excess data from the end of the packet).

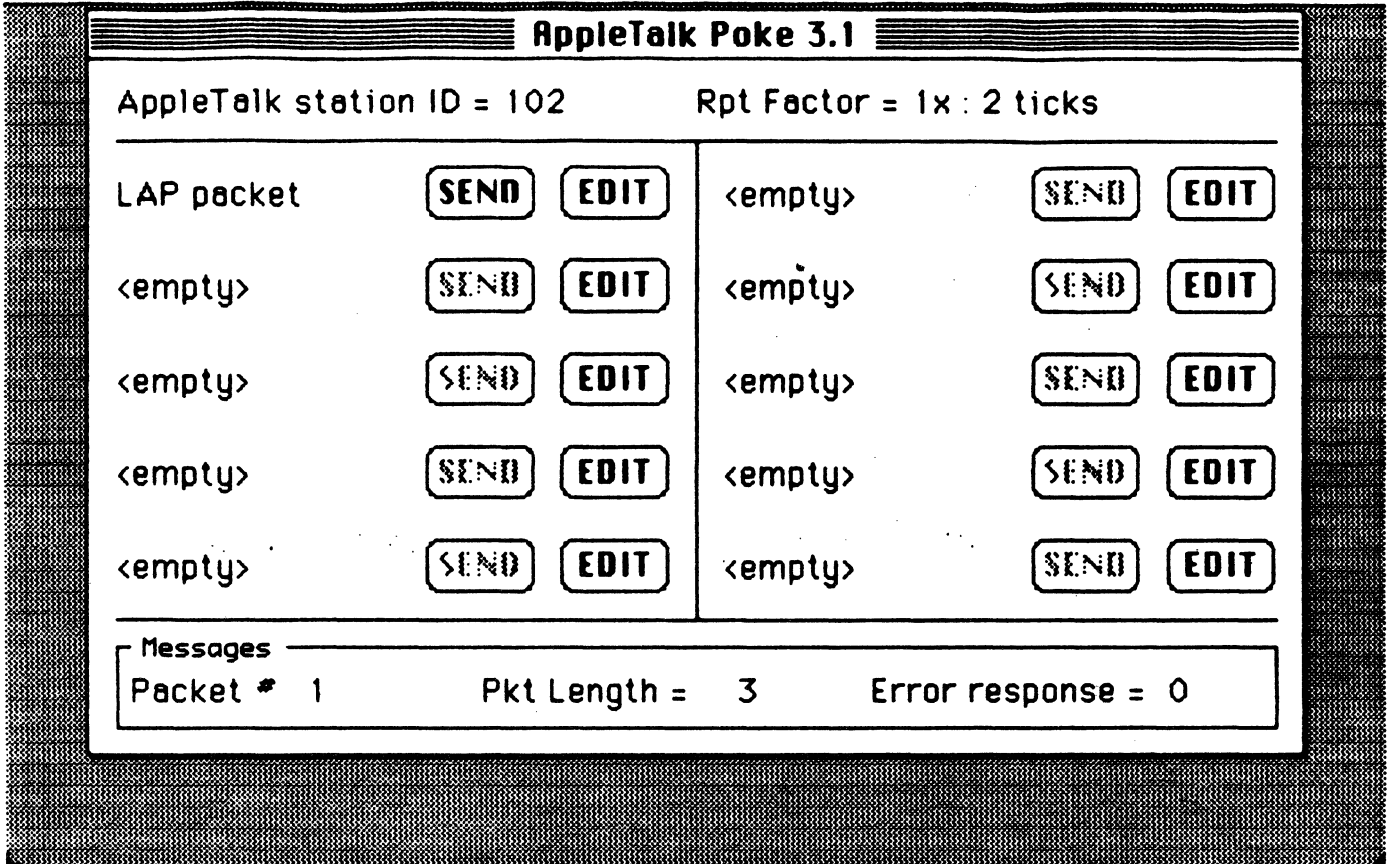


Figure 1. Main Window

Packet Name: Packet Type: LAP DDP ATP

Header Data

LAP

Dest Node Addr: LAP Type:

DDP

Hop Count: Dest Skt #: Src Skt #:
DDP Type: Dest Node Addr: Src Node Addr:
Checksum: Dest Net #: Src Net #:

ATP

Req Rsp Rel Trans ID: HO EOM STS

BitMap: U1: U2: U3: U4:

Packet Data

This is ASCII data. If I wish to enter unprintable characters, I enter characters like: \$00,\$01,\$24 (dollar sign).

Packet Data Display

Hex ASCII

Figure 2. ASCII Display

Packet Name: Packet Type: LAP DDP ATP

Header Data

LAP
Dest Node Addr: LAP Type:

DDP
Hop Count: Dest Skt #: Src Skt #:
DDP Type: Dest Node Addr: Src Node Addr:
Checksum: Dest Net #: Src Net #:

ATP

Packet Data

\$54\$68\$69\$73\$20\$69\$73\$20\$4 1\$53\$43\$49\$49\$20\$64\$6 1\$74\$6 1\$00\$00\$00\$00\$00\$00\$54\$
68\$69\$73\$20\$69\$73\$20\$68\$65\$78\$20\$64\$6 1\$74\$6 1\$20\$69\$6E\$20\$74\$68\$65\$20\$77\$69\$6
E\$64\$6F\$77

Packet Data Display
 Hex ASCII

OK CANCEL CLEAR

Figure 3. Hex Data Display

